SIMULIA

3DEXPERIENCE

# ABAQUS 2026 FD04
# USER SUBROUTINES GUIDE

DASSAULT SYSTEMES

# Contents

**Trademarks and Legal Notices....19**
**Abaqus User Subroutines Guide....20**
**What's New....21**
**About User Subroutines and Utilities....28**
Including User Subroutines in a Model....28
Managing External Databases in Abaqus and Exchanging Information with Other Software....28
Writing a User Subroutine....29
Required INCLUDEs....29
Naming Convention....29
Redefining Variables....29
Compilation and Linking Problems....30
Memory Allocation Considerations....30
Testing and Debugging....30
Terminating an Analysis....30
Models Defined in Terms of an Assembly of Part Instances....30
Reference Coordinate System....30
Node and Element Numbers....31
Set and Surface Names....31
Solution-Dependent State Variables....31
Defining and Updating....31
Allocating Space for Solution-Dependent State Variables....32
Defining Initial Values....33
Element Deletion Controlled by Solution-Dependent State Variables....34
Output....34
Element Solution-Dependent Variables....34
Defining and Updating....34
Allocating Space for Element Solution-Dependent Variables....35
Defining Initial Values....35
Element Deletion Controlled by Element Solution-Dependent Variables....35
Output....36
Alphanumeric Data....36
Precision in Abaqus/Explicit....36
Vectorization in Abaqus/Explicit....36
Parallelization....36
User Subroutine Calls....37
Subroutines That Define Material, Element, or Interface Behavior....37
Subroutines That Define Initial Conditions or Orientations....37
Subroutines That Define Predefined Fields....37
Verification of Subroutine Calls....37
Utility Routines....37
Variables Provided for Use in Utility Routines....38
**About Table Collections, Parameter Tables, and Property Tables....39**
Introduction....39

Table Collection Definition....40
Parameter Table Definition....40
Property Table Definition....41
Defining Tables within a Material Definition....41
Interpolation of Property Table Data....41
Regularizing Property Table Data in Abaqus/Explicit....42
**About Event Series....43**
Introduction....43
Event Series Definition....43
Reading Event Series Data from a SIM Database File....44
Reading Event Series Data from an Alternate Input File....44
Positioning of Event Series Data....45
**About Neural Networks....46**
Introduction....46
Feed-Forward Neural Network Definition....46
Activation Functions....48
Scaling....49
Usage....49
**Abaqus/Standard User Subroutines....51**
CREEP....54
Metals....55
Drucker-Prager Materials....55
Capped Drucker-Prager Materials....57
Gaskets....58
Integration Schemes....58
Constant Stress Assumption When Defining Creep and Swelling....59
Defining Both Plasticity and Creep....59
Interpretation of Stress and Strain Variables....59
User Subroutine Interface....59
Variables to Be Defined....60
Variables That Can Be Updated....61
Variables Passed in for Information....62
Example: Hyperbolic Sine Creep Law....63
DECURRENT....65
User Subroutine Interface....65
Variables to Be Defined....66
Variables Passed in for Information....66
DFLOW....68
User Subroutine Interface....68
Variables to Be Defined....68
Variables Passed in for Information....69
DFLUX....70
User Subroutine Interface....70
Variables to Be Defined....70
Variables Passed in for Information....71
DISP....73
User Subroutine Interface....73
Variables to Be Defined....73

Variables Passed in for Information....74
DLOAD....75
User Subroutine Interface....75
Variables to Be Defined....76
Variables Passed in for Information....76
FILM....78
User Subroutine Interface....78
Variables to Be Defined....78
Variables Passed in for Information....79
FLOW....80
User Subroutine Interface....80
Variables to Be Defined....80
Variables Passed in for Information....80
FRIC....82
User Subroutine Interface....82
Variables to Be Defined....83
Variables That Can Be Updated....84
Variables Passed in for Information....85
FRIC_COEF....87
User Subroutine Interface....87
Variables to Be Defined....89
Variables That Can Be Updated....89
Variables Passed in for Information....90
GAPCON....92
Usage with Contact Pairs, General Contact, and Gap Elements....92
User Subroutine Interface....92
Variables to Be Defined....93
Variables Passed in for Information....93
GAPELECTR....95
User Subroutine Interface....95
Variables to Be Defined....95
Variables Passed in for Information....96
HARDINI....97
Defining Backstress Components....97
User Subroutine Interface....97
Variables to Be Defined....98
Variables Passed in for Information....98
HETVAL....99
User Subroutine Interface....99
Variables to Be Defined....100
Variables That Can Be Updated....100
Variables Passed in for Information....100
MPC....101
Coding Methods....101
Constraints That Involve Degrees of Freedom That Are Not Active in the Model....101
Use with Nodal Coordinate Systems....102
Degree of Freedom Version of User Subroutine MPC....102
Nodal Version of User Subroutine MPC....106

ORIENT....114
User Subroutine Interface....114
Variables to Be Defined....115
Variables Passed in for Information....115
RSURFU....117
Overpenetration Constraint....117
Calculations to Be Performed....117
Defining the Local Surface Geometry....118
User Subroutine Interface....118
Variables to Be Defined....119
Variables Passed in for Information....119
Example: Rigid Punch....120
SDVINI....125
Use of Solution-Dependent State Variables in Other User Subroutines....125
User Subroutine Interface....125
Variables to Be Defined....126
Variables Passed in for Information....126
SIGINI....127
Stress Components....127
Initial Stress Field Equilibrium....127
Stress in Multiscale Material....127
User Subroutine Interface....127
Variables to Be Defined....128
Variables Passed in for Information....128
UAMP....129
Explicit Solution Dependence....129
User Subroutine Interface....129
Variables to Be Defined....130
Variables That Can Be Updated....130
Variables Passed in for Information....131
Example: Amplitude Definition Using Sensor and State Variables....131
UANISOHYPER_INV....134
Enumeration of Invariants....134
Storage of Arrays of Derivatives of the Energy Function....135
Special Considerations for Various Element Types....136
User Subroutine Interface....136
Variables to Be Defined....136
Variables Passed in for Information....137
Example: Anisotropic Hyperelastic Model of Kaliske and Schmidt....138
Additional Reference....142
UANISOHYPER_STRAIN....143
Storage of Strain Components....143
Storage of Arrays of Derivatives of the Energy Function....143
Special Considerations for Various Element Types....145
User Subroutine Interface....145
Variables to Be Defined....145
Variables Passed in for Information....146
Example: Orthotropic Saint-Venant Kirchhoff Model....146

UCOHESIVEOFFSET....151
User Subroutine Interface....151
Variables to Be Defined....151
Variables That Can Be Updated....152
Variables Passed in for Information....152
Example: Keeping Fracture Prop Open....153
UCORR....155
Cross-Correlation for Base Motion Excitation....155
Cross-Correlation for Point Loads and Distributed Loads....155
User Subroutine Interface....155
Variables to Be Defined....156
Variables Passed in for Information....156
UCOUPLEDCONSTRESP....157
User Subroutine Interface....157
Variables to Be Defined....158
Variables That Can Be Updated....162
Variables Passed in for Information....162
UCREEPNETWORK....165
Model Description....165
User Subroutine Interface....166
Variables to Be Defined....168
Variables That Can Be Updated....168
Variables Passed in for Information....168
Example: Bergstrom-Boyce Model....169
UCURE....173
User Subroutine Interface....173
Variables to Be Defined....174
Variables That Can Be Updated....174
Variables Passed in for Information....174
Example: Kamal Model....175
UDAMAGEMF....177
User Subroutine Interface....177
Variables to Be Defined....178
Variables That Can Be Updated....178
Variables Passed in for Information....178
Example: User-Defined Damage Model....179
UDECURRENT....181
User Subroutine Interface....181
Variables to Be Defined....182
Variables Passed in for Information....182
UDEMPOTENTIAL....184
User Subroutine Interface....184
Variables to Be Defined....185
Variables Passed in for Information....185
UDMGINI....186
User Subroutine Interface....186
Variables to Be Defined....186
Variables That Can Be Updated....187

Variables Passed in for Information....187
Example: User-Defined Damage Initiation Criterion with Two Different Failure Mechanisms....188
UDSECURRENT....190
User Subroutine Interface....190
Variables to Be Defined....191
Variables Passed in for Information....191
UECHEMECDBV....193
User Subroutine Interface....193
Variables to Be Defined....194
Variables That Can Be Updated....194
Variables Passed in for Information....194
UECHEMECDDISS....197
User Subroutine Interface....197
Variables to Be Defined....198
Variables That Can Be Updated....198
Variables Passed in for Information....198
UECHEMECDLPL....201
User Subroutine Interface....201
Variables to Be Defined....202
Variables That Can Be Updated....202
Variables Passed in for Information....202
UECHEMECDSEI....205
User Subroutine Interface....205
Variables to Be Defined....206
Variables That Can Be Updated....206
Variables Passed in for Information....206
UECHEMECDSEIREFORM....209
User Subroutine Interface....209
Variables to Be Defined....210
Variables That Can Be Updated....210
Variables Passed in for Information....210
UEIGENSTRAIN....213
User Subroutine Interface....213
Variables to Be Defined....214
Variables That Can Be Updated....214
Variables Passed in for Information....214
UEL....217
Wave Kinematic Data....217
User Subroutine Interface....217
Variables to Be Defined....218
Variables That Can Be Updated....219
Variables Passed in for Information....219
UEL Conventions....222
Usage with General Nonlinear Procedures....222
Usage with Linear Perturbation Procedures....226
Nondiagonal Damping in Linear Perturbation Procedures....228
Example: Structural and Heat Transfer User Element....228
UELEMDRESP....233

User-Defined Element Design Response....233
User Subroutine Interface....233
Variables to Be Defined....234
Variables Passed in for Information....234
UELMAT....236
User Subroutine Interface....236
Variables to Be Defined....237
Variables That Can Be Updated....238
Variables Passed in for Information....238
UELMAT Conventions....241
Usage with General Nonlinear Procedures....241
Usage with Linear Perturbation Procedures....243
Example: Structural User Element with Abaqus Isotropic Linearly Elastic Material....244
UEPACTIVATIONFACET....251
User Subroutine Interface....251
Variables to Be Defined....252
Variables That Can Be Updated....252
Variables Passed in for Information....252
UEPACTIVATIONSETUP....254
User Subroutine Interface....254
Variables to Be Defined....255
Variables That Can Be Updated....255
Variables Passed in for Information....255
UEPACTIVATIONVOL....256
User Subroutine Interface....256
Variables to Be Defined....257
Variables That Can Be Updated....257
Variables Passed in for Information....257
UEREF....260
User Subroutine Interface....260
Variables to Be Defined....261
Variables Passed in for Information....261
UEXPAN....263
User Subroutine Interface....263
Variables to Be Defined....263
Variables That Can Be Updated....264
Variables Passed in for Information....264
UEXTERNALDB....266
User Subroutine Interface....266
Variables to Be Defined....266
Variables Passed in for Information....266
UFIELD....268
Updating Field Variables....268
User Subroutine Interface....268
Variables to Be Defined....269
Variables Passed in for Information....270
UFLUID....271
Density and Fluid Mass....271

User Subroutine Interface....271
Variables to Be Defined....271
Variables That Can Be Updated....272
Variables Passed in for Information....272
UFLUIDCONNECTORLOSS....274
User Subroutine Interface....274
Variables to Be Defined....274
Variables Passed in for Information....275
UFLUIDCONNECTORVALVE....276
User Subroutine Interface....276
Variables to Be Defined....276
Variables Passed in for Information....277
UFLUIDLEAKOFF....278
Leak-Off Models....278
Simple Leak-Off Model....278
Physics-Based Leak-Off Model....280
UFLUIDPIPEFRICTION....282
User Subroutine Interface....282
Variables to Be Defined....282
Variables Passed in for Information....283
UGENS....284
Storage of Membrane and Bending Components....284
Increments for Which Only the Section Stiffness Can Be Defined....284
Stability....285
Convergence Rate....285
Use with Shells That Have Transverse Shear and/or Hourglass Stiffness....285
Use with Continuum Shell Elements....285
User Subroutine Interface....285
Variables to Be Defined....285
Variables That Can Be Updated....286
Variables Passed in for Information....286
UHARD....289
User Subroutine Interface....289
Variables to Be Defined....289
Variables Passed in for Information....290
UHYPEL....291
Special Considerations for Various Element Types....291
User Subroutine Interface....291
Variables to Be Defined....292
Variables Passed in for Information....292
UHYPER....293
Special Considerations for Various Element Types....293
User Subroutine Interface....293
Variables to Be Defined....294
Variables Passed in for Information....295
UHYPER_STRETCH....296
Shells That Calculate Transverse Shear Energy....296
Elements with Hourglassing Modes....296

User Subroutine Interface....296
Variables to Be Defined....297
Variables Passed in for Information....297
UINTER....299
User Subroutine Interface....299
Variables to Be Defined....300
Variables That Can Be Updated....301
Variables Passed in for Information....303
UINTFLUXGEN....305
User Subroutine Interface....305
Variables to Be Defined....306
Variables That Can Be Updated....306
Variables Passed in for Information....307
UMASFL....309
User Subroutine Interface....309
Variables to Be Defined....309
Variables Passed in for Information....309
UMAT....311
Storage of Stress and Strain Components....311
Defining Local Orientations....311
Stability....312
Convergence Rate....312
Viscoelastic Behavior in Frequency Domain....312
Special Considerations for Various Element Types....312
Large Volume Changes with Geometric Nonlinearity....313
Use with Almost Incompressible or Fully Incompressible Elastic Materials....314
Increments for Which Only the Jacobian Can Be Defined....315
Utility Routines....315
User Subroutine Interface....315
Variables to Be Defined....315
Variables That Can Be Updated....317
Variables Passed in for Information....317
Example: Using More than One User-Defined Mechanical Material Model....319
Example: Simple Linear Viscoelastic Material....319
UMATHT....324
Use of Subroutine UMATHT with Coupled Temperature-Displacement and Coupled Thermal-Electrical-Structural Elements....324
User Subroutine Interface....324
Variables to Be Defined....325
Variables That Can Be Updated....325
Variables Passed in for Information....326
Example: Using More than One User-Defined Thermal Material Model....327
Example: Uncoupled Heat Transfer....327
UMDFLUX....330
User Subroutine Interface....330
Variables to Be Defined....330
Variables That Can Be Updated....331
Variables Passed in for Information....331
UMDFLUXSETUP....333

User Subroutine Interface....333
Variables to Be Defined....333
Variables That Can Be Updated....333
Variables Passed in for Information....334
UMESHMOTION....335
Accessing Node Point Data....335
User Subroutine Interface....335
Variables to Be Defined....336
Variables That Can Be Updated....336
Variables Passed in for Information....336
UMIXMODEFATIGUE....338
User Subroutine Interface....338
Variables to Be Defined....338
Variables That Can Be Updated....339
Variables Passed in for Information....339
Example: User-Defined Fatigue Crack Growth Criterion....340
UMOTION....341
User Subroutine Interface....341
Variables to Be Defined....341
Variables Passed in for Information....342
UMULLINS....343
User Subroutine Interface....343
Variables to Be Defined....343
Variables That Can Be Updated....344
Variables Passed in for Information....344
UPOREP....345
User Subroutine Interface....345
Variables to Be Defined....345
Variables Passed in for Information....345
UPRESS....346
User Subroutine Interface....346
Variables to Be Defined....346
Variables Passed in for Information....346
UPSD....348
User Subroutine Interface....348
Variables to Be Defined....348
Variables Passed in for Information....348
URDFIL....349
User Subroutine Interface....349
Variables to Be Defined....349
Variables Passed in for Information....350
Example: Terminating an Analysis upon Exceeding a Mises Stress Limit....350
Example: Terminating an Analysis When the Maximum Mises Stress Value Stops Increasing....351
USDFLD....353
Explicit Solution Dependence....353
Defining Field Variables....353
Accessing Material Point Data....354
State Variables....354

User Subroutine Interface....354
Variables to Be Defined....354
Variables That Can Be Updated....355
Variables Passed in for Information....355
Example: Damaged Elasticity Model....356
USETTLING....359
User Subroutine Interface....359
Variables to Be Defined....359
Variables That Can Be Updated....360
Variables Passed in for Information....360
USUPERELASHARDMOD....362
User Subroutine Interface....362
Variables to Be Defined....362
Variables Passed in for Information....363
UTEMP....365
User Subroutine Interface....365
Variables to Be Defined....365
Variables Passed in for Information....366
UTRACLOAD....367
User Subroutine Interface....367
Variables to Be Defined....367
Variables Passed in for Information....368
UTRS....370
User Subroutine Interface....370
Variables to Be Defined....370
Variables That Can Be Updated....371
Variables Passed in for Information....371
UTRSNETWORK....372
User Subroutine Interface....372
Variables to Be Defined....373
Variables That Can Be Updated....373
Variables Passed in for Information....373
Example: Williams-Landel-Ferry Shift Function....374
UVAREL....377
Accessing Material Point Data....377
Using Element User-Defined Output Variables....377
User Subroutine Interface....378
Variables to Be Defined....378
Variables Passed in for Information....378
UVARM....380
Accessing Material Point Data....380
Using User-Defined Output Variables....380
User Subroutine Interface....381
Variables to Be Defined....381
Variables Passed in for Information....381
Example: Calculation of Stress Relative to Shift Tensor....382
UWAVE....384
User Subroutine Interface....384

Variables to Be Defined....384
Variables Passed in for Information....385
UXFEMCRACK....387
User Subroutine Interface....387
Variables to Be Defined....387
Variables That Can Be Updated....388
Variables Passed in for Information....388
Example: User-Defined Offset for Enriched Cracked Element Surfaces....389
UXFEMNONLOCALWEIGHT....391
User Subroutine Interface....391
Variables to Be Defined....391
Variables Passed in for Information....391
VOIDRI....393
User Subroutine Interface....393
Variables to Be Defined....393
Variables Passed in for Information....393
**Abaqus/Explicit User Subroutines....394**
VDFLUX....396
User Subroutine Interface....396
Variables to Be Defined....396
Variables Passed in for Information....397
VDISP....399
Initial Velocity....399
Acceleration....399
User Subroutine Interface....400
Variables to Be Defined....400
Variables Passed in for Information....400
Example: Imposition of Acceleration on a Rigid Body with Nonzero Initial Velocity....401
VDLOAD....404
User Subroutine Interface....404
Variables to Be Defined....405
Variables Passed in for Information....405
VEXTERNALDB....407
Dynamic Exchange of Data with Other Abaqus User Subroutines and External Programs....407
User Subroutine Interface....407
Variables to Be Defined....409
Variables That Can Be Updated....409
Variables Passed in for Information....410
VFABRIC....412
Component Ordering in Tensors....412
Initial Calculations and Checks....413
Orientation of the Fabric Yarn....413
Material Point Deletion....413
User Subroutine Interface....413
Variables to Be Defined....414
Variables That Can Be Updated....414
Variables Passed in for Information....415
Example: Using More than One User-Defined Material Model....416

Example: Influence of Nonorthogonal Material Directions in Highly Anisotropic Elastic Material.....417
VFILM.....419
User Subroutine Interface.....419
Variables to Be Defined.....420
Variables Passed in for Information.....420
VFRIC.....422
Terminology.....422
Local Coordinate System.....422
Frictional Forces.....423
User Subroutine Interface.....424
Variables to Be Defined.....424
Variables That Can Be Updated.....425
Variables Passed in for Information.....425
VFRIC_COEF.....428
User Subroutine Interface.....428
Variables to Be Defined.....429
Variables That Can Be Updated.....430
Variables Passed in for Information.....430
VFRICTION.....432
Contact Points.....432
Local Coordinate System.....432
Frictional Forces.....433
User Subroutine Interface.....434
Variables to Be Defined.....435
Variables That Can Be Updated.....435
Variables Passed in for Information.....435
VHETVAL.....439
User Subroutine Interface.....439
Variables to Be Defined.....440
Variables Passed in for Information.....440
VSDVINI.....441
Use of Solution-Dependent State Variables in Other User Subroutines.....441
User Subroutine Interface.....441
Variables to Be Defined.....442
Variables Passed in for Information.....442
VUAMP.....443
Explicit Solution Dependence.....443
User Subroutine Interface.....443
Variables to Be Defined.....444
Variables That Can Be Updated.....444
Variables Passed in for Information.....445
Example: Amplitude Definition Using Sensor and State Variables.....445
VUANISOHYPER_INV.....448
Enumeration of Invariants.....448
Storage of Arrays of Derivatives of Energy Function.....449
Special Considerations for Shell Elements.....450
Material Point Deletion.....450
User Subroutine Interface.....450

Variables to Be Defined....451
Variables Passed in for Information....451
Example: Using More than One User-Defined Anisotropic Hyperelastic Material Model....452
Example: Anisotropic Hyperelastic Model of Kaliske and Schmidt....452
Additional Reference....456
VUANISOHYPER_STRAIN....457
Component Ordering in Tensors....457
Special Consideration for Shell Elements....459
Material Point Deletion....459
User Subroutine Interface....459
Variables to Be Defined....460
Variables Passed in for Information....460
Example: Using More than One User-Defined Anisotropic Hyperelastic Material Model....461
Example: Orthotropic Saint-Venant Kirchhoff Model....461
VUCHARLENGTH....466
Defining Characteristic Element Length....466
Array of Element Type and Geometric Properties....466
Elements....467
Special Consideration for 8-Node Continuum Shell Elements....467
User Subroutine Interface....467
Variables to Be Defined....468
Variables Passed in for Information....468
VUCREEPNETWORK....470
Model Description....470
User Subroutine Interface....471
Variables to Be Defined....472
Variables That Can Be Updated....473
Variables Passed in for Information....473
Example: Power Law Strain Hardening Model....474
VUEL....477
User Subroutine Interface....477
Variables to Be Defined....478
Variables Passed in for Information....479
VUEL Conventions....481
Usage with General Nonlinear Procedures....482
Example: Structural User Element....482
VUEOS....486
User Subroutine Interface....486
Variables to Be Defined....487
Variables That Can Be Updated....487
Variables Passed in for Information....487
Example: User Subroutine VUEOS to Reproduce Results Obtained with *EOS, TYPE=USUP....488
VUEXPAN....490
User Subroutine Interface....490
Variables to Be Defined....491
Variables Passed in for Information....491
VUFIELD....493
Updating Field Variables....493

User Subroutine Interface....493
Variables to Be Defined....494
Variables Passed in for Information....495
VUFLUIDEXCH....497
Conventions for Defining Mass Flow/Heat Energy Flow Rate....497
User Subroutine Interface....497
Variables to Be Defined....498
Variables That Can Be Updated....498
Variables Passed in for Information....498
VUFLUIDEXCHEFFAREA....500
Defining Effective Area....500
User Subroutine Interface....500
Variables to Be Defined....501
Variables Passed in for Information....501
VUGENS....503
Storage of Membrane and Bending Components....503
Stability....503
Use with Shells That Have Transverse Shear....503
Use with Continuum Shell Elements....504
User Subroutine Interface....504
Variables to Be Defined....505
Variables That Can Be Updated....505
Variables Passed in for Information....505
VUHARD....508
User Subroutine Interface....508
Variables to Be Defined....509
Variables Passed in for Information....509
VUHYPER....511
Special Considerations for Shell Elements....511
Material Point Deletion....511
User Subroutine Interface....511
Variables to Be Defined....512
Variables Passed in for Information....513
Example: Neo-Hookean Hyperelastic Model....513
VUHYPER_STRETCH....515
Special Considerations for Shell Elements....515
Material Point Deletion....515
User Subroutine Interface....515
Variables to Be Defined....516
Variables Passed in for Information....517
Example: Hencky Hyperelastic Model....518
VUINTER....520
Terminology....520
Local Coordinate System....520
Conventions for Heat Flux and Stress....521
User Subroutine Interface....521
Variables to Be Defined....522
Variables That Can Be Updated....522

Variables Passed in for Information....523
VUINTERACTION....525
Terminology....525
Local Coordinate System....526
Conventions for Stress and Heat Flux....527
User Subroutine Interface....527
Variables to Be Defined....528
Variables That Can Be Updated....528
Variables Passed in for Information....529
VUMAT....533
Component Ordering in Tensors....533
Initial Calculations and Checks....534
Defining Local Orientations....534
Special Considerations for Various Element Types....534
Deformation Gradient....535
Objective Stress Rates....535
Defining Effective Modulus to Control Time Incrementation in Abaqus/Explicit....536
Material Point Deletion....536
User Subroutine Interface....536
Variables to Be Defined....537
Variables That Can Be Updated....537
Variables Passed in for Information....537
Example: Using More than One User-Defined Material Model....540
Example: Elastic/Plastic Material with Kinematic Hardening....540
VUMATHT....545
User Subroutine Interface....545
Variables to Be Defined....546
Variables That Can Be Updated....546
Variables Passed in for Information....546
Example: Using More than One User-Defined Material Model....547
Example: A Simple Thermal Material....547
VUMULLINS....550
Material Point Deletion....550
User Subroutine Interface....550
Variables to Be Defined....551
Variables That Can Be Updated....551
Variables Passed in for Information....551
Example: Hyperelasticity with Softening....552
Additional Reference....553
VUSDFLD....554
Explicit Solution Dependence....554
Defining Field Variables....554
State Variables....555
Accessing Material Point Data....555
Component Ordering in Symmetric Tensors....555
User Subroutine Interface....555
Variables to Be Defined....556
Variables That Can Be Updated....556

Variables Passed in for Information....556
Example: Damaged Elasticity Model....557
VUSUPERELASHARDMOD....561
User Subroutine Interface....561
Variables to Be Defined....562
Variables Passed in for Information....562
VUTEMP....564
User Subroutine Interface....564
Variables to Be Defined....565
Variables Passed in for Information....566
VUTRS....567
User Subroutine Interface....567
Variables to Be Defined....568
Variables That Can Be Updated....568
Variables Passed in for Information....568
VUTRSNETWORK....570
User Subroutine Interface....570
Variables to Be Defined....571
Variables That Can Be Updated....571
Variables Passed in for Information....571
Example: Williams-Landel-Ferry Shift Function....572
VUVISCOSITY....574
User Subroutine Interface....574
Variables to Be Defined....575
Variables Passed in for Information....575
Example: Cross Viscosity Model....575
VWAVE....577
User Subroutine Interface....577
Variables to Be Defined....579
Variables That Can Be Updated....579
Variables Passed in for Information....579
**General Utility Routines....582**
Execution....583
Obtaining Abaqus Environment Variables....584
Obtaining the Abaqus Job Name....585
Obtaining the Abaqus Output Directory Name....586
Obtaining Parallel Processes Information....587
Printing Messages to the Message or Status File....590
Terminating an Analysis....592
Ensuring Thread Safety....593
Allocatable Arrays....595
Model Data....610
Obtaining Part Information....611
Obtaining Physical Constants in an Abaqus/Standard Analysis....613
Accessing Abaqus Table Collections....614
Accessing Abaqus Event Series Data....626
Evaluating Abaqus Neural Networks....629
History Data....632

Setting a User-Defined Value ....633
Obtaining Sensor Information....634
Element Data....635
Obtaining Material Point Information from User Subroutine UVAREL....636
Obtaining the Element Orientation Name from User Subroutine UVAREL....639
Obtaining the Material Name from User Subroutine UVAREL....640
Obtaining Element Section Information from User Subroutine UVAREL....641
Obtaining Wave Kinematic Data in an Abaqus/Aqua Analysis....643
Node Data....646
Obtaining Node Point Information....647
Obtaining Node to Element Connectivity....649
Material Point Data....651
Obtaining Material Point Data in Abaqus/Standard....652
Obtaining Material Point Data in Abaqus/Explicit....671
Contact Data: Obtaining Contact Constraint Point Information in an Abaqus/Standard Analysis ....679
Utility Routine Interface....679
Variables to Be Provided to the Utility Routine....679
Variables Returned from the Utility Routine....679
Available Variable Keys....680
**Toolpath-Mesh Intersection Utility Routines....682**
Setup Utility Routines....683
PtkSetMeshAndEventSeries....683
PtkSetEventSeriesProperties....683
PtkGetEventSeriesNumDivisionPoints....686
PtkGetEventSeriesDivisionCoordinates....686
PtkSetEventSeriesDivisionPointWeights....687
PtkSetMeshAndScanStrategies....687
PtkSetEigLibrary....690
PtkAssignEigLibToScanStrategies....691
PtkAssignEigLibToES....691
Compute Utility Routine ....692
PtkCompute....692
Data Retrieval Utility Routines ....693
PtkGetScanPatternData....693
PtkGetNumIntersectedElements....693
PtkGetIntersectedElements....694
PtkGetDataAccess....694
PtkGetData....695
**Index....699**
User Subroutines Index....700
User subroutine functions listing....702

# Trademarks and Legal Notices

## Trademarks

Abaqus, **3D**EXPERIENCE®, the 3DS logo, the Compass icon, IFWE, 3DEXCITE, 3DVIA, BIOVIA, CATIA, CENTRIC PLM, DELMIA, ENOVIA, GEOVIA, MEDIDATA, NETVIBES, OUTSCALE, SIMULIA and SOLIDWORKS are commercial trademarks or registered trademarks of Dassault Systèmes, a European company (Societas Europaea) incorporated under French law, and registered with the Versailles trade and companies registry under number 322 306 440, or its subsidiaries in the United States and/or other countries. All other trademarks are owned by their respective owners. Use of any Dassault Systèmes or its subsidiaries trademarks is subject to their express written approval.

## Legal Notices

Abaqus and this documentation may be used or reproduced only in accordance with the terms of the software license agreement signed by the customer, or, absent such agreement, the then current software license agreement to which the documentation relates.

This documentation and the software described in this documentation are subject to change without prior notice.

Dassault Systèmes or its Affiliates shall not be responsible for the consequences of any errors or omissions that may appear in this documentation.

© Dassault Systèmes Americas Corp., 2026.

For a full list of the third-party software contained in this release, please go to the Legal Notices in the Abaqus 2026 HTML documentation, which can be obtained from a documentation installation, or in the SIMULIA Established Products 2026 Program Directory, which is available on *www.3ds.com*.

# Abaqus User Subroutines Guide

The Abaqus User Subroutines Guide describes all the user subroutines available for use in Abaqus analyses and the utility routines that can be used when coding subroutines.

This guide is a part of the Abaqus® documentation collection, which describes all the capabilities of the Abaqus finite element analysis technology used in SIMULIA® applications.

User subroutines:

- are provided to increase the functionality of several Abaqus capabilities for which the usual data input methods alone may be too restrictive;
- provide an extremely powerful and flexible tool for analysis;
- are written as C, C++, or Fortran code and must be included in a model when you execute the analysis, as discussed below;
- must be included and, if desired, can be revised in a restarted run, since they are not saved to the restart files);
- cannot be called one from another; and
- can in some cases call utility routines that are also available in Abaqus.

A variety of utility routines are available to assist in the coding of user subroutines. You include the utility routine inside a user subroutine. When called, the utility routine will perform a predefined function or action whose output or results can be integrated into the user subroutine. Some utility routines are only applicable to particular user subroutines.

For information about the **abaqus make** utility, which is used to create user postprocessing executables and user-defined libraries of Abaqus user subroutines, see *Making User-Defined Executables and Subroutines*.

Most user subroutine interfaces in this guide use the Fortran language, although user subroutines can be written using the C and C++ languages. Similarly, the utility routines can be invoked from within these C and C++ user subroutines. For more information, see *Writing a User Subroutine* and the Knowledge Base item QA00000008005 in the Dassault Systèmes Knowledge Base.

# What's New

This page describes recent changes in Abaqus User Subroutines.

### 2026 FD04

#### Obtaining the Cap Position for Cap Plasticity Materials

You can use utility subroutines `GETVRM` and `GETVRMAVGATNODE` in Abaqus/Standard and `VGETVRM` in Abaqus/Explicit to obtain the cap position (PBCAP) for materials with cap plasticity.

**Benefits:** Using the new output variable provides a meaningful label for identifying the cap position in the cap plasticity model.

For more information, see *Obtaining Material Point Information in an Abaqus/Standard Analysis*, *Obtaining Material Point Information Averaged at a Node*, and *Obtaining Material Point Information in an Abaqus/Explicit Analysis*.

#### Rate-Dependent and Rate-Independent Responses of Rate-Dependent User-Defined Materials

In Abaqus/Explicit, you can use user subroutines `VUCREEPNETWORK`, `VUHARD`, and `VUMAT` to define both the rate-dependent and rate-independent responses of rate-dependent user-defined materials.

**Benefits:** You can run explicit dynamic simulations with rate-dependent or rate-independent material response without having to modify the material definition.

For more information, see *VUCREEPNETWORK*, *VUHARD*, and *VUMAT*.

### 2026 FD03

#### `VUHYPER_STRETCH` (new)

You can use user subroutine `VUHYPER_STRETCH` to define stretch-based hyperelastic models in Abaqus/Explicit.

**Benefits:** The user subroutine extends the functionality of Abaqus/Explicit, enabling you to define stretch-based hyperelastic models that are not otherwise available in Abaqus.
You can now model hyperelastic material behavior in Abaqus/Explicit using user subroutine `VUHYPER_STRETCH` to define the stretch-based strain energy function. Equivalent functionality is available in Abaqus/Standard with user subroutine `UHYPER_STRETCH`.

`VUHYPER_STRETCH` can be used in combination with other material behaviors, such as linear and nonlinear viscoelasticity. When used with the parallel rheological framework, the subroutine has access to the network identifier (`networkID`), which allows the definition of different hyperelastic models for different nonlinear viscoelastic networks.

For more information, see *VUHYPER_STRETCH*.

#### User-Defined Output Variables at Beam Elements

You can use user subroutine UVAREL to define output variables at beam elements in Abaqus/Standard. In earlier releases, this functionality was available only for continuum and shell elements.

**Benefits:** The user subroutine makes it easier for you to compute, create, and elaborate user-defined element output variables and gives you the flexibility to implement a broad variety of data consolidation schemes that can reduce the number of results that must be stored in the output database.
For beam elements, you can also invoke the following new utility routines from user subroutine UVAREL:

- getElMPtVar returns material point information;
- getSecVar returns integration point information;
- getUVarElOrientationName returns the name of the element orientation; and
- getUVarElMaterialName returns the name of the material.

For more information, see *UVAREL*, *Obtaining Material Point Information from User Subroutine UVAREL*, and *Obtaining Element Section Information from User Subroutine UVAREL*.

## 2026 FD02

### Strain Tensors for User-Defined Design Response and Sensitivities in UELEMDRESP

User subroutine UELEMDRESP now allows definition of the user-defined design response and corresponding sensitivities using the strain tensor, the logarithmic strain tensor, or the nominal strain tensor.

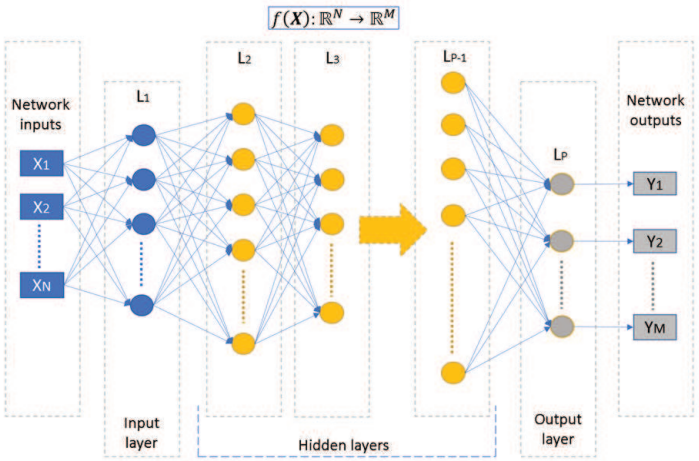
**Benefits:** User-defined strain-based failure criteria provide improved flexibility for evaluating the sensitivity of allowable responses in design and optimization workflows.
You can now define a user-defined failure criterion (as a design response) using the strain tensor, the logarithmic strain tensor, or the nominal strain tensor using user subroutine UELEMDRESP. This new functionality is similar to the user-defined design responses using the stress tensor or the plastic strain tensor available in earlier releases. You can use UELEMDRESP to calculate the strain-based user-defined design responses for static, quasi-static, and implicit transient dynamic analyses.

For more information, see *UELEMDRESP*.

### Evaluating Feed-Forward Neural Networks During Simulations Using Abaqus/Standard

You can now use specialized utility routines in Abaqus/Standard to evaluate feed-forward trained neural networks that were defined in the input file.

**Benefits:** This new functionality bridges advanced machine learning capabilities with finite element analysis, allowing you to dynamically incorporate trained feed-forward neural network predictions—such as constitutive material responses, complex boundary conditions, or load definitions—into simulations.
You can now define trained feed-forward (dense) neural networks in the input file. In a single input file, you can define and use multiple neural networks. Each neural network can have an arbitrary configuration comprised of different numbers of layers, with each layer having a different size and a different activation function. You can then call the neural network evaluation utility routines to obtain the neural network output and the output derivatives with respect to the input vector. The figure below displays a feed-forward neural network with P layers, mapping an $n$-dimensional input vector to an $m$-dimensional output vector.




For more information, see *About Neural Networks* and *Evaluating Abaqus Neural Networks*.

### 2026 FD01

## C/C++ Interfaces to Abaqus Utility Routines

You can now invoke Abaqus utility routines from user subroutines written in C/C++ without any Fortran-CC interoperability issues.

**Benefits:** This extends the functionality of Abaqus/Standard.

In earlier releases, only Fortran interfaces were available. Now you can invoke these utility routines directly from C/C++ user subroutines without any Fortran-C interoperability issues.

For more information, see *Accessing Abaqus Table Collections*, *Obtaining Material Point Information from User Subroutine UVAREL*, and *Obtaining Material Point Information in an Abaqus/Standard Analysis*.

## UHYPER

You can now access the values of the solution-dependent state variables at the beginning of the increment in user subroutine `UHYPER`.

**Benefits:** This extends the functionality of Abaqus/Standard.

The values of the solution-dependent state variables at the beginning of the increment can now be accessed in user subroutine `UHYPER` in the `STATEV(*,2)` array. This allows you to use those values in the definition of your user-defined isotropic hyperelastic materials. In earlier releases, only the current values of the solution-dependent state variables (now in the `STATEV(*,1)` array) were accessible.

For more information, see *UHYPER*.

## 2026 GA

### VUTRSNETWORK (new)

You can use user subroutine VUTRSNETWORK to define a temperature-time shift for a nonlinear viscoelastic network for models defined using the parallel rheological framework in Abaqus/Explicit.

**Benefits:** The user subroutine extends the functionality of Abaqus/Explicit, enabling you to define forms of temperature-time shifts that are not otherwise available in Abaqus.
You can now model temperature-time shift in Abaqus/Explicit using user subroutine VUTRSNETWORK to define the form of the temperature-time shift. This functionality is equivalent to user subroutine UTRSNETWORK previously available in Abaqus/Standard. The VUTRSNETWORK subroutine has access to the network identifier (networkID), which allows the definition of different forms of temperature-time shifts for different nonlinear viscoelastic networks.

For more information, see *VUTRSNETWORK*.

## 2025 FD03

### Electrochemical Procedures

You can use new user subroutines in electrochemical analysis.

**Benefits:** You can simulate a variety of complex electrochemical effects for battery simulations in Abaqus/Standard.

| | |
|---|---|
| DECURRENT | Define electrical current magnitude as a function of position, time, element number, field variables, etc. for coupled thermal-electrical, coupled temperature-displacement electrical, coupled thermal-electrochemical, and coupled temperature-displacement electrochemical analyses |
| UCOUPLEDCONSTRESP | Define a fully coupled constitutive response based on the different physics that is being modeled |
| UECHEMECDBV | Define the Butler-Volmer intercalation current to be used for elements that utilize porous electrode theory in an electrochemical analysis |
| UECHEMECDDISS | Define the electrode dissolution aging current to be used for elements that utilize porous electrode theory in an electrochemical analysis |
| UECHEMECDLPL | Define the lithium plating (LPL) aging current to be used for elements that utilize porous electrode theory in an electrochemical analysis |
| UECHEMECDSEI | Define the Solid Electrolyte Interface (SEI) aging current to be used for elements that utilize porous electrode theory in an electrochemical analysis |
| UECHEMECDSEIREFORM | Define the Solid Electrolyte Interface (SEI) reformation current to be used for elements that utilize porous electrode theory in an electrochemical analysis |
| UEIGENSTRAIN | Define the eigenstrain in a material from different physics fields |
| UEREF | Define the Open Circuit Potential (OCP) to be used for elements that utilize porous electrode theory in an electrochemical analysis |

| | |
|---|---|
| UINTFLUXGEN | Define the internal heat generation in a material from different physics fields |

For more information, see *DECURRENT*, *UCOUPLEDCONSTRESP*, *UECHEMECDBV*, *UECHEMECDDISS*, *UECHEMECDLPL*, *UECHEMECDSEI*, *UECHEMECDSEIREFORM*, *UEIGENSTRAIN*, *UEREF*, and *UINTFLUXGEN*.

### 2025 FD01

#### UHYPER

You can now use user subroutine UHYPER to define different hyperelastic models for different viscoelastic networks when the subroutine is used with the parallel rheological framework.

**Benefits:** The user subroutine extends the functionality of Abaqus/Standard.

The network identification number (networkID) is now passed as a parameter to the user subroutine UHYPER when the subroutine is used with the parallel rheological framework. This allows you to define different hyperelastic models for different nonlinear viscoelastic networks. In earlier releases, this capability was available only with element overlay.

For more information, see *UHYPER*.

#### UHYPER_STRETCH

You can now use user subroutine UHYPER_STRETCH to define different hyperelastic models for different viscoelastic networks when the subroutine is used with the parallel rheological framework.

**Benefits:** The user subroutine extends the functionality of Abaqus/Standard.

The network identification number (networkID) is now passed as a parameter to user subroutine UHYPER_STRETCH when the subroutine is used with the parallel rheological framework. This allows you to define different hyperelastic models for different nonlinear viscoelastic networks. In earlier releases, this capability was available only with element overlay.

For more information, see *UHYPER_STRETCH*.

#### VUHYPER (new)

You can use user subroutine VUHYPER to define hyperelastic material models in Abaqus/Explicit.

**Benefits:** The user subroutine extends the functionality of Abaqus/Explicit, enabling you to define strain energy functions that are not otherwise available in Abaqus.
You can now model hyperelastic material behavior in Abaqus/Explicit using user subroutine VUHYPER to define the strain energy function. This functionality is equivalent to user subroutine UHYPER previously available in Abaqus/Standard. VUHYPER can be used in combination with other material behaviors, such as linear and nonlinear viscoelasticity. When used with the parallel rheological framework, the subroutine has access to the network identifier (networkID), which allows the definition of different hyperelastic models for different nonlinear viscoelastic networks.

For more information, see *VUHYPER*.

### PDF Guides Available

You can download PDF versions of the Abaqus guides from each guide overview.

**Benefits:** You can easily access the PDF versions of the guides.
In earlier releases, PDF versions of the guides were updated only for the GA (General Availability) release and could be downloaded from the Dassault Systèmes Knowledge Base. As of 2025 FD01, the PDF guides will be updated with each release of the HTML versions of the guides.

For more information, see *Abaqus User Subroutines Guide*.

## 2024 FD03

### `UVAREL` (new)

You can use the new user subroutine `UVAREL` to define output variables at the element and integration point level in Abaqus/Standard.

**Benefits:** The user subroutine makes it easier for you to compute, create, and elaborate user-defined element output variables and provides you with the flexibility to implement a broad variety of data consolidation schemes that can reduce the number of results that must be stored in the output database.

User subroutine `UVAREL` gives you a method to define variables at the element level to reduce the amount of output data if the results at every material point are not needed. The reduction of output data can be significant, especially for composite elements with many layers.

The user-defined element output variables that are defined in user subroutine `UVAREL` usually depend on other Abaqus output variables such as stresses, strains, and section forces. You can access these output quantities by calling the new utility routines `getElMPtVar` and `getSecVar`.

You can invoke the following new utility routines from user subroutine `UVAREL`:

- `getElMPtVar` returns material point information
- `getSecVar` returns integration point information
- `getUVarElOrientationName` returns the name of the element orientation
- `getUVarElMaterialName` returns the name of the material

For more information, see *UVAREL*, **ELEMENT USER OUTPUT VARIABLES*, *Obtaining Material Point Information from User Subroutine UVAREL*, *Obtaining the Element Orientation Name from User Subroutine UVAREL*, *Obtaining the Material Name from User Subroutine UVAREL*, *Obtaining Element Section Information from User Subroutine UVAREL*, and *UVAREL*.

### User Subroutine and Utilities Assistance Consolidated in this Guide

Conceptual information about user subroutines and utilities available in Abaqus is now consolidated in this guide.

**Benefits:** Moving the content from the *Abaqus Analysis Guide* makes it easier for you to access the information as you review the reference topics in this guide.

For more information, see *About User Subroutines and Utilities*, *About Table Collections, Parameter Tables, and Property Tables*, and *About Event Series*.

### 2024 FD02

VDLOAD

You can now access displacement and acceleration information in user subroutine VDLOAD.

**Benefits:** Accessing the displacements and accelerations information in the user subroutine is useful when the distributed loads are a function of the displacements and accelerations.

The array velocity available in earlier releases in the VDLOAD argument list is now named vua. It is also extended to include velocities, displacements, and accelerations, giving you the option to define distributed loads as function of these quantities with user subroutine VDLOAD. This change does not affect existing VDLOAD implementations based on the interface available in earlier releases.

For more information, see *VDLOAD*.

### 2024 FD01

UEXPAN

You can now define field expansion strains and, in a porous medium, pore fluid field expansion strains directly in user subroutine UEXPAN.

**Benefits:** Defining the values in the user subroutine is particularly useful when the field expansion strains are complicated functions of temperature, field variables, or state variables.

The argument list of user subroutine UEXPAN has been enhanced to provide additional information that you can use to discern if the call corresponds to a thermal or a field expansion definition, the field number and, in the case of a porous medium, if the call corresponds to the expansion definition for the pore fluid. This change does not affect existing UEXPAN implementations based on the interface available in earlier releases.

For more information, see *UEXPAN* and *UEXPAN*.

# About User Subroutines and Utilities

## Overview

User subroutines:

- are provided to increase the functionality of several Abaqus capabilities for which the usual data input methods alone might be too restrictive;
- provide an extremely powerful and flexible tool for analysis;
- are written as C, C++, or Fortran code and must be included in a model when you execute the analysis, as discussed below;
- must be included and, if desired, can be revised in a restarted run, since they are not saved to the restart files (see *Restarting an Analysis*);
- cannot be called one from another; and
- can in some cases call utility routines that are also available in Abaqus.

The available user subroutines for Abaqus are listed in *Abaqus/Standard User Subroutines* and *Abaqus/Explicit User Subroutines*.

## Including User Subroutines in a Model

You can include one or more user subroutines in a model by specifying the name of a C, C++, or Fortran source or precompiled object file that contains the subroutines. Details are provided in *Abaqus/Standard and Abaqus/Explicit Execution*.

**Input File Usage:** Enter the following input on the command line:

**abaqusjob**=*job-name* **user**={*source-file* | *object-file*}

**Abaqus/CAE Usage:** Job module: job editor: **General**: **User subroutine file**

## Managing External Databases in Abaqus and Exchanging Information with Other Software

In Abaqus it is sometimes desirable to set up the runtime environment and manage interactions with external data files or parallel processes that are used in conjunction with user subroutines. For example, there might be history-dependent quantities to be computed externally, once per increment, for use during the analysis; or output quantities that are accumulated over multiple elements in COMMON block variables within user subroutines might need to be written to external files at the end of a converged increment for postprocessing. Such operations

can be performed with user subroutine *UEXTERNALDB* in Abaqus/Standard and *VEXTERNALDB* in Abaqus/Explicit. This user interface can potentially be used to exchange data with another code, allowing for "stagger" between Abaqus and another code.

# Writing a User Subroutine

User subroutines should be written with great care. To ensure their successful implementation, the rules and guidelines below should be followed. For a detailed discussion of the individual subroutines, including coding interfaces and requirements, refer to the *Abaqus User Subroutines Guide*.

## Required INCLUDEs

Every user subroutine written in Fortran must include one of the following statements as the first statement after the argument list:

- Abaqus/Standard:

```
include 'aba_param.inc'
```

- Abaqus/Explicit:

```
include 'vaba_param.inc'
```

If variables are exchanged between the main user subroutine and subsequent subroutines, you should specify the above include statement in all the subroutines to preserve precision.

Every C and C++ user subroutine must include the statement

```
#include <omi_for_c.h>
```

This file contains macros for the Fortran-to-C interface interoperability.

The files `aba_param.inc`, `vaba_param.inc`, and `omi_for_c.h` are installed on the system by the Abaqus installation procedure and contain important installation parameters. These statements tell the Abaqus execution procedure, which compiles and links the user subroutine with the rest of Abaqus, to include the `aba_param.inc` or `vaba_param.inc` file automatically. It is not necessary to find the file and copy it to any particular directory; Abaqus will know where to find it.

## Naming Convention

If user subroutines call other subroutines or use COMMON blocks to pass information, such subroutines or COMMON blocks should begin with the letter K since this letter is never used to start the name of any subroutine or COMMON block in Abaqus.

User subroutines written in C or C++ will be called from Fortran; therefore, they must conform to the Fortran calling conventions: the name of a C or C++ subroutine must be wrapped in a `FOR_NAME` macro; for example,

```
 extern "C" void   FOR_NAME(film,FILM) (double & arg1, ...) { ... }
```

and the arguments must be passed and received by reference.

## Redefining Variables

User subroutines must perform their intended function without overwriting other parts of Abaqus. In particular, you should redefine only those variables identified in this chapter as "variables to be defined." Redefining "variables passed in for information" will have unpredictable effects.

### Compilation and Linking Problems

If problems are encountered during compilation or linking of the subroutine, make sure that the Abaqus environment file (the default location for this file is the `site` subdirectory of the Abaqus installation) contains the correct compile and link commands as specified in *System customization parameters*. These commands should have been set up by the Abaqus site manager during installation. The number and type of arguments must correspond to what is specified in the documentation. Mismatches in type or number of arguments might lead to platform-dependent linking or runtime errors.

### Memory Allocation Considerations

Your user subroutine will share memory resources with Abaqus. When you need to use large arrays or other large data structures, you should allocate their memory dynamically, so that memory is allocated from the heap and not the stack. Failure to dynamically allocate large arrays might result in stack overflow errors and an exit of your Abaqus analysis. For an example of dynamic allocation using native Fortran allocatable arrays, refer to *Creation of a data file to facilitate the postprocessing of elbow element results: FELBOW*. Abaqus also provides another, more convenient way for users to allocate their own storage (see *Allocatable Arrays*).

### Testing and Debugging

When developing user subroutines, test them thoroughly on smaller examples in which the user subroutine is the only complicated aspect of the model before attempting to use them in production analysis work.

If needed, debug output can be written to the Abaqus/Standard message (`.msg`) file using Fortran unit 7 or to the Abaqus/Standard data (`.dat`) file or the Abaqus/Explicit log (`.log`) file using Fortran unit 6; these units should not be opened by your routines since they are already opened by Abaqus.

Fortran units 15 through 18 or units greater than 100 can be used to read or write other user-specified information. The use of other Fortran units might interfere with Abaqus file operations; see *Fortran Unit Numbers*. You must open these Fortran units; and because of the use of scratch directories, the full pathname for the file must be used in the OPEN statement.

Environment variable `ABA_PARALLEL_DEBUG` can be set to turn on verbosity, to display compilation and linking commands, and to enable debugging of user subroutines.

### Terminating an Analysis

Utility routine `XIT` (Abaqus/Standard) or `XPLB_EXIT` (Abaqus/Explicit) should be used instead of `STOP` when terminating an analysis from within a user subroutine. This will ensure that all files associated with the analysis are closed properly (*Terminating an Analysis*).

## Models Defined in Terms of an Assembly of Part Instances

An Abaqus model can be defined in terms of an assembly of part instances (see *Assembly Definition*).

### Reference Coordinate System

Although a local coordinate system can be defined for each part instance, all variables (such as current coordinates) are passed to a user subroutine in the global coordinate system, not in a part-local coordinate system. The only exception to this rule is when the user subroutine interface specifically indicates that a variable is in a user-defined local coordinate system (*Orientations*, or *Transformed Coordinate Systems*). The local coordinate system

originally might have been defined relative to a part coordinate system, but it was transformed according to the positioning data given for the part instance. As a result, a new local coordinate system was created relative to the assembly (global) coordinate system. This new coordinate system definition is the one used for local orientations in user subroutines.

### Node and Element Numbers

The node and element numbers passed to a user subroutine are internal numbers generated by Abaqus. These numbers are global in nature; all internal node and element numbers are unique. If the original number and the part instance name are required, call the utility subroutine `GETPARTINFO` (Abaqus/Standard) or `VGETPARTINFO` (Abaqus/Explicit) from within your user subroutine (see *Obtaining Part Information*). The expense of calling these routines is not trivial, so minimal use of them is recommended.

Another utility subroutine, `GETINTERNAL` (Abaqus/Standard) or `VGETINTERNAL` (Abaqus/Explicit), can be used to retrieve the internal node or element number corresponding to a given part instance name and local number.

### Set and Surface Names

Set and surface names passed to user subroutines are always prefixed by the assembly and part instance names, separated by underscores. For example, a surface named `surf1` belonging to part instance `Part1-1` in assembly `Assembly1` will be passed to a user subroutine as

```
Assembly1_Part1-1_surf1
```

## Solution-Dependent State Variables

Solution-dependent state variables are values that can be defined to evolve with the solution of an analysis.

### Defining and Updating

Any number of solution-dependent state variables can be used in the following user subroutines:

- *CREEP*
- *FRIC*
- *FRIC_COEF*
- *HETVAL*
- *UANISOHYPER_INV*
- *UANISOHYPER_STRAIN*
- *UCREEPNETWORK*
- *UEL*
- *UEXPAN*
- *UGENS*
- *UHARD*
- *UHYPER*
- *UHYPER_STRETCH*
- *UINTER*

- *UMAT*
- *UMATHT*
- *UMULLINS*
- *USDFLD*
- *UTRS*
- *UTRSNETWORK*
- *VFABRIC*
- *VFRIC*
- *VFRICTION*
- *VHETVAL*
- *VUANISOHYPER_INV*
- *VUANISOHYPER_STRAIN*
- *VUCREEPNETWORK*
- *VUFLUIDEXCH*
- *VUHARD*
- *VUINTER*
- *VUINTERACTION*
- *VUMAT*
- *VUMATHT*
- *VUMULLINS*
- *VUSDFLD*
- *VUTRS*
- *VUTRSNETWORK*
- *VUVISCOSITY*
- *VWAVE*

The state variables can be defined as a function of any other variables appearing in these subroutines and can be updated accordingly. Solution-dependent state variables should not be confused with field variables, which might also be needed in the constitutive routines and can vary with time; field variables are discussed in detail in *Predefined Fields*.

Solution-dependent state variables used in *VFRIC*, *VUINTER*, *VFRICTION*, and *VUINTERACTION* are defined as state variables at secondary nodes and are updated with other contact variables.

## Allocating Space for Solution-Dependent State Variables

You must allocate space for each of the solution-dependent state variables at every applicable integration point or contact secondary node.

Separate user subroutine groups have been identified that differ in the way the number of solution-dependent state variables is defined. These groups are described below. Solution-dependent state variables can be shared by subroutines within the same group; they cannot be shared between subroutines belonging to different groups.

**Input File Usage:** For most subroutines the number of such variables required at the points or nodes is entered as the only value on the first data line of the **DEPVAR* option, which should

be included as part of the material definition for every material in which solution-dependent state variables are to be considered:

**DEPVAR*

For subroutines that do not use the material behavior defined with the **MATERIAL* option, the **DEPVAR* option is not used.

For subroutine *UEL*:

**USER ELEMENT*, VARIABLES=*number of variables*

For subroutine *UGENS*:

**SHELL GENERAL SECTION*, USER, VARIABLES=*number of variables*

For subroutines *FRIC* and *VFRIC*:

**FRICTION*, USER, DEPVAR=*number of variables*

For subroutines *UINTER* and *VUINTER*:

**SURFACE INTERACTION*, USER, DEPVAR=*number of variables*

For subroutine *VFRICTION*:

**FRICTION*, USER=FRICTION, DEPVAR=*number of variables*

For subroutine *VUFLUIDEXCH*:

**FLUID EXCHANGE PROPERTY*, TYPE=USER, DEPVAR=*number of variables*

For subroutine *VUINTERACTION*:

**SURFACE INTERACTION*, USER=INTERACTION, DEPVAR=*number of variables*

For subroutine *VWAVE*:

**WAVE*, TYPE=USER, DEPVAR=*number of variables*

**Abaqus/CAE Usage:** For most subroutines the number of such variables required at the points or nodes is entered as part of the material definition for every material in which solution-dependent state variables are to be considered:

Property module: material editor: **General->Depvar**: **Number of solution-dependent state variables**

## Defining Initial Values

You can define the initial values of solution-dependent state variable fields directly or in Abaqus/Standard through a user subroutine. The initial values of solution-dependent state variables for contact or for user subroutine *VWAVE* in Abaqus/Explicit are assigned as zero internally.

#### Defining Initial Values Directly

You can define the initial values in a tabular format for elements and/or element sets. See *Initial Conditions* for additional details.

**Input File Usage:** **INITIAL CONDITIONS*, TYPE=SOLUTION

#### Defining Initial Values in a User Subroutine in Abaqus/Standard

For complicated cases in Abaqus/Standard you can call user subroutine *SDVINI* so that dependencies on coordinates, element numbers, etc. can be used in the definition of the variable field.

**Input File Usage:** **INITIAL CONDITIONS*, TYPE=SOLUTION, USER

### Element Deletion Controlled by Solution-Dependent State Variables

If element deletion controlled by state variables is defined in an analysis (see *User-Defined Mechanical Material Behavior*), the value of the state variable that is controlling the deletion flag can be modified by any of the user subroutines in which state variables are used, provided that the user subroutine is called at a material point.

### Output

User-defined, solution-dependent state variables can be written to the data (`.dat`) file, the output database (`.odb`) file, and the results (`.fil`) file; the output identifiers SDV and SDV*n* are available as element integration variables (see *Abaqus/Standard Output Variable Identifiers* and *Abaqus/Explicit Output Variable Identifiers*). Output of these variables is not available for user subroutines *VFRIC*, *VUINTER*, *VFRICTION*, *VUINTERACTION*, and *VWAVE*.

## Element Solution-Dependent Variables

Element solution-dependent variables are values that can be defined at each applicable element to evolve with the solution of an Abaqus/Standard analysis. Element solution-dependent variables are different from solution-dependent state variables, which are defined at each applicable integration point or node.

### Defining and Updating

Any number of element solution-dependent variables can be used in the following user subroutines:

- *UEPACTIVATIONFACET*
- *UEPACTIVATIONVOL*
- *UMDFLUX*

Element solution-dependent variables can be defined as a function of any other variables appearing in these subroutines and can be updated accordingly. Element solution-dependent variables should not be confused with field variables, which might also be needed in the constitutive routines and can vary with time; field variables are discussed in detail in *Predefined Fields*.

## Allocating Space for Element Solution-Dependent Variables

You must allocate space for element solution-dependent variables at every applicable element. In each allocation you specify the number of element solution-dependent variables required at each element. You can optionally specify user-defined output keys and descriptions for some or all variables. The allocation should be used in conjunction with an element section definition, which defines section properties for elements in which element solution-dependent variables are to be considered.

**Input File Usage:** Use the following options to allocate space for element solution-dependent variables in shell elements:

**SHELL SECTION*, ELSET=*shell_set_name*
**ELEMENT SOLUTION-DEPENDENT VARIABLES*

Use the following options to allocate space for element solution-dependent variables in solid elements:

**SOLID SECTION*, ELSET=*solid_set_name*
**ELEMENT SOLUTION-DEPENDENT VARIABLES*

**Abaqus/CAE Usage:** Defining element solution-dependent variables is not supported in Abaqus/CAE.

## Defining Initial Values

You can define the initial values of element solution-dependent variables in a tabular format for elements and/or element sets. See *Initial Conditions* for additional details.

**Input File Usage:** **INITIAL CONDITIONS*, TYPE=ESDV

**Abaqus/CAE Usage:** Defining initial values of element solution-dependent variables is not supported in Abaqus/CAE.

## Element Deletion Controlled by Element Solution-Dependent Variables

Element deletion in a mesh can be controlled during a stress/displacement analysis by element solution-dependent variables in Abaqus/Standard. Deleted elements have no ability to carry stresses and, therefore, have no contribution to the stiffness of the model. You specify the element solution-dependent variable number controlling the element deletion flag. The deletion variable can be set to a value of one or zero. A value of one indicates that the element is active, while a value of zero indicates that Abaqus should delete the element from the model by setting the stresses to zero. The deletion variable is initialized as a value of one at the beginning of the analysis. This initial value can be overwritten by user-specified initial condition values. Once an element is flagged as deleted, it cannot be reactivated. The status of an element can be determined by requesting output of the variable STATUS. This variable is equal to one if the element is active and is equal to zero if the element is deleted.

**Input File Usage:** **ELEMENT SOLUTION-DEPENDENT VARIABLES*, DELETE=*variable number*

**Abaqus/CAE Usage:** Defining element deletion controlled by element solution-dependent variables is not supported in Abaqus/CAE.

### Output

Element solution-dependent variables can be written to the output database (`.odb`) file; the output identifiers ESDV and ESDV*n* are available as whole element variables (see *Abaqus/Standard Output Variable Identifiers*).

## Alphanumeric Data

Alphanumeric data, such as labels (names) of surfaces or materials, are always passed into user subroutines in the upper case. As a result, direct comparison of these labels with corresponding lower-case characters will fail. Upper case must be used for all such comparisons. An example of such a comparison can be found in *UMAT*. It illustrates the code setup inside user subroutine *UMAT* when more than one user-defined material model needs to be defined. The variable `CMNAME` is compared against `MAT1` and `MAT2` (even in situations where the material names might have been defined as `mat1` and `mat2`, respectively.)

## Precision in Abaqus/Explicit

Abaqus/Explicit is installed with both single precision and double precision executables. To use the double precision executable, you must specify double precision when you run the analysis (see *Abaqus/Standard and Abaqus/Explicit Execution*). All variables in the user subroutines that start with the letters a to h and o to z will automatically be defined in the precision of the executable that you run. The precision of the executable is defined in the `vaba_param.inc` file, and it is not necessary to define the precision of the variables explicitly.

## Vectorization in Abaqus/Explicit

Abaqus/Explicit user subroutines are written with a vector interface, which means that blocks of data are passed to the user subroutines. For example, the vectorized user material routines (*VFABRIC* and *VUMAT*) are passed stresses, strains, state variables, etc. for `nblock` material points. One of the parameters defined by `vaba_param.inc` is `maxblk`, the maximum block size. If the user subroutine requires the dimensioning of temporary arrays, they can be dimensioned by `maxblk`.

## Parallelization

User subroutines can be used when running jobs in parallel. In thread-parallel mode access to common blocks, common files, and other shared resources need to be guarded against race conditions. Special utility routines are provided for that purpose. For details, see *Ensuring Thread Safety*. An environment variable `ABA_PARALLEL_DEBUG` can be set to increase verbosity and help troubleshoot problems should they arise in parallel runs.

## User Subroutine Calls

Most of the user subroutines available in Abaqus are called at least once for each increment during an analysis step. However, as discussed below, many subroutines are called more or less often.

### Subroutines That Define Material, Element, or Interface Behavior

Most user subroutines that are used to define material, element, or interface behavior are called twice per material point, element, or secondary surface node in the first iteration of every increment such that the model's initial stiffness matrix can be formulated appropriately for the step procedure chosen. The subroutines are called only once per material point, element, or secondary surface node in each succeeding iteration within the increment.

By default, in transient implicit dynamic analyses (*Implicit Dynamic Analysis Using Direct Integration*) Abaqus/Standard calculates accelerations at the beginning of each dynamic step. Abaqus/Standard must call user subroutines that are used to define material, element, or interface behavior two extra times for each material point, element, or secondary surface node prior to the zero increment. The extra calls to the user subroutines are not made if the initial acceleration calculations are suppressed. If the half-increment residual tolerances are being checked in a transient implicit dynamic step, Abaqus/Standard must call these user subroutines (except `UVARM`) one extra time for each material point, element, or secondary surface node at the end of each increment. If the calculation of the half-increment residual is suppressed, the extra call to the user subroutines is not made.

User subroutines `UHARD`, `UHYPEL`, `UHYPER`, and `UMULLINS`, when used in plane stress analyses, are called more often.

### Subroutines That Define Initial Conditions or Orientations

User subroutines that are used to define initial conditions or orientations are called before the first iteration of the first step's initial increment within an analysis.

### Subroutines That Define Predefined Fields

User subroutines that are used to define predefined fields are called prior to the first iteration of the relevant step's first increment for all iterations of all increments whenever the current field variable is needed.

### Verification of Subroutine Calls

If there is any doubt as to how often a user subroutine is called, this information can be obtained upon testing the subroutine on a small example, as suggested earlier. The current step and increment numbers are commonly passed into these subroutines, and they can be printed out as debug output (also discussed earlier). The iteration number for which the subroutine is called might not be passed into the user subroutine; however, if printed output is sent from the subroutine to the message (`.msg`) file (*About Output*), the location of the output within this file will give the iteration number, provided that the output to the message file is written at every increment.

## Utility Routines

A variety of utility routines are available to assist in the coding of user subroutines. You include the utility routine inside a user subroutine. When called, the utility routine will perform a predefined function or action whose output or results can be integrated into the user subroutine. Some utility routines are only applicable to particular user subroutines. Each utility routine is discussed in detail in *General Utility Routines*. For information about

the utility routines that are available for use with the toolpath-mesh intersection module, see *Toolpath-Mesh Intersection Utility Routines*.

## Variables Provided for Use in Utility Routines

The following utility routines require the use of Abaqus-provided variables passed into the user subroutines from which they are called:

- GETNODETOELEMCONN
- GETVRM
- GETVRMAVGATNODE
- GETVRN
- IGETSENSORID
- IVGETSENSORID
- MATERIAL_LIB_MECH
- MATERIAL_LIB_HT
- getElMPtVar

These variables will be defined properly when passed into your user subroutine; you cannot modify the variables or create alternative variables for use in the utility routines.

For example, the GETVRM utility routine requires the variable JMAC, which is passed from Abaqus/Standard into user subroutine *UVARM* and other user subroutines for which GETVRM is a supported utility. The variable JMAC represents an Abaqus data structure that requires no further manipulation on your part. If you use the GETVRM utility routine from within user subroutine *UVARM*, you will pass the JMAC variable from *UVARM* into GETVRM.

# About Table Collections, Parameter Tables, and Property Tables

**Products:** Abaqus/Standard Abaqus/Explicit

**References:**

- *Special-Purpose Techniques for Additive Manufacturing*
- **TABLE COLLECTION*
- **PARAMETER TABLE*
- **PROPERTY TABLE*
- **PARAMETER TABLE TYPE*
- **PROPERTY TABLE TYPE*
- *Accessing Abaqus Table Collections*

## Overview

Table collections:

- are used to facilitate access to user-defined data from user subroutines;
- can contain parameter tables, which define constants of type integer, float, or string;
- can contain property tables, which define properties as a function of independent variables, temperature, or predefined field variables; and
- work in combination with a library of utility routines (including table lookup operations) that are accessible within user subroutines to query and evaluate the tables.

## Introduction

The table collection functionality in Abaqus is used to provide access to user-defined data (in the form of parameter and property tables) from user subroutines. The functionality allows you to pass data to user subroutines and provides a library of utility routines that can be invoked (within the user subroutine) to query or evaluate the data, including table lookup evaluations. For example, you can define material properties and access them in a material-related user subroutine. This is particularly useful when dealing with user subroutines that do not provide a mechanism to pass properties, because it avoids using hard-coded values inside the user code. You can also use table collections to define analysis parameters relevant to a particular application, such as the examples described for additive manufacturing process simulation (see *Special-Purpose Techniques for Additive Manufacturing*).

## Table Collection Definition

Table collections are containers that encapsulate parameter and/or property tables. Any number of table collections can be defined in an analysis. Each table collection definition can contain any number of parameter or property tables.

You must assign a name to each table collection definition. This name allows the table collection to be referenced from user subroutines where the parameter or property tables are accessed.

A material definition can be considered a special case of table collection (see *Defining Tables within a Material Definition*).

**Input File Usage:** **TABLE COLLECTION* (or **MATERIAL*), NAME=*name*

Each table collection definition is specified in a data block, which is initiated by a **TABLE COLLECTION* option. The table collection definition continues with the definition of the parameter or property tables until an option that does not define a table (such as another **TABLE COLLECTION* option) is introduced, at which point the table collection definition is assumed to be complete. The order of the parameter or property tables within a table collection is not important.

## Parameter Table Definition

To define a parameter table, you must first declare the type of the parameter table, which determines the number and the type of parameters in the table. Optionally, this type declaration can also include a default value for each parameter.

The type declaration has a global scope and must be unique for the model. You can define multiple parameter tables of the same type in the model. If multiple tables of the same type are defined within the same table collection (or material definition), a label must be assigned to these parameter tables.

**Input File Usage:** Use the following options to define a parameter table:

**PARAMETER TABLE TYPE*, NAME=*type_name*, PARAMETERS=*n*
**TABLE COLLECTION* (or **MATERIAL*), NAME=*name*
**PARAMETER TABLE*, TYPE=*type_name*, LABEL=*label*

For example, a parameter table with three parameters of type integer, float, and string can be defined as follows:

**PARAMETER TABLE TYPE*, NAME=*type_name*, PARAMETERS=3
INTEGER
FLOAT
STRING
**TABLE COLLECTION*, NAME=*table_name*
**PARAMETER TABLE*, TYPE=*type_name*, LABEL=*label_1*
*integer value, float value, string value*
**PARAMETER TABLE*, TYPE=*type_name*, LABEL=*label_2*
*integer value, float value, string value*

## Property Table Definition

Property tables are used to define real-valued properties that can be a function of one or more independent variables, temperature, or predefined field variables. The number of properties and independent variables in the table are defined with a type declaration. This type declaration has a global scope and must be unique for the model. You can define multiple property tables of the same type in the model. The dependence of the properties on temperature and or predefined field variables is optional and is not part of the type declaration; instead, the dependence is declared when the property table is defined. If multiple tables of the same type are defined within the same table collection (or material definition), a label must be assigned to these property tables.

The definition of the properties as a function of independent variables, temperature, or predefined field variables, must follow the same convention used for the definition of material data (see *Material Data Definition*). A table that depends on temperature and independent variable is illustrated in the example of an elastic-plastic material (see *Specifying Material Data as Functions of Temperature and Independent Field Variables*).

**Input File Usage:** Use the following options to define a property table:

**PROPERTY TABLE TYPE*, NAME=*type_name*, PROPERTIES=*n*, INDEPENDENT VARIABLES=*n*
**TABLE COLLECTION* (or **MATERIAL*), NAME=*name*
**PROPERTY TABLE*, TYPE=*type_name*, LABEL=*label*, TEMPERATURE, DEPENDENCIES=*n*

## Defining Tables within a Material Definition

As mentioned above, a material definition block is a special case of a table collection. Therefore, you can define parameter and property tables within a material definition. Abaqus creates an internal table collection associated with this material name, and the tables are available automatically in all user subroutines invoked with this material.

**Input File Usage:** Use the following options to define tables within a material definition:

**MATERIAL*, NAME=*material_name*
**PROPERTY TABLE*, TYPE=*property_table_type*
**PARAMETER TABLE*, TYPE=*parameter_table_type*

## Interpolation of Property Table Data

When the property table data are a function of only one variable, the data must be given in order of increasing values of the independent variable, temperature, or predefined field variable. Abaqus then interpolates linearly for values between the given variables. If no linear extrapolation is requested, the properties are assumed to be constant outside the range of variables given. You can specify linear extrapolation to extrapolate properties outside the specified range of the independent variables assuming that the slope given by the end points of the curve remains constant. If the property table data depend on a variable in a strongly nonlinear manner, you must specify enough data points so that a linear interpolation captures the nonlinear behavior accurately.

When properties depend on several variables, the variation of the properties with respect to the first variable must be given at fixed values of the other variables, in ascending values of the second variable, then of the third variable, and so on. The data must always be ordered so that the independent variables are given increasing values. This process ensures that the value of the material property is completely and uniquely defined at any values of the variables upon which the property depends. See *Input Syntax Rules* for further explanation and an example.

**Input File Usage:** Use the following option to use constant extrapolation outside the specified range of the independent variables (default):

**PROPERTY TABLE*, EXTRAPOLATION=CONSTANT

Use the following option to request linear extrapolation outside the specified range of the independent variables:

**PROPERTY TABLE*, EXTRAPOLATION=LINEAR

## Regularizing Property Table Data in Abaqus/Explicit

Interpolating properties as functions of independent variables requires table lookups during the analysis, which occur frequently in Abaqus/Explicit and are most economical if the interpolation is from regular intervals of the independent variables. By default, Abaqus/Explicit regularizes property table data automatically. The default regularization tolerance is 0.03. Optionally, you can turn off regularization of the data, but this results in some performance degradation. For more information on the regularization process, see *Regularizing User-Defined Data in Abaqus/Explicit*.

Sometimes it desirable to preserve certain points of the input data during regularization. For example, if the force is zero at zero displacement in the displacement-force relation, you can keep this data point unchanged during regularization.

**Input File Usage:** Use the following option to modify the regularization tolerance:

**PROPERTY TABLE*, RTOL=*tolerance*, REGULARIZE=ON (default)

Use the following option to use property table data without regularization:

**PROPERTY TABLE*, REGULARIZE=OFF

Use the following option to preserve the property at the zero value of the first variable during regularization:

**PROPERTY TABLE*, REGULARIZE=ORIGIN

# About Event Series

**Products:** Abaqus/Standard

**References:**

- *Special-Purpose Techniques for Additive Manufacturing*
- **EVENT SERIES*
- **EVENT SERIES TYPE*
- *Accessing Abaqus Event Series Data*

## Overview

Event series:

- are used to define fields that vary with both time and space; and
- work in combination with a library of utility routines that are accessible within some user subroutines to query and access events.

## Introduction

The event series functionality in Abaqus/Standard allows you to define fields that vary with time and space (event series) and to provide access to them from user subroutines `UMDFLUX` and `UEPACTIVATIONVOL`. During the analysis, you can access only a slice of event series data, where a slice is a series of events from a current time increment. The utility routines that can be invoked to query or access the data are described in *Accessing Abaqus Event Series Data*. Examples of using event series functionality to specify raw material deposition and moving heat source for additive manufacturing process simulation are presented in *Special-Purpose Techniques for Additive Manufacturing*.

## Event Series Definition

To define an event series, you must first declare the type of the event series, which determines the number of fields to include in the definition. The time and space variation of these fields are specified in tabular form as a list of discrete events corresponding to increasing values of time. In general, Abaqus does not make any assumptions about how the fields vary between discrete events (no interpolation of the data is performed); the interpretation of the discrete events is determined by the user.

The type and event series definitions have a global scope and must be unique for the model. By default, the time of events is assumed to be the step time, but you can define events using the total time instead. See *Conventions* for a discussion of these time measures.

**Input File Usage:** Use the following option to define the type of event series:

**EVENT SERIES TYPE*, NAME=*type_name*, FIELDS=*n*

Use the following option to define an event series using the step time (default):

**EVENT SERIES*, NAME=*event_series_name*, TYPE=*type_name*, TIME=STEP TIME

Use the following option to define an event series using the total time:

**EVENT SERIES*, NAME=*event_series_name*, TYPE=*type_name*, TIME=TOTAL TIME

For example, an event series with one field and two events can be defined as follows:

**EVENT SERIES TYPE*, NAME=*type_name*, FIELDS=1
**EVENT SERIES*, NAME=*event_series_name*, TYPE=*type_name*
*t_1*, *x_1*, *y_1*, *z_1*, *field_value_1*
*t_2*, *x_2*, *y_2*, *z_2*, *field_value_2*

## Reading Event Series Data from a SIM Database File

To read event series data from a SIM database file, you must specify the name of the file. The SIM file must contain the complete definition of the event series, including the type declaration. You do not need to specify the file extension. By default, Abaqus uses the name of the event series from the SIM database, but you can modify this name.

**Input File Usage:** Use the following option to read event series data from a SIM database file:

**EVENT SERIES*, NAME=*sim_event_series_name*, FILE=*file_name*

Use the following option to change the name of the event series:

**EVENT SERIES*, NAME=*new_event_series_name*, FILE=*file_name*, SOURCE NAME=*sim_event_series_name*

## Reading Event Series Data from an Alternate Input File

The event series data can be contained in a separate input file (see *Input Syntax Rules*).

**Input File Usage:** Use the following option to read event series data from an alternate file:

**EVENT SERIES*, INPUT=*file_name*

## Positioning of Event Series Data

You can modify the position coordinates in the event series by specifying a translation and/or rotation for events relative to the origin of the global coordinate system.

**Input File Usage:** Use the following option to position the event series:

**EVENT SERIES*, TRANSFORM
*positioning data*

# About Neural Networks

**Products:** Abaqus/Standard

**References:**

- **NEURAL NETWORK*
- **NEURAL NETWORK DATA*
- *Evaluating Abaqus Neural Networks*
- *Activation functions used by feed-forward neural networks*
- *Plate with elliptical hole under thermal loading*

## Overview

Neural networks:

- are used to define arbitrary mappings (mathematical functions) between a set of inputs and a corresponding set of outputs;
- are standalone model data;
- must be defined in the input file and assigned a unique ID;
- can be of arbitrary size and configuration;
- can be evaluated from any user subroutine using a set of dedicated utility routines;
- are assumed to be “trained”; and
- can be used for a variety of applications, including surrogate constitutive modeling and complex boundary and loading conditions.

## Introduction

You can define feed-forward, dense (fully connected) neural networks in Abaqus/Standard and evaluate their output and derivatives using dedicated utility routines. You can evaluate a neural network from any Abaqus/Standard user subroutine. *Evaluating Abaqus Neural Networks* describes the evaluation utility routines. You can define multiple neural networks in a single input file. Each neural network can have an arbitrary configuration.

## Feed-Forward Neural Network Definition

To fully define a trained feed-forward neural network, you need to define the neural network configuration and the neural network data (weights and biases). To define the configuration of the network, you must specify the input and output size of the neural network, the number of layers in the neural network, and the size and activation function type for each layer. Some of the supported activation functions employ an activation parameter with a

default value. You can provide a nondefault value for this activation parameter. You need to specify whether or not each layer uses biases.

To define the neural network data, you must provide the values of the weights and the values of the biases (if used) for each layer. The neural network definition must explicity specify all layers of the neural network. The size of the last layer ("output" layer) must be identical to the neural network output size. Each neural network can have an arbitrary number of layers. Abaqus/Standard assumes that the weights and biases are dimensionless.

You must assign a unique integer ID to each neural network definition. This unique ID is used by the evaluation utility routines to access the neural network to be evaluated. Optionally, you can assign a (string) label to individual neural networks. However, you cannot use this label to access the neural network during the evaluation.

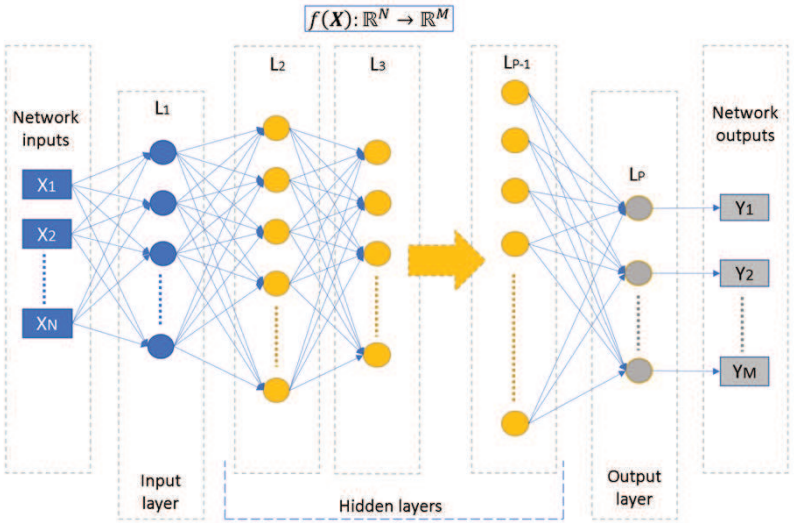



**Figure 1: Diagram of a feed-forward neural network with P layers, mapping an $n$-dimensional input vector to an $m$-dimensional output vector.**

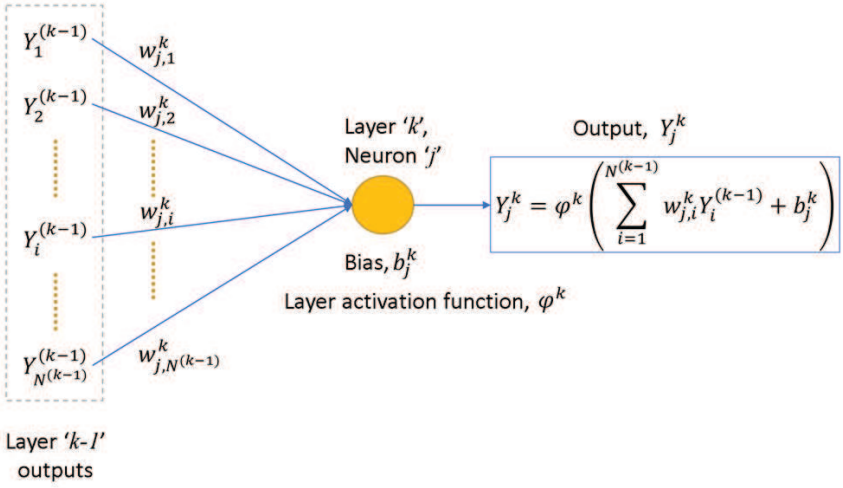



**Figure 2: Computational unit ("neuron") in a feed-forward neural network.**

## Activation Functions

The following activation functions are supported:

- CELU (Continuously differentiable exponential linear unit activation function) defined as

  $$\varphi(x) = \begin{cases} x, & x \geq 0 \\ \alpha\left(e^{x/\alpha} - 1\right), & x < 0 \end{cases}.$$

  You can specify a nondefault value for the activation parameter $\alpha$. The default value is $\alpha = 1$.

- ELU (Exponential linear unit activation function) defined as $\varphi(x) = \begin{cases} x, & x \geq 0 \\ \alpha(e^x - 1), & x < 0 \end{cases}$.

  You can specify a nondefault value for the activation parameter $\alpha$. The default value is $\alpha = 1$.
- FastTanh (Hyperbolic tangent approximation).
- Gauss (Gauss activation function) defined as $\varphi(x) = e^{-x^2}$.
- GELU (Gaussian error linear unit activation function) defined using an approximation of

  $\varphi(x) = 0.5x\left[1 + \operatorname{erf}\left(x/\sqrt{2}\right)\right]$.

- LeakyReLU (Leaky rectified linear unit activation function) defined as $\varphi(x) = \begin{cases} x, & x \geq 0 \\ \alpha x, & x < 0 \end{cases}$.

  You can specify a nondefault value for the activation parameter $\alpha$. The default value is $\alpha = 0.01$.
- Linear (Linear activation function) defined as $\varphi(x) = x$.

- ReLU (Rectified linear unit activation function) defined as $\varphi(x) = \begin{cases} x, & x > 0 \\ 0, & x \leq 0 \end{cases}$.
- Sigmoid (Logistic sigmoid activation function) defined as $\varphi\left(x\right) = (1 + e^{-x})^{-1}$.
- SiLU (Sigmoid linear unit activation function) defined as $\varphi\left(x\right) = x(1 + e^{-x})^{-1}$.
- Sin (Sinusoid activation function) defined as $\varphi(x) = \sin(x)$.
- SoftPlus (Softplus activation function) defined as $\varphi(x) = \ln(1 + e^x)$.
- SoftSign (Softsign activation function) defined as $\varphi\left(x\right) = x(1 + |x|)^{-1}$.
- Swish (Swish activation function) defined as $\varphi\left(x\right) = x\left(1 + e^{-\beta x}\right)^{-1}$.

  You can specify a nondefault value for the activation parameter $\beta$. The default value is $\beta = 1$.
- Tanh (Hyperbolic tangent activation function) defined as $\varphi(x) = \tanh(x)$.

All neurons in a layer use the same activation function. You can assign each layer a different activation function.

## Scaling

Abaqus/Standard assumes that the defined feed-forward neural networks are "trained." If any scaling of the data sets was performed prior to training of the neural network, you must explicitly account for the scaling effects (on the neural network inputs, outputs, and derivatives) before and after the evaluation of the neural network in the user subroutines.

## Usage

**Input File Usage:** Use the following options to define a feed-forward neural network.

**NEURAL NETWORK*,TYPE=FFD, ID=*neural_network_id*, NINPUT=*neural_network_input_size*, NOUTPUT=*neural_network_output_size*, LABEL=*neural_network_label*

A simple example of a feed-forward neural network using two layers (with two neurons in the first layer and one neuron in the second layer, both layers using biases) and approximating an XY curve:

**NEURAL NETWORK*, ID=*id*, TYPE=FFD, NINPUT=1, NOUTPUT=1, LABEL=*label*
*2*
*2*, YES, *Tanh*
*1*, YES, *Linear*
**NEURAL NETWORK DATA*
*1*
*weight_layer1_neuron1_for_NN_input1, weight_layer1_neuron2_for_NN_input1, bias_layer1_neuron1, bias_layer1_neuron2*
*2*
*weight_layer2_neuron1_for_layer1_output1, weight_layer2_neuron1_for_layer1_output2, bias_layer2_neuron1*

A more complex example of a feed-forward neural network using three layers and mapping an *n*-dimensional input quantity to an *m*-dimensional output quantity (weights and biases are assumed to be defined using eight values per data line):

**NEURAL NETWORK*, ID=*id*, TYPE=FFD, NINPUT=*N*, NOUTPUT=*M*, LABEL=*label*
*3*
*layer1_size*, *layer1_uses_biases_flag* (YES/NO), *layer1_activation_function_type*, *layer1_activation_param_value*
*layer2_size*, *layer2_uses_biases_flag* (YES/NO), *layer2_activation_function_type*, *layer2_activation_param_value*
*M*, *layer3_uses_biases_flag* (YES/NO), *layer3_activation_function_type*, *layer3_activation_param_value*
**NEURAL NETWORK DATA*
*1*
*weight_layer1_neuron1_for_NN_input1, weight_layer1_neuron1_for_NN_input2,...,weight_layer1_neuron1_for_NN_input_N,*

*weight_layer1_neuron2_for_NN_input1, weight_layer1_neuron2_for_NN_input2, … , weight_layer1_neuron2_for_NN_input_N, ...,bias_layer1_neuron1, bias_layer1_neuron2,…*
*2*
*multiple_weights_layer2_neuron1_for_layer1_outputs, multiple_weights_layer2_neuron2_for_layer1_outputs,…,multiple_biases_for_all_layer2_neurons*
*3*
*multiple_weights_layer3_neuron1_for_layer2_outputs, multiple_weights_layer3_neuron2_for_layer2_outputs,…,multiple_biases_for_all_layer3_neurons*

# Abaqus/Standard User Subroutines

Abaqus/Standard user subroutines increase the functionality of several Abaqus capabilities for which the usual data input methods alone may be too restrictive. They provide an extremely powerful and flexible tool for analysis.

**In this section:**


- *CREEP*
- *DECURRENT*
- *DFLOW*
- *DFLUX*
- *DISP*
- *DLOAD*
- *FILM*
- *FLOW*
- *FRIC*
- *FRIC_COEF*
- *GAPCON*
- *GAPELECTR*
- *HARDINI*
- *HETVAL*
- *MPC*
- *ORIENT*
- *RSURFU*
- *SDVINI*
- *SIGINI*
- *UAMP*
- *UANISOHYPER_INV*
- *UANISOHYPER_STRAIN*
- *UCOHESIVEOFFSET*
- *UCORR*
- *UCOUPLEDCONSTRESP*
- *UCREEPNETWORK*
- *UCURE*
- *UDAMAGEMF*
- *UDECURRENT*
- *UDEMPOTENTIAL*
- *UDMGINI*
- *UDSECURRENT*
- *UECHEMECDBV*
- *UECHEMECDDISS*
- *UECHEMECDLPL*
- *UECHEMECDSEI*

- *UECHEMECDSEIREFORM*
- *UEIGENSTRAIN*
- *UEL*
- *UELEMDRESP*
- *UELMAT*
- *UEPACTIVATIONFACET*
- *UEPACTIVATIONSETUP*
- *UEPACTIVATIONVOL*
- *UEREF*
- *UEXPAN*
- *UEXTERNALDB*
- *UFIELD*
- *UFLUID*
- *UFLUIDCONNECTORLOSS*
- *UFLUIDCONNECTORVALVE*
- *UFLUIDLEAKOFF*
- *UFLUIDPIPEFRICTION*
- *UGENS*
- *UHARD*
- *UHYPEL*
- *UHYPER*
- *UHYPER_STRETCH*
- *UINTER*
- *UINTFLUXGEN*
- *UMASFL*
- *UMAT*
- *UMATHT*
- *UMDFLUX*
- *UMDFLUXSETUP*
- *UMESHMOTION*
- *UMIXMODEFATIGUE*
- *UMOTION*
- *UMULLINS*
- *UPOREP*
- *UPRESS*
- *UPSD*
- *URDFIL*
- *USDFLD*
- *USETTLING*
- *USUPERELASHARDMOD*
- *UTEMP*
- *UTRACLOAD*
- *UTRS*
- *UTRSNETWORK*
- *UVAREL*
- *UVARM*

- *UWAVE*
- *UXFEMCRACK*
- *UXFEMNONLOCALWEIGHT*
- *VOIDRI*

# CREEP: User subroutine to define time-dependent, viscoplastic behavior (creep and swelling).

**Products:** Abaqus/Standard

**References:**

- *Rate-Dependent Plasticity: Creep and Swelling*
- *Extended Drucker-Prager Models*
- *Modified Drucker-Prager/Cap Model*
- *Defining the Gasket Behavior Directly Using a Gasket Behavior Model*
- **CAP CREEP*
- **CREEP*
- **DRUCKER PRAGER CREEP*
- **SWELLING*
- *Verification of creep integration*

## Overview

User subroutine CREEP will be called at all integration points of elements for which the material definition contains user-subroutine-defined metal creep, time-dependent volumetric swelling, Drucker-Prager creep, or cap creep behavior, during procedures that allow viscoplastic response of the above type to occur (such as the quasi-static procedure). This subroutine will also be called at all integration points of gasket elements for which the behavior definition contains user-subroutine-defined creep.

If user subroutine CREEP is used to define a material behavior, the subroutine:

- is intended to provide the "uniaxial" creep laws that are to be included in a general time-dependent, viscoplastic material formulation;
- can be used in the coupled-temperature displacement (*Fully Coupled Thermal-Stress Analysis*), coupled thermal-electrical-structural (*Fully Coupled Thermal-Electrical-Structural Analysis*), soils (*Coupled Pore Fluid Diffusion and Stress Analysis*), and quasi-static (*Quasi-Static Analysis*) procedures;
- allows for the definition of creep laws for which the meaning and internal use depend on the material model with which they are being used;
- allows creep and swelling to be combined with rate-independent plastic behavior in a coupled manner, or they may simply be the only inelastic behaviors of the material, in which case Mises behavior is assumed;
- can use and update solution-dependent state variables; and
- can be used in conjunction with user subroutine USDFLD to redefine any field variables before they are passed in.

If user subroutine CREEP is used to define rate-dependent behavior in the thickness direction for a gasket, the subroutine:

- is intended to provide the creep laws that are used to prescribe the thickness-direction behavior for a gasket;
- can be used only in a quasi-static (*Quasi-Static Analysis*) procedure;
- is used in a coupled form with the elastic-plastic model used to define the rate-independent part of the thickness-direction behavior of the gasket; and
- can use and update solution-dependent variables.

## Metals

For metals whose material behavior includes metal creep and/or time-dependent volumetric swelling, the routine allows any "creep" and "swelling" laws (viscoplastic behavior) of the following general form to be defined:

$$\dot{\bar{\varepsilon}}^{cr} = g^{cr}\left(p, \tilde{q}, \bar{\varepsilon}^{sw}, \bar{\varepsilon}^{cr}, \text{time}, \ldots\right),$$
$$\dot{\bar{\varepsilon}}^{sw} = g^{sw}\left(p, \tilde{q}, \bar{\varepsilon}^{sw}, \bar{\varepsilon}^{cr}, \text{time}, \ldots\right),$$

where

$\bar{\varepsilon}^{cr}$

is the uniaxial equivalent "creep" strain, conjugate to $\tilde{q}$, the Mises or Hill equivalent stress;

$\bar{\varepsilon}^{sw}$

is the volumetric swelling strain;

$p$

is the equivalent pressure stress, $p = -\frac{1}{3}\left(\sigma_{11} + \sigma_{22} + \sigma_{33}\right)$; and

$\tilde{q}$

is the equivalent deviatoric stress (Mises' or, if anisotropic creep behavior is defined, Hill's definition).

The user subroutine must define the increments of inelastic strain, $\Delta\bar{\varepsilon}^{cr}$ and $\Delta\bar{\varepsilon}^{sw}$, as functions of $p$ and $\tilde{q}$ and any other variables used in the definitions of $g^{cr}$ and $g^{sw}$ (such as solution-dependent state variables introduced by you) and of the time increment, $\Delta t$. If any solution-dependent state variables are included in the definitions of $g^{cr}$ and $g^{sw}$, they must also be integrated forward in time in this routine.

Abaqus computes the incremental creep strain (or the incremental viscoplastic strain) components as

$$\Delta\boldsymbol{\varepsilon}^{cr} = \frac{1}{3}\Delta\bar{\varepsilon}^{sw}\mathbf{R} + \Delta\bar{\varepsilon}^{cr}\mathbf{n},$$

where $\mathbf{n}$ is the gradient of the deviatoric stress potential, defined as

$$\mathbf{n} = \frac{\partial\tilde{q}}{\partial\boldsymbol{\sigma}},$$

and $\mathbf{R}$ is a matrix with the anisotropic swelling ratios in the diagonal if anisotropic swelling is defined; otherwise, $\mathbf{R} = \mathbf{I}$.

## Drucker-Prager Materials

For materials that yield according to the extended Drucker-Prager plasticity models using Drucker-Prager creep, the routine allows any "creep" laws (viscoplastic behavior) of the following general form to be defined:

$$\dot{\bar{\varepsilon}}^{cr} = g^{cr}\left(\bar{\sigma}^{cr}, \bar{\varepsilon}^{cr}, \text{time}, \ldots\right),$$

where

$\overline{\sigma}^{cr}$

is the equivalent creep stress defined as

$\frac{q-p\tan\beta}{1-\frac{1}{3}\tan\beta}$

if creep is defined in terms of uniaxial compression,

$\frac{q-p\tan\beta}{1+\frac{1}{3}\tan\beta}$

if creep is defined in terms of uniaxial tension, and

$q-p\tan\beta$

if creep is defined in terms of pure shear,

where $q$ is the equivalent deviatoric Mises' stress, $p$ is the pressure stress, and $\beta$ is the friction angle, and

$\bar{\varepsilon}^{cr}$

is the uniaxial equivalent “creep” strain, conjugate to $\overline{\sigma}^{cr}$ such that $\overline{\sigma}^{cr}\Delta\bar{\varepsilon}^{cr}=\sigma_{ij}\Delta\varepsilon_{ij}^{cr}$.

The user subroutine must define the increment of inelastic strain, $\Delta\bar{\varepsilon}^{cr}$, as a function of $\overline{\sigma}^{cr}$ and any other variables used in the definitions of $g^{cr}$ (such as solution-dependent state variables introduced by you) and of the time increment, $\Delta t$. If any solution-dependent state variables are included in the definitions of $g^{cr}$, they must also be integrated forward in time in this routine.

Abaqus computes the incremental creep strain (or the incremental viscoplastic strain) components as

$$\Delta\boldsymbol{\varepsilon}^{cr}=\frac{\Delta\bar{\varepsilon}^{cr}}{f^{cr}}\left(\frac{q}{\sqrt{(\epsilon\overline{\sigma}|_0\tan\psi)^2+q^2}}\mathbf{n}+\frac{1}{3}\tan\psi\mathbf{I}\right),$$

where $\mathbf{n}=\partial\tilde{q}/\partial\boldsymbol{\sigma}$. The variable $f^{cr}$ is determined in such a way that

$$f^{cr}=\frac{1}{\overline{\sigma}^{cr}}\boldsymbol{\sigma}:\frac{\partial G^{cr}}{\partial\boldsymbol{\sigma}},$$

and

$$G^{cr}=\sqrt{(\epsilon\overline{\sigma}|_0\tan\psi)^2+q^2}-p\tan\psi$$

is the hyperbolic creep potential, where $\psi(\theta,f^{\alpha})$ is the dilation angle measured in the $p$–$q$ plane at high confining pressure, $\overline{\sigma}|_0=\overline{\sigma}|_{\bar{\varepsilon}^{pl}=0,\dot{\bar{\varepsilon}}^{pl}=0}$ is the initial yield stress, and $\epsilon$ is the eccentricity. See *Extended Drucker-Prager Models* for a discussion of $\psi$, $\epsilon$, and $\overline{\sigma}|_0$.

## Capped Drucker-Prager Materials

For materials that yield according to the modified Drucker-Prager/Cap plasticity model using cap creep, the routine allows any "cohesion creep" and "consolidation creep" laws (viscoplastic behavior) of the following general form to be defined:

$$\dot{\bar{\varepsilon}}_s^{cr} = g_s^{cr}\left(\bar{\sigma}^{cr}, \bar{\varepsilon}_s^{cr}, \text{time}, \ldots\right),$$
$$\dot{\bar{\varepsilon}}_c^{cr} = g_c^{cr}\left(\bar{p}^{cr}, \bar{\varepsilon}_c^{cr}, \text{time}, \ldots\right),$$

where

$\bar{\sigma}^{cr}$

is the equivalent creep stress defined from uniaxial compression test data as

$$\bar{\sigma}^{cr} = \frac{q - p\tan\beta}{\left(1 - \frac{1}{3}\tan\beta\right)},$$

where $q$ is the equivalent deviatoric Mises' stress, $p$ is the pressure stress, and $\beta$ is the friction angle;

$\bar{\varepsilon}_s^{cr}$

is the equivalent cohesion creep uniaxial strain, conjugate to $\bar{\sigma}^{cr}$ such that $\bar{\sigma}^{cr}\Delta\bar{\varepsilon}_s^{cr} = \boldsymbol{\sigma} : \Delta\boldsymbol{\varepsilon}_s^{cr}$, where $\Delta\boldsymbol{\varepsilon}_s^{cr}$ is defined below;

$\bar{p}^{cr} = p - p_a$

is the effective creep pressure ($p = -\frac{1}{3}\left(\sigma_{11} + \sigma_{22} + \sigma_{33}\right)$ and $p_a$ is the cap hardening parameter); and

$\bar{\varepsilon}_c^{cr}$

is the volumetric consolidation creep strain.

The user subroutine must define the increments of inelastic strain, $\Delta\bar{\varepsilon}_s^{cr}$ and/or $\Delta\bar{\varepsilon}_c^{cr}$, as functions of $\bar{\sigma}^{cr}$ and/or $\bar{p}^{cr}$ and any other variables used in the definitions of $g_s^{cr}$ and $g_c^{cr}$ (such as solution-dependent state variables introduced by you) and of the time increment, $\Delta t$. If any solution-dependent state variables are included in the definitions of $g_s^{cr}$ and $g_c^{cr}$, they must also be integrated forward in time in this routine.

### Calculation of Incremental Creep Strains for the Cohesion Mechanism

Abaqus computes the incremental creep strain (or the incremental viscoplastic strain) components of the cohesion mechanism as

$$\Delta\boldsymbol{\varepsilon}_s^{cr} = \frac{\Delta\bar{\varepsilon}_s^{cr}}{f^{cr}}\left(\frac{q}{\sqrt{\left(0.1\frac{d}{\left(1 - \frac{1}{3}\tan\beta\right)}\tan\beta\right)^2 + q^2}}\mathbf{n} + \frac{1}{3}\tan\beta\mathbf{I}\right),$$

where $\mathbf{n} = \partial \tilde{q} / \partial \boldsymbol{\sigma}$, $d$ is the material cohesion, the variable $f^{cr}$ is determined in such a way that

$$f^{cr} = \frac{1}{\bar{\sigma}^{cr}} \boldsymbol{\sigma} : \frac{\partial G_s^{cr}}{\partial \boldsymbol{\sigma}},$$

and $G_s^{cr}$ is the cohesion creep potential

$$G_s^{cr} = \sqrt{\left(0.1 \frac{d}{\left(1 - \frac{1}{3}\tan\beta\right)} \tan\beta\right)^2 + q^2} - p\tan\beta.$$

### Calculation of Incremental Creep Strains for the Consolidation Mechanism

Abaqus computes the incremental creep strain (or the incremental viscoplastic strain) components of the consolidation mechanism as

$$\Delta\boldsymbol{\varepsilon}_c^{cr} = \frac{\Delta\bar{\varepsilon}_c^{cr}}{G_c^{cr}} \left(R^2 \; q \; \mathbf{n} - \frac{1}{3}(p - p_a) \; \mathbf{I}\right),$$

where $R$ controls the shape of the cap, and $G_c^{cr}$ is the consolidation creep potential

$$G_c^{cr} = \sqrt{(p - p_a)^2 + (Rq)^2}.$$

Cohesion material properties are determined with a uniaxial compression test in which $d\bar{\varepsilon}_s^{cr} = \|d\epsilon_{11}^{cr}\|$, and consolidation material properties are determined with a volumetric compression test in which $d\bar{\varepsilon}_c^{cr} = \|d\epsilon_{vol}^{cr}\|$. Most likely, $g_s^{cr}$ is a positive function of $\bar{\sigma}^{cr}$, and $g_c^{cr}$ is a positive function of $\bar{p}^{cr}$.

## Gaskets

For gaskets whose behavior includes creep, the routine allows any “creep” law of the following general form to be defined:

$$\dot{\varepsilon}^{cr} = g^{cr}\left(\sigma, \varepsilon^{cr}, \text{time}, \ldots\right),$$

where $\varepsilon^{cr}$ is the compressive creep strain, conjugate to $\sigma$, the compressive stress in the gasket.

The user subroutine must define the increments of inelastic creep strain, $\Delta\varepsilon^{cr}$, as functions of $\sigma$ and any other variables used in the definitions of $g^{cr}$ (such as solution-dependent state variables introduced by you) and of the time increment, $\Delta t$. If any solution-dependent state variables are included in the definitions of $g^{cr}$, they must also be integrated forward in time in this routine. Abaqus will automatically multiply this creep strain by the proper thickness (see *Defining the Gasket Behavior Directly Using a Gasket Behavior Model*) to obtain a creep closure.

## Integration Schemes

Abaqus provides both explicit and implicit time integration of creep and swelling behavior defined in this routine. The choice of the time integration scheme depends on the procedure type, the procedure definition, and whether a geometric linear or nonlinear analysis is requested (see *Rate-Dependent Plasticity: Creep and Swelling*).

Implicit integration is generally more effective when the response period is long relative to typical relaxation times for the material. Simple high-temperature structural design applications usually do not need implicit integration, but more complicated problems (such as might arise in manufacturing processes), creep buckling applications, or nonstructural problems (such as geotechnical applications) often are integrated more efficiently by the implicit method provided in the program. If implicit integration is used with this subroutine, nonlinear

equations must be solved at each time step and the variations of $\Delta\bar{\varepsilon}^{cr}$, $\Delta\bar{\varepsilon}^{sw}$, $\Delta\bar{\varepsilon}_s^{cr}$, or $\Delta\bar{\varepsilon}_c^{cr}$ with respect to $\bar{\varepsilon}^{cr}$, $\bar{\varepsilon}^{sw}$, $\bar{\varepsilon}_s^{cr}$, $\bar{\varepsilon}_c^{cr}$, $p$, $\tilde{q}$, $\bar{p}^{cr}$, or $\bar{\sigma}^{cr}$ must be defined in the subroutine. To obtain good convergence during implicit integration, it is essential to define these quantities accurately.

At the start of a new increment the subroutine is called once for each integration point to calculate the estimated creep strain based on the state at the start of the increment. Subsequently, it is called twice for each iteration if explicit integration is used: once to calculate the creep strain increment at the start of the increment and once to calculate it at the end of the increment. This is needed to test the validity of the time increment with respect to the user-specified maximum allowable difference in the creep strain increment. The flag `LEND` indicates whether the routine is called at the start or the end of the increment. The subroutine must use the corresponding values of time, temperature, field variables, and solution-dependent state variables in the calculation of the creep strain increment.

For implicit integration Abaqus uses a local iteration procedure to solve the nonlinear constitutive equations, and the subroutine is called multiple times. The exact number of calls depends on the convergence rate of the local iteration procedure and, hence, will vary from point to point. During these iterations it is possible for the values of the state variables to be far from their final values when the equations are solved. Therefore, the coding in the subroutine must adequately protect against arithmetic failures (such as floating point overflows) even when variables are passed in with physically unreasonable values. As in explicit integration, the variable `LEND` indicates whether the routine is called at the start or the end of the increment.

## Constant Stress Assumption When Defining Creep and Swelling

When the creep and swelling behavior are defined by simple formulæ, it is often possible to calculate the increments of equivalent creep and swelling strain exactly if it is assumed that the stress is constant during the increment. This approach has the advantage that it provides very good accuracy within the constant stress assumption. It also avoids the problem that arises for some creep behavior definitions: that the creep strain rate becomes infinite at zero time (or strain). Otherwise, in such a case you must protect against causing arithmetic failures at the start of the solution.

## Defining Both Plasticity and Creep

If both plasticity and creep are defined for a material, Abaqus will calculate the creep strain before entering the plasticity routines. The stresses passed into the creep routine may, therefore, exceed the yield stress.

## Interpretation of Stress and Strain Variables

In finite-strain applications strain variables should be interpreted as logarithmic strains and stresses as “true” stress.

## User Subroutine Interface

```
      SUBROUTINE CREEP(DECRA,DESWA,STATEV,SERD,EC,ESW,P,QTILD,
     1 TEMP,DTEMP,PREDEF,DPRED,TIME,DTIME,CMNAME,LEXIMP,LEND,
     2 COORDS,NSTATV,NOEL,NPT,LAYER,KSPT,KSTEP,KINC)
C
      INCLUDE 'ABA_PARAM.INC'
C
      CHARACTER*80 CMNAME
C
      DIMENSION DECRA(5),DESWA(5),STATEV(*),PREDEF(*),DPRED(*),
     1 TIME(3),EC(2),ESW(2),COORDS(*)

      user coding to define DECRA,  DESWA
```

```
      RETURN
      END
```

## Variables to Be Defined

**In all cases**

`DECRA(1)` The definition depends on the usage:

- Metal creep: $\Delta\bar{\varepsilon}^{cr}$, equivalent (uniaxial) deviatoric creep strain increment.
- Drucker-Prager creep: $\Delta\bar{\varepsilon}^{cr}$, equivalent (uniaxial) creep strain increment.
- Capped Drucker-Prager creep: $\Delta\bar{\varepsilon}^{cr}_{s}$, equivalent (uniaxial) cohesion creep strain increment.
- Gasket creep: $\Delta\varepsilon^{cr}$, uniaxial compressive creep strain increment.

`DESWA(1)` The definition depends on the usage:

- Metal creep: $\Delta\bar{\varepsilon}^{sw}$, volumetric swelling strain increment.
- Capped Drucker-Prager creep: $\Delta\bar{\varepsilon}^{cr}_{c}$, equivalent (volumetric) consolidation creep strain increment.
- Drucker-Prager and gasket creep: = 0.

**For implicit creep integration (`LEXIMP=1`, see below)**

`DECRA(2)` The definition depends on the usage:

- Metal creep and Drucker-Prager creep: $\partial\Delta\bar{\varepsilon}^{cr}/\partial\bar{\varepsilon}^{cr}$.
- Capped Drucker-Prager creep: $\partial\Delta\bar{\varepsilon}^{cr}_{s}/\partial\bar{\varepsilon}^{cr}_{s}$.
- Gasket creep: $\partial\Delta\varepsilon^{cr}/\partial\varepsilon^{cr}$.

`DECRA(3)` The definition depends on the usage:

- Metal creep: $\partial\Delta\bar{\varepsilon}^{cr}/\partial\bar{\varepsilon}^{sw}$.
- Drucker-Prager creep, gasket creep, and capped Drucker-Prager creep: = 0.

`DECRA(4)` The definition depends on the usage:

- Metal creep: $\partial\Delta\bar{\varepsilon}^{cr}/\partial p$.
- Drucker-Prager creep, gasket creep, and capped Drucker-Prager creep: = 0.

`DECRA(5)` The definition depends on the usage:

- Metal creep: $\partial\Delta\bar{\varepsilon}^{cr}/\partial\tilde{q}$.
- Drucker-Prager creep: $\partial\Delta\bar{\varepsilon}^{cr}/\partial\bar{\sigma}^{cr}$.
- Capped Drucker-Prager creep: $\partial\Delta\bar{\varepsilon}^{cr}_{s}/\partial\bar{\sigma}^{cr}$.

**For implicit creep integration (`LEXIMP`=1, see below)**

- Gasket creep: $\partial \Delta \varepsilon^{cr} / \partial \sigma$.

`DESWA(2)` The definition depends on the usage:

- Metal creep: $\partial \Delta \bar{\varepsilon}^{sw} / \partial \bar{\varepsilon}^{cr}$.
- Drucker-Prager creep, gasket creep, and capped Drucker-Prager creep: = 0.

`DESWA(3)` The definition depends on the usage:

- Metal creep: $\partial \Delta \bar{\varepsilon}^{sw} / \partial \bar{\varepsilon}^{sw}$.
- Capped Drucker-Prager creep: $\partial \Delta \bar{\varepsilon}_c^{cr} / \partial \bar{\varepsilon}_c^{cr}$.
- Drucker-Prager and gasket creep: = 0.

`DESWA(4)` The definition depends on the usage:

- Metal creep: $\partial \Delta \bar{\varepsilon}^{sw} / \partial p$.
- Capped Drucker-Prager creep: $\partial \Delta \bar{\varepsilon}_c^{cr} / \partial \bar{p}^{cr}$.
- Drucker-Prager and gasket creep: = 0.

`DESWA(5)` The definition depends on the usage:

- Metal creep: $\partial \Delta \bar{\varepsilon}^{sw} / \partial \tilde{q}$.
- Drucker-Prager creep, gasket creep, and capped Drucker-Prager creep: = 0.

## Variables That Can Be Updated

`STATEV` An array containing the user-defined solution-dependent state variables at this point. This array will be passed in containing the values of these variables at the start of the increment unless they are updated in user subroutine *USDFLD* or *UEXPAN*, in which case the updated values are passed in. If any of the solution-dependent variables are being used in conjunction with the creep behavior and the routine was called at the end of the increment (`LEND`=1, see the definition of `LEND` below), they must be updated in this subroutine to their values at the end of the increment. Furthermore, if the solution-dependent state variables are defined as a function of the creep (swelling) strain increment, they must be updated based on the creep (swelling) strain increment computed as `EC(2)`−`EC(1)` (likewise `ESW(2)`−`ESW(1)`), where `EC(1)`, `EC(2)`, `ESW(1)`, and `ESW(2)` are defined below. You define the size of this array by allocating space for it (see *Allocating Space for Solution-Dependent State Variables* for more information).

`SERD` Magnitude of the strain energy rate density, $\dot{W}$ (required only in $C_t$-integral calculations). The strain energy rate density is defined as

$$\dot{W} = \int_0^{\dot{\bar{\varepsilon}}^{cr}} q \, d\dot{\bar{\varepsilon}}^{cr} + \int_0^{\dot{\bar{\varepsilon}}^{sw}} p \, d\dot{\bar{\varepsilon}}^{sw} = \int_0^{\dot{\varepsilon}} \boldsymbol{\sigma} : (d\dot{\boldsymbol{\varepsilon}}^{cr} + d\dot{\boldsymbol{\varepsilon}}^{sw}) .$$

Elastic rates are ignored in the calculation of $\dot{W}$. The contour integral will, therefore, be path independent only for steady-state creep conditions; that is, when the creep straining dominates throughout the specimen.

## Variables Passed in for Information

`EC(1)` The definition depends on the usage:

- Metal creep and Drucker-Prager creep: $\bar{\varepsilon}^{cr}$ at the start of the increment.
- Capped Drucker-Prager creep: $\bar{\varepsilon}_s^{cr}$ at the start of the increment.
- Gasket creep: $\varepsilon^{cr}$ at the start of the increment.

`EC(2)` The definition depends on the usage:

- Metal creep and Drucker-Prager creep: $\bar{\varepsilon}^{cr}$ at the end of the increment.
- Capped Drucker-Prager creep: $\bar{\varepsilon}_s^{cr}$ at the end of the increment.
- Gasket creep: $\varepsilon^{cr}$ at the end of the increment.

`ESW(1)` The definition depends on the usage:

- Metal creep: $\bar{\varepsilon}^{sw}$ at the start of the increment.
- Capped Drucker-Prager creep: $\bar{\varepsilon}_c^{cr}$ at the start of the increment.
- Drucker-Prager and gasket creep: = 0.

`ESW(2)` The definition depends on the usage:

- Metal creep: $\bar{\varepsilon}^{sw}$ at the end of the increment.
- Capped Drucker-Prager creep: $\bar{\varepsilon}_c^{cr}$ at the end of the increment.
- Drucker-Prager and gasket creep: = 0.

`P` The definition depends on the usage:

- Metal creep and Drucker-Prager creep: $p = -\frac{1}{3}(\sigma_{11} + \sigma_{22} + \sigma_{33})$, equivalent pressure stress (in soils analysis this is the equivalent effective pressure stress).
- Capped Drucker-Prager creep: $\bar{p}^{cr} = p - p_a$, effective creep pressure (in soils analysis $p$ is the effective pressure stress).
- Gasket creep: = 0.

If `LEND=0`, the value is $p$ or $\bar{p}^{cr}$ at the beginning of the increment. If `LEND=1`, the value is $p$ or $\bar{p}^{cr}$ at the end of the increment.

`QTILD` The definition depends on the usage:

- Metal creep: $\tilde{q}$, Mises or Hill equivalent stress (the Hill formula is used if anisotropic creep is defined; see *Anisotropic Creep*).
- Gasket creep: $\sigma$, the uniaxial compressive stress.
- Drucker-Prager creep: $\bar{\sigma}^{cr}$, equivalent creep stress (in soils analysis this is based on effective stresses).
- Capped Drucker-Prager creep: $\bar{\sigma}^{cr}$, equivalent creep stress (in soils analysis this is based on effective stresses).

| | |
|---|---|
| | If LEND=0, the value is $\tilde{q}$ or $\bar{\sigma}^{cr}$ at the beginning of the increment. If LEND=1, the value is $\tilde{q}$ or $\bar{\sigma}^{cr}$ at the end of the increment. |
| TEMP | Temperature at the end of the increment. |
| DTEMP | Increment of temperature during the time increment. |
| PREDEF | An array containing the values of all of the user-specified predefined variables at this point at the end of the increment (initial values at the beginning of the analysis and current values during the analysis). |
| DPRED | An array containing the increments of all of the predefined variables during the time increment. |
| TIME(1) | Value of step time at the end of the increment. |
| TIME(2) | Value of total time at the end of the increment. |
| TIME(3) | Value of creep time at the end of the increment. |
| DTIME | Time increment. |
| CMNAME | User-specified material name or gasket behavior name, left justified. Some internal creep models are given names starting with the "ABQ_" character string. To avoid conflict, you should not use "ABQ_" as the leading string for CMNAME. |
| LEXIMP | Explicit/implicit flag. |
| | If LEXIMP=0, explicit creep integration is being used and only DECRA(1) and DESWA(1) need be defined; DECRA(I) and DESWA(I), I=2,5, need not be defined. |
| | If LEXIMP=1, implicit creep integration is being used. The derivatives, DECRA(I) and DESWA(I), I=2,5, should be defined accurately to achieve rapid convergence of the solution. |
| LEND | Start/end of increment flag. |
| | If LEND=0, the routine is being called at the start of the increment. In this case DECRA(1) and DESWA(1) must be defined as the equivalent creep and swelling rates calculated at the beginning of the increment, multiplied by the time increment. |
| | If LEND=1, the routine is being called at the end of the increment. In this case DECRA(1) and DESWA(1) must be defined as the equivalent creep and swelling rates calculated at the end of the increment, multiplied by the time increment. If applicable, the solution-dependent state variables STATEV must be updated as well. |
| COORDS(3) | An array containing the current coordinates of this point. |
| NSTATV | Number of solution-dependent state variables associated with this material or gasket behavior type (specified when space is allocated for the array; see *Allocating Space for Solution-Dependent State Variables*). |
| NOEL | Element number. |
| NPT | Integration point number. |
| LAYER | Layer number (for composite shells and layered solids). |
| KSPT | Section point number within the current layer. |
| KSTEP | Step number. |
| KINC | Increment number. |

## Example: Hyperbolic Sine Creep Law

Suppose that we wish to model a metal using the creep behavior

$$\dot{\bar{\varepsilon}}^{cr} = A\left(\sinh\frac{q}{\sigma_0}\right)^n, \quad \dot{\bar{\varepsilon}}^{sw} = 0,$$

where $A$, $\sigma_0$, and $n$ are constants.

User subroutine CREEP can be coded as follows:

```
      SUBROUTINE CREEP(DECRA,DESWA,STATEV,SERD,EC,ESW,P,QTILD,
     1 TEMP,DTEMP,PREDEF,DPRED,TIME,DTIME,CMNAME,LEXIMP,LEND,
     2 COORDS,NSTATV,NOEL,NPT,LAYER,KSPT,KSTEP,KINC)
C
      INCLUDE 'ABA_PARAM.INC'
C
      CHARACTER*80 CMNAME
C
      DIMENSION DECRA(5),DESWA(5),STATEV(*),PREDEF(*),DPRED(*),
     1 TIME(3),COORDS(*),EC(2),ESW(2)
C
C DEFINE CONSTANTS
C
      A=
      SIG0=
      AN=
C
      T1=EXP(QTILD/SIG0)
      T2=EXP(-QTILD/SIG0)
      DECRA(1) = A*(.5*(T1-T2))**AN*DTIME
      IF(LEXIMP.EQ.1) THEN
       DECRA(5) = AN*A*(.5*(T1-T2))**(AN-1.)*DTIME/
     1             SIG0*.5*(T1+T2)
      END IF
C
      RETURN
      END
```

The derivative

$$\frac{\partial \Delta \bar{\varepsilon}^{cr}}{\partial q} = \frac{nA\Delta t}{\sigma_0}\left(\sinh \frac{q}{\sigma_0}\right)^{n-1} \cosh \frac{q}{\sigma_0}$$

has been defined on the assumption that the subroutine will be used with implicit integration.

# DECURRENT: User subroutine to define electrical current density magnitude as a function of position, time, element number, field variables, etc. for coupled thermal-electrical, coupled temperature-displacement electrical, coupled thermal-electrochemical, and coupled temperature-displacement electrochemical analyses.

**Products:** Abaqus/Standard

## Overview

User subroutine *DECURRENT*:

- can be used to define the current density value as a function of position, time, element number, field variables, etc. for coupled thermal-electrical, coupled temperature-displacement electrical, coupled thermal-electrochemical, and coupled temperature-displacement electrochemical analyses;
- will be called at each load integration point for each element-based nonuniform volume or facet and for each surface-based current density definition; and
- ignores any amplitude references that might appear with the associated step definition or current density load definition.

## User Subroutine Interface

```
      subroutine decurrent(
C        Write
     *   currentsource, dsourcedsol,
C        Read only
     *   solutionvar, dsolutionvar, nsolutionvar,
     *   predef, dpredef, npredef,
     *   coords, ncoords,
     *   i_array, niarray,
     *   r_array, nrarray,
     *   c_array, ncarray)
C
include 'aba_param.inc'
C
      dimension dsourcedsol(*),solutionvar(*),dsolutionvar(*)
      dimension predef(*),dpredef(*),coords(*)
      dimension i_array(*),r_array(*)
      character*8 c_array(*)
C
      parameter (i_decurrent_jltyp   = 1,
     *           i_decurrent_kstep   = 2,
     *           i_decurrent_kinc    = 3,
     *           i_decurrent_noel    = 4,
     *           i_decurrent_npt     = 5,
     *           i_decurrent_proc    = 6)
C
      parameter (ir_decurrent_timestep = 1,
     *           ir_decurrent_timetotal = 2,
     *           ir_decurrent_dtime = 3)
C
       user coding to define currentsource and dsourcedsol
```

```
return
end
```

## Variables to Be Defined

`currentsource` The current density value. The units are $CL^{-2}T^{-1}$ for element facets or surfaces and $CL^{-3}T^{-1}$ for element volumes. The `currentsource` is passed into the routine as the value specified as part of the corresponding current density definition. If the value is not defined, `currentsource` is passed in as zero.

`dsourcedsol(1)` The derivative of current density value with respect to electric potential. The value is passed in as zero.

`dsourcedsol(2)` The derivative of current density value with respect to temperature. The value is passed in as zero. In general, the number of entries to be defined in the `dsourcedsol` array must be equal to the `nsolutionvar` parameter defined below.

## Variables Passed in for Information

`solutionvar (*)` An array containing the latest solution components' values (`solutionvar(1)` for electric potential and `solutionvar(2)` for temperature) at the load integration point, based on interpolation from the values at the nodes.

`dsolutionvar (*)` An array containing an increment in solution components from the previous time at the load integration point, based on interpolation from the values at the nodes.

`nsolutionvar` The number of solution components.

`predef(*)` An array containing the latest values of all the predefined field variables at the integration point, based on interpolation from the values at the nodes.

`dpredef(*)` An array containing an increment in all the predefined field variables from the previous time at the load integration point, based on interpolation from the values at the nodes.

`npredef` The number of predefined field variables.

`coords(*)` An array containing the coordinates of the load integration point.

`ncoords` The number of coordinates.

`i_array(i_decurrent_jltyp)` Identifies the electric current type for which this call to the user subroutine is being made. The current type can be a body current, a surface-based current, or an element-based surface current. This information is useful when several different nonuniform distributed fluxes are being imposed on an element at the same time. See *About the Element Library* for element face identification. The key is as follows:

| i_array(i_decurrent_jltyp) value | Electric current type |
|---|---|
| 17 | CBFNU, element volume |
| 27 | CS1NU, element facet 1 |
| 28 | CS2NU, element facet 2 |
| 29 | CS3NU, element facet 3 |
| 30 | CS4NU, element facet 4 |
| 31 | CS5NU, element facet 5 |
| 32 | CS6NU, element facet 6 |

| i_array(i_decurrent_jltyp) value | Electric current type |
|---|---|
| 33 | CSNU, element facet 1 on surface |
| 34 | CSNU, element facet 2 on surface |
| 35 | CSNU, element facet 3 on surface |
| 36 | CSNU, element facet 4 on surface |
| 37 | CSNU, element facet 5 on surface |
| 38 | CSNU, element facet 6 on surface |

`i_array(i_decurrent_kstep)` Step number.

`i_array(i_decurrent_kinc)` Increment number.

`i_array(i_decurrent_noel)` Element number.

`i_array(i_decurrent_npt)` Load integration point number.

`i_array(i_decurrent_proc)`

| i_array(i_decurrent_proc) value | Procedure |
|---|---|
| 1 | Coupled thermal-electrical steady-state |
| 2 | Coupled thermal-electrical transient |
| 3 | Coupled temperature-displacement electrical steady-state |
| 4 | Coupled temperature-displacement electrical transient |
| 5 | Coupled temperature-displacement electrochemical steady-state or coupled thermal-electrochemical steady-state |
| 6 | Coupled temperature-displacement electrochemical transient or coupled thermal-electrochemical transient |
| 0 | Other procedures |

`niarray` Size of `array i_array`. Currently equal to 6.

`r_array(ir_decurrent_timestep)` The latest step time.

`r_array(ir_decurrent_timetotal)` The latest total time.

`r_array(ir_decurrent_dtime)` The time increment.

`nrarray` Size of `array r_array`. Currently equal to 3.

`c_array(1)` Surface name for surface-based current, blank otherwise.

`ncarray` Size of `array c_array`. Currently equal to 1.

# DFLOW: User subroutine to define nonuniform pore fluid velocity in a consolidation analysis.

**Products:** Abaqus/Standard

**References:**

- *Pore Fluid Flow*
- **DFLOW*
- **DSFLOW*

## Overview

User subroutine *DFLOW*:

- can be used to define the variation of the seepage magnitude as a function of position, time, pore pressure, etc. in a soils consolidation analysis;
- will be called at each flow integration point for each element-based or surface-based nonuniform flow definition in the analysis; and
- ignores any amplitude references that may appear with the associated nonuniform flow definition.

## User Subroutine Interface

```
      SUBROUTINE DFLOW(FLOW,U,KSTEP,KINC,TIME,NOEL,NPT,COORDS,
     1 JLTYP,SNAME)
C
      INCLUDE 'ABA_PARAM.INC'
C
      DIMENSION TIME(2),COORDS(3)
      CHARACTER*80 SNAME


      user coding to define FLOW


      RETURN
      END
```

## Variables to Be Defined

`FLOW` Effective velocity of pore fluid crossing the surface at this point from the inside of the region modeled to the outside of the region modeled. Units are $LT^{-1}$. Effective velocity is the volumetric flow rate per unit area (refer to *Permeability*).

`FLOW` will be passed into the routine as the magnitude of the seepage specified as part of the element-based or surface-based flow definition. If the magnitude is not defined, `FLOW` will be passed in as zero.

The effective velocity is not available for output purposes.

## Variables Passed in for Information

`U` Estimated pore pressure at this time at this point.

`KSTEP` Step number.

`KINC` Increment number.

`TIME(1)` Current value of step time (defined only in transient analysis).

`TIME(2)` Current value of total time (defined only in transient analysis).

`NOEL` Element number.

`NPT` Integration point number on the element's surface.

`COORDS` An array containing the coordinates of this point. These are the current coordinates if geometric nonlinearity is accounted for during the step (see *Defining an Analysis*); otherwise, the array contains the original coordinates of the point.

`JLTYP` Identifies the element face for which this call to *DFLOW* is being made through the element-based flow definition. This information is useful when several different nonuniform distributed flows are being imposed on an element at the same time. See *About the Element Library* for identification of element faces. The key is as follows:

| JLTYP | Flow type |
|---|---|
| 0 | Surface-based load |
| 11 | S1NU |
| 12 | S2NU |
| 13 | S3NU |
| 14 | S4NU |
| 15 | S5NU |
| 16 | S6NU |

`SNAME` Surface name for which this call to *DFLOW* is being made through the surface-based flow definition (`JLTYP`=0). For an element-based flow definition the surface name is passed in as a blank.

# DFLUX: User subroutine to define nonuniform distributed flux in a heat transfer or mass diffusion analysis.

**Products:** Abaqus/Standard

**References:**

- *Thermal Loads*
- *Mass Diffusion Analysis*
- **DFLUX*
- **DSFLUX*
- *DFLUX*

## Overview

User subroutine *DFLUX*:

- can be used to define a nonuniform distributed flux as a function of position, time, temperature, element number, integration point number, etc. in a heat transfer or mass diffusion analysis;
- will be called at each flux integration point for each element-based or surface-based (heat transfer only) nonuniform distributed flux definition in the analysis;
- ignores any amplitude references that may appear with the associated nonuniform distributed flux definition; and
- uses the nodes as flux integration points for first-order heat transfer, first-order coupled temperature-displacement, first-order coupled thermal-electrical-structural, and mass diffusion elements.

## User Subroutine Interface

```
      SUBROUTINE DFLUX(FLUX,SOL,KSTEP,KINC,TIME,NOEL,NPT,COORDS,
     1 JLTYP,TEMP,PRESS,SNAME)
C
      INCLUDE 'ABA_PARAM.INC'
C
      DIMENSION FLUX(2), TIME(2), COORDS(3)
      CHARACTER*80 SNAME


      user coding to define FLUX(1) and FLUX(2)


      RETURN
      END
```

## Variables to Be Defined

`FLUX(1)` Magnitude of flux flowing into the model at this point. In heat transfer cases the units are $JT^{-1}L^{-2}$ for surface fluxes and $JT^{-1}L^{-3}$ for body flux. In transient heat transfer cases where a nondefault amplitude is used to vary the applied fluxes, the time average flux over the time increment must be defined rather than the value at the end of the time increment. In mass diffusion cases the units are $PLT^{-1}$ for surface fluxes and $PT^{-1}$ for body flux.

`FLUX(1)` will be passed into the routine as the magnitude of the flux specified as part of the element-based or surface-based flux definition. If the magnitude is not defined, `FLUX(1)` will be passed in as zero.

This flux is not available for output purposes.

`FLUX(2)` In heat transfer cases $dq/d\theta$, the rate of change of the flux with respect to the temperature at this point. The units are $JT^{-1}L^{-2}\theta^{-1}$ for surface fluxes and $JT^{-1}L^{-3}\theta^{-1}$ for body flux.

In mass diffusion cases: $dq/dc$, the rate of change of the flux with respect to the mass concentration at this point. The units are $LT^{-1}$ for surface fluxes and $T^{-1}$ for body flux.

The convergence rate during the solution of the nonlinear equations in an increment is improved by defining this value, especially when the flux is a strong function of temperature in heat transfer analysis or concentration in mass diffusion analysis.

## Variables Passed in for Information

`SOL` Estimated value of the solution variable (temperature in a heat transfer analysis or concentration in a mass diffusion analysis) at this time at this point.

`KSTEP` Step number.

`KINC` Increment number.

`TIME(1)` Current value of step time (defined only in transient analysis).

`TIME(2)` Current value of total time (defined only in transient analysis).

`NOEL` Element number.

`NPT` Integration point number in the element or on the element's surface. The integration scheme depends on whether this is a surface or a body flux.

`COORDS` An array containing the coordinates of this point. These are the current coordinates if geometric nonlinearity is accounted for during the step (see *Defining an Analysis*); otherwise, the array contains the original coordinates of the point.

`JLTYP` Identifies the flux type for which this call to `DFLUX` is being made. The flux type may be a body flux, a surface-based flux, or an element-based surface flux. For element-based surface fluxes, this variable identifies the element face for which this call to *DFLUX* is being made. This information is useful when several different nonuniform distributed fluxes are being imposed on an element at the same time. See *About the Element Library* for element face identification. The key is as follows:

| JLTYP | Flux type |
|---|---|
| 0 | Surface-based flux |
| 1 | BFNU |
| 11 | S1NU (SNEGNU for heat transfer shells) |
| 12 | S2NU (SPOSNU for heat transfer shells) |
| 13 | S3NU |
| 14 | S4NU |
| 15 | S5NU |
| 16 | S6NU |

`TEMP` Current value of temperature at this integration point (defined only for a mass diffusion analysis). Temperature for a heat transfer analysis is passed in as variable `SOL`.

PRESS Current value of the equivalent pressure stress at this integration point (defined only for a mass diffusion analysis).

SNAME Surface name for a surface-based flux definition (JLTYP=0). For a body flux or an element-based surface flux the surface name is passed in as blank.

# DISP: User subroutine to specify prescribed boundary conditions.

**Products:** Abaqus/Standard

**References:**

- *Boundary Conditions*
- *Connector Actuation*
- **BOUNDARY*
- **CONNECTOR MOTION*
- *Riser dynamics*
- *DISP*
- *Boundary conditions*

## Overview

User subroutine *DISP*:

- can be used to define the magnitudes of prescribed boundary conditions or connector motions;
- requires incremental values to be defined for prescribed rotation boundary conditions;
- is called for all degrees of freedom listed in a user-subroutine-defined boundary condition or connector motion definition;
- redefines any magnitudes that may be specified (and possibly modified by an amplitude) as part of the associated boundary condition or connector motion definition; and
- ignores the specified type, if any, of the associated boundary condition or connector motion definition.

## User Subroutine Interface

```
      SUBROUTINE  DISP(U,KSTEP,KINC,TIME,NODE,NOEL,JDOF,COORDS)
C
      INCLUDE 'ABA_PARAM.INC'
C
      DIMENSION U(3),TIME(3),COORDS(3)
C


      user coding to define U


      RETURN
      END
```

## Variables to Be Defined

`U(1)` All variable types except rotation: the total value of the prescribed variable at this point. The variable may be displacement, pore pressure, temperature, etc., depending on the degree of freedom constrained. `U(1)` will be passed into the user subroutine as the value defined by any magnitude and/or amplitude specification for the boundary condition or connector motion.

Rotation variable type: the incremental value of the prescribed rotation at this point. The time increment, passed into the user subroutine through `TIME(3)`, should be used to calculate the incremental value. In addition, `U(1)` will be passed into user subroutine `DISP` as the value defined by any magnitude or amplitude specification for the boundary condition.

If the analysis procedure requires that the time derivatives of prescribed variables be defined (for example, in a dynamic analysis the velocity and acceleration, as well as the value of the variable, are needed), $du/dt$ must be given in `U(2)` and $d^2u/dt^2$ in `U(3)`. The total value of the variable (incremental value in the case of rotation) and its time derivatives must be given in user subroutine `DISP`, regardless of the type of boundary condition or connector motion.

## Variables Passed in for Information

| | |
|---|---|
| `KSTEP` | Step number. |
| `KINC` | Increment number. |
| `TIME(1)` | Current value of step time. |
| `TIME(2)` | Current value of total time. |
| `TIME(3)` | Current value of time increment. |
| `NODE` | Node number. This variable cannot be used if user subroutine *DISP* is used to prescribe connector motions. |
| `NOEL` | Element number. This variable cannot be used if user subroutine *DISP* is used to prescribe boundary conditions. |
| `JDOF` | Degree of freedom. |
| `COORDS` | An array containing the current coordinates of this point. These are the coordinates at the end of the prior increment if geometric nonlinearity is accounted for during the step (see *Defining an Analysis*); otherwise, the array contains the original coordinates of the node. This array cannot be used if user subroutine *DISP* is used to prescribe connector motions. |

# DLOAD: User subroutine to specify nonuniform distributed loads.

**Products:** Abaqus/Standard

**References:**

- *Distributed Loads*
- **DLOAD*
- **DSLOAD*
- *Nonuniform crack-face loading and J-integrals*
- *Pure bending of a cylinder: CAXA elements*
- *Cylinder subjected to asymmetric pressure loads: CAXA elements*
- *Patch test for axisymmetric elements*
- *Transient internal pressure loading of a viscoelastic cylinder*
- *DLOAD*

## Overview

User subroutine *DLOAD*:

- can be used to define the variation of the distributed load magnitude as a function of position, time, element number, load integration point number, etc.;
- will be called at each load integration point for each element-based or surface-based nonuniform distributed load definition during stress analysis;
- will be called at each stiffness integration point for computing the effective axial force, ESF1, for pipe elements subjected to nonuniform load types PENU and PINU;
- cannot be used in mode-based procedures to describe the time variation of the load; and
- ignores any amplitude references that may appear with the associated step definition or nonuniform distributed load definition.

## User Subroutine Interface

```
      SUBROUTINE DLOAD(F,KSTEP,KINC,TIME,NOEL,NPT,LAYER,KSPT,
     1 COORDS,JLTYP,SNAME)
C
      INCLUDE 'ABA_PARAM.INC'
C
      DIMENSION TIME(2), COORDS (3)
      CHARACTER*80 SNAME

      user coding to define F


      RETURN
      END
```

## Variables to Be Defined

`F` Magnitude of the distributed load. Units are $FL^{-2}$ for surface loads and $FL^{-3}$ for body forces. `F` will be passed into the routine as the magnitude of the load specified as part of the element-based or surface-based distributed load definition. If the magnitude is not defined, `F` will be passed in as zero. For a static analysis that uses the modified Riks method (*Static Stress Analysis*) `F` must be defined as a function of the load proportionality factor, $\lambda$. The distributed load magnitude is not available for output purposes.

## Variables Passed in for Information

`KSTEP` Step number.

`KINC` Increment number.

`TIME(1)` Current value of step time or current value of the load proportionality factor, $\lambda$, in a Riks step.

`TIME(2)` Current value of total time.

`NOEL` Element number.

`NPT` Load integration point number within the element or on the element's surface, depending on the load type. (Stiffness integration point number while computing effective axial force, ESF1, for pipe elements subjected to load types PENU and PINU.)

`LAYER` Layer number (for body forces in layered solids).

`KSPT` Section point number within the current layer.

`COORDS` An array containing the coordinates of the load integration point. These are the current coordinates if geometric nonlinearity is accounted for during the step (see *Defining an Analysis*); otherwise, the array contains the original coordinates of the point. For axisymmetric elements that allow nonaxisymmetric deformation, `COORDS(3)` is the angular position of the integration point, in degrees.

`JLTYP` Identifies the load type for which this call to *DLOAD* is being made. The load type may be a body force, a surface-based load, or an element-based surface load. For element-based surface loads, this variable identifies the element face for which this call to *DLOAD* is being made. This information is useful when several different nonuniform distributed loads are being imposed on an element at the same time. See *About the Element Library* for element face identification. The key is as follows:

| JLTYP | Load type |
|---|---|
| 0 | Surface-based load |
| 1 | BXNU |
| 1 | BRNU |
| 2 | BYNU (except for axisymmetric elements) |
| 2 | BZNU (for axisymmetric elements only) |
| 3 | BZNU (for three-dimensional elements and asymmetric-axisymmetric elements) |
| 20 | PNU |
| 21 | P1NU |
| 22 | P2NU |
| 23 | P3NU |
| 24 | P4NU |
| 25 | P5NU |
| 26 | P6NU |

| JLTYP | Load type |
|---|---|
| 27 | PINU |
| 28 | PENU |
| 41 | PXNU |
| 42 | PYNU |
| 43 | PZNU |

SNAME Surface name for a surface-based load definition (JLTYP=0). For a body force or an element-based surface load the surface name is passed in as blank.

# FILM: User subroutine to define nonuniform film coefficient and associated sink temperatures for heat transfer analysis.

**Products:** Abaqus/Standard

**References:**

- *Thermal Loads*
- **CFILM*
- **FILM*
- **SFILM*
- *Temperature-dependent film condition*
- *Specifying Element-Based Film Conditions on Evolving Faces of an Element in Abaqus/Standard*

## Overview

User subroutine *FILM*:

- can be used to define a node-based, element-based, or surface-based nonuniform film coefficient;
- can be used to define sink temperatures as functions of position, time, temperature, node number, element number, integration point number, etc.;
- will be called during procedures that allow heat transfer analysis at each node or surface integration point of those surfaces and elements for which node-based, element-based, or surface-based nonuniform film conditions are defined;
- ignores any amplitude references for the sink temperature or film coefficient that may appear with the associated nonuniform film definition; and
- uses the nodes for first-order heat transfer elements as surface integration points for both element-based and surface-based films.

## User Subroutine Interface

```
      SUBROUTINE FILM(H,SINK,TEMP,KSTEP,KINC,TIME,NOEL,NPT,
     1 COORDS,JLTYP,FIELD,NFIELD,SNAME,NODE,AREA)
C
      INCLUDE 'ABA_PARAM.INC'
C
      DIMENSION H(2),TIME(2),COORDS(3), FIELD(NFIELD)
      CHARACTER*80 SNAME

      user coding to define H(1), H(2), and SINK

      RETURN
      END
```

## Variables to Be Defined

`H(1)` Film coefficient at this point. Units are $JT^{-1}L^{-2}\theta^{-1}$. `H(1)` will be passed into the routine as the magnitude of the film coefficient specified as part of the node-based, element-based, or surface-based film condition definition. If the magnitude is not defined, `H(1)` will be initialized to zero.

H(2) $dh/d\theta$, rate of change of the film coefficient with respect to the surface temperature at this point. Units are $JT^{-1}L^{-2}\theta^{-2}$. The rate of convergence during the solution of the nonlinear equations in an increment is improved by defining this value, especially when the film coefficient is a strong function of surface temperature.

SINK Sink temperature. SINK will be passed into the routine as the sink temperature specified as part of the node-based, element-based, or surface-based film condition definition. If the sink temperature is not defined, SINK will be initialized to zero.

## Variables Passed in for Information

TEMP Estimated surface temperature at this time at this point.

KSTEP Step number.

KINC Increment number.

TIME(1) Current value of step time.

TIME(2) Current value of total time.

NOEL Element number. This variable is passed in as zero for node-based films.

NPT Surface integration point number. This variable is passed in as zero for node-based films.

COORDS An array containing the coordinates of this point. These are the current coordinates if geometric nonlinearity is accounted for during the step (see *Defining an Analysis*); otherwise, the array contains the original coordinates of the point.

JLTYP Identifies the element face for which this call to *FILM* is being made for an element-based film coefficient specification. This information is useful when several different nonuniform film conditions are being imposed on an element at the same time. See *About the Element Library* for element face identification. The key is as follows:

| JLTYP | Film type |
|---|---|
| 0 | Node-based or surface-based loading |
| 11 | F1NU (FNEGNU for heat transfer shells) |
| 12 | F2NU (FPOSNU for heat transfer shells) |
| 13 | F3NU |
| 14 | F4NU |
| 15 | F5NU |
| 16 | F6NU |
| -1 | Free surface interior to the element at the current time, applicable to the FFSNU *FILM* type only |

FIELD Interpolated values of field variables at this point.

NFIELD Number of field variables.

SNAME Surface name for which this call to *FILM* is being made for a surface-based film coefficient specification (JLTYP=0). This variable is passed in as blank for both node-based and element-based films.

NODE Node number. This variable is passed in as zero for both element-based and surface-based films.

AREA Nodal area for node-based films. AREA will be passed into the routine as the nodal area specified as part of the node-based film coefficient specification. This nodal area is not available for output purposes. This variable is passed in as zero for both element-based and surface-based films.

# FLOW: User subroutine to define nonuniform seepage coefficient and associated sink pore pressure for consolidation analysis.

**Products:** Abaqus/Standard

**References:**

- *Pore Fluid Flow*
- **FLOW*
- **SFLOW*

## Overview

User subroutine *FLOW*:

- can be used in a soils consolidation analysis to define the variation of the reference pore pressure and the seepage coefficient as functions of position, time, pore pressure, element number, integration point number, etc.;
- will be called at each integration point of element surfaces for which element-based or surface-based nonuniform surface seepage flow is defined; and
- ignores any amplitude references that may appear with the associated nonuniform flow definition.

## User Subroutine Interface

```
      SUBROUTINE FLOW(H,SINK,U,KSTEP,KINC,TIME,NOEL,NPT,COORDS,
     1 JLTYP,SNAME)
C
      INCLUDE 'ABA_PARAM.INC'
C
      DIMENSION TIME(2), COORDS(3)
      CHARACTER*80 SNAME


      user coding to define H and SINK


      RETURN
      END
```

## Variables to Be Defined

`H` Seepage coefficient at this point. Units are $F^{-1}L^{3}T^{-1}$. `H` will be passed into the routine as the reference seepage coefficient value specified as part of the element-based or surface-based flow definition. If the reference value is not defined, `H` will be passed in as zero.

`SINK` Sink pore pressure. `SINK` will be passed into the routine as the reference pore pressure value specified as part of the element-based or surface-based flow definition. If the reference value is not defined, `SINK` will be passed in as zero.

## Variables Passed in for Information

`U` Estimated surface total pore pressure at this time and at this point.

KSTEP — Step number.

KINC — Increment number.

TIME(1) — Current value of step time (defined only in transient analysis).

TIME(2) — Current value of total time (defined only in transient analysis).

NOEL — Element number.

NPT — Surface integration point number.

COORDS — An array containing the coordinates of this integration point. These are the current coordinates if geometric nonlinearity is accounted for during the step (see *Defining an Analysis*); otherwise, the array contains the original coordinates of the point.

JLTYP — Identifies the element face for which this call to *FLOW* is being made for an element-based flow. This information is useful when several nonuniform flow conditions are being imposed on an element at the same time. See *About the Element Library* for identification of the element faces. The key is as follows:

| JLTYP | Flow type |
|---|---|
| 0 | Surface-based flow |
| 61 | Q1NU |
| 62 | Q2NU |
| 63 | Q3NU |
| 64 | Q4NU |
| 65 | Q5NU |
| 66 | Q6NU |

SNAME — Surface name for which this call to *FLOW* is being made for a surface-based flow (JLTYP=0). For an element-based flow the surface name is passed in as a blank.

# FRIC: User subroutine to define frictional behavior for contact surfaces.

**Products:** Abaqus/Standard

**References:**

- *Frictional Behavior*
- **FRICTION*
- *Thermal-stress analysis of a disc brake*
- *FRIC*

## Overview

User subroutine *FRIC*:

- can be used to define the frictional behavior between contacting surfaces;
- can be used when the extended versions of the classical Coulomb friction model provided in Abaqus are too restrictive and a more complex definition of shear transmission between contacting surfaces is required;
- will be called at points on the secondary surface of a contact pair and at the integration points in a contact element (only when the contact point is closed) for which the contact interaction property model contains user-subroutine-defined friction;
- must provide the entire definition of shear interaction between the contacting surfaces; and
- can use and update solution-dependent state variables.

## User Subroutine Interface

```
      SUBROUTINE FRIC(LM,TAU,DDTDDG,DDTDDP,DSLIP,SED,SFD,
     1 DDTDDT,PNEWDT,STATEV,DGAM,TAULM,PRESS,DPRESS,DDPDDH,SLIP,
     2 KSTEP,KINC,TIME,DTIME,NOEL,CINAME,SECNAME,MAINNAME,NPT,NODE,
     3 NPATCH,COORDS,RCOORD,DROT,TEMP,PREDEF,NFDIR,MCRD,NPRED,
     4 NSTATV,CHRLNGTH,PROPS,NPROPS)
C
      INCLUDE 'ABA_PARAM.INC'
C
      CHARACTER*80 CINAME,SECNAME,MAINNAME
C
      DIMENSION TAU(NFDIR),DDTDDG(NFDIR,NFDIR),DDTDDP(NFDIR),
     1 DSLIP(NFDIR),DDTDDT(NFDIR,2),STATEV(*),DGAM(NFDIR),
     2 TAULM(NFDIR),SLIP(NFDIR),TIME(2),COORDS(MCRD),
     3 RCOORD(MCRD),DROT(2,2),TEMP(2),PREDEF(2,*),PROPS(NPROPS)


      user coding to define LM, TAU, DDTDDG, DDTDDP,
      and, optionally, DSLIP, SED, SFD, DDTDDT, PNEWDT, STATEV


      RETURN
      END
```

## Variables to Be Defined

**In all cases**

LM Relative motion flag. User subroutine *FRIC* is called only if the contact point is determined to be closed; that is, if the contact pressure is positive (the contact point was closed in the previous iteration) or if the contact point is overclosed (the contact point was open in the previous iteration).

During iterations LM is passed into the subroutine as the value defined during the previous iteration. At the start of an increment or if the contact point opened during the previous iteration, this variable will be passed into the routine depending on the contact condition in the previous increment. If the contact point was slipping, LM is equal to 0; if the contact point was sticking, LM is equal to 1; and if the contact point was open, LM is equal to 2.

Set LM equal to 0 if relative motion is allowed (either due to slip or elastic stick). In this case the subroutine must specify the frictional stress $\tau_1$ (and $\tau_2$ for three-dimensional analysis) as a function of the relative sliding motion $\gamma_1$ (and $\gamma_2$), the interface contact pressure $p$, and other predefined or user-defined state variables. In addition, the subroutine must define the derivatives of the frictional stress with respect to $\gamma_1$, ($\gamma_2$), and $p$. For example, in the case of isotropic elastic sticking, $\partial\tau_1/\partial\gamma_1 = \partial\tau_2/\partial\gamma_2 = k_{elas}$, $\partial\tau_1/\partial\gamma_2 = \partial\tau_2/\partial\gamma_1 = 0$, where $k_{elas}$ is the elastic stiffness of the interface.

Set LM equal to 1 if no relative motion is allowed; a rigid sticking condition at the interface is enforced by a Lagrange multiplier method. In this case no further variables need to be updated. If LM is always set to 1, a "perfectly rough" interface is created. It is not advisable to set LM to 1 when the finite-sliding, surface-to-surface contact formulation is used.

Set LM equal to 2 if friction is ignored (frictionless sliding is assumed). In this case no further variables need to be updated. If LM is always set to 2, a "perfectly smooth" interface is created.

You can make decisions about the stick/slip condition based on incremental slip information and calculated frictional stresses. These quantities are passed in by Abaqus/Standard, as discussed below.

To avoid convergence problems for the general class of frictional contact problems, set LM to 2 and exit this routine if the contact point was open at the end of the previous increment; that is, if Abaqus/Standard sets LM=2 when it calls this routine, exit the routine.

**If the return value of LM is 0**

TAU(NFDIR) These values are passed in as the values of the frictional stress components, $\tau_\alpha$, at the beginning of the increment and must be updated to the values at the end of the increment. Here, and in the rest of this description, Greek subscripts ($\alpha$, $\beta$) refer to frictional shear directions. The orientation of these directions on contact surfaces is defined in *Contact Formulations in Abaqus/Standard*.

DDTDDG(NFDIR,NFDIR) $\partial\Delta\tau_\alpha/\partial\Delta\gamma_\beta$, partial derivative of the frictional stress in direction $\alpha$ with respect to the relative motion in direction $\beta$.

DDTDDP(NFDIR) $\partial\Delta\tau_\alpha/\partial\Delta p$, partial derivative of the frictional stress in direction $\alpha$ with respect to the contact pressure. Since these terms yield an unsymmetric contribution to the stiffness matrix, they are used only if the unsymmetric equation solver is used (see *Defining an Analysis*).

## Variables That Can Be Updated

DSLIP(NFDIR) $\Delta\gamma_\alpha^{sl}$, increment in nonrecoverable sliding motion (slip). If LM was 0 in the previous iteration, this array is passed in as the user-defined values during the previous iteration; otherwise, it will be zero. The array should be updated only if the return value of LM is 0.

This array is useful to detect slip reversals between iterations. It is used by the output options to indicate whether this point is sticking or slipping. Upon convergence of an increment, the values in DSLIP(NFDIR) are accumulated in SLIP(NFDIR), which are stored as the plastic strains.

SED This variable is passed in as the value of the elastic energy density at the start of the increment and should be updated to the elastic energy density at the end of the increment. This variable is used for output only and has no effect on other solution variables.

SFD This variable should be defined as the incremental frictional dissipation. The units are energy per unit area if the contact element or contact pair calling *FRIC* uses stresses as opposed to forces. For regular stress analysis this variable is used for output only and has no effect on other solution variables. In coupled temperature-displacement and coupled thermal-electrical-structural analyses the dissipation is converted into heat if the gap heat generation model is used. If SFD is not defined, the heat generation is calculated based on the dissipation obtained as the product of the slip increment, DSLIP, and the frictional stress, TAU.

DDTDDT(NFDIR,2) $\partial\Delta\tau_\alpha/\partial\Delta\theta_1$, $\partial\Delta\tau_\alpha/\partial\Delta\theta_2$ partial derivatives of the frictional stress in direction $\alpha$ with respect to the temperatures of the two surfaces. This is required only for coupled temperature-displacement and coupled thermal-electrical-structural elements, in which the frictional stress is a function of the surface temperatures.

PNEWDT Ratio of suggested new time increment to the time increment being used (DTIME, see below). This variable allows you to provide input to the automatic time incrementation algorithms in Abaqus/Standard (if automatic time incrementation is chosen).

PNEWDT is set to a large value before each call to *FRIC*.

If PNEWDT is redefined to be less than 1.0, Abaqus/Standard must abandon the time increment and attempt it again with a smaller time increment. The suggested new time increment provided to the automatic time integration algorithms is PNEWDT × DTIME, where the PNEWDT used is the minimum value for all calls to user subroutines that allow redefinition of PNEWDT for this iteration.

If PNEWDT is given a value that is greater than 1.0 for all calls to user subroutines for this iteration and the increment converges in this iteration, Abaqus/Standard may increase the time increment. The suggested new time increment provided to the automatic time integration algorithms is PNEWDT × DTIME, where the PNEWDT used is the minimum value for all calls to user subroutines for this iteration.

If automatic time incrementation is not selected in the analysis procedure, values of PNEWDT greater than 1.0 will be ignored and values of PNEWDT less than 1.0 will cause the job to terminate.

STATEV(NSTATV) An array containing the user-defined solution-dependent state variables. You specify the number of available state variables; see *Allocating Space for Solution-Dependent State Variables* for details. This array will be passed in containing the values of these variables at the start of the increment. If any of the solution-dependent state variables is being used with the friction behavior, they must be updated in this subroutine to their values at the end of the increment.

## Variables Passed in for Information

DGAM(NFDIR) If LM was set to 0 in the previous iteration, this value is the increment of sliding motion in the current increment, $\Delta\gamma_\alpha$. Otherwise, it will be zero. Comparison with DSLIP(NFDIR) makes it possible to determine whether slip changes to stick at this point and/or if there is a slip direction reversal occurring at this point.

TAULM(NFDIR) If LM was set to 1 in the previous iteration, this value is the current value of the constraint stress at the end of the increment, $\tau_\alpha^{LM}$. Otherwise, it will be zero. Comparison with the critical shear stress makes it possible to determine whether stick changes to slip at this point.

PRESS $p$, contact pressure at end of increment.

DPRESS $\Delta p$, increment in contact pressure.

DDPDDH $\partial \Delta p / \partial \Delta h$, current contact stiffness, in the case of soft contact (*Contact Pressure-Overclosure Relationships*).

SLIP(NFDIR) Total nonrecoverable sliding motion (slip) at the beginning of the increment, $\gamma_\alpha^{sl}$. This value is the accumulated value of DSLIP(NFDIR) from previous increments.

KSTEP Step number.

KINC Increment number.

TIME(1) Value of step time at the end of the increment.

TIME(2) Value of total time at the end of the increment.

DTIME Current increment in time.

NOEL Element label for contact elements. Passed in as zero if contact surfaces are defined.

CINAME User-specified surface interaction name associated with the friction definition, left justified. For contact elements it is the element set name given for the interface definition associated with the friction definition; if an optional name is assigned to the interface definition, CINAME is passed in as this name, left justified.

SECNAME Secondary surface name. Passed in as blank if contact elements are used.

MAINNAME Main surface name. Passed in as blank if contact elements are used.

NPT Integration point number for contact elements. Passed in as zero if contact surfaces are defined.

NODE User-defined global secondary node number (or internal node number for models defined in terms of an assembly of part instances) involved with this contact point. Corresponds to the predominant secondary node of the constraint if the surface-to-surface contact formulation is used. Passed in as zero if called from a contact element.

NPATCH Not used.

COORDS(MCRD) An array containing the current coordinates of this point.

RCOORD(MCRD) If the main surface is defined as a rigid surface, this array is passed in containing the coordinates of the opposing point on the rigid surface in its current position and orientation.

DROT(2,2) Rotation increment matrix. For contact with a three-dimensional rigid surface, this matrix represents the incremental rotation of the surface directions relative to the rigid surface. It is provided so that vector- or tensor-valued state variables can be rotated appropriately in this subroutine. Stress and slip components are already rotated by this amount before *FRIC* is called. This matrix is passed in as a unit matrix for two-dimensional and axisymmetric contact problems.

TEMP(2) Current temperature at the secondary node and the opposing main surface, respectively.

| | |
|---|---|
| PREDEF(2,NPRED) | An array containing pairs of values of all the user-specified field variables at the end of the current increment (initial values at the beginning of the analysis and current values during the analysis). If *FRIC* is called from a contact pair, the first value in a pair corresponds to the secondary node and the second value corresponds to the nearest point on the main surface. If *FRIC* is called from a large-sliding contact element, PREDEF(1,NPRED) corresponds to the value at the integration point of the element and PFREDEF(2,NPRED) corresponds to the nearest point on the opposing surface. If *FRIC* is called from a small-sliding contact element, PREDEF(1,NPRED) corresponds to the value at the integration point of the first side and PFREDEF(2,NPRED) corresponds to the value at the integration point on the opposite face of the element. |
| NFDIR | Number of friction directions. |
| MCRD | Number of coordinate directions at the contact point. |
| NPRED | Number of predefined field variables. |
| NSTATV | Number of user-defined state variables. |
| CHRLNGTH | Characteristic contact surface face dimension, which can be used to define the maximum allowable elastic slip. |
| PROPS(NPROPS) | Array of user-specified property values that are used to define the frictional behavior between the contacting surfaces. |
| NPROPS | User-specified number of property values associated with this friction model. |

# FRIC_COEF: User subroutine to define the frictional coefficient for contact surfaces.

**Products:** Abaqus/Standard

**References:**

- *Frictional Behavior*
- **FRICTION*
- *FRIC_COEF*

## Overview

User subroutine *FRIC_COEF*:

- can be used to define the isotropic and anisotropic frictional coefficients between contacting surfaces;
- corresponds to the classical Coulomb friction model; and
- can be used with the contact pair and general contact algorithms.

## User Subroutine Interface

```
      subroutine fric_coef (
C Write only -
     *   fCoef, fCoefDeriv,
C Read only, except state variables within rData array -
     *   nBlock, nProps, nTemp, nFields,
     *   iData, rData,
     *   surfInt, surfSec, surfMain,
     *   props, slipData, pressure,
     *   tempAvg, fieldAvg )
C
      include 'aba_param.inc'
C
      dimension fCoef(nBlock,2),
     *   fCoefDeriv(nBlock,20,2),
     *   props(nProps),
     *   slipData(nBlock,6),
     *   pressure(nBlock),
     *   tempAvg(nBlock),
     *   fieldAvg(nBlock,nFields)
c
c    Offset parameters to various integer quantities in iData(nDataI)
c
      parameter( i_ifc_KStep   = 1,
     *           i_ifc_KInc    = 2,
     *           i_ifc_Kit     = 3,
     *           i_ifc_Node    = 4,
     *           i_ifc_nStateV = 5,
     *           i_ifc_nFDir   = 6,
     *           nDataI        = 6 )
c
c    Offset parameters to various real quantities in rData(*)
c
```

```
      parameter( i_rfc_TimStep = 1,
     *           i_rfc_TimGlb  = 2,
     *           i_rfc_DTimCur = 3,
     *           i_rfc_Drot    = 4,
     *           i_rfc_StateV  = 8 )
c
c     Offset parameters to slip variables in slipData
c
      parameter( i_SD_SlipRateMag  = 1,
     *           i_SD_SlipRate1    = 2,
     *           i_SD_SlipRate2    = 3,
     *           i_SD_SlipEq       = 4,
     *           i_SD_Slip1        = 5,
     *           i_SD_Slip2        = 6,
     *           n_SD_Length       = 6 )
c
c     Offset parameters to derivatives in fCoefDeriv
c
      parameter( i_fDer_SlipRateMag  = 1,
     *           i_fDer_Pressure     = 2,
     *           i_fDer_Temperature  = 3,
     *           i_fDer_SlipRate1    = 4,
     *           i_fDer_SlipRate2    = 5,
     *           i_fDer_SlipEq       = 6,
     *           i_fDer_Slip1        = 7,
     *           i_fDer_Slip2        = 8 )
c
      dimension iData(nDataI), rData(*)
c
      character*80 surfInt, surfSec, surfMain
c
C
      user coding to define fCoef
c
c     Pointers to access blocks of data from rData
c     Solution dependent state variables indexed from
c           1 to number of state variables nStateV where
c           nStateV = iData(i_ifc_nStateV)
c
      pointer (ptrStateVariables,StateVariables)
      real*8 StateVariables(*) !indexed 1 to iData(i_ifc_nStateV)
      ......
c
c     Access state variables
c
      ptrStateVariables = loc(rData(i_rfc_StateV))
      ......

c
c     Compute fCoef
c
      ......
C
      return
      end
```

## Variables to Be Defined

`fCoef(nBlock,2)` For isotropic friction `fCoef(1,1)` must be updated to the current value of the friction coefficient at the contact point. `fCoef(1,2)` should not be updated.

For anisotropic friction `fCoef(1,1)` and `fCoef(1,2)` must be updated to the current values of the friction coefficients in the first and second local tangent directions, respectively.

`fCoef(1,1)` is initialized to 0.0, and `fCoef(1,2)` is initialized to -1.0 before calling *FRIC_COEF*. Abaqus/Standard assumes that isotropic friction applies to contact points for which `fCoef(1,2)` remains at -1.0 on returning from *FRIC_COEF*.

`fCoefDeriv(nBlock,20,2)` For isotropic friction the entries of `fCoefDeriv(1,*,1)` are updated to derivatives of the friction coefficient. The entries of `fCoefDeriv(1,*,2)` should not be updated.

For anisotropic friction the entries of `fCoefDeriv(1,*,1)` are updated to derivatives of the friction coefficient in the first local tangent direction, while the entries of `fCoefDeriv(1,*,2)` must be updated to derivatives of the friction coefficient in the second local tangent direction. See the table below for the meaning of individual derivatives and their parameterized access to update them.

| Derivative | Definition |
|---|---|
| `fCoefDeriv(1,i_fDer_SlipRateMag,k)` | Derivative with respect to slip rate magnitude |
| `fCoefDeriv(1,i_fDer_Pressure,k)` | Derivative with respect to contact pressure |
| `fCoefDeriv(1,i_fDer_Temperature,k)` | Derivative with respect to temperature |
| `fCoefDeriv(1,i_fDer_SlipRate1,k)` | Derivative with respect to slip rate component in the first local tangent direction |
| `fCoefDeriv(1,i_fDer_SlipRate2,k)` | Derivative with respect to slip rate component in the second local tangent direction |
| `fCoefDeriv(1,i_fDer_SlipEq,k)` | Derivative with respect to equivalent slip accumulated across increments |
| `fCoefDeriv(1,i_fDer_Slip1,k)` | Derivative with respect to slip component in the first local tangent direction |
| `fCoefDeriv(1,i_fDer_Slip2,k)` | Derivative with respect to slip component in the second local tangent direction |

## Variables That Can Be Updated

`rData(i_rfc_StateV)` The offset parameter `i_rfc_StateV=8`. The array of real values starting from this offset corresponds to the solution-dependent state variables. You specify the number of available state variables (*Allocating Space for Solution-Dependent State Variables*). These state variables are passed in with values at the start of the increment. Update them to their corresponding values at the end of the increment. See the user subroutine interface provided earlier for an example of how to access the array of state variables from the buffer `rData`.

## Variables Passed in for Information

| | |
|---|---|
| `nBlock` | Equal to 1. |
| `nProps` | User-specified number of property values associated with this friction model. |
| `nTemp` | 1 if the temperature is defined, and 0 if the temperature is not defined. |
| `nFields` | Number of user-specified field variables. |
| `iData(i_ifc_KStep)` | The offset parameter `i_ifc_KStep=1`, and the integer value at this offset denotes the step number. |
| `iData(i_ifc_KInc)` | The offset parameter `i_ifc_KStep=2`, and the integer value at this offset denotes the increment number. |
| `iData(i_ifc_Kit)` | The offset parameter `i_ifc_Kit=3` and the integer value at this offset denotes the counter for the element pass calculations; for example, the counter starts out at 1 before the first equation solver pass, and the counter is reset to 1 after a cutback of an increment. |
| `iData(i_ifc_Node)` | The offset parameter `i_ifc_Node=4`, and the integer value at this offset denotes the user-defined secondary node label (or an internal node number for models defined in terms of an assembly of part instances). For the surface-to-surface formulation, this secondary node corresponds to the predominant secondary node of the constraint. |
| `iData(i_ifc_nStateV)` | The offset parameter `i_ifc_nStateV=5`, and the integer value at this offset denotes the number of state variables you have specified. The default value is 0. |
| `iData(i_ifc_nFDir)` | The offset parameter `i_ifc_nFDir=6`, and the integer value at this offset denotes the number of friction directions. |
| `rData(i_rfc_TimStep)` | The offset parameter `i_rfc_TimStep=1`, and the real value at this offset is the step time. |
| `rData(i_rfc_TimGlb)` | The offset parameter `i_rfc_TimGlb=2`, and the real value at this offset is the total time. |
| `rData(i_rfc_DTimCur)` | The offset parameter `i_rfc_DTimCur=3`, and the real value at this offset is the current increment size from $t = t_{curr} - \Delta t$ to $t = t_{curr}$. |
| `rData(i_rfc_DRot)` | The offset parameter `i_rfc_DRot=4`. The list of four real value entries starting from this offset corresponds to the column major storage of a $2 \times 2$ rotation increment matrix. For contact with a three-dimensional rigid surface, this matrix represents the incremental rotation of the surface directions relative to the rigid surface. It is provided so that vector- or tensor-valued state variables can be rotated appropriately in this subroutine. For other cases including two-dimensional and axisymmetric contact, the rotation increment matrix is the identity matrix. |
| `surfInt` | User-specified surface interaction name, left justified. |
| `surfSec` | Secondary surface name, left justified. |
| `surfMain` | Main surface name, left justified. |
| `props(nProps)` | User-specified vector of property values to define the frictional coefficient at the contact point. |

`slipData(nBlock,6)` Array containing contact slip variables for the current time increment at the contact point. See the table below for the meaning of individual variables and their access through parameterized locations.

| Variable | Definition |
|---|---|
| `slipData(1,i_SD_SlipRateMag)` | Slip rate magnitude |
| `slipData(1,i_SD_SlipRate1)` | Slip rate in the first local tangent direction |
| `slipData(1,i_SD_SlipRate2)` | Slip rate in the second local tangent direction; zero for two-dimensional problems |
| `slipData(1,i_SD_SlipEq)` | Equivalent slip distance accumulated across increments |
| `slipData(1,i_SD_Slip1)` | Slip component in the first local tangent direction |
| `slipData(1,i_SD_Slip2)` | Slip component in the second local tangent direction; zero for two-dimensional problems |

`pressure(nBlock)` Array containing the pressure at the contact point projected at the end of the current time increment.

`tempAvg(nBlock)` Average current temperature between the main and secondary surfaces at the contact point.

`fieldAvg(nBlock,nFields)` Average current value of all the user-specified field variables between the main and secondary surfaces at the contact point.

# GAPCON: User subroutine to define conductance between contact surfaces or nodes in a fully coupled temperature-displacement analysis, coupled thermal-electrical-structural analysis, or pure heat transfer analysis.

**Products:** Abaqus/Standard

**References:**

- *Thermal Contact Properties*
- **GAP CONDUCTANCE*
- *GAPCON*

## Overview

User subroutine *GAPCON*:

- assumes that the heat transfer between surfaces is modeled as $q = k(\theta_A - \theta_B)$, where $q$ is the heat flux per unit area flowing between corresponding points $A$ and $B$ on the surfaces, $k$ is the gap conductance, and $\theta_A$ and $\theta_B$ are the surface temperatures;
- is used to define $k$, providing greater flexibility than direct gap conductance definition in specifying the dependencies of $k$ (for example, it is not necessary to define the gap conductance as a function of the average of the two surfaces' temperatures, mass flow rates, or field variables);
- will be called at the secondary nodes of general contact (*About General Contact in Abaqus/Standard*) or contact pairs (*About Contact Pairs in Abaqus/Standard*) and at the integration points in a contact or a gap element for which the heat conductance definition contains a user-subroutine-defined gap conductance; and
- ignores any dependencies or data specified for the gap conductance outside the user subroutine.

## Usage with Contact Pairs, General Contact, and Gap Elements

When this subroutine is used with general contact or a contact pair, point $A$ is on the secondary surface and point $B$ is on the main surface.

When *GAPCON* is used with gap elements of type DGAP or GAPUNIT, point $A$ is on the first node of the element and point $B$ is the second node of the element.

## User Subroutine Interface

```
      SUBROUTINE GAPCON(AK,D,FLOWM,TEMP,PREDEF,TIME,CINAME,SECNAME,
     1 MAINNAME,COORDS,NOEL,NODE,NPRED,KSTEP,KINC)
C
      INCLUDE 'ABA_PARAM.INC'
C
      CHARACTER*80 CINAME,SECNAME,MAINNAME
C
      DIMENSION AK(5),D(2), FLOWM(2), TEMP(2), PREDEF(2,*),
     1 TIME(2), COORDS(3)
      user coding to define AK(1) -- AK(5)
      RETURN
      END
```

## Variables to Be Defined

`AK(1)` Gap conductance, $k$. The units of $k$ are energy per time (flux) per area per temperature ($JT^{-1}L^{-2}\theta^{-1}$).

`AK(2)` $\partial k/\partial d$, derivative of the gap conductance with respect to the clearance between the bodies. If the gap conductance is not a function of gap clearance, `AK(2)`=0.0. This variable needs to be defined only for fully coupled temperature-displacement and coupled thermal-electrical-structural analyses.

`AK(3)` $\partial k/\partial p$, derivative of the gap conductance with respect to the pressure between the bodies. If the gap conductance is not a function of the pressure, `AK(3)`=0.0. This variable needs to be defined only for fully coupled temperature-displacement and coupled thermal-electrical-structural analyses.

`AK(4)` $\partial k/\partial \theta_A$, derivative of the gap conductance with respect to the temperature of point $A$ on the first surface of the interface.

`AK(5)` $\partial k/\partial \theta_B$, derivative of the gap conductance with respect to the temperature of point $B$ on the second surface of the interface.

## Variables Passed in for Information

`D(1)` Separation between the surfaces, $d$.

`D(2)` Pressure transmitted across the surfaces, $p$. This pressure is zero in pure heat transfer analysis.

`FLOWM(2)` $|\dot{m}|_A$, $|\dot{m}|_B$, magnitudes of the mass flow rate per unit area at points $A$ and $B$.

`TEMP(2)` Current temperature at points $A$ and $B$.

`PREDEF(2,NPRED)` An array containing pairs of values of all the user-specified field variables at the end of the current increment at points $A$ and $B$ (initial values at the beginning of the analysis and current values during the analysis).

`TIME(1)` Value of step time at the end of the increment.

`TIME(2)` Value of total time at the end of the increment.

`CINAME` User-specified surface interaction name associated with the heat conductance definition, left justified. For contact elements it is the element set name given for the interface definition associated with the heat conductance definition; if an optional name is assigned to the interface definition, `CINAME` is passed in as this name, left justified. For gap elements it is the element set name for the element definition associated with the heat conductance definition.

`SECNAME` Secondary surface name. Passed in as blank if contact or gap elements are used.

`MAINNAME` Main surface name. Passed in as blank if contact or gap elements are used.

`COORDS` An array containing the coordinates of point $A$. These are the current coordinates if geometric nonlinearity is accounted for during the step (see *Defining an Analysis*); otherwise, the array contains the original coordinates of the point.

`NOEL` Element label for contact or gap elements. Passed in as zero if contact surfaces are defined.

`NODE` Secondary node number (point $A$) if *GAPCON* is called for a contact pair or a general contact.

`NPRED` Number of predefined field variables.

| | |
|---|---|
| KSTEP | Step number. |
| KINC | Increment number. |

# GAPELECTR: User subroutine to define electrical conductance between surfaces in a coupled thermal-electrical or a coupled thermal-electrical-structural analysis.

**Products:** Abaqus/Standard

**References:**

- *Electrical Contact Properties*
- **GAP ELECTRICAL CONDUCTANCE*

## Overview

User subroutine *GAPELECTR*:

- assumes that the electrical current flowing between the interface surfaces is modeled as $J = \sigma_g (\varphi_A - \varphi_B)$, where $J$ is the electrical current density flowing across the interface from point $A$ (the secondary surface) to point $B$ (the main surface), $\varphi_A$ and $\varphi_B$ are the electrical potential on opposite points of the surfaces, and $\sigma_g$ is the surface electrical conductance;
- is used to define $\sigma_g$, providing much greater flexibility than direct gap electrical conductance definition in specifying the dependencies of $\sigma_g$ (for example, it is not necessary to define the gap electrical conductance as a function of the average of the two surfaces' temperatures and/or field variables);
- will be called at the secondary nodes of general contact (*About General Contact in Abaqus/Standard*) or contact pairs (*About Contact Pairs in Abaqus/Standard*) for which the gap electrical conductance is defined in a user subroutine; and
- ignores any dependencies or data specified for the gap electrical conductance outside the user subroutine.

## User Subroutine Interface

```
      SUBROUTINE GAPELECTR(SIGMA,D,TEMP,PREDEF,TIME,CINAME,
     1 SECNAME,MAINNAME,COORDS,NODE,NPRED,KSTEP,KINC)
C
      INCLUDE 'ABA_PARAM.INC'
C
      CHARACTER*80 CINAME,SECNAME,MAINNAME
C
      DIMENSION SIGMA(5),D(2),TEMP(2),PREDEF(2,*),TIME(2),
     1 COORDS(2,3)

      user coding to define SIGMA(1) -- SIGMA(5)

      RETURN
      END
```

## Variables to Be Defined

`SIGMA(1)` Gap electrical conductance, $\sigma_g$.

SIGMA(2) $\partial \sigma_g / \partial \theta_A$, derivative of the gap electrical conductance with respect to the temperature of point *A*. If the gap electrical conductance is not a function of $\theta_A$, SIGMA(2) = 0.0.

SIGMA(3) $\partial \sigma_g / \partial \theta_B$, derivative of the gap electrical conductance with respect to the temperature of point *B*. If the gap electrical conductance is not a function of $\theta_B$, SIGMA(3) = 0.0.

SIGMA(4) $\partial \sigma_g / \partial d$, derivative of the gap electrical conductance with respect to the clearance between the bodies. If the gap electrical conductance is not a function of gap clearance, SIGMA(4) = 0.0. This variable needs to be defined only for a fully coupled thermal-electrical-structural analysis.

SIGMA(5) $\partial \sigma_g / \partial p$, derivative of the gap electrical conductance with respect to the pressure between the bodies. If the gap electrical conductance is not a function of the pressure, SIGMA(5) = 0.0. This variable needs to be defined only for a fully coupled thermal-electrical-structural analysis.

## Variables Passed in for Information

| | |
|---|---|
| D(1) | Separation between the interface surfaces, *d*. |
| D(2) | Pressure transmitted across the surfaces, *p*. |
| TEMP(2) | Current temperature at points *A* and *B*. |
| PREDEF(2,NPRED) | An array containing pairs of values of all the user-specified field variables at the end of the current increment at points *A* and *B* (initial values at the beginning of the analysis and current values during the analysis). |
| TIME(1) | Value of step time at the end of the increment. |
| TIME(2) | Value of total time at the end of the increment. |
| CINAME | User-specified surface interaction name, left justified. |
| SECNAME | Secondary surface name. |
| MAINNAME | Main surface name. |
| COORDS | An array containing the current coordinates of points *A* and *B*. COORDS(1,K1) are the coordinates at point *A*, and COORDS(2,K1) are the coordinates at point *B*. |
| NODE | Secondary node number (point *A*). |
| NPRED | Number of predefined field variables. |
| KSTEP | Step number. |
| KINC | Increment number. |

# HARDINI: User subroutine to define initial equivalent plastic strain and initial backstress tensor.

**Products:** Abaqus/Standard

**References:**

- *Initial Conditions*
- *Classical Metal Plasticity*
- *Models for Metals Subjected to Cyclic Loading*
- *Extended Drucker-Prager Models*
- **INITIAL CONDITIONS*
- *HARDINI*

## Overview

User subroutine *HARDINI*:

- can be used only for material models that use metal plasticity or Drucker-Prager plasticity;
- can be used to provide initial equivalent plastic strain values as a function of element number, material point number, and/or material point coordinates for isotropic and combined hardening;
- enables you to specify initial conditions for the backstress tensor as a function of element number, material point number, and/or material point coordinates for kinematic and combined hardening;
- will be called to define the initial equivalent plastic strain and, if relevant, the initial backstresses at material points for which user-subroutine-defined initial hardening conditions are specified; and
- is intended for use when the initial equivalent plastic strain and/or backstress distributions are too complicated to specify directly as initial hardening conditions.

## Defining Backstress Components

The number of backstress components that must be defined depends on the element type for which this routine is being called. *Abaqus Elements Guide* describes the number of stress components for each element type; the number of backstress components is identical to the number of stress components. The order of the backstress components is the same as the order of the stress components. For example, in three-dimensional continuum elements six backstress components must be defined in the order $\alpha_{11}, \alpha_{22}, \alpha_{33}, \alpha_{12}, \alpha_{13}, \alpha_{23}$.

## User Subroutine Interface

```
      SUBROUTINE HARDINI(ALPHA,EQPS,COORDS,NTENS,NCRDS,NOEL,NPT,
     1 LAYER,KSPT,LFLAGS,NAMES)
C
      INCLUDE 'ABA_PARAM.INC'
C
      DIMENSION ALPHA(NTENS,*),COORDS(NCRDS),LFLAGS(2)
      CHARACTER*80 NAMES(2)


      user coding to define EQPS and, if relevant, ALPHA(NTENS)
```

```
      RETURN
      END
```

## Variables to Be Defined

The variables described below are element-type dependent.

| | |
|---|---|
| `EQPS` | Equivalent plastic strain. |
| `ALPHA(1,1)` | First backstress component of the first backstress. |
| `ALPHA(2,1)` | Second backstress component of the first backstress. |
| `ALPHA(3,1)` | Third backstress component of the first backstress. |
| `Etc.` | `NTENS` backstress component values should be defined for each backstress. |

## Variables Passed in for Information

| | |
|---|---|
| `COORDS` | An array containing the initial coordinates of this point. |
| `NTENS` | Number of backstress values to be defined. This number depends on the element type. |
| `NCRDS` | Number of coordinates. |
| `NOEL` | Element number. |
| `NPT` | Integration point number. |
| `LAYER` | Layer number (for composite shells and layered solids). |
| `KSPT` | Section point number within the current layer. |
| `LFLAGS(1)` | Rebar flag. If `LFLAGS(1)`=1, the current integration point is associated with element rebar. Otherwise, `LFLAGS(1)`=0. |
| `LFLAGS(2)` | Constituent flag. If `LFLAGS(2)`=1, the current integration point is associated with a constituent in the multiscale material. Otherwise, `LFLAGS(2)`=0. |
| `NAMES(1)` | Name of the rebar to which the current integration point belongs, which is the name given in the rebar or rebar layer definition (*Defining Reinforcement* or *Defining Rebar as an Element Property*). If no name was given in the rebar or rebar layer definition, this variable will be blank. This variable is relevant only when `LFLAGS(1)`=1. |
| `NAMES(2)` | Constituent name. This variable is relevant only when `LFLAGS(2)`=1. |

# HETVAL: User subroutine to provide internal heat generation in heat transfer analysis.

**Products:** Abaqus/Standard

**References:**

- *Uncoupled Heat Transfer Analysis*
- *Fully Coupled Thermal-Stress Analysis*
- *Fully Coupled Thermal-Electrical-Structural Analysis*
- **HEAT GENERATION*
- *HETVAL*

## Overview

User subroutine *HETVAL*:

- can be used to define a heat flux due to internal heat generation in a material, for example, as might be associated with phase changes occurring during the solution;
- allows for the dependence of internal heat generation on state variables (such as the fraction of material transformed) that themselves evolve with the solution and are stored as solution-dependent state variables;
- will be called at all material calculation points for which the material definition contains volumetric heat generation during heat transfer, coupled temperature-displacement, coupled thermal-electrical, or coupled thermal-electrical-structural analysis procedures;
- can be useful if it is necessary to include a kinetic theory for a phase change associated with latent heat release (for example, in the prediction of crystallization in a polymer casting process);
- can be used in conjunction with user subroutine *USDFLD* if it is desired to redefine any field variables before they are passed in; and

## User Subroutine Interface

```
      SUBROUTINE HETVAL(CMNAME,TEMP,TIME,DTIME,STATEV,FLUX,
     1 PREDEF,DPRED)
C
      INCLUDE 'ABA_PARAM.INC'
C
      CHARACTER*80 CMNAME
C
      DIMENSION TEMP(2),STATEV(*),PREDEF(*),TIME(2),FLUX(2),
     1 DPRED(*)


      user coding to define FLUX and update STATEV


      RETURN
      END
```

## Variables to Be Defined

`FLUX(1)` Heat flux, $r$ (thermal energy per time per volume: $JT^{-1}L^{-3}$), at this material calculation point.

`FLUX(2)` Rate of change of heat flux per temperature, $\partial r / \partial \theta$. This variable is nonzero only if the heat flux depends on temperature. It is needed to define a correct Jacobian matrix.

## Variables That Can Be Updated

`STATEV(*)` An array containing the user-defined solution-dependent state variables at this point.

In an uncoupled heat transfer analysis `STATEV` is passed into subroutine *HETVAL* as the values of these variables at the beginning of the increment. However, any updating of `STATEV` in user subroutine *USDFLD* will be included in the values passed into subroutine *HETVAL* since this routine is called before *HETVAL*. In addition, if *HETVAL* is being used in a fully coupled temperature-displacement or coupled thermal-electrical-structural analysis and user subroutine *UEXPAN*, user subroutine *CREEP*, user subroutine *UMAT*, or user subroutine *UTRS* is used to define the mechanical behavior of the material, those routines are called before this routine; therefore, any updating of `STATEV` done in *UEXPAN*, *CREEP*, *UMAT*, or *UTRS* will be included in the values passed into this routine.

In all cases `STATEV` should be passed back from user subroutine *HETVAL* containing the values of the state variables at the end of the current increment.

## Variables Passed in for Information

`CMNAME` User-specified material name, left justified.

`TEMP(1)` Current temperature.

`TEMP(2)` Temperature increment.

`TIME(1)` Step time at the end of the increment.

`TIME(2)` Total time at the end of the increment.

`DTIME` Time increment.

`PREDEF(*)` An array containing the values of all of the user-specified field variables at this point (initial values at the beginning of the analysis and current values during the analysis).

`DPRED(*)` Array of increments of predefined field variables.

# MPC: User subroutine to define multi-point constraints.

**Products:** Abaqus/Standard

**References:**

- *General Multi-Point Constraints*
- **MPC*

## Overview

User subroutine *MPC*:

- is called to impose a user-defined multi-point constraint and is intended for use when general constraints cannot be defined with one of the MPC types provided by Abaqus/Standard;
- can use only degrees of freedom that also exist on an element somewhere in the same model (methods for overcoming this limitation are discussed below);
- can generate linear as well as nonlinear constraints;
- allows definition of constraints involving finite rotations; and
- makes it possible to switch constraints on and off during an analysis.

## Coding Methods

There are two methods for coding this routine. By default, the subroutine operates in a *degree of freedom* mode. In this mode each call to this subroutine allows one individual degree of freedom to be constrained. Alternatively, you can specify that the subroutine operate in a *nodal* mode. In this mode each call to this subroutine allows a set of constraints to be imposed all at once; that is, on multiple degrees of freedom of the dependent node. In either case, the routine will be called for each user-subroutine-defined multi-point constraint or set of constraints. See *General Multi-Point Constraints* for details.

### Constraints That Involve Rotational Degrees of Freedom

In geometrically nonlinear analyses Abaqus/Standard compounds three-dimensional rotations based on a finite-rotation formulation and not by simple addition of the individual rotation components (see *Conventions* and *Rotation variables*). An incremental rotation involving one component usually results in changes in all three total rotation components. Therefore, any general constraint that involves large three-dimensional rotations should be implemented using the nodal mode of user subroutine *MPC*. The single degree of freedom version of user subroutine *MPC* can be used for geometrically linear problems, geometrically nonlinear problems with planar rotations, and constraints that do not involve rotation components.

## Constraints That Involve Degrees of Freedom That Are Not Active in the Model

The degrees of freedom involved in user MPCs must appear on some element or Abaqus/StandardMPC type in the model: user MPCs cannot use degrees of freedom that have not been introduced somewhere on an element. For example, a mesh that uses only continuum (solid) elements cannot have user MPCs that involve rotational degrees of freedom. The simplest way to overcome this limitation is to introduce an element somewhere in the model that uses the required degrees of freedom but does not affect the solution in any other way. Alternatively, if the degrees of freedom are rotations, they can be activated by the use of a library BEAM-type MPC somewhere in the model.

## Use with Nodal Coordinate Systems

When a local coordinate system (*Transformed Coordinate Systems*) and a user MPC are both used at a node, the variables at the node are first transformed before the MPC is imposed. Therefore, user-supplied MPCs must be based on the transformed degrees of freedom. The local-to-global transformation matrices $\mathbf{T}^I$ for the individual nodes:

$$\mathbf{u}^I_{global} = \mathbf{T}^I \cdot \mathbf{u}^I_{local}$$

are passed in for information.

## Degree of Freedom Version of User Subroutine MPC

This version of user subroutine *MPC* allows for one individual degree of freedom to be constrained and, thus, eliminated at a time. The constraint can be quite general and nonlinear of the form:

$$f\left(u^1, u^2, u^3, \ldots, u^N, \text{geometry}, \text{temperature}, \text{field variables}\right) = 0.$$

The first degree of freedom in this function, $u^1$, is the degree of freedom that will be eliminated to impose the constraint. $u^2$, $u^3$, etc. are any other degrees of freedom that are involved in the constraint. Since $u^1$ will be eliminated to impose the constraint, it cannot be used in subsequent kinematic constraints (multi-point constraints, linear equation constraints, or boundary conditions). Therefore, the user MPCs are imposed in the order given in the input.

You must provide, at all times, two items of information in user subroutine *MPC*:

1. A list of degree of freedom identifiers at the nodes that are listed in the corresponding multi-point constraint definition. This list corresponds to $u^1, u^2, u^3$, etc. in the constraint as given above.
2. An array of the derivatives

   $$A^1 = \frac{\partial f}{\partial u^1}, \quad A^2 = \frac{\partial f}{\partial u^2}, \quad A^3 = \frac{\partial f}{\partial u^3}, \quad \cdots$$

   of the constraint function with respect to the degrees of freedom involved. This array is needed for the redistribution of loads from degree of freedom $u^1$ to the other degrees of freedom and for the elimination of $u^1$ from the system matrices.

   In addition, you can provide the value of the dependent degree of freedom $u^1$ as a function of the independent degrees of freedom $u^2, u^3,$ etc. If this value is not provided, Abaqus/Standard will update $u^1$ based on the linearized form of the constraint equation as

   $$u^1 = -\frac{1}{A^1}\sum_{i=2}^{N} A^i u^i.$$

   Subroutine *MPC* should be coded and checked with care: if the array of derivatives $\partial f / \partial u^1$, etc. does not correspond to the definition of $u^1$ in terms of $u^2$, $u^3$, etc., forces will be transmitted improperly by the MPC and violations of equilibrium may occur. In addition, convergence of the solution may be adversely affected.

### User Subroutine Interface

```
      SUBROUTINE MPC(UE,A,JDOF,MDOF,N,JTYPE,X,U,UINIT,MAXDOF,
     * LMPC,KSTEP,KINC,TIME,NT,NF,TEMP,FIELD,LTRAN,TRAN)
C
```

```
      INCLUDE 'ABA_PARAM.INC'
C
      DIMENSION A(N),JDOF(N),X(6,N),U(MAXDOF,N),UINIT(MAXDOF,N),
     * TIME(2),TEMP(NT,N),FIELD(NF,NT,N),LTRAN(N),TRAN(3,3,N)


      user coding to define UE, A, JDOF, and, optionally, LMPC


      RETURN
      END
```

## Variables to Be Defined

`A(N)` An array containing the derivatives of the constraint function,

$$A(1) = \frac{\partial f}{\partial u^1}, \quad A(2) = \frac{\partial f}{\partial u^2}, \quad \cdots$$

The coding in the subroutine must define `N` entries in `A`, where `N` is defined below.

`JDOF(N)` An array containing the degree of freedom identifiers at the nodes that are involved in the constraint. For example, if $u^1$ is the $z$-displacement at a node, give `JDOF`(1) = 3; if $u^2$ is the $x$-displacement at a node, give `JDOF`(2) = 1. The coding in the subroutine must define `N` entries in `JDOF`, where `N` is defined below.

## Variables That Can Be Updated

`UE` This variable is passed in as the total value of the eliminated degree of freedom, $u^1$. This variable will either be zero or have the current value of $u^1$ based on the linearized constraint equation, depending at which stage of the iteration the user subroutine is called. If the constraint is linear and is used in a small-displacement analysis (nonlinear geometric effects are not considered) or in a perturbation analysis, this variable need not be defined: Abaqus/Standard will compute $u^1$ as

$$u^1 = -\frac{1}{A^1}\sum_{i=2}^{N} A^i u^i.$$

If the constraint is nonlinear, this variable should be updated to the value of $u^1$ at the end of the increment to satisfy the constraint exactly. If the return value is the same as the incoming value, Abaqus/Standard will update the eliminated degree of freedom based on the linearized form of the constraint equation. In this case the constraint is not likely to be satisfied exactly.

`LMPC` Set this variable to zero to avoid the application of the multi-point constraint. The MPC will be applied if the variable is not changed. This variable must be set to zero every time the subroutine is called if the user MPC is to remain deactivated. This MPC variable is useful for switching the MPC on and off during an analysis. However, the option should be used with care: switching off an MPC may cause a sudden disturbance in equilibrium, which can lead to convergence problems. If this variable is used to switch on an MPC during an analysis, the variable `UE` should be defined; otherwise, the constraint may not be satisfied properly.

## Variables Passed in for Information

`MDOF` Maximum number of active degrees of freedom per node involved in the MPC. For the degree of freedom mode of user subroutine *MPC*, `MDOF`= 1.

| | |
|---|---|
| N | Number of degrees of freedom that are involved in the constraint, defined as the number of nodes given in the corresponding multi-point constraint definition. If more than one degree of freedom at a node is involved in a constraint, the node must be repeated as needed or, alternatively, the nodal mode should be used. |
| JTYPE | Constraint identifier given for the corresponding multi-point constraint definition. |
| X(6,N) | An array containing the original coordinates of the nodes involved in the constraint. |
| U(MAXDOF,N) | An array containing the values of the degrees of freedom at the nodes involved in the constraint. These values will be either the values at the end of the previous iteration or the current values based on the linearized constraint equation, depending at which stage of the iteration the user subroutine is called. |
| UINIT(MAXDOF,N) | An array containing the values at the beginning of the current iteration of the degrees of freedom at the nodes involved in the constraint. This information is useful for decision-making purposes when you do not want the outcome of a decision to change during the course of an iteration. For example, there are constraints in which the degree of freedom to be eliminated changes during the course of the analysis, but it is necessary to prevent the choice of the dependent degree of freedom from changing during the course of an iteration. |
| MAXDOF | Maximum degree of freedom number at any node in the analysis. For example, for a coupled temperature-displacement analysis with continuum elements, MAXDOF will be equal to 11. |
| KSTEP | Step number. |
| KINC | Increment number within the step. |
| TIME(1) | Current value of step time. |
| TIME(2) | Current value of total time. |
| NT | Number of positions through a section where temperature or field variable values are stored at a node. In a mesh containing only continuum elements, NT=1. For a mesh containing shell or beam elements, NT is the largest of the values specified for the number of temperature points in the shell or beam section definition (or 2 for temperatures specified together with gradients for shells or two-dimensional beams, 3 for temperatures specified together with gradients for three-dimensional beams). |
| NF | Number of different predefined field variables requested for any node (including field variables defined as initial conditions). |
| TEMP(NT,N) | An array containing the temperatures at the nodes involved in the constraint. This array is not used for a heat transfer, coupled temperature-displacement, coupled thermal-electrical, or coupled thermal-electrical-structural analysis since the temperatures are degrees of freedom of the problem. |
| FIELD(NF,NT,N) | An array containing all field variables at the nodes involved in the constraint. |
| LTRAN(N) | An integer array indicating whether the nodes in the MPC are transformed. If LTRAN(I)=1, a transformation is applied to node I; if LTRAN(I)=0, no transformation is applied. |
| TRAN(3,3,N) | An array containing the local-to-global transformation matrices for the nodes used in the MPC. If no transformation is present at node I, TRAN(*,*,I) is the identity matrix. |

## Example: Nonlinear Single Degree of Freedom MPC

An example of a nonlinear single degree of freedom MPC is a geometrically nonlinear two-dimensional slider involving nodes *a*, *b*, and *c*. The constraint forces node *a* to be on the straight line connecting nodes *b* and *c* (see *Figure 1*).

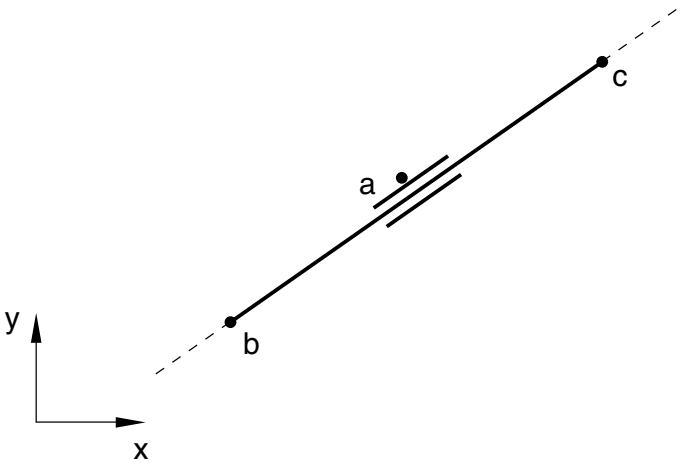



**Figure 1: Nonlinear MPC example: two-dimensional slider.**

The constraint equation can be written in the form

$$f\left(u^a, v^a, u^b, v^b, u^c, v^c\right) = \left(x^a - x^b\right)\left(y^c - y^b\right) - \left(y^a - y^b\right)\left(x^c - x^b\right) = 0,$$

where $\left(x^a, y^a\right)$, $\left(x^b, y^b\right)$, and $\left(x^c, y^c\right)$ are the current locations of *a*, *b*, and *c*. The derivatives are readily obtained as

$$\frac{\partial f}{u^a} = y^c - y^b\,, \quad \frac{\partial f}{u^b} = y^a - y^c\,, \quad \frac{\partial f}{u^c} = y^b - y^a,$$
$$\frac{\partial f}{v^a} = x^b - x^c\,, \quad \frac{\partial f}{v^b} = x^c - x^a\,, \quad \frac{\partial f}{v^c} = x^a - x^b.$$

Depending on the orientation of the segment $(b, c)$, we choose either $u^a$ or $v^a$ as the degree of freedom to be eliminated. If $\left|x^c - x^b\right| \geq \left|y^c - y^b\right|$, we choose $v^a$ as the dependent degree of freedom. If $\left|x^c - x^b\right| < \left|y^c - y^b\right|$, we choose $u^a$ as the dependent degree of freedom. Moreover, if points *b* and *c* are coincident, the constraint is not applied.

To prevent the choice of either $u^a$ or $v^a$ as the dependent degree of freedom from changing during the course of an iteration, the orientation of the segment $(b, c)$ is tested based on the geometry at the beginning of the iteration. The dependent degree of freedom is allowed to change from increment to increment.

Suppose the above multi-point constraint is defined as type 1, with nodes *a*, *a*, *b*, *b*, *c*, *c*. The user subroutine *MPC* could be coded as follows:

```
      SUBROUTINE MPC(UE,A,JDOF,MDOF,N,JTYPE,X,U,UINIT,MAXDOF,
     * LMPC,KSTEP,KINC,TIME,NT,NF,TEMP,FIELD,LTRAN,TRAN)
C
      INCLUDE 'ABA_PARAM.INC'
C
      DIMENSION A(N),JDOF(N),X(6,N),U(MAXDOF,N),UINIT(MAXDOF,N),
     * TIME(2),TEMP(NT,N),FIELD(NF,NT,N),LTRAN(N),TRAN(3,3,N)
      PARAMETER( PRECIS = 1.D-15 )
C
      IF (JTYPE .EQ. 1) THEN
        DYBC0 = X(2,5) + UINIT(2,5) - X(2,3) - UINIT(2,3)
        DXBC0 = X(1,3) + UINIT(1,3) - X(1,5) - UINIT(1,5)
        DYBC = X(2,5) + U(2,5) - X(2,3) - U(2,3)
        DXBC = X(1,3) + U(1,3) - X(1,5) - U(1,5)
        A(3) = X(2,1) + U(2,1) - X(2,5) - U(2,5)
        A(4) = X(1,5) + U(1,5) - X(1,1) - U(1,1)
```

```
        A(5) = X(2,3) + U(2,3) - X(2,1) - U(2,1)
        A(6) = X(1,1) + U(1,1) - X(1,3) - U(1,3)
        JDOF(3) = 1
        JDOF(4) = 2
        JDOF(5) = 1
        JDOF(6) = 2
        IF (ABS(DYBC0).LE.PRECIS .AND. ABS(DXBC0).LE.PRECIS) THEN
C
C         POINTS B AND C HAVE COLLAPSED. DO NOT APPLY CONSTRAINT.
C
          LMPC = 0
        ELSE IF (ABS(DXBC0) .LT. ABS(DYBC0)) THEN
C
C         MAKE U_A DEPENDENT DOF.
C
          JDOF(1) = 1
          JDOF(2) = 2
          A(1) = DYBC
          A(2) = DXBC
          UE = A(5)*A(2)/A(1) + X(1,3) + U(1,3) - X(1,1)
        ELSE
C
C         MAKE V_A DEPENDENT DOF.
C
          JDOF(1) = 2
          JDOF(2) = 1
          A(1) = DXBC
          A(2) = DYBC
          UE = -A(6)*A(2)/A(1) + X(2,3) + U(2,3) - X(2,1)
        END IF
      END IF
C
      RETURN
      END
```

## Nodal Version of User Subroutine MPC

The nodal version of user subroutine *MPC* allows for multiple degrees of freedom of a node to be eliminated simultaneously. The set of constraints can be quite general and nonlinear, of the form

$$f_i\left(u^1, u^2, u^3, \ldots, u^N,\ \text{geometry},\ \text{temperature},\ \text{field variables}\right) = 0 \quad i = 1, 2, \cdots, \text{NDEP}.$$

NDEP is the number of dependent degrees of freedom that are involved in the constraint and should have a value between 1 and MDOF, which is the number of active degrees of freedom per node in the analysis. $N$ is the number of nodes involved in the constraint. The scalar constraint functions $f_i$ can also be considered as a vector function $\boldsymbol{f}$, and the first set of degrees of freedom $u^1$ in the vector function $\boldsymbol{f}$ will be eliminated to impose the constraint. The sets $u^2$, $u^3$, etc. are the independent degrees of freedom at nodes 2, 3, etc. involved in the constraint. The set $u^1$ must be composed of NDEP degrees of freedom at the first node of the MPC definition. For example, if the dependent degrees of freedom are the $x$-displacement, the $z$-displacement, and the $y$-rotation at the first node, $u^1 = \left(u_x^1, u_z^1, \phi_y^1\right)$. $u^2$, $u^3$, etc. can be composed of any number of degrees of freedom, depending on which ones play a role in the constraint, and need not be of the same size; for example, $u^2 = \left(u_y^2\right)$ and $u^3 = \left(u_x^3, u_y^3, u_z^3\right)$.

The dependent node can also reappear as an independent node in the MPC. However, since the dependent degrees of freedom of this node will be eliminated, they cannot be used as independent degrees of freedom in this MPC. For example, if the rotations at node $a$ are constrained by the MPC, the displacements of node $a$ can still be used as independent degrees of freedom in the MPC, but the rotations themselves cannot. Similarly, the degrees of

freedom that will be eliminated to impose the constraint cannot be used in subsequent kinematic constraints (multi-point constraints, linear equation constraints, or boundary conditions). The MPCs are imposed in the order given in the input for this purpose.

The nodal version of user subroutine *MPC* was designed with the application of nonlinear constraints involving large three-dimensional rotations in mind. Due to the incremental nature of the solution procedure in Abaqus/Standard, a linearized set of constraints

$$\delta f_i = A_i^1 \cdot \delta u^1 + A_i^2 \cdot \delta u^2 + A_i^3 \cdot \delta u^3 + \ldots = 0 \qquad i = 1, 2, \cdots, \mathtt{NDEP},$$

where $A_i^1 = A_i^1\left(u^1, u^2, \ldots\right)$, $A_i^2 = A_i^2\left(u^1, u^2, \ldots\right)$, etc. is applied during each iteration. This linearized set of constraints is used for the calculation of equilibrium. For finite rotations the linearized equation is given in terms of the linearized rotations $\delta\theta^1$, $\delta\theta^2$, $\delta\theta^3$, … , yielding

$$\delta f_i = A_i^1 \cdot \delta\theta^1 + A_i^2 \cdot \delta\theta^2 + A_i^3 \cdot \delta\theta^3 + \ldots = 0 \qquad i = 1, 2, \cdots, \mathtt{NDEP}.$$

Since the linearized rotation field, $\delta\theta$, is not the variation of the total rotation vector, $\phi$ (see *Rotation variables*), you cannot obtain the linearized constraint equation by simply taking derivatives of the vector function, $f$, with respect to the rotational degrees of freedom involved. The formulation of the linearized constraint in $\delta\theta$ is equivalent to the formulation of a geometrically linear constraint in the deformed configuration and is generally easier to formulate than the constraint in terms of $\delta\phi$. For an exact formulation of the constraint, the dependent components of the total rotation vector $\phi^1$ must be defined exactly (see *Rotation variables*).

You must provide, at all times, two items of information in subroutine *MPC*:

1. A matrix of degree of freedom identifiers at the nodes that are listed in the corresponding multi-point constraint definition. The columns of this matrix correspond to $u^1, u^2, u^3$, etc. in the set of constraints as given above, where unused entries are padded with zeros. The number of nonzero entries in $u^1$ will implicitly determine the number of dependent degrees of freedom, NDEP.
2. The matrices representing the linearized constraint function with respect to the degrees of freedom involved. These matrices are needed for the redistribution of loads from degrees of freedom $u^1$ to the other degrees of freedom and for the elimination of $u^1$ from the system matrices. For constraints that do not involve three-dimensional rotations and constraints with planar rotations, these matrices can be readily obtained from the derivatives of the total constraint function with respect to the degrees of freedom involved:

   $$A_{ij}^1 = \frac{\partial f_i}{\partial u_j^1}, \quad A_{ij}^2 = \frac{\partial f_i}{\partial u_j^2}, \quad A_{ij}^3 = \frac{\partial f_i}{\partial u_j^3}, \quad \cdots$$

   For constraints that involve finite rotations, the matrices follow from the linearized form:

   $$A_{ij}^1 \delta\theta_j^1 + A_{ij}^2 \delta\theta_j^2 + A_{ij}^3 \delta\theta_j^3 + \ldots = 0.$$

   In addition, you can provide the values of the dependent degrees of freedom $u^1$, as a function of the independent degrees of freedom $u^2, u^3$, etc. For finite rotations, $\phi^1$ must be specified as a function of $\phi^2$, $\phi^2$, etc. If these values are not provided, Abaqus/Standard will update $u^1$ based on the linearized form of the constraint equations. Subroutine *MPC* should be coded and checked with care: if the matrices of derivatives $A_{ij}^1$, etc. do not correspond to the definition of $u^1$ in terms of $u^2$, $u^3$, etc., forces will be transmitted improperly by the MPC and violations of equilibrium may occur. In addition, convergence of the solution may be adversely affected.

## User Subroutine Interface

```
      SUBROUTINE MPC(UE,A,JDOF,MDOF,N,JTYPE,X,U,UINIT,MAXDOF,
     * LMPC,KSTEP,KINC,TIME,NT,NF,TEMP,FIELD,LTRAN,TRAN)
C
      INCLUDE 'ABA_PARAM.INC'
C
      DIMENSION UE(MDOF),A(MDOF,MDOF,N),JDOF(MDOF,N),X(6,N),
     * U(MAXDOF,N),UINIT(MAXDOF,N),TIME(2),TEMP(NT,N),
     * FIELD(NF,NT,N),LTRAN(N),TRAN(3,3,N)


      user coding to define JDOF, UE, A and, optionally, LMPC


      RETURN
      END
```

## Variables to Be Defined

`JDOF(MDOF,N)` Matrix of degrees of freedom identifiers at the nodes involved in the constraint. Before each call to *MPC*, Abaqus/Standard will initialize all of the entries of `JDOF` to zero. All active degrees of freedom for a given column (first index ranging from 1 to `MDOF`) must be defined starting at the top of the column with no zeros in between. A zero will mark the end of the list for that column. The number of nonzero entries in the first column will implicitly determine the number of dependent degrees of freedom (`NDEP`). For example, if the dependent degrees of freedom are the $z$-displacement, the $x$-rotation, and the $z$-rotation at the first node, `NDEP` $= 3$ and

$$\mathbf{JDOF}(1,1) = 3\,, \quad \mathbf{JDOF}(2,1) = 4\,, \quad \mathbf{JDOF}(3,1) = 6\,.$$

If the degrees of freedom at the third node involved in the MPC are the $x$-displacement and the $y$-rotation, define

$$\mathbf{JDOF}(1,3) = 1\,, \quad \mathbf{JDOF}(2,3) = 5.$$

`A(MDOF,MDOF,N)` Submatrices of coefficients of the linearized constraint function,

$$\mathbf{A}(\mathbf{I},\mathbf{J},\mathbf{1}) = A^1_{ij}\,, \quad \mathbf{A}(\mathbf{I},\mathbf{J},\mathbf{2}) = A^2_{ij}\,, \quad \cdots$$

Before each call to user subroutine *MPC*, Abaqus/Standard will initialize all of the entries of `A` to zero; therefore, only nonzero entries need to be defined. If the coding in the subroutine defines `NDEP` nonzero entries in the column `JDOF(J,1)`, it should define `NDEP` × `NDEP` entries in the submatrix `A(I,J,1)`. Since this submatrix will be inverted to impose the MPC, it must be nonsingular. A maximum of `NDEP` × `MDOF` entries can be defined for the remaining submatrices `A(I,J,K)`, `K = 2,`$\cdots$,`N`. The number of columns in each submatrix `A(I,J,K)` must correspond to the number of nonzero entries in the corresponding column of the matrix `JDOF(J,K)`.

## Variables That Can Be Updated

`UE(NDEP)` This array is passed in as the total value of the eliminated degrees of freedom, $\boldsymbol{u}^{\mathbf{1}}$. This array will either be zero or contain the current values of $\boldsymbol{u}^{\mathbf{1}}$ based on the linearized constraint equations, depending at which stage of the iteration the user subroutine is called. For small-displacement analysis or perturbation analysis this array need not be defined: Abaqus/Standard will compute $\boldsymbol{u}^{\mathbf{1}}$ as

$$u_i^1 = -\sum_{r=1}^{\mathtt{NDEP}} A_{ir}^{1\,-1} \sum_{s=2}^{\mathtt{N}} \sum_{t=1}^{\mathtt{MDOF}} A_{rt}^s \, u_{\mathtt{JDOF}(t,s)}^s \qquad i = 1, \cdots, \mathtt{NDEP}.$$

For large-displacement analysis this array can be updated to the value of $\boldsymbol{u}^1$ at the end of the increment to satisfy the constraint exactly. If the return values are the same as the incoming values, Abaqus/Standard will update the eliminated degrees of freedom based on the linearized form of the constraint equations. In this case the constraint is not likely to be satisfied exactly.

`LMPC` Set this variable to zero to avoid the application of the multi-point constraint. If the variable is not changed, the MPC will be applied. This variable must be set to zero every time the subroutine is called if the user MPC is to remain deactivated. This MPC variable is useful for switching the MPC on and off during an analysis. This option should be used with care: switching off an MPC may cause a sudden disturbance in equilibrium, which can lead to convergence problems.

## Variables Passed in for Information

`MDOF` Number of active degrees of freedom per node in the analysis. For example, for a coupled temperature-displacement analysis with two-dimensional continuum elements, the active degrees of freedom are 1, 2, and 11 and, hence, `MDOF` will be equal to 3.

`N` Number of nodes involved in the constraint. The value of `N` is defined as the number of nodes given in the corresponding multi-point constraint definition.

`JTYPE` Constraint identifier given for the corresponding multi-point constraint definition.

`X(6,N)` An array containing the original coordinates of the nodes involved in the constraint.

`U(MAXDOF,N)` An array containing the values of the degrees of freedom at the nodes involved in the constraint. These values will either be the values at the end of the previous iteration or the current values based on the linearized constraint equation, depending at which stage of the iteration the user subroutine is called.

`UINIT(MAXDOF,N)` An array containing the values at the beginning of the current iteration of the degrees of freedom at the nodes involved in the constraint. This information is useful for decision-making purposes when you do not want the outcome of a decision to change during the course of an iteration. For example, there are constraints in which the degrees of freedom to be eliminated change during the course of the analysis, but it is necessary to prevent the choice of the dependent degrees of freedom from changing during the course of an iteration.

`MAXDOF` Maximum degree of freedom number at any node in the analysis. For example, for a coupled temperature-displacement analysis with continuum elements, `MAXDOF` is equal to 11.

`KSTEP` Step number.

`KINC` Increment number within the step.

`TIME(1)` Current value of step time.

`TIME(2)` Current value of total time.

`NT` Number of positions through a section where temperature or field variable values are stored at a node. In a mesh containing only continuum elements, `NT`=1. For a mesh containing shell or beam elements, `NT` is the largest of the values specified for the number of temperature points in the shell or beam section definition (or 2 for temperatures specified together with gradients for shells or two-dimensional beams, 3 for temperatures specified together with gradients for three-dimensional beams).

`NF` Number of different predefined field variables requested for any node (including field variables defined as initial conditions).

| | |
|---|---|
| `TEMP(NT,N)` | An array containing the temperatures at the nodes involved in the constraint. This array is not used for a heat transfer, coupled temperature-displacement, coupled thermal-electrical, or coupled thermal-electrical-structural analysis since the temperatures are degrees of freedom of the problem. |
| `FIELD(NF,NT,N)` | An array containing all field variables at the nodes involved in the constraint. |
| `LTRAN(N)` | An integer array indicating whether the nodes in the MPC are transformed. If `LTRAN(I)=1`, a transformation is applied to node `I`; if `LTRAN(I)=0`, no transformation is applied. |
| `TRAN(3,3,N)` | An array containing the local-to-global transformation matrices for the nodes used in the MPC. If no transformation is present at node `I`, `TRAN(*,*,I)` is the identity matrix. |

## Example: Nonlinear MPC

As an example of a nonlinear MPC, consider the insertion of a rigid beam in a large-displacement, planar (two-dimensional) problem. This MPC is the two-dimensional version of library BEAM-type MPC. It can be implemented as a set of three different single degree of freedom MPCs or as a single nodal MPC. Here, the second method will be worked out because it is simpler and requires less data input.

Let *a* and *b* (see *Figure 2*) be the ends of the beam, with *a* the dependent end.

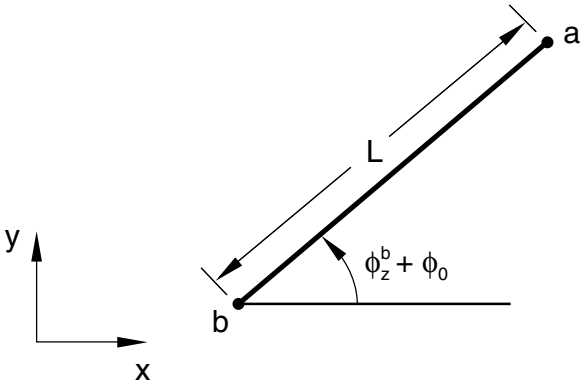



**Figure 2: Nonlinear MPC example: rigid beam.**

The rigid beam will then define both components of displacement and the rotation at *a* in terms of the displacements and rotation at end *b* according to the set of equations:

$$f_1\left(u^a, u^b\right) = x^a - x^b - L\cos\left(\phi_z^b + \phi_0\right) = 0,$$
$$f_2\left(u^a, u^b\right) = y^a - y^b - L\sin\left(\phi_z^b + \phi_0\right) = 0,$$
$$f_3\left(u^a, u^b\right) = \phi_z^a - \phi_z^b = 0,$$

where $u^a = \left(u_x^a, u_y^a, \phi_z^a\right)$ and $u^b = \left(u_x^b, u_y^b, \phi_z^b\right)$, $\left(x^a, y^a\right)$ and $\left(x^b, y^b\right)$ are the current locations of *a* and *b*, $\phi_z^a$ and $\phi_z^b$ are the rotations at *a* and *b* about the *z*-axis, *L* is the length of the link, and $\phi_0$ is the original orientation of the link.

In terms of the original positions $\left(X^a, Y^a\right)$ and $\left(X^b, Y^b\right)$ of *a* and *b*,

$$L = \sqrt{L_X^2 + L_Y^2}$$

and

$$\cos\phi_0 = L_X/L,$$
$$\sin\phi_0 = L_Y/L,$$

where $L_X = X^a - X^b$ and $L_Y = Y^a - Y^b$. Thus, the constraint equations can be expressed as

$$f_1\left(u^a, u^b\right) = u_x^a - u_x^b + L_X - L_X\cos\phi_z^b + L_Y\sin\phi_z^b = 0,$$
$$f_2\left(u^a, u^b\right) = u_y^a - u_y^b + L_Y - L_X\sin\phi_z^b - L_Y\cos\phi_z^b = 0,$$
$$f_3\left(u^a, u^b\right) = \phi_z^a - \phi_z^b = 0.$$

In light of the above formulation, the nontrivial portions of the matrices A and JDOF are

$$\mathbf{A}(1:3,1:3,1) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad \mathbf{A}(1:3,1:3,2) = \begin{pmatrix} -1 & 0 & L_X\sin\phi_z^b + L_Y\cos\phi_z^b \\ 0 & -1 & -L_X\cos\phi_z^b + L_Y\sin\phi_z^b \\ 0 & 0 & -1 \end{pmatrix}$$

and

$$\mathbf{JDOF}(1:3,1) = \mathbf{JDOF}(1:3,2) = \begin{pmatrix} 1 \\ 2 \\ 6 \end{pmatrix}.$$

Since degree of freedom 6 ($\phi_z$) appears in this constraint, there must be an element in the mesh that uses that degree of freedom—a B21 beam element, for example.

If the above multi-point constraint is defined as type 1 with nodes *a* and *b*, the user subroutine *MPC* could be coded as follows:

```
      SUBROUTINE MPC(UE,A,JDOF,MDOF,N,JTYPE,X,U,UINIT,MAXDOF,LMPC,
     * KSTEP,KINC,TIME,NT,NF,TEMP,FIELD,LTRAN,TRAN)
C
      INCLUDE 'ABA_PARAM.INC'
C
      DIMENSION UE(MDOF),A(MDOF,MDOF,N),JDOF(MDOF,N),X(6,N),
     * U(MAXDOF,N),UINIT(MAXDOF,N),TIME(2),TEMP(NT,N),
     * FIELD(NF,NT,N),LTRAN(N),TRAN(3,3,N)
C
      IF (JTYPE .EQ. 1) THEN
        COSFIB = COS(U(6,2))
        SINFIB = SIN(U(6,2))
        ALX = X(1,1) - X(1,2)
        ALY = X(2,1) - X(2,2)
C
        UE(1) = U(1,2) + ALX*(COSFIB-1.) - ALY*SINFIB
        UE(2) = U(2,2) + ALY*(COSFIB-1.) + ALX*SINFIB
        UE(3) = U(6,2)
C
        A(1,1,1) =  1.
        A(2,2,1) =  1.
        A(3,3,1) =  1.
        A(1,1,2) = -1.
        A(1,3,2) =  ALX*SINFIB + ALY*COSFIB
        A(2,2,2) = -1.
        A(2,3,2) = -ALX*COSFIB + ALY*SINFIB
        A(3,3,2) = -1.
C
        JDOF(1,1) = 1
```

```
          JDOF(2,1) = 2
          JDOF(3,1) = 6
          JDOF(1,2) = 1
          JDOF(2,2) = 2
          JDOF(3,2) = 6
        END IF
C
        RETURN
        END
```

## Example: Nonlinear MPC Involving Finite Rotations

As an example of a nonlinear MPC involving finite rotations, consider a two-dimensional constant velocity joint that might be part of a robotics application. Let *a*, *b*, *c* (see *Figure 3*) be the nodes making up the joint, with *a* the dependent node.

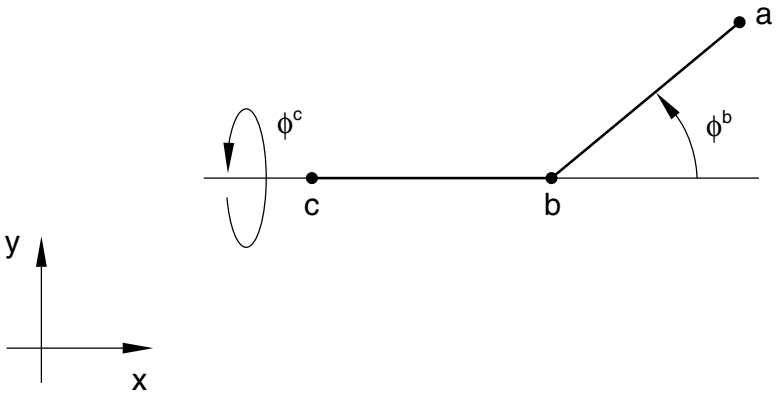



**Figure 3: Nonlinear MPC example: constant velocity joint.**

The joint is operated by prescribing an axial rotation $\boldsymbol{\phi}^c = \phi^c \mathbf{e}_x$ at *c* and an out-of-plane rotation $\boldsymbol{\phi}^b = \phi^b \mathbf{e}_z$ at *b*. The compounding of these two prescribed rotation fields will determine the total rotation at *a*. We can formally write this constraint as follows:

$$\boldsymbol{f}\left(\boldsymbol{\phi}^a, \boldsymbol{\phi}^b, \boldsymbol{\phi}^c\right) = \boldsymbol{\phi}^a - \boldsymbol{\phi}^b \circ \boldsymbol{\phi}^c = \mathbf{0},$$

where $\circ$ denotes the rotation product. The formulation of the linearized constraint can be readily achieved from geometrically linear considerations in the deformed state.

In geometrically linear problems compound rotations are obtained simply as the linear superposition of individual rotation vectors. Consider the geometry depicted in *Figure 3* and assume that the infinitesimal rotations $\boldsymbol{\delta\theta}^c = \delta\theta^c \mathbf{e}_x$ and $\boldsymbol{\delta\theta}^b = \delta\theta^b \mathbf{e}_z$ are applied at *c* and *b*, respectively. The rotation $\boldsymbol{\delta\theta}^a$ at *a* will simply be the sum of the vector $\boldsymbol{\delta\theta}^b$ to the vector $\boldsymbol{\delta\theta}^c$ rotated by an angle $\boldsymbol{\phi}^b$ about the *z*-axis. Thus, the linearized constraint can be written directly as

$$\begin{aligned}
\delta f_1\left(\boldsymbol{\phi}^a, \boldsymbol{\phi}^b, \boldsymbol{\phi}^c\right) &= \delta\theta_x^a - \cos\left(\phi^b\right)\delta\theta^c = 0, \\
\delta f_2\left(\boldsymbol{\phi}^a, \boldsymbol{\phi}^b, \boldsymbol{\phi}^c\right) &= \delta\theta_y^a - \sin\left(\phi^b\right)\delta\theta^c = 0, \\
\delta f_3\left(\boldsymbol{\phi}^a, \boldsymbol{\phi}^b, \boldsymbol{\phi}^c\right) &= \delta\theta_z^a - \delta\theta^b = 0.
\end{aligned}$$

In light of this formulation, the nontrivial portions of the matrices `JDOF` and `A` are

$$\mathbf{JDOF}(1:3,1) = \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix}, \quad \mathbf{JDOF}(1,2) = 6, \quad \mathbf{JDOF}(1,3) = 4,$$

and

$$\mathbf{A}(1:3,1:3,1) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad \mathbf{A}(1:3,1,2) = \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}, \quad \mathbf{A}(1:3,1,3) = \begin{pmatrix} -\cos(\phi^b) \\ -\sin(\phi^b) \\ 0 \end{pmatrix}.$$

Since degrees of freedom 4 ($\phi_x$), 5($\phi_y$), and 6($\phi_z$) appear in this constraint, there must be an element in the mesh that uses these degrees of freedom—a B31 beam element, for example. The *MPC* subroutine has been coded with just this information. In this case Abaqus/Standard updates the dependent rotation field, $\boldsymbol{\phi}^a$, based on the linearized constraint equations. Although the constraint is not satisfied exactly, good results are obtained as long as the rotation increments are kept small enough. A more rigorous derivation of the linearized constraint and the exact nonlinear recovery of the dependent degrees of freedom is presented in *Rotation variables*.

If the above multi-point constraint is defined as type 1 with nodes *a*, *b*, and *c*, user subroutine *MPC* could be coded as follows:

```
      SUBROUTINE MPC(UE,A,JDOF,MDOF,N,JTYPE,X,U,UINIT,MAXDOF,LMPC,
     * KSTEP,KINC,TIME,NT,NF,TEMP,FIELD,LTRAN,TRAN)
C
      INCLUDE 'ABA_PARAM.INC'
C
      DIMENSION UE(MDOF),A(MDOF,MDOF,N),JDOF(MDOF,N),X(6,N),
     * U(MAXDOF,N),UINIT(MAXDOF,N),TIME(2),TEMP(NT,N),
     * FIELD(NF,NT,N),LTRAN(N),TRAN(3,3,N)
C
      IF (JTYPE .EQ. 1) THEN
        A(1,1,1) =  1.
        A(2,2,1) =  1.
        A(3,3,1) =  1.
        A(3,1,2) = -1.
        A(1,1,3) = -COS(U(6,2))
        A(2,1,3) = -SIN(U(6,2))
C
        JDOF(1,1) = 4
        JDOF(2,1) = 5
        JDOF(3,1) = 6
        JDOF(1,2) = 6
        JDOF(1,3) = 4
      END IF
C
      RETURN

END
```

# ORIENT: User subroutine to provide an orientation for defining local material directions or local directions for kinematic coupling constraints or local rigid body directions for inertia relief.

**Products:** Abaqus/Standard

**References:**

- *Orientations*
- **ORIENTATION*
- *Eigenvalue analysis of a piezoelectric transducer*

## Overview

User subroutine *ORIENT*:

- will be called at the start of the analysis at each location (material point, special-purpose element, coupling node, or reference point for inertia relief) for which local directions are defined with a user-subroutine-defined orientation;
- is used to define the direction cosines of a local system of (material) directions with respect to the default basis directions (default basis directions are defined as the global directions for continuum elements and as the default surface directions for shell, membrane, and surface elements, as described in *Conventions*);
- can be used to define the direction cosines orienting the layer of reinforcing material in membrane, shell, or surface elements (see *Defining Reinforcement*);
- can be used to provide a local system for defining the direction of action of rotary inertia, spring, dashpot, flexible joint, and elastic-plastic joint elements;
- can be used with gasket elements to define the local in-plane directions for three-dimensional area and three-dimensional link elements that consider transverse shear and membrane deformations (see *Defining the Gasket Behavior Directly Using a Gasket Behavior Model*);
- can be used to define a local system in which coupling constraints are applied (see *Coupling Constraints* and *Kinematic Coupling Constraints*);
- can be used to define a local system at the reference point for the rigid body directions in which inertia relief loads are applied for the entire model (see *Inertia Relief*);
- will ignore rotation angles defined for layers of composite solids (see *Solid (Continuum) Elements*) but will take into account rotation angles defined for layers of composite shells (see *Using a Shell Section Integrated during the Analysis to Define the Section Behavior* and *Using a General Shell Section to Define the Section Behavior*); and
- ignores any data specified for the associated orientation definition outside the user subroutine.

The local directions defined by user subroutine *ORIENT* must be specified relative to the default basis directions.

## User Subroutine Interface

```
      SUBROUTINE ORIENT(T,NOEL,NPT,LAYER,KSPT,COORDS,BASIS,
     1 ORNAME,NNODES,CNODES,JNNUM)
C
      INCLUDE 'ABA_PARAM.INC'
C
      CHARACTER*80 ORNAME
C
```

```
      DIMENSION T(3,3),COORDS(3),BASIS(3,3),CNODES(3,NNODES)
      DIMENSION JNNUM(NNODES)

      user coding to define T

      RETURN
      END
```

## Variables to Be Defined

`T` An array containing the direction cosines of the preferred orientation in terms of the default basis directions. `T(1,1)`, `T(2,1)`, `T(3,1)` give the (1, 2, 3) components of the first direction; `T(1,2)`, `T(2,2)`, `T(3,2)` give the second direction; etc. For shell and membrane elements only the first and second directions are used. The directions do not have to be normalized. If the second direction is not orthogonal to the first direction, Abaqus/Standard will orthogonalize and normalize the second direction with respect to the first. The third direction is then determined by taking the cross product of the first and second directions. For planar elements the first two directions must lie in the plane of the element.

For use with coupling constraints (*Coupling Constraints*), the local basis directions are used as the local constraint directions for application of the kinematic constraint.

For use with inertia relief loads, the local basis directions are used as the rigid body direction vectors for computing the loads.

## Variables Passed in for Information

`NOEL` Element number. This value is zero when the subroutine is called for use with coupling constraints or inertia relief loads.

`NPT` Integration point number. This variable is set only for relevant uses.

`LAYER` Layer number (for composite shells and layered solids). This variable is set only when relevant. It is equal to zero when it is irrelevant, such as in a regular solid element or in a shell element when transverse shear stiffness calculations are performed.

`KSPT` Section point number within the current layer. This variable is set only when relevant. It is equal to zero when it is irrelevant, such as in a regular solid element or in a shell element when transverse shear stiffness calculations are performed.

`COORDS` An array containing the initial coordinates of this point. This array contains the coordinates of the reference point for inertia relief loads.

`BASIS` An array containing the direction cosines of the normal material basis directions in terms of the global coordinates in the original configuration. `BASIS(1,1)`, `BASIS(2,1)`, `BASIS(3,1)` give the 1-direction, etc. This is useful only in shells or membranes since in all other cases the basis is the global coordinate system.

`ORNAME` User-specified orientation name, left justified, with one exception. When an overall section orientation is specified for a composite solid or shell section and the individual layer orientations are specified by an orientation angle, Abaqus defines an internal orientation name to represent the actual orientation of the layer. To avoid internal names, provide an orientation name rather than an orientation angle as part of the layer definition for each individual layer of a composite section.

`NNODES` Number of element nodes. This value is two when the subroutine is called for use with a kinematic coupling definition, where the two nodes are the reference and current coupling node. When used with a distributing coupling definition, this number is equal to the number of coupling nodes plus one for the reference node. It is one when used with inertia relief loads since the local basis is defined at the reference point.

CNODES An array containing the original coordinates of the nodes. When used with a kinematic coupling definition, the first entry defines the reference node coordinates, and the second entry defines the coupling node coordinates. When used with a distributing coupling definition, the first entry defines the reference node coordinates, and the subsequent entries define the coupling node coordinates in the order defined by the JNNUM array. When used with inertia relief loads, this array is not used. For all other uses the entry order follows that of the element definition node ordering.

JNNUM An array containing the NNODES node numbers. When used with a kinematic coupling definition, the first entry is the reference node number, and the second entry is the node number for the current coupling node. When used with a distributing coupling definition, the first entry is the reference node number followed by the node numbers of all coupling nodes. When used with inertia relief loads, this array is not used. For all other uses the entry order follows that of the element definition node ordering.

# RSURFU: User subroutine to define a rigid surface.

**Products:** Abaqus/Standard

**References:**

- *Analytical Rigid Surface Definition*
- **SURFACE*
- **RIGID BODY*
- *RSURFU*

## Overview

User subroutine *RSURFU*:

- is used to define the surface of a rigid body for use in contact pairs;
- can be used to define a complex rigid surface if the various capabilities provided for defining a surface in Abaqus (see *Analytical Rigid Surface Definition*) are too restrictive;
- will be called at points on the secondary surface of a contact pair or, if contact elements are used, at each integration point of each contact element with which the rigid surface is associated; and
- requires the definition of the closest point on the rigid surface, the normal and tangent directions, and the surface curvature.

## Overpenetration Constraint

This routine must determine if a point on the secondary surface has penetrated the rigid surface and define the local surface geometry. If the deforming and rigid surfaces are in contact at this point, Abaqus/Standard will impose a constraint at the point to prevent overpenetration. The local surface geometry must be defined to provide the necessary orientation for the constraint equations and friction directions and to allow Abaqus/Standard to compute the rate of change of these equations as the point moves around on the surface—the "tangent stiffness matrix" for the surface in the Newton algorithm. For the purpose of these calculations, it is best to define a smooth surface. If the surface is defined in a discontinuous manner, convergence may be adversely affected.

## Calculations to Be Performed

Each time *RSURFU* is called, Abaqus/Standard gives the current position of point $A$ on the surface of the deforming structure, $\mathbf{x}_A$; the current position of the rigid body reference point, $\mathbf{x}_C$; the total displacements of both of these points, $\mathbf{u}_A$ and $\mathbf{u}_C$; and the total rotation of the rigid body reference point, $\boldsymbol{\phi}_C$.

The routine should perform the following calculations:

1. A point, $A$ , must be found on the rigid surface at which the normal to the surface passes through $\mathbf{x}_A$. If there is not a unique point $A$ , the routine must choose the most suitable point (usually the closest $A$ to $A$). The routine must pass back the coordinates of $A$ to Abaqus/Standard. For the surface-to-surface contact formulation, the secondary normal, not the main normal, should be used.
2. *RSURFU* must define the distance, $h$, by which $A$ has penetrated the surface below $A$ . A negative value of $h$ means that $A$ is outside the surface of the rigid body.

3. If the surfaces are in contact, which may sometimes be the case even if $h$ is negative, *RSURFU* must define the local surface geometry.

## Defining the Local Surface Geometry

There are two scenarios under which it is mandatory that the routine define the local surface geometry: if $A$ has penetrated the surface—$h > 0$ if the surface behavior is truly rigid, or $h$ is greater than the maximum overclosure value specified for modified surface behavior using either contact controls (see *Generally Applicable Contact Controls in Abaqus/Standard*) or a modified pressure-overclosure relationship (see *Contact Pressure-Overclosure Relationships*)—and if $A$ was in contact at the beginning of the increment, in which case the flag LCLOSE=1 (see the variable list for the definition of LCLOSE). The variable LCLOSE is not relevant for the surface-to-surface contact formulation and is always passed in as 0. The routine can be coded so that local surface geometry definitions are always provided regardless of the scenario.

The local surface geometry is specified by two orthogonal tangents to the rigid surface at $A$ , as well as the rates of change of the outward pointing normal at $A$ , $\mathbf{n}$, with respect to local surface coordinates that are distance measuring along the tangents, $S^1$ and $S^2$ (see *Figure 1*).

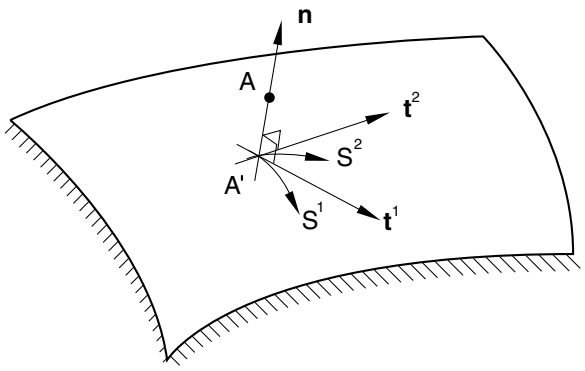



**Figure 1: Local geometry on a rigid surface.**

The tangents to the surface at $A$ must be defined so that their positive, right-handed cross product is the outward normal to the surface. For two-dimensional cases Abaqus/Standard assumes that the second tangent is (0, 0, −1), so that when you give the direction cosines of the first tangent as ($t_1$, $t_2$, 0), the outward normal will be ($-t_2$, $t_1$, 0). The rates of change of the normal with respect to $S^1$ and $S^2$ are required to define the local curvature of the surface.

## User Subroutine Interface

```
      SUBROUTINE RSURFU(H,P,TGT,DNDS,X,TIME,U,CINAME,SECNAME,
     1 MAINNAME,NOEL,NODE,LCLOSE)
C
      INCLUDE 'ABA_PARAM.INC'
C
      CHARACTER*80 CINAME,SECNAME,MAINNAME
C
      DIMENSION P(3),TGT(3,2),DNDS(3,2),X(3,3),TIME(2),U(6,2)


      user coding to define H, P, TGT, and DNDS


      RETURN
      END
```

## Variables to Be Defined

`H` Penetration of the point $A$ on the deforming structure into the surface of the rigid body, measured down the outward normal to the rigid surface. A negative value of `H` indicates that $A$ is outside the rigid surface. Even for a completely rigid surface, $A$ may appear to penetrate the surface during the iterations because the kinematic constraints are not fully satisfied until an increment converges.

`P(3)` Position of the point $A$ on the surface of the rigid body closest to point $A$ on the surface of the deforming structure.

`TGT(3,2)` Direction cosines of the two unit tangents to the surface, $\mathbf{t}^1$ and $\mathbf{t}^2$, at point $A$. For two-dimensional cases only the first two components of $\mathbf{t}^1$ need be given since in this case Abaqus/Standard assumes that $\mathbf{t}^2$ is (0, 0, −1).

`DNDS(3,2)` Rates of change of the surface normal, $\mathbf{n}$, at $A$, with respect to distance measuring coordinates, $S^1$ and $S^2$, along $\mathbf{t}^1$ and $\mathbf{t}^2$. For two-dimensional cases only the first two entries in the first column of `DNDS` ($\partial n_1/\partial S^1$, $\partial n_2/\partial S^1$) are required. The array `DNDS` is not required to be assigned for the surface-to-surface contact formulation.

## Variables Passed in for Information

`X(K1,1)` Current coordinates of point $A$ on the surface of the deforming structure.

`X(K1,2)` Current coordinates of the rigid body reference point.

`X(K1,3)` Unit normal vector for point $A$; relevant only for the surface-to-surface contact formulation.

`TIME(1)` Value of step time at the end of the increment.

`TIME(2)` Value of total time at the end of the increment.

`U(K1,1)` Total displacement of point $A$ on the surface of the deforming structure.

`U(K1,2)` Total displacement and rotation of the rigid body reference point; $k_1 = 1, 2, 3$ are the displacement components, $k_1 = 4, 5, 6$ are the rotation components. For two-dimensional cases the only nonzero rotation component is $k_1 = 6$: `U(4,2)` and `U(5,2)` are both zero.

`CINAME` User-specified surface interaction name, left justified. For user-defined contact elements, it is either the element set name given for the interface definition or the optional name assigned to the interface definition.

`SECNAME` Secondary surface name. Passed in as blank if *RSURFU* is called for contact elements.

`MAINNAME` Main surface name. Passed in as blank if *RSURFU* is called for contact elements.

`NOEL` Element label for contact elements. Passed in as zero if *RSURFU* is called for a contact pair.

`NODE` Node number for point $A$. For the surface-to-surface contact formulation, this quantity is passed in as 0.

`LCLOSE` Flag indicating contact status at the beginning of the increment. `LCLOSE`=1 indicates that $A$ is in contact (closed) at the beginning of the increment. `LCLOSE`=0 indicates that $A$ is not in contact (open) at the beginning of the increment. If `LCLOSE`=1, `P`, `TGT`, and `DNDS` must be defined even if $A$ opens during this increment. `LCLOSE` is not used for the surface-to-surface contact formulation and is passed in as 0.

## Example: Rigid Punch

The input files for the following examples can be found in *RSURFU*. The following discussion pertains only to the node-to-surface contact formulation.

Consider the punch shown in *Figure 2*.

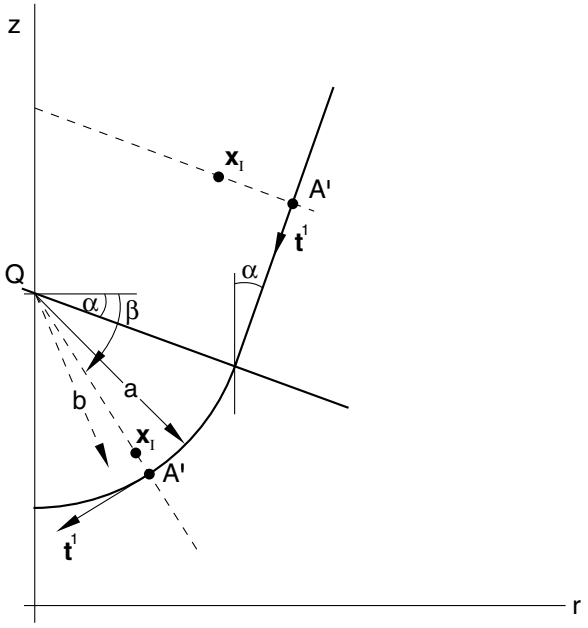



**Figure 2: Cross-section of a rigid punch.**

It consists of a spherical head of radius $a$, smoothly merging into a conical section with cone angle $\alpha$. The center of the sphere lies on the $z$-axis at $Q$. We assume that the punch is being driven down the $z$-axis by a prescribed displacement at the rigid body reference node defined as a boundary condition. (This same surface could be defined directly as a three-dimensional surface of revolution, as described in *Analytical Rigid Surface Definition*. We define it here in `RSURFU` as an illustration.)

A point (secondary node) on the surface of the deforming body will be associated with the spherical head or with the conical part of the punch, depending on whether it lies above or below the cone that passes through $Q$ and the circle of intersection of the sphere and cone. Thus, define

$$r = \sqrt{x_1^2 + x_2^2}\,, \quad z = x_3$$

in the three-dimensional case or

$$r = x_1\,, \quad z = x_2$$

in the axisymmetric case. Then, if $r \tan\alpha < z_Q - z$, the point is associated with the spherical surface. Otherwise, it is associated with the cone (both cases are indicated in *Figure 2*).

Consider first the axisymmetric case. Then, for $r \tan\alpha < z_Q - z$ (the sphere) the overclosure is

$$h = a - b,$$

where

$$b = \sqrt{r^2 + (z - z_Q)^2}.$$

The position of the point $A$ on the rigid surface is ($a\cos\beta$, $z_Q - a\sin\beta$, $0$), where

$$\cos\beta = r/b\,, \quad \text{and} \quad \sin\beta = (z_Q - z)/b.$$

The tangent to the rigid surface at $A$ is $\mathbf{t}^1 = (-\sin\beta, -\cos\beta, 0)$. The positive direction for $\mathbf{t}^1$ must be chosen so that the normal satisfies the right-hand rule with respect to $\mathbf{t}^1$ and $\mathbf{t}^2$ and points out of the rigid body. In addition, $dS^1 = a\,d\beta$, so that

$$\frac{\partial \mathbf{n}}{\partial S^1} = \left(-\frac{1}{a}\sin\beta, -\frac{1}{a}\cos\beta, 0\right).$$

For $r\tan\alpha > z_Q - z$ (the conical surface) the clearance is

$$h = -r\cos\alpha + (z - z_Q)\sin\alpha + a,$$

and the position of the point $A$ on the rigid surface is $(r + h\cos\alpha, z - h\sin\alpha)$. The surface tangent is $\mathbf{t}^1 = (-\sin\alpha, -\cos\alpha, 0)$ and there is no change in $\mathbf{n}$ with position, so that

$$\frac{\partial \mathbf{n}}{\partial S^1} = (0, 0, 0).$$

The routine can then be coded as follows:

```
      SUBROUTINE RSURFU(H,P,TGT,DNDS,X,TIME,U,CINAME,SECNAME,
     1   MAINNAME,NOEL,NODE,LCLOSE)
C
      INCLUDE 'ABA_PARAM.INC'
C
      CHARACTER*80 CINAME,SECNAME,MAINNAME
      DIMENSION P(3),TGT(3,2),DNDS(3,2),X(3,2),TIME(2),U(6,2)
C
C     DEFINE THE FOLLOWING QUANTITIES:
C     A = RADIUS 'A' OF THE SPHERICAL HEAD
C     SINA = SINE (CONE ANGLE ALPHA)
C     COSA = COSINE (CONE ANGLE ALPHA)
C     Z0 = ORIGINAL 'Z' COORDINATE OF POINT 'Q'
C
      A=5.0
      SINA=0.5
      COSA=0.86602
      Z0=6.0
      ZQ=Z0 + U(2,2)
C
C     TEST FOR SEGMENT
C
      IF(X(1,1)*SINA/COSA.LT.ZQ-X(2,1))THEN
C
C     SPHERE
C
         B=SQRT(X(1,1)**2 + (X(2,1)-ZQ)**2)
         H=A-B
         COSB=X(1,1)/B
         SINB=(ZQ-X(2,1))/B
         P(1)=A*COSB
         P(2)=ZQ-A*SINB
         TGT(1,1)=-SINB
         TGT(2,1)=-COSB
         DNDS(1,1)=-SINB/A
         DNDS(2,1)=-COSB/A
      ELSE
```

```
C     CONE
         H=-X(1,1)*COSA+(X(2,1)-ZQ)*SINA+A
         P(1)=X(1,1) + H*COSA
         P(2)=X(2,1)- H*SINA
         TGT(1,1)=-SINA
         TGT(2,1)=-COSA
         DNDS(1,1)=0.
         DNDS(2,1)=0.
      END IF
      RETURN
      END
```

The above case can be directly extended to three dimensions. For this purpose we assume that the radial axis, $r$, is in the global ($x$–$y$) plane, so that

$$r = \sqrt{x_1^2 + x_2^2}\,, \quad z = x_3.$$

For $r\tan\alpha < z_Q - z$ (the sphere), the overclosure is $h = a - b$, where again

$$b = \sqrt{r^2 + (z - z_Q)^2}.$$

The point $A$ on the rigid surface is ($a\cos\beta\cos\gamma$, $a\cos\beta\sin\gamma$, $z_Q - a\sin\beta$), where

$$\cos\gamma = \tfrac{x_1}{r}\,, \quad \sin\gamma = \tfrac{x_2}{r}.$$

For $r = 0$, $\gamma$ is not defined uniquely; in that case we arbitrarily choose $\gamma = 0$. We now need two tangents to the surface. The tangent $\mathbf{t}^1$ used in the axisymmetric case is now

$$\mathbf{t}^1 = (-\sin\beta\cos\gamma, -\sin\beta\sin\gamma, -\cos\beta)$$

and the orthogonal tangent is

$$\mathbf{t}^2 = (-\sin\gamma, \cos\gamma, 0)\,.$$

Again, the positive directions of $\mathbf{t}^1$and $\mathbf{t}^2$ are chosen so that $\mathbf{t}^1 \times \mathbf{t}^2$ defines an outward normal to the surface. The distance measures on the surface are

$$dS^1 = a\,d\beta\,, \quad dS^2 = a\cos\beta\,d\gamma$$

so that

$$\frac{\partial \mathbf{n}}{\partial S^1} = \left(-\frac{1}{a}\sin\beta\cos\gamma, -\frac{1}{a}\sin\beta\sin\gamma, -\frac{1}{a}\cos\beta\right),$$
$$\frac{\partial \mathbf{n}}{\partial S^2} = \left(-\frac{1}{a}\sin\gamma, \frac{1}{a}\cos\gamma, 0\right).$$

For the conical surface ($r\tan\alpha \geq z_Q - z$ ), the surface separation is

$$h = -r\cos\alpha + (z - z_Q)\,\sin\alpha + a.$$

The point $A$ on the rigid surface is $((r + h\cos\alpha)\,\cos\gamma, (r + h\cos\alpha)\,\sin\gamma, z - h\sin\alpha)$; and the surface tangents are

$$\mathbf{t}^1 = (-\sin\alpha\cos\gamma, -\sin\alpha\sin\gamma, -\cos\alpha)$$
$$\mathbf{t}^2 = (-\sin\gamma, \cos\gamma, 0)\,.$$

There is no change of $\mathbf{n}$ with respect to $S^1$, and, in this case $dS^2 = c\,d\gamma$, where $c = r + h\cos\alpha$ so that

$$\frac{\partial \mathbf{n}}{\partial S^2} = \left(-\frac{1}{c}\cos\alpha\sin\gamma, +\frac{1}{c}\cos\alpha\cos\gamma, 0\right).$$

The routine can then be coded as follows:

```
      SUBROUTINE RSURFU(H,P,TGT,DNDS,X,TIME,U,CINAME,SECNAME,
     1     MAINNAME,NOEL,NODE,LCLOSE)
C
      INCLUDE 'ABA_PARAM.INC'
C
      CHARACTER*80 CINAME,SECNAME,MAINNAME
      DIMENSION P(3), TGT(3,2),DNDS(3,2), X(3,2), TIME(2), U(6,2)
C
C     DEFINE THE FOLLOWING QUANTITIES:
C     A = RADIUS 'A' OF THE SPHERICAL HEAD
C     SINA = SINE (CONE ANGLE ALPHA)
C     COSA = COSINE (CONE ANGLE ALPHA)
C     Z0 = ORIGINAL 'Z' COORDINATE OF POINT 'Q'
C
      A=5.0
      SINA=0.5
      COSA=0.86603
      Z0=5.0
      ZQ= Z0 + U(3,2)
C
C     TEST FOR SEGMENT
C
      R  = SQRT(X(1,1)*X(1,1)+X(2,1)*X(2,1))
      IF(R .GT. 0.0) THEN
         COSG = X(1,1)/R
         SING = X(2,1)/R
      ELSE
         COSG = 1.0
         SING = 0.0
      END IF
      IF(R*SINA/COSA .LT. ZQ -X(3,1)) THEN
C
C     SPHERE
C
         B=SQRT(R*R+(X(3,1)-ZQ)**2)
         H=A-B
         COSB=R/B
         SINB=(ZQ-X(3,1))/B
         P(1)=A*COSB*COSG
         P(2)=A*COSB*SING
         P(3)=ZQ-A*SINB
         TGT(1,1)=-SINB*COSG
         TGT(2,1)=-SINB*SING
         TGT(3,1)=-COSB
         TGT(1,2)=-SING
         TGT(2,2)=COSG
         TGT(3,2)=0.0
         DNDS(1,1)=-SINB*COSG/A
         DNDS(2,1)=-SINB*SING/A
         DNDS(3,1)=-COSB/A
         DNDS(1,2)=-SING/A
         DNDS(2,2)=COSG/A
         DNDS(3,2)=0.0
      ELSE
```

```
C
C     CONE
C
         H=-R*COSA+(X(3,1)-ZQ)*SINA+A
         P(1)=(R+H*COSA)*COSG
         P(2)=(R+H*COSA)*SING
         P(3)=X(3,1)-H*SINA
         TGT(1,1)=-SINA*COSG
         TGT(2,1)=-SINA*SING
         TGT(3,1)=-COSA
         TGT(1,2)=-SING
         TGT(2,2)=COSG
         TGT(3,2)=0.0
         DNDS(1,1)=0.0
         DNDS(2,1)=0.0
         DNDS(3,1)=0.0
         C=R+H*COSA
         DNDS(1,2)=-COSA*SING/C
         DNDS(2,2)=COSA*COSG/C
         DNDS(3,2)=0.0
      END IF
C
      RETURN
      END
```

# SDVINI: User subroutine to define initial solution-dependent state variable fields.

**Products:** Abaqus/Standard

**References:**

- *About User Subroutines and Utilities*
- **INITIAL CONDITIONS*
- *SDVINI*

## Overview

User subroutine *SDVINI*:

- will be called for user-subroutine-defined initial solution-dependent state variable fields at particular material points, shell section points, contact secondary nodes, or for user elements (see *Initial Conditions*);
- can be used to initialize solution-dependent state variables allocated as described in *Allocating Space for Solution-Dependent State Variables*; and
- returns a value of zero for any solution-dependent state variables that have no defined initial condition.

## Use of Solution-Dependent State Variables in Other User Subroutines

Solution-dependent state variables initialized in *SDVINI* can be used and updated in the following user subroutines:

- *CREEP*
- *FRIC*
- *HETVAL*
- *UEL*
- *UEXPAN*
- *UGENS*
- *UHARD*
- *UMAT*
- *UMATHT*
- *USDFLD*
- *UTRS*

The solution-dependent state variables are passed into these routines in the order in which they are entered in *SDVINI*.

## User Subroutine Interface

```
      SUBROUTINE SDVINI(STATEV,COORDS,NSTATV,NCRDS,NOEL,NPT,
     1 LAYER,KSPT)
C
      INCLUDE 'ABA_PARAM.INC'
```

```
C
      DIMENSION STATEV(NSTATV),COORDS(NCRDS)

      user coding to define STATEV(NSTATV)

      RETURN
      END
```

## Variables to Be Defined

| | |
|---|---|
| `STATEV(1)` | First solution-dependent state variable. |
| `STATEV(2)` | Second solution-dependent state variable. |
| `STATEV(3)` | Third solution-dependent state variable. |
| `Etc.` | Only `NSTATV` solution-dependent state variable values should be defined. |

## Variables Passed in for Information

| | |
|---|---|
| `COORDS` | An array containing the initial coordinates of this point. Coordinates are not available for user elements. |
| `NSTATV` | User-defined number of solution-dependent state variables (see *Allocating Space for Solution-Dependent State Variables*). |
| `NCRDS` | Number of coordinates. This value is zero for user elements. |
| `NOEL` | Element number. |
| `NPT` | Integration point number in the element (not relevant for user elements). |
| `LAYER` | Layer number (for composite shells and layered solids). |
| `KSPT` | Section point number within the current layer or section. Section point 1 is used for all pure heat transfer, coupled temperature-displacement, and coupled thermal-electrical-structural analyses. |

# SIGINI: User subroutine to define an initial stress field.

**Products:** Abaqus/Standard

**References:**

- *Initial Conditions*
- **INITIAL CONDITIONS*

## Overview

User subroutine *SIGINI*:

- will be called for user-subroutine-defined initial stress fields at particular material points (these are the effective stress values for soils analysis);
- is called at the start of the analysis for each applicable material calculation point in the model; and
- can be used to define all active initial stress components at material points as functions of coordinates, element number, integration point number, etc.

## Stress Components

The number of stress components that must be defined depends on the element type for which this call is being made. *About the Element Library* describes the element stresses. The order in which the components must be defined is the same as in the element definition. For example, in three-dimensional continuum elements six stress components must be defined in the order $\sigma_{11}, \sigma_{22}, \sigma_{33}, \sigma_{12}, \sigma_{13}, \sigma_{23}$.

## Initial Stress Field Equilibrium

You should ensure that the initial stress field is in equilibrium with the applied forces and distributed loads by using a static step or a geostatic step to check the equilibrium of the initial stress field before starting the response history. See *Geostatic Stress State* for a discussion of defining initial equilibrium conditions for problems that include pore fluid pressure.

## Stress in Multiscale Material

For multiscale material (*Mean-Field Homogenization*), you can define the stress components at either the composite level or at the constituent level. When defining the stresses at the constituent level, you should define stresses at all constituents to ensure that the stresses at the composite level are computed correctly through homogenization.

## User Subroutine Interface

```
      SUBROUTINE SIGINI(SIGMA,COORDS,NTENS,NCRDS,NOEL,NPT,LAYER,
     1 KSPT,LFLAGS,NAMES)
C
      INCLUDE 'ABA_PARAM.INC'
C
      DIMENSION SIGMA(NTENS),COORDS(NCRDS),LFLAGS(2)
      CHARACTER NAMES(3)*80
```

```
      user coding to define SIGMA(NTENS)


      RETURN
      END
```

## Variables to Be Defined

| | |
|---|---|
| SIGMA(1) | First stress component. |
| SIGMA(2) | Second stress component. |
| SIGMA(3) | Third stress component. |
| Etc. | Only NTENS stress values should be defined, where NTENS depends on the element type. |

## Variables Passed in for Information

| | |
|---|---|
| COORDS | An array containing the initial coordinates of this point. |
| NTENS | Number of stresses to be defined, which depends on the element type. |
| NCRDS | Number of coordinates. |
| NOEL | Element number. |
| NPT | Integration point number in the element. |
| LAYER | Layer number (for composite shells and layered solids). |
| KSPT | Section point number within the current layer. |
| LFLAGS(1) | Rebar flag. If LFLAGS(1)=1, the current integration point is associated with element rebar. Otherwise, LFLAGS(1)=0. |
| LFLAGS(2) | Constituent flag. If LFLAGS(2)=1, the current integration point is associated with a constituent in the multiscale material. Otherwise, LFLAGS(2)=0. |
| NAMES(1) | Name of the rebar to which the current integration point belongs, which is the name given in the rebar or rebar layer definition (*Defining Reinforcement* or *Defining Rebar as an Element Property*). If no name was given in the rebar or rebar layer definition, this variable will be blank. This variable is relevant only when LFLAGS(1)=1. |
| NAMES(2) | Element type name (see *Abaqus/Standard Element Index*). |
| NAMES(3) | Constituent name. This variable is relevant only when LFLAGS(2)=1. |

# UAMP: User subroutine to specify amplitudes.

**Products:** Abaqus/Standard

### References:

- *Amplitude Curves*
- **AMPLITUDE*
- **OUTPUT*

## Overview

User subroutine *UAMP*:

- allows you to define the current value of an amplitude definition as a function of time;
- can be used to model control engineering aspects of your system when sensors are used (sensor values are from the beginning of the increment);
- can use a predefined number of state variables in their definition; and
- can optionally compute the derivatives and integrals of the amplitude function.

## Explicit Solution Dependence

The solution dependence introduced in this user subroutine is explicit: all data passed in the subroutine for information or to be updated are values at the beginning of that increment.

## User Subroutine Interface

```
      SUBROUTINE UAMP(
     *     ampName, time, ampValueOld, dt, nProps, props, nSvars,
     *     svars, lFlagsInfo,
     *     nSensor, sensorValues, sensorNames, jSensorLookUpTable,
     *     AmpValueNew,
     *     lFlagsDefine,
     *     AmpDerivative, AmpSecDerivative, AmpIncIntegral,
     *     AmpDoubleIntegral)
C
      INCLUDE 'ABA_PARAM.INC'

C     time indices
      parameter (iStepTime        = 1,
     *           iTotalTime       = 2,
     *           nTime            = 2)
C     flags passed in for information
      parameter (iInitialization   = 1,
     *           iRegularInc       = 2,
     *           iCuts             = 3,
     *           ikStep            = 4,
     *           nFlagsInfo        = 4)
C     optional flags to be defined
      parameter (iComputeDeriv       = 1,
     *           iComputeSecDeriv    = 2,
     *           iComputeInteg       = 3,
```

```
     *           iComputeDoubleInteg = 4,
     *           iStopAnalysis       = 5,
     *           iConcludeStep       = 6,
     *           nFlagsDefine        = 6)
      dimension time(nTime), lFlagsInfo(nFlagsInfo),
     *          lFlagsDefine(nFlagsDefine)
      dimension jSensorLookUpTable(*)
      dimension sensorValues(nSensor), svars(nSvars), props(nProps)
      character*80 sensorNames(nSensor)
      character*80 ampName

      user coding to define AmpValueNew, and
      optionally lFlagsDefine, AmpDerivative, AmpSecDerivative,
      AmpIncIntegral, AmpDoubleIntegral

      RETURN
      END
```

## Variables to Be Defined

`AmpValueNew` Current value of the amplitude.

## Variables That Can Be Updated

`lFlagsDefine` Integer flag array to determine whether the computation of additional quantities is necessary or to set step continuation requirements.

| | |
|---|---|
| `lFlagsDefine(iComputeDeriv)` | If set to 1, you must provide the computation of the amplitude derivative. The default is 0, which means that Abaqus computes the derivative automatically. |
| `lFlagsDefine(iComputeSecDeriv)` | If set to 1, you must provide the computation of the amplitude second derivative. The default is 0, which means that Abaqus computes the second derivative automatically. |
| `lFlagsDefine(iComputeInteg)` | If set to 1, you must provide the computation of the amplitude incremental integral. The default is 0, which means that Abaqus computes the incremental integral automatically. |
| `lFlagsDefine(iComputeDoubleInteg)` | If set to 1, you must provide the computation of the amplitude incremental double integral. The default is 0, which means that Abaqus computes the incremental integral automatically. |
| `lFlagsDefine(iStopAnalysis)` | If set to 1, the analysis will be stopped and an error message will be issued. The default is 0, which means that Abaqus will not stop the analysis. |
| `lFlagsDefine(iConcludeStep)` | If set to 1, Abaqus will conclude the step execution and advance to the next step (if a next step exists). The default is 0. |

`svars` An array containing the values of the solution-dependent state variables associated with this amplitude definition. The number of such variables is `nsvars` (see above). You define the meaning of these variables.

This array is passed into *UAMP* containing the values of these variables at the start of the current increment. In most cases they should be updated to be the values at the end of the increment.

`AmpDerivative` Current value of the amplitude derivative.

`AmpSecDerivative` Current value of the amplitude second derivative.

`AmpIncIntegral` Current value of the amplitude incremental integral.

`AmpDoubleIntegral` Current value of the amplitude incremental double integral.

## Variables Passed in for Information

`ampName` User-specified amplitude name, left justified.

`time(iStepTime)` Current value of step time or frequency.

`time(iTotalTime)` Current value of total time.

`ampValueOld` Old value of the amplitude from the previous increment.

`dt` Time increment.

`props` User-specified array of material constants associated with this amplitude definition.

`nProps` User-defined number of material constants associated with this amplitude definition.

`nSvars` User-defined number of solution-dependent state variables associated with this amplitude definition.

`lFlagsInfo` Integer flag array with information regrading the current call to *UAMP*.

| | |
|---|---|
| `lFlagsInfo(iInitialization)` | This flag is equal to 1 if *UAMP* is called from the initialization phase of the first analysis step and is set to 0 otherwise. |
| `lFlagsInfo(iRegularInc)` | This flag is equal to 1 if *UAMP* is called from a regular increment and is set to 0 if called from the initialization phase of the first analysis step. |
| `lFlagsInfo(iCuts)` | Number of cutbacks in this increment. |
| `lFlagsInfo(ikStep)` | Step number. |

`nSensor` Total number of sensors in the model.

`sensorValues` Array with sensor values at the end of the previous increment. Each sensor value corresponds to a history output variable associated with the output database request defining the sensor.

`sensorNames` Array with user-defined sensor names in the entire model, left justified. Each sensor name corresponds to a sensor value provided with the output database request. All names will be converted to uppercase characters if lowercase or mixed-case characters were used in their definition.

`jSensorLookUpTable` Variable that must be passed into the utility functions `IGETSENSORID` and `GETSENSORVALUE`.

## Example: Amplitude Definition Using Sensor and State Variables

```
c     user amplitude subroutine
      Subroutine UAMP(
C           passed in for information and state variables
     *     ampName, time, ampValueOld, dt, nProps, props, nSvars,
     *     svars, lFlagsInfo,
     *     nSensor, sensorValues, sensorNames,
     *     jSensorLookUpTable,
C           to be defined
     *     ampValueNew,
     *     lFlagsDefine,
     *     AmpDerivative, AmpSecDerivative, AmpIncIntegral,
     *     AmpDoubleIntegral)
```

```
      include 'aba_param.inc'

C     svars - additional state variables, similar to (V)UEL
      dimension sensorValues(nSensor), svars(nSvars),
     * props(nProps)
      character*80 sensorNames(nSensor)
      character*80 ampName

C     time indices
      parameter( iStepTime        = 1,
     *           iTotalTime       = 2,
     *           nTime            = 2)
C     flags passed in for information
      parameter( iInitialization   = 1,
     *           iRegularInc       = 2,
     *           iCuts             = 3,
     *           ikStep            = 4,
     *           nFlagsInfo        = 4)
C     optional flags to be defined
      parameter( iComputeDeriv        = 1,
     *           iComputeSecDeriv     = 2,
     *           iComputeInteg        = 3,
     *           iComputeDoubleInteg  = 4,
     *           iStopAnalysis        = 5,
     *           iConcludeStep         = 6,
     *           nFlagsDefine         = 6)

      parameter( tStep=0.18d0, tAccelerateMotor = .00375d0,
     *            omegaFinal=23.26d0,
     * zero=0.0d0, one=1.0d0, two=2.0d0, four=4.0d0)

      dimension time(nTime), lFlagsInfo(nFlagsInfo),
     *          lFlagsDefine(nFlagsDefine)
      dimension jSensorLookUpTable(*)

      lFlagsDefine(iComputeDeriv)       = 1
      lFlagsDefine(iComputeSecDeriv)    = 1
      lFlagsDefine(iComputeInteg)       = 1
      lFlagsDefine(iComputeDoubleInteg) = 1

c     get sensor value
      vTrans_CU1  = GetSensorValue('HORIZ_TRANSL_MOTION',
     *                             jSensorLookUpTable,
     *                             sensorValues)

      if (ampName(1:22) .eq. 'MOTOR_WITH_STOP_SENSOR' ) then
         if (lFlagsInfo(iInitialization).eq.1) then
            AmpSecDerivative  = zero
            AmpDerivative     = omegaFinal/tAccelerateMotor
            ampValueNew       = zero
            AmpIncIntegral    = zero
            AmpDoubleIntegral = zero

            svars(1) = zero
            svars(2) = zero
         else
            tim = time(iStepTime)

c           ramp up the angular rot velocity  of the
c           electric motor
c           after which hold constant
```

```
            if (tim .le. tAccelerateMotor) then
               AmpSecDerivative  = zero
               AmpDerivative     = omegaFinal/tAccelerateMotor
               ampValueNew       = omegaFinal*tim/tAccelerateMotor
               AmpIncIntegral    = dt*(ampValueOld+ampValueNew)/
               two
               AmpDoubleIntegral = dt**2*(ampValueOld+ampValueNew)/
               four
            else
               AmpSecDerivative  = zero
               AmpDerivative     = zero
               ampValueNew       = omegaFinal
               AmpIncIntegral    = dt*(ampValueOld+ampValueNew)/
               two
               AmpDoubleIntegral = dt**2*(ampValueOld+ampValueNew)/
               four
            end if

c           retrieve old sensor value
            vTrans_CU1_old = svars(1)

c           detect a zero crossing and count the number of
c           crossings
            if (vTrans_CU1_old*vTrans_CU1 .le. zero .and.
     *          tim .gt. tAccelerateMotor ) then
               svars(2) = svars(2) + one
            end if
            nrCrossings = int(svars(2))

c           stop the motor if sensor crosses zero the second
            time
            if (nrCrossings.eq.2) then
               ampValueNew =  zero
               lFlagsDefine(iConcludeStep)=1
            end if

c           store sensor value
            svars(1) = vTrans_CU1

         end if
      end if

      return
      end
```

# UANISOHYPER_INV: User subroutine to define anisotropic hyperelastic material behavior using the invariant formulation.

**Products:** Abaqus/Standard

**References:**

- *Anisotropic Hyperelastic Behavior*
- **ANISOTROPIC HYPERELASTIC*
- *UANISOHYPER_INV and VUANISOHYPER_INV*

## Overview

User subroutine *UANISOHYPER_INV*:

- can be used to define the strain energy potential of anisotropic hyperelastic materials as a function of an irreducible set of scalar invariants;
- is called at all material calculation points of elements for which the material definition contains user-defined anisotropic hyperelastic behavior with an invariant-based formulation (*Anisotropic Hyperelastic Behavior*);
- can include material behavior dependent on field variables or state variables;
- requires that the values of the derivatives of the strain energy density function of the anisotropic hyperelastic material be defined with respect to the scalar invariants; and
- is called twice per material point in each iteration.

## Enumeration of Invariants

To facilitate coding and provide easy access to the array of invariants passed to user subroutine *UANISOHYPER_INV*, an enumerated representation of each invariant is introduced. Any scalar invariant can, therefore, be represented uniquely by an *enumerated* invariant, $\boldsymbol{I}_{\boldsymbol{n}}^{\boldsymbol{*}}$, where the subscript $n$ denotes the order of the invariant according to the enumeration scheme in the following table:

| Invariant | Enumeration, $n$ |
|---|---|
| $\bar{I}_1$ | $1$ |
| $\bar{I}_2$ | $2$ |
| $J$ | $3$ |
| $\bar{I}_{4(\alpha\beta)}$ | $4+2(\alpha-1)+\beta(\beta-1)\,;\qquad \alpha\le\beta$ |
| $\bar{I}_{5(\alpha\beta)}$ | $5+2(\alpha-1)+\beta(\beta-1)\,;\qquad \alpha\le\beta$ |

For example, in the case of three families of fibers there are a total of 15 invariants: $\bar{I}_1$, $\bar{I}_2$, $J$, six invariants of type $\bar{I}_{4(\alpha\beta)}$, and six invariants of type $\bar{I}_{5(\alpha\beta)}$, with $\alpha,\beta=1,2,3 \quad (\alpha\le\beta)$. The following correspondence exists between each of these invariants and their enumerated counterpart:

| Enumerated invariant | Invariant |
|---|---|
| $I_1^*$ | $\overline{I}_1$ |
| $I_2^*$ | $\overline{I}_2$ |
| $I_3^*$ | $J$ |
| $I_4^*$ | $\overline{I}_{4(11)}$ |
| $I_5^*$ | $\overline{I}_{5(11)}$ |
| $I_6^*$ | $\overline{I}_{4(12)}$ |
| $I_7^*$ | $\overline{I}_{5(12)}$ |
| $I_8^*$ | $\overline{I}_{4(22)}$ |
| $I_9^*$ | $\overline{I}_{5(22)}$ |
| $I_{10}^*$ | $\overline{I}_{4(13)}$ |
| $I_{11}^*$ | $\overline{I}_{5(13)}$ |
| $I_{12}^*$ | $\overline{I}_{4(23)}$ |
| $I_{13}^*$ | $\overline{I}_{5(23)}$ |
| $I_{14}^*$ | $\overline{I}_{4(33)}$ |
| $I_{15}^*$ | $\overline{I}_{5(33)}$ |

A similar scheme is used for the array ZETA of terms $\zeta_{\alpha\beta} = \mathbf{A}_\alpha \cdot \mathbf{A}_\beta$. Each term can be represented uniquely by an enumerated counterpart $\zeta_m^*$, as shown below:

| Dot product | Enumeration, $m$ |
|---|---|
| $\zeta_{\alpha\beta}$ | $\alpha + \frac{1}{2}(\beta - 2)(\beta - 1)\,; \quad \alpha < \beta$ |

As an example, for the case of three families of fibers there are three $\zeta_{\alpha\beta}$ terms: $\zeta_{12}$, $\zeta_{13}$, and $\zeta_{23}$. These are stored in the ZETA array as $(\zeta_1^*, \zeta_2^*, \zeta_3^*)$.

## Storage of Arrays of Derivatives of the Energy Function

The components of the array UI1 of first derivatives of the strain energy potential with respect to the scalar invariants, $\partial U / \partial I_i^*$, are stored using the enumeration scheme discussed above for the scalar invariants.

The elements of the array UI2 of second derivatives of the strain energy function, $\partial^2 U / \partial I_i^* \partial I_j^*$, are laid out in memory using triangular storage: if $k$ denotes the component in this array corresponding to the term $\partial^2 U / \partial I_i^* \partial I_j^*$,

then $k = i + j \times (j-1)/2; \quad (i \le j)$. For example, the term $\partial^2 U / \partial I_2^* \partial I_5^*$ is stored in component $k = 2 + (5 \times 4)/2 = 12$ in the UI2 array.

## Special Considerations for Various Element Types

There are several special considerations that need to be noted.

### Shells That Calculate Transverse Shear Energy

When *UANISOHYPER_INV* is used to define the material response of shell elements that calculate transverse shear energy, Abaqus/Standard cannot calculate a default value for the transverse shear stiffness of the element. Hence, you must define the material transverse shear modulus (see *Defining the Elastic Transverse Shear Modulus*) or the element's transverse shear stiffness (see *Shell Section Behavior*).

### Elements with Hourglassing Modes

When *UANISOHYPER_INV* is used to define the material response of elements with hourglassing modes, you must define the hourglass stiffness for hourglass control based on the total stiffness approach. The hourglass stiffness is not required for enhanced hourglass control, but you can define a scaling factor for the stiffness associated with the drill degree of freedom (rotation about the surface normal). See *Section Controls*.

## User Subroutine Interface

```
      SUBROUTINE UANISOHYPER_INV (AINV, UA, ZETA, NFIBERS, NINV,
     1     UI1, UI2, UI3, TEMP, NOEL, CMNAME, INCMPFLAG, IHYBFLAG,
     2     NUMSTATEV, STATEV, NUMFIELDV, FIELDV, FIELDVINC,
     3     NUMPROPS, PROPS)
C
      INCLUDE 'ABA_PARAM.INC'
C
      CHARACTER*80 CMNAME
      DIMENSION AINV(NINV), UA(2),
     2     ZETA(NFIBERS*(NFIBERS-1)/2)), UI1(NINV),
     3     UI2(NINV*(NINV+1)/2), UI3(NINV*(NINV+1)/2),
     4     STATEV(NUMSTATEV), FIELDV(NUMFIELDV),
     5     FIELDVINC(NUMFIELDV), PROPS(NUMPROPS)


        user coding to define UA,UI1,UI2,UI3,STATEV


      RETURN
      END
```

## Variables to Be Defined

UA(1) — $U$, strain energy density function. For a compressible material at least one derivative involving $J$ should be nonzero. For an incompressible material all derivatives involving $J$ are ignored.

UA(2) — $\tilde{U}_{dev}$, the deviatoric part of the strain energy density of the primary material response. This quantity is needed only if the current material definition also includes Mullins effect (see *Stress Softening in Rubberlike Materials*).

`UI1(NINV)` Array of derivatives of strain energy potential with respect to the scalar invariants, $\partial U / \partial I_i^*$, ordered using the enumeration scheme discussed above.

`UI2(NINV*(NINV+1)/2)` Array of second derivatives of strain energy potential with respect to the scalar invariants (using triangular storage), $\partial^2 U / \partial I_i^* \partial I_j^*$.

`UI3(NINV*(NINV+1)/2)` Array of derivatives with respect to $J$ of the second derivatives of the strain energy potential (using triangular storage), $\partial^3 U / \partial I_i^* \partial I_j^* \partial J$. This quantity is needed only for compressible materials with a hybrid formulation (when `INCMPFLAG` = 0 and `IHYBFLAG` = 1).

`STATEV` Array containing the user-defined solution-dependent state variables at this point. These are supplied as values at the start of the increment or as values updated by other user subroutines (see *About User Subroutines and Utilities*) and must be returned as values at the end of the increment.

## Variables Passed in for Information

`NFIBERS` Number of families of fibers defined for this material.

`NINV` Number of scalar invariants.

`TEMP` Current temperature at this point.

`NOEL` Element number.

`CMNAME` User-specified material name, left justified.

`INCMPFLAG` Incompressibility flag defined to be 1 if the material is specified as incompressible or 0 if the material is specified as compressible.

`IHYBFLAG` Hybrid formulation flag defined to be 1 for hybrid elements; 0 otherwise.

`NUMSTATEV` User-defined number of solution-dependent state variables associated with this material (see *Allocating Space for Solution-Dependent State Variables*).

`NUMFIELDV` Number of field variables.

`FIELDV` Array of interpolated values of predefined field variables at this material point at the end of the increment based on the values read in at the nodes (initial values at the beginning of the analysis and current values during the analysis).

`FIELDVINC` Array of increments of predefined field variables at this material point for this increment, including any values updated by user subroutine `USDFLD`.

`NUMPROPS` Number of material properties entered for this user-defined hyperelastic material.

`PROPS` Array of material properties entered for this user-defined hyperelastic material.

`AINV(NINV)` Array of scalar invariants, $I_i^*$, at each material point at the end of the increment. The invariants are ordered using the enumeration scheme discussed above.

`ZETA(NFIBERS*(NFIBERS-1)/2))` Array of dot product between the directions of different families of fiber in the reference configuration, $\zeta_{\alpha\beta} = \mathbf{A}_\alpha \cdot \mathbf{A}_\beta$. The array contains the enumerated values $\zeta_m^*$ using the scheme discussed above.

## Example: Anisotropic Hyperelastic Model of Kaliske and Schmidt

As an example of the coding of user subroutine *UANISOHYPER_INV*, consider the model proposed by Kaliske and Schmidt (2005) for nonlinear anisotropic elasticity with two families of fibers. The strain energy function is given by a polynomial series expansion in the form

$$U = \frac{1}{D}(J-1)^2 + \sum_{i=1}^{3} a_i \left(\bar{I}_1 - 3\right)^i + \sum_{j=1}^{3} b_j \left(\bar{I}_2 - 3\right)^j + \sum_{k=2}^{6} c_k \left(\bar{I}_{4(11)} - 1\right)^k + \sum_{l=2}^{6} d_l \left(\bar{I}_{5(11)} - 1\right)^l$$
$$+ \sum_{m=2}^{6} e_m \left(\bar{I}_{4(22)} - 1\right)^m + \sum_{n=2}^{6} f_n \left(\bar{I}_{5(22)} - 1\right)^n + \sum_{p=2}^{6} g_p \left(\zeta_{12}\bar{I}_{4(12)} - \zeta_{12}^2\right)^p.$$

The code in user subroutine *UANISOHYPER_INV* must return the derivatives of the strain energy function with respect to the scalar invariants, which are readily computed from the above expression. In this example auxiliary functions are used to facilitate enumeration of pseudo-invariants of type $\bar{I}_{4(\alpha\beta)}$ and $\bar{I}_{5(\alpha\beta)}$, as well as for indexing into the array of second derivatives using symmetric storage. The user subroutine would be coded as follows:

```
      subroutine uanisohyper_inv (aInv, ua, zeta, nFibers, nInv,
     *                            ui1, ui2, ui3, temp, noel,
     *                            cmname, incmpFlag, ihybFlag,
     *                            numStatev, statev,
     *                            numFieldv, fieldv, fieldvInc,
     *                            numProps, props)
C
      include 'aba_param.inc'
C
      character *80 cmname
      dimension aInv(nInv), ua(2), zeta(nFibers*(nFibers-1)/2)
      dimension ui1(nInv), ui2(nInv*(nInv+1)/2)
      dimension ui3(nInv*(nInv+1)/2), statev(numStatev)
      dimension fieldv(numFieldv), fieldvInc(numFieldv)
      dimension props(numProps)
C
      parameter ( zero  = 0.d0,
     *            one   = 1.d0,
     *            two   = 2.d0,
     *            three = 3.d0,
     *            four  = 4.d0,
     *            five  = 5.d0,
     *            six   = 6.d0 )
C
C Kaliske-Schmidtt energy function (3D)
C
C     Read material properties
      d=props(1)
      dInv = one / d
      a1=props(2)
      a2=props(3)
      a3=props(4)
      b1=props(5)
      b2=props(6)
      b3=props(7)
      c2=props(8)
      c3=props(9)
      c4=props(10)
      c5=props(11)
      c6=props(12)
```

```
      d2=props(13)
      d3=props(14)
      d4=props(15)
      d5=props(16)
      d6=props(17)
      e2=props(18)
      e3=props(19)
      e4=props(20)
      e5=props(21)
      e6=props(22)
      f2=props(23)
      f3=props(24)
      f4=props(25)
      f5=props(26)
      f6=props(27)
      g2=props(28)
      g3=props(29)
      g4=props(30)
      g5=props(31)
      g6=props(32)
C
C     Compute Udev and 1st and 2nd derivatives w.r.t invariants
C     - I1
      bi1 = aInv(1)
      term = bi1-three
      ua(2)  = a1*term + a2*term**2 + a3*term**3
      ui1(1) = a1 + two*a2*term + three*a3*term**2
      ui2(indx(1,1)) = two*a2 + three*two*a3*term
C     - I2
      bi2 = aInv(2)
      term = bi2-three
      ua(2)  = ua(2) + b1*term + b2*term**2 + b3*term**3
      ui1(2) = b1 + two*b2*term + three*b3*term**2
      ui2(indx(2,2)) = two*b2 + three*two*b3*term
C     - I3 (=J)
      bi3 = aInv(3)
      term = bi3-one
      ui1(3) = two*dInv*term
      ui2(indx(3,3)) = two*dInv
C     - I4(11)
      nI411 = indxInv4(1,1)
      bi411 = aInv(nI411)
      term = bi411-one
      ua(2) = ua(2)
     *      + c2*term**2 + c3*term**3 + c4*term**4
     *      + c5*term**5 + c6*term**6
      ui1(nI411) =
     *          two*c2*term
     *        + three*c3*term**2
     *        + four*c4*term**3
     *        + five*c5*term**4
     *        + six*c6*term**5
      ui2(indx(nI411,nI411)) =
     *          two*c2
     *        + three*two*c3*term
     *        + four*three*c4*term**2
     *        + five*four*c5*term**3
     *        + six*five*c6*term**4
C     - I5(11)
      nI511 = indxInv5(1,1)
      bi511 = aInv(nI511)
```

```
      term = bi511-one
      ua(2) = ua(2)
     *      + d2*term**2 + d3*term**3 + d4*term**4
     *      + d5*term**5 + d6*term**6
      ui1(nI511) =
     *          two*d2*term
     *        + three*d3*term**2
     *        + four*d4*term**3
     *        + five*d5*term**4
     *        + six*d6*term**5
      ui2(indx(nI511,nI511)) =
     *          two*d2
     *        + three*two*d3*term
     *        + four*three*d4*term**2
     *        + five*four*d5*term**3
     *        + six*five*d6*term**4
C     - I4(22)
      nI422 = indxInv4(2,2)
      bi422 = aInv(nI422)
      term = bi422-one
      ua(2) = ua(2)
     *      + e2*term**2 + e3*term**3 + e4*term**4
     *        + e5*term**5 + e6*term**6
      ui1(nI422) =
     *          two*e2*term
     *        + three*e3*term**2
     *        + four*e4*term**3
     *        + five*e5*term**4
     *        + six*e6*term**5
      ui2(indx(nI422,nI422)) =
     *          two*e2
     *        + three*two*e3*term
     *        + four*three*e4*term**2
     *        + five*four*e5*term**3
     *        + six*five*e6*term**4
C     - I5(22)
      nI522 = indxInv5(2,2)
      bi522 = aInv(nI522)
      term = bi522-one
      ua(2) = ua(2)
     *      + f2*term**2 + f3*term**3 + f4*term**4
     *        + f5*term**5 + f6*term**6
      ui1(nI522) =
     *          two*f2*term
     *        + three*f3*term**2
     *        + four*f4*term**3
     *        + five*f5*term**4
     *        + six*f6*term**5
      ui2(indx(nI522,nI522)) =
     *          two*f2
     *        + three*two*f3*term
     *        + four*three*f4*term**2
     *        + five*four*f5*term**3
     *        + six*five*f6*term**4
C     - I4(12)
      nI412 = indxInv4(1,2)
      bi412 = aInv(nI412)
      term = zeta(1)*(bi412-zeta(1))
      ua(2) = ua(2)
     *        + g2*term**2 + g3*term**3
     *        + g4*term**4 + g5*term**5
```

```
     *        + g6*term**6
      ui1(nI412) = zeta(1) * (
     *          two*g2*term
     *        + three*g3*term**2
     *        + four*g4*term**3
     *        + five*g5*term**4
     *        + six*g6*term**5 )
      ui2(indx(nI412,nI412)) = zeta(1)**2 * (
     *          two*g2
     *        + three*two*g3*term
     *        + four*three*g4*term**2
     *        + five*four*g5*term**3
     *        + six*five*g6*term**4 )
C
C Add volumetric energy
C
      term = aInv(3) - one
      ua(1) = ua(2) + dInv*term*term
C
      return
      end
C
C Maps index from Square to Triangular storage of symmetric
C matrix
C
      integer function indx( i, j )
C
      include 'aba_param.inc'
C
      ii = min(i,j)
      jj = max(i,j)
C
      indx = ii + jj*(jj-1)/2
C
      return
      end
C
C
C Generate enumeration of Anisotropic Pseudo Invariants of
C type 4
C
      integer function indxInv4( i, j )
C
      include 'aba_param.inc'
C
      ii = min(i,j)
      jj = max(i,j)
C
      indxInv4 = 4 + jj*(jj-1) + 2*(ii-1)
C
      return
      end
C
C
C Generate enumeration of Anisotropic Pseudo Invariants of
C type 5
C
      integer function indxInv5( i, j )
C
      include 'aba_param.inc'
C
```

```
      ii = min(i,j)
      jj = max(i,j)
C
      indxInv5 = 5 + jj*(jj-1) + 2*(ii-1)
C
      return
      end
```

## Additional Reference


- Kaliske, M., and J. Schmidt, “Formulation of Finite Nonlinear Anisotropic Elasticity,” *CADFEM GmbH Infoplaner 2/2005*, vol. 2, pp. 22–23, 2005.

# UANISOHYPER_STRAIN: User subroutine to define anisotropic hyperelastic material behavior based on Green strain.

**Products:** Abaqus/Standard

**References:**

- *Anisotropic Hyperelastic Behavior*
- **ANISOTROPIC HYPERELASTIC*
- *UANISOHYPER_INV and VUANISOHYPER_INV*

## Overview

User subroutine *UANISOHYPER_STRAIN*:

- can be used to define the strain energy potential of anisotropic hyperelastic materials as a function of the components of the Green strain tensor;
- is called at all material calculation points of elements for which the material definition contains user-defined anisotropic hyperelastic behavior with a Green strain-based formulation (*Anisotropic Hyperelastic Behavior*);
- can include material behavior dependent on field variables or state variables;
- requires that the values of the derivatives of the strain energy density function of the anisotropic hyperelastic material be defined with respect to the components of the modified Green strain tensor; and
- is called twice per material point in each iteration.

## Storage of Strain Components

In the array of modified Green strain, EBAR, direct components are stored first, followed by shear components. There are NDI direct and NSHR tensor shear components. The order of the components is defined in *Conventions*. Since the number of active stress and strain components varies between element types, the routine must be coded to provide for all element types with which it will be used.

## Storage of Arrays of Derivatives of the Energy Function

The array of first derivatives of the strain energy function, DU1, contains NTENS+1 components, with NTENS=NDI+NSHR. The first NTENS components correspond to the derivatives with respect to each component of the modified Green strain, $\partial U/\partial \bar{\varepsilon}^G_{ij}$. The last component contains the derivative with respect to the volume ratio, $\partial U/\partial J$.

The array of second derivatives of the strain energy function, DU2, contains (NTENS+1)*(NTENS+2)/2 components. These components are ordered using the following triangular storage scheme:

| Component | 2D Case | 3D Case |
|---|---|---|
| 1 | $\partial^2 U/\partial \bar{\varepsilon}^G_{11}\,\partial \bar{\varepsilon}^G_{11}$ | $\partial^2 U/\partial \bar{\varepsilon}^G_{11}\,\partial \bar{\varepsilon}^G_{11}$ |
| 2 | $\partial^2 U/\partial \bar{\varepsilon}^G_{11}\,\partial \bar{\varepsilon}^G_{22}$ | $\partial^2 U/\partial \bar{\varepsilon}^G_{11}\,\partial \bar{\varepsilon}^G_{22}$ |
| 3 | $\partial^2 U/\partial \bar{\varepsilon}^G_{22}\,\partial \bar{\varepsilon}^G_{22}$ | $\partial^2 U/\partial \bar{\varepsilon}^G_{22}\,\partial \bar{\varepsilon}^G_{22}$ |

| Component | 2D Case | 3D Case |
|---|---|---|
| 4 | $\partial^2 U/\partial \bar{\varepsilon}_{11}^G \partial \bar{\varepsilon}_{33}^G$ | $\partial^2 U/\partial \bar{\varepsilon}_{11}^G \partial \bar{\varepsilon}_{33}^G$ |
| 5 | $\partial^2 U/\partial \bar{\varepsilon}_{22}^G \partial \bar{\varepsilon}_{33}^G$ | $\partial^2 U/\partial \bar{\varepsilon}_{22}^G \partial \bar{\varepsilon}_{33}^G$ |
| 6 | $\partial^2 U/\partial \bar{\varepsilon}_{33}^G \partial \bar{\varepsilon}_{33}^G$ | $\partial^2 U/\partial \bar{\varepsilon}_{33}^G \partial \bar{\varepsilon}_{33}^G$ |
| 7 | $\partial^2 U/\partial \bar{\varepsilon}_{11}^G \partial \bar{\varepsilon}_{12}^G$ | $\partial^2 U/\partial \bar{\varepsilon}_{11}^G \partial \bar{\varepsilon}_{12}^G$ |
| 8 | $\partial^2 U/\partial \bar{\varepsilon}_{22}^G \partial \bar{\varepsilon}_{12}^G$ | $\partial^2 U/\partial \bar{\varepsilon}_{22}^G \partial \bar{\varepsilon}_{12}^G$ |
| 9 | $\partial^2 U/\partial \bar{\varepsilon}_{33}^G \partial \bar{\varepsilon}_{12}^G$ | $\partial^2 U/\partial \bar{\varepsilon}_{33}^G \partial \bar{\varepsilon}_{12}^G$ |
| 10 | $\partial^2 U/\partial \bar{\varepsilon}_{12}^G \partial \bar{\varepsilon}_{12}^G$ | $\partial^2 U/\partial \bar{\varepsilon}_{12}^G \partial \bar{\varepsilon}_{12}^G$ |
| 11 | $\partial^2 U/\partial \bar{\varepsilon}_{11}^G \partial J$ | $\partial^2 U/\partial \bar{\varepsilon}_{11}^G \partial \bar{\varepsilon}_{13}^G$ |
| 12 | $\partial^2 U/\partial \bar{\varepsilon}_{22}^G \partial J$ | $\partial^2 U/\partial \bar{\varepsilon}_{22}^G \partial \bar{\varepsilon}_{13}^G$ |
| 13 | $\partial^2 U/\partial \bar{\varepsilon}_{33}^G \partial J$ | $\partial^2 U/\partial \bar{\varepsilon}_{33}^G \partial \bar{\varepsilon}_{13}^G$ |
| 14 | $\partial^2 U/\partial \bar{\varepsilon}_{12}^G \partial J$ | $\partial^2 U/\partial \bar{\varepsilon}_{12}^G \partial \bar{\varepsilon}_{13}^G$ |
| 15 | $\partial^2 U/\partial J^2$ | $\partial^2 U/\partial \bar{\varepsilon}_{13}^G \partial \bar{\varepsilon}_{13}^G$ |
| 16 | | $\partial^2 U/\partial \bar{\varepsilon}_{11}^G \partial \bar{\varepsilon}_{23}^G$ |
| 17 | | $\partial^2 U/\partial \bar{\varepsilon}_{22}^G \partial \bar{\varepsilon}_{23}^G$ |
| 18 | | $\partial^2 U/\partial \bar{\varepsilon}_{33}^G \partial \bar{\varepsilon}_{23}^G$ |
| 19 | | $\partial^2 U/\partial \bar{\varepsilon}_{12}^G \partial \bar{\varepsilon}_{23}^G$ |
| 20 | | $\partial^2 U/\partial \bar{\varepsilon}_{13}^G \partial \bar{\varepsilon}_{23}^G$ |
| 21 | | $\partial^2 U/\partial \bar{\varepsilon}_{23}^G \partial \bar{\varepsilon}_{23}^G$ |
| 22 | | $\partial^2 U/\partial \bar{\varepsilon}_{11}^G \partial J$ |
| 23 | | $\partial^2 U/\partial \bar{\varepsilon}_{22}^G \partial J$ |
| 24 | | $\partial^2 U/\partial \bar{\varepsilon}_{33}^G \partial J$ |
| 25 | | $\partial^2 U/\partial \bar{\varepsilon}_{12}^G \partial J$ |
| 26 | | $\partial^2 U/\partial \bar{\varepsilon}_{13}^G \partial J$ |
| 27 | | $\partial^2 U/\partial \bar{\varepsilon}_{23}^G \partial J$ |
| 28 | | $\partial^2 U/\partial J^2$ |

Finally, the array of third derivatives of the strain energy function, DU3, also contains (NTENS+1)*(NTENS+2)/2 components, each representing the derivative with respect to $J$ of the corresponding component of DU2. It follows the same triangular storage scheme as DU2.

## Special Considerations for Various Element Types

There are several special considerations that need to be noted.

### Shells That Calculate Transverse Shear Energy

When *UANISOHYPER_STRAIN* is used to define the material response of shell elements that calculate transverse shear energy, Abaqus/Standard cannot calculate a default value for the transverse shear stiffness of the element. Hence, you must define the material transverse shear modulus (see *Defining the Elastic Transverse Shear Modulus*) or the element's transverse shear stiffness (see *Shell Section Behavior*).

### Elements with Hourglassing Modes

When *UANISOHYPER_STRAIN* is used to define the material response of elements with hourglassing modes, you must define the hourglass stiffness for hourglass control based on the total stiffness approach. The hourglass stiffness is not required for enhanced hourglass control, but you can define a scaling factor for the stiffness associated with the drill degree of freedom (rotation about the surface normal). See *Section Controls*.

## User Subroutine Interface

```
      SUBROUTINE UANISOHYPER_STRAIN (EBAR, AJ, UA, DU1, DU2, DU3,
     1 TEMP, NOEL, CMNAME, INCMPFLAG, IHYBFLAG, NDI, NSHR, NTENS,
     2 NUMSTATEV, STATEV, NUMFIELDV, FIELDV, FIELDVINC,
     3 NUMPROPS, PROPS)
C
       INCLUDE 'ABA_PARAM.INC'
C
      CHARACTER*80 CMNAME
C
      DIMENSION EBAR(NTENS), UA(2), DU1(NTENS+1),
     2     DU2((NTENS+1)*(NTENS+2)/2),
     3     DU3((NTENS+1)*(NTENS+2)/2),
     4     STATEV(NUMSTATEV), FIELDV(NUMFIELDV),
     5     FIELDVINC(NUMFIELDV), PROPS(NUMPROPS)


    user coding to define UA,DU1,DU2,DU3,STATEV


      RETURN
      END
```

## Variables to Be Defined

UA(1)

$U$, strain energy density function. For a compressible material at least one derivative involving $J$ should be nonzero. For an incompressible material all derivatives involving $J$ are ignored.

UA(2)

$\tilde{U}_{dev}$, the deviatoric part of the strain energy density of the primary material response. This quantity is needed only if the current material

definition also includes Mullins effect (see *Stress Softening in Rubberlike Materials*).

`DU1(NTENS+1)` Derivatives of strain energy potential with respect to the components of the modified Green strain tensor, $\partial U/\partial \bar{\varepsilon}_{ij}^{G}$, and with respect to the volume ratio, $\partial U/\partial J$.

`DU2((NTENS+1)*(NTENS+2)/2)` Second derivatives of strain energy potential with respect to the components of the modified Green strain tensor and the volume ratio (using triangular storage, as mentioned earlier).

`DU3((NTENS+1)*(NTENS+2)/2)` Derivatives with respect to $J$ of the second derivatives of the strain energy potential (using triangular storage, as mentioned earlier). This quantity is needed only for compressible materials with a hybrid formulation (when `INCMPFLAG` = 0 and `IHYBFLAG` = 1).

`STATEV` Array containing the user-defined solution-dependent state variables at this point. These are supplied as values at the start of the increment or as values updated by other user subroutines (see *About User Subroutines and Utilities*) and must be returned as values at the end of the increment.

## Variables Passed in for Information

`TEMP` Current temperature at this point.

`NOEL` Element number.

`CMNAME` User-specified material name, left justified.

`NDI` Number of direct stress components at this point.

`NSHR` Number of shear components at this point.

`NTENS` Size of the stress or strain component array (`NDI` + `NSHR`).

`INCMPFLAG` Incompressibility flag defined to be 1 if the material is specified as incompressible or 0 if the material is specified as compressible.

`IHYBFLAG` Hybrid formulation flag defined to be 1 for hybrid elements; 0 otherwise.

`NUMSTATEV` User-defined number of solution-dependent state variables associated with this material (see *Allocating Space for Solution-Dependent State Variables*).

`NUMFIELDV` Number of field variables.

`FIELDV` Array of interpolated values of predefined field variables at this material point at the end of the increment based on the values read in at the nodes (initial values at the beginning of the analysis and current values during the analysis).

`FIELDVINC` Array of increments of predefined field variables at this material point for this increment, including any values updated by user subroutine `USDFLD`.

`NUMPROPS` Number of material properties entered for this user-defined hyperelastic material.

`PROPS` Array of material properties entered for this user-defined hyperelastic material.

`EBAR(NTENS)` Modified Green strain tensor, $\bar{\boldsymbol{\varepsilon}}^{G}$, at the material point at the end of the increment.

`AJ` $J$, determinant of deformation gradient (volume ratio) at the end of the increment.

## Example: Orthotropic Saint-Venant Kirchhoff Model

As a simple example of the coding of user subroutine `UANISOHYPER_STRAIN`, consider the generalization to anisotropic hyperelasticity of the Saint-Venant Kirchhoff model. The strain energy function of the Saint-Venant Kirchhoff model can be expressed as a quadratic function of the Green strain tensor, $\boldsymbol{\varepsilon}^{G}$, as

$$U\left(\boldsymbol{\varepsilon}^G\right) = \frac{1}{2}\boldsymbol{\varepsilon}^G : \mathbf{D} : \boldsymbol{\varepsilon}^G,$$

where $\mathbf{D}$ is the fourth-order elasticity tensor. The derivatives of the strain energy function with respect to the Green strain are given as

$$\frac{\partial U}{\partial \boldsymbol{\varepsilon}^G} = \mathbf{D} : \boldsymbol{\varepsilon}^G,$$
$$\frac{\partial^2 U}{\partial \boldsymbol{\varepsilon}^G \partial \boldsymbol{\varepsilon}^G} = \mathbf{D}.$$

However, user subroutine UANISOHYPER_STRAIN must return the derivatives of the strain energy function with respect to the modified Green strain tensor, $\overline{\boldsymbol{\varepsilon}}^G$, and the volume ratio, $J$, which can be accomplished easily using the following relationship between $\boldsymbol{\varepsilon}^G$, $\overline{\boldsymbol{\varepsilon}}^G$, and $J$:

$$\boldsymbol{\varepsilon}^G = J^{\frac{2}{3}}\overline{\boldsymbol{\varepsilon}}^G + \frac{1}{2}\left(J^{\frac{2}{3}} - 1\right)\mathbf{I},$$

where $\mathbf{I}$ is the second-order identity tensor. Thus, using the chain rule we find

$$\frac{\partial U}{\partial \overline{\boldsymbol{\varepsilon}}^G} = J^{\frac{2}{3}}\frac{\partial U}{\partial \boldsymbol{\varepsilon}^G},$$
$$\frac{\partial U}{\partial J} = \frac{\partial \boldsymbol{\varepsilon}^G}{\partial J} : \frac{\partial U}{\partial \boldsymbol{\varepsilon}^G},$$
$$\frac{\partial^2 U}{\partial \overline{\boldsymbol{\varepsilon}}^G \partial \overline{\boldsymbol{\varepsilon}}^G} = J^{\frac{4}{3}}\frac{\partial^2 U}{\partial \boldsymbol{\varepsilon}^G \partial \boldsymbol{\varepsilon}^G},$$
$$\frac{\partial^2 U}{\partial J^2} = \frac{\partial^2 \boldsymbol{\varepsilon}^G}{\partial J^2} : \frac{\partial U}{\partial \boldsymbol{\varepsilon}^G} + \frac{\partial \boldsymbol{\varepsilon}^G}{\partial J} : \frac{\partial^2 U}{\partial \boldsymbol{\varepsilon}^G \partial \boldsymbol{\varepsilon}^G} : \frac{\partial \boldsymbol{\varepsilon}^G}{\partial J},$$
$$\frac{\partial^2 U}{\partial \overline{\boldsymbol{\varepsilon}}^G \partial J} = \frac{2}{3J}J^{\frac{2}{3}}\frac{\partial U}{\partial \boldsymbol{\varepsilon}^G} + J^{\frac{2}{3}}\frac{\partial^2 U}{\partial \boldsymbol{\varepsilon}^G \partial \boldsymbol{\varepsilon}^G} : \frac{\partial \boldsymbol{\varepsilon}^G}{\partial J},$$

where

$$\frac{\partial \boldsymbol{\varepsilon}^G}{\partial J} = \frac{2}{3J}J^{\frac{2}{3}}\left(\overline{\boldsymbol{\varepsilon}}^G + \frac{1}{2}\mathbf{I}\right) = \frac{2}{3J}\left(\boldsymbol{\varepsilon}^G + \frac{1}{2}\mathbf{I}\right)$$

and

$$\frac{\partial^2 \boldsymbol{\varepsilon}^G}{\partial J^2} = -\frac{1}{3J}\frac{\partial \boldsymbol{\varepsilon}^G}{\partial J}.$$

In this example an auxiliary function is used to facilitate indexing into a fourth-order symmetric tensor. The user subroutine would be coded as follows:

```
      subroutine uanisohyper_strain (
     *     ebar, aj, ua, du1, du2, du3, temp, noel, cmname,
     *     incmpFlag, ihybFlag, ndi, nshr, ntens,
     *     numStatev, statev, numFieldv, fieldv, fieldvInc,
     *     numProps, props)
c
      include 'aba_param.inc'
c
      dimension ebar(ntens), ua(2), du1(ntens+1)
      dimension du2((ntens+1)*(ntens+2)/2)
      dimension du3((ntens+1)*(ntens+2)/2)
      dimension statev(numStatev), fieldv(numFieldv)
      dimension fieldvInc(numFieldv), props(numProps)
c
      character*80 cmname
c
      parameter ( half    = 0.5d0,
```

```
     $            one     = 1.d0,
     $            two     = 2.d0,
     $            third   = 1.d0/3.d0,
     $            twothds = 2.d0/3.d0,
     $            four    = 4.d0 )
*
* Orthotropic Saint-Venant Kirchhoff strain energy function (3D)
*
      D1111=props(1)
      D1122=props(2)
      D2222=props(3)
      D1133=props(4)
      D2233=props(5)
      D3333=props(6)
      D1212=props(7)
      D1313=props(8)
      D2323=props(9)
*
      d2UdE11dE11 = D1111
      d2UdE11dE22 = D1122
      d2UdE11dE33 = D1133
*
      d2UdE22dE11 = d2UdE11dE22
      d2UdE22dE22 = D2222
      d2UdE22dE33 = D2233
*
      d2UdE33dE11 = d2UdE11dE33
      d2UdE33dE22 = d2UdE22dE33
      d2UdE33dE33 = D3333
*
      d2UdE12dE12 = D1212
*
      d2UdE13dE13 = D1313
*
      d2UdE23dE23 = D2323
*
      xpow = exp ( log(aj) * twothds )
      detuInv = one / aj
*
      E11 = xpow * ebar(1) + half * ( xpow - one )
      E22 = xpow * ebar(2) + half * ( xpow - one )
      E33 = xpow * ebar(3) + half * ( xpow - one )
      E12 = xpow * ebar(4)
      E13 = xpow * ebar(5)
      E23 = xpow * ebar(6)
*
      term1 = twothds * detuInv
      dE11Dj = term1 * ( E11 + half )
      dE22Dj = term1 * ( E22 + half )
      dE33Dj = term1 * ( E33 + half )
      dE12Dj = term1 * E12
      dE13Dj = term1 * E13
      dE23Dj = term1 * E23
      term2 = - third * detuInv
      d2E11DjDj = term2 * dE11Dj
      d2E22DjDj = term2 * dE22Dj
      d2E33DjDj = term2 * dE33Dj
      d2E12DjDj = term2 * dE12Dj
      d2E13DjDj = term2 * dE13Dj
      d2E23DjDj = term2 * dE23Dj
*
```

```
      dUdE11 = d2UdE11dE11 * E11
     *       + d2UdE11dE22 * E22
     *       + d2UdE11dE33 * E33
      dUdE22 = d2UdE22dE11 * E11
     *       + d2UdE22dE22 * E22
     *       + d2UdE22dE33 * E33
      dUdE33 = d2UdE33dE11 * E11
     *       + d2UdE33dE22 * E22
     *       + d2UdE33dE33 * E33
      dUdE12 = two * d2UdE12dE12 * E12
      dUdE13 = two * d2UdE13dE13 * E13
      dUdE23 = two * d2UdE23dE23 * E23
*
      U = half * ( E11*dUdE11 + E22*dUdE22 + E33*dUdE33 )
     *           + E12*dUdE12 + E13*dUdE13 + E23*dUdE23
*
      ua(2) = U
      ua(1) = ua(2)
*
      du1(1) = xpow * dUdE11
      du1(2) = xpow * dUdE22
      du1(3) = xpow * dUdE33
      du1(4) = xpow * dUdE12
      du1(5) = xpow * dUdE13
      du1(6) = xpow * dUdE23
      du1(7) = dUdE11*dE11Dj + dUdE22*dE22Dj + dUdE33*dE33Dj
     *     + two * ( dUdE12*dE12Dj
     *               +dUdE13*dE13Dj
     *               +dUdE23*dE23Dj )
*
      xpow2 = xpow * xpow
*
      du2(indx(1,1)) = xpow2 * d2UdE11dE11
      du2(indx(1,2)) = xpow2 * d2UdE11dE22
      du2(indx(2,2)) = xpow2 * d2UdE22dE22
      du2(indx(1,3)) = xpow2 * d2UdE11dE33
      du2(indx(2,3)) = xpow2 * d2UdE22dE33
      du2(indx(3,3)) = xpow2 * d2UdE33dE33
      du2(indx(1,4)) = zero
      du2(indx(2,4)) = zero
      du2(indx(3,4)) = zero
      du2(indx(4,4)) = xpow2 * d2UdE12dE12
      du2(indx(1,5)) = zero
      du2(indx(2,5)) = zero
      du2(indx(3,5)) = zero
      du2(indx(4,5)) = zero
      du2(indx(5,5)) = xpow2 * d2UdE13dE13
      du2(indx(1,6)) = zero
      du2(indx(2,6)) = zero
      du2(indx(3,6)) = zero
      du2(indx(4,6)) = zero
      du2(indx(5,6)) = zero
      du2(indx(6,6)) = xpow2 * d2UdE23dE23
*
      du2(indx(1,7)) = xpow * ( term1 * dUdE11
     *     + d2UdE11dE11 * dE11Dj
     *     + d2UdE11dE22 * dE22Dj
     *     + d2UdE11dE33 * dE33Dj )
      du2(indx(2,7)) = xpow * ( term1 * dUdE22
     *     + d2UdE22dE11 * dE11Dj
     *     + d2UdE22dE22 * dE22Dj
```

```
     *     + d2UdE22dE33 * dE33Dj )
      du2(indx(3,7)) = xpow * ( term1 * dUdE33
     *     + d2UdE33dE11 * dE11Dj
     *     + d2UdE33dE22 * dE22Dj
     *     + d2UdE33dE33 * dE33Dj )
      du2(indx(4,7)) = xpow * ( term1 * dUdE12
     *     + two * d2UdE12dE12 * dE12Dj )
      du2(indx(5,7)) = xpow * ( term1 * dUdE13
     *     + two * d2UdE13dE13 * dE23Dj )
      du2(indx(6,7)) = xpow * ( term1 * dUdE23
     *     + two * d2UdE23dE23 * dE13Dj )
      du2(indx(7,7))= dUdE11*d2E11DjDj
     *               +dUdE22*d2E22DjDj
     *               +dUdE33*d2E33DjDj
     *     + two*( dUdE12*d2E12DjDj
     *            +dUdE13*d2E13DjDj
     *            +dUdE23*d2E23DjDj)
     *     + d2UdE11dE11 * dE11Dj * dE11Dj
     *     + d2UdE22dE22 * dE22Dj * dE22Dj
     *     + d2UdE33dE33 * dE33Dj * dE33Dj
     *     + two  * ( d2UdE11dE22 * dE11Dj * dE22Dj
     *               +d2UdE11dE33 * dE11Dj * dE33Dj
     *               +d2UdE22dE33 * dE22Dj * dE33Dj )
     *     + four * ( d2UdE12dE12 * dE12Dj * dE12Dj
     *               +d2UdE13dE13 * dE13Dj * dE13Dj
     *               +d2UdE23dE23 * dE23Dj * dE23Dj )
*
      return
      end
*
* Maps index from Square to Triangular storage
* of symmetric matrix
*
      integer function indx( i, j )
*
      include 'aba_param.inc'
*
      ii = min(i,j)
      jj = max(i,j)
*
      indx = ii + jj*(jj-1)/2
*
      return
      end
```

# UCOHESIVEOFFSET: User subroutine to define the cohesive offset for pore pressure cohesive elements.

**Products:** Abaqus/Standard

**References:**

- *Defining the Constitutive Response of Fluid within the Cohesive Element Gap*
- *Linear Elastic Behavior*
- **ELASTIC*
- *Propagation of hydraulically driven fracture*

## Overview

User subroutine *UCOHESIVEOFFSET*:

- can be used to define a cohesive offset for pore pressure, coupled temperature-pore pressure, and coupled slurry-temperature-pore pressure cohesive elements; and
- is called at all material calculation points of elements for which the material definition contains user-defined cohesive offsets.

## User Subroutine Interface

```
      SUBROUTINE UCOHESIVEOFFSET( OFFSETOLD, OFFSETNEW, DELTA, OPENMAX,
     1 NOEL, NPT, COORDS, KSTEP, KINC, STIME, DTIME, TEMP, DTEMP, PREDEF,
DPRED, NFIELD,
     2 JMAC, JMATYP, MATLAYO, LACCFLA, SDEGOLD, GAPINI, THICKINI, STRESSOLD,
 STRESSNEW,
     3 DDSDDE, NTENS, NDI, NSHR, DP, CV, JFLGOPT, STATEV, NSTATEV, NIARRAY,
 I_ARRAY,
     4 NRARRAY, R_ARRAY, NCARRAY, C_ARRAY)
C
      INCLUDE 'ABA_PARAM.INC'
C
      CHARACTER*80 CMNAME
     DIMENSION DELTA(*), COORDS(*), PREDEF(*), DPRED(*), JMAC(*), JMATYP(*),

     1 STRESSOLD(NTENS), STRESSNEW(NTENS), DDSDDE(NTENS,NTENS), STATEV(
NSTATEV),
     2 I_ARRAY(NIARRAY), R_ARRAY(NRARRAY), C_ARRAY(NCARRAY)

      user coding to define JFLGOPT, OFFSETNEW, DDSDDE,
      STRESSNEW, and, if necessary, STATEV

      RETURN
      END
```

## Variables to Be Defined

`OFFSETNEW` Cohesive offset at the end of the increment, $\delta_{off}$. If this value is not defined, it is assumed to be zero.

JFLGOPT — An integer flag that must be set to either 1 or 2.

Set JFLGOPT=1 to update OFFSETNEW only.

Set JFLGOPT=2 to update OFFSETNEW; the Jacobian matrix, DDSDDE; and the stress at the end of the increment, STRESSNEW.

## Variables That Can Be Updated

DDSDDE — Jacobian matrix for the traction-separation response, $C = \partial \Delta t / \partial \epsilon$, where $\Delta t$ are the nominal traction stress vector increments, and $\Delta \epsilon$ are the nominal strain increments. DDSDDE(I,J) defines the change in the $I$th stress component at the end of the time increment caused by an infinitesimal perturbation in the $J$th component of the strain increment array.

DDSDDE is initialized to the Jacobian matrix (stiffness matrix) of the underlying traction-separation response before *UCOHESIVEOFFSET* is called, and Abaqus uses these values for all subsequent calculations if DDSDDE is not updated in *UCOHESIVEOFFSET*.

STRESSNEW — This variable is initialized to the nominal traction stress vector at the end of the previous increment and can be updated to the corresponding values at the end of the increment.

If you set JFLGOPT=1, Abaqus computes STRESSNEW internally. Any updates you make to STRESSNEW in this case are ignored.

If you set JFLGOPT=2, you must update STRESSNEW.

STATEV — An array containing the solution-dependent state variables. The size of the array is defined as described in *Allocating Space for Element Solution-Dependent Variables*.

## Variables Passed in for Information

OPENMAX — $\delta_{off}^{max}$, the maximum value of $\delta_{off}$ at a material point over the deformation history.

NSTATEV — Number of solution-dependent state variables that are associated with this material type (defined as described in *Allocating Space for Element Solution-Dependent Variables*).

OFFSETOLD — $\delta_{off}$, the cohesive separation offset at the beginning of the increment.

STRESSOLD — Nominal traction stress vector at the beginning of the increment.

DELTA(NTENS) — Cohesive separations including one normal and one (2D) or two (3D) shear components.

NOEL — Element number.

NPT — Integration point number.

COORDS — Current coordinates of this point.

KSTEP — Step number.

KINC — Increment number.

STIME — Step time.

DTIME — Time increment.

TEMP — Temperature at the start of the increment.

DTEMP — Temperature increment.

PREDEF — Array of predefined field variables.

DPRED — Array of increments of predefined field variables.

| | |
|---|---|
| NFIELD | Number of predefined field variables. |
| JMAC | Variable that must be passed into the GETVRM utility routine to access an output variable. |
| JMATYP | Variable that must be passed into the GETVRM utility routine to access an output variable. |
| MATLAYO | Variable that must be passed into the GETVRM utility routine to access an output variable. |
| LACCFLA | Variable that must be passed into the GETVRM utility routine to access an output variable. |
| SDEGOLD | Damage variable at the beginning of the increment. |
| GAPINI | Initial gap. |
| THICKINI | Initial thickness of the cohesive element. |
| NDI | Number of direct stress components at this point. |
| NSHR | Number of shear stress components at this point. |
| NTENS | Size of the stress or strain component array (NDI + NSHR). |
| DP | Diameter of the proppant particle. |
| CV | Slurry concentration. |
| NIARRAY | Size of array I_ARRAY. |
| I_ARRAY | Not used. |
| NRARRAY | Size of array R_ARRAY. |
| R_ARRAY | Not used. |
| NCARRAY | Size of array C_ARRAY. |
| C_ARRAY | Not used. |

## Example: Keeping Fracture Prop Open

```
      SUBROUTINE UCOHESIVEOFFSET( OFFSETCUR, OFFSETNEW, DELTA, OPENMAX,
     1     NOEL, NPT, COORDS, KSTEP, KINC, TIME, DTIME, TEMP, DTEMP, PREDEF,

     2     DPRED, NFIELD, JMAC, JMATYP, MATLAYO, LACCFLA, SDEG, GAPINI,
     3     THICKINI, STRESSOLD, STRESSNEW, STIFFNESS, NTENS, NDI, NSHR,
     4     DIAMETERP, CV, JFLGOPT, STATEV, NSTATEV, NIARRAY, I_ARRAY,
NRARRAY,
     5     R_ARRAY, NCARRAY, C_ARRAY)
C
      INCLUDE 'ABA_PARAM.INC'
C
      DIMENSION COORDS(*), TIME(*), TEMP(*), DTEMP(*), PREDEF(NFIELD),
     $     DPRED(*), JMAC(*), JMATYP(*), STRESSOLD(*), STRESSNEW(*),
     $     STIFFNESS( NTENS, NTENS), DELTA(*), STATEV(*)
      DIMENSION I_ARRAY(*), R_ARRAY(*)
      CHARACTER*80 C_ARRAY(*)
C
CC   THIS EXAMPLE DEMONSTRATES HOW TO SPECIFY AN OFFSET IN STEP 2 TO KEEP
THE
CC   FRACTURE "PROPPED OPEN" WHEN EXTERNAL LOADING CONDITIONS WOULD
CC   OTHERWISE HAVE LED TO CLOSURE OF THE FRACTURE

      JFLGOPT = 1
      IF(SDEG.LT.0.99999 .OR. KSTEP.EQ.1) THEN
         OFFSETNEW = 0.0
      ELSE
```

```
         OFFSETNEW = 0.005
      END IF

      RETURN
      END
```

# UCORR: User subroutine to define cross-correlation properties for random response loading.

**Products:** Abaqus/Standard

**References:**

- *Random Response Analysis*
- **CORRELATION*
- *Random response to jet noise excitation*

## Overview

User subroutine *UCORR*:

- can be used to define the coefficients for the cross-correlation matrix in a random response analysis;
- will be called once for the combination of any two degrees of freedom with nonzero prescribed loads for each load case specified as a concentrated or distributed load or once for the combination of any two excitation directions specified as a base motion;
- allows correlation coefficients to be defined as a function of nodal coordinates; and
- ignores any data specified outside the user subroutine for the associated cross-correlation matrix.

## Cross-Correlation for Base Motion Excitation

The spatial correlation matrix for base motion excitation is defined by the coefficients $\Psi_{ij}^{IJ}$ in user subroutine *UCORR*, where $i, j$ are excitation directions and $J$ corresponds to the $J$th frequency function referenced under load case $I$.

## Cross-Correlation for Point Loads and Distributed Loads

The spatial correlation matrix of the load is defined as follows. Let $F_{(N,i)}^{I}$ be the load applied to degree of freedom $i$ at node $N$ in load case $I$, through the use of a concentrated or distributed load. Let $J$ correspond to the $J$th frequency function referenced under load case $I$. The spatial correlation matrix used in the random response analysis for this load case is then

$$\Psi_{(N,i)(M,j)}^{IJ} = C_{(N,i)(M,j)}^{IJ} F_{(N,i)}^{I} F_{(M,j)}^{I},$$

where $C_{(N,i)(M,j)}^{IJ}$ are the coefficients defined in user subroutine *UCORR*. Typically the load magnitude is given as 1.0; therefore, the load definition is simply selecting the nonzero terms that will appear in $\Psi_{(N,i)(M,j)}^{IJ}$.

## User Subroutine Interface

```
      SUBROUTINE UCORR(PSD,CORRR,CORRI,KSTEP,LCASE,JNODE1,JDOF1,
     1 JNODE2,JDOF2,COOR1,COOR2)
C
      INCLUDE 'ABA_PARAM.INC'
C
```

```
      DIMENSION COOR1(3),COOR2(3)
      CHARACTER*80 PSD

      user coding to define CORRR and CORRI

      RETURN
      END
```

## Variables to Be Defined

| | |
|---|---|
| CORRR | Real part of the cross-correlation scaling factor. |
| CORRI | Imaginary part of the cross-correlation scaling factor. |

## Variables Passed in for Information

| | |
|---|---|
| PSD | User-specified name for the frequency function that references this correlation, left justified. |
| KSTEP | Step number. |
| LCASE | Load case number, *I*. |
| JNODE1 | First node involved, *N* (not used for base motion excitation). |
| JDOF1 | Degree of freedom *i* at the first node (for concentrated or distributed load excitation) or global excitation direction *i* (for base motion excitation). |
| JNODE2 | Second node involved, *M* (not used for base motion excitation). |
| JDOF2 | Degree of freedom *j* at the second node (for concentrated or distributed load excitation) or global excitation direction *j* (for base motion excitation). |
| COOR1 | An array containing the coordinates of the first node (not used for base motion excitation). |
| COOR2 | An array containing the coordinates of the second node (not used for base motion excitation). |

# UCOUPLEDCONSTRESP: User subroutine to define a fully coupled constitutive response based on the different physics that is being modeled.

**Products:** Abaqus/Standard

**References:**

## Overview



**Warning:** The use of this subroutine generally requires considerable expertise. You are cautioned that the implementation of any realistic constitutive model requires extensive development and testing. Initial testing on a single-element model with prescribed loading in different fields is strongly recommended.

User subroutine *UCOUPLEDCONSTRESP*:

- can be used to define a fully coupled multiphysics constitutive behavior of a material;
- will be called at all material calculation points of elements for which the material definition includes a user-defined coupled constitutive behavior;
- can be used only in procedures that support electrochemical analyses (see *Electrochemical Analysis*);
- can use solution-dependent state variables; and
- can be used in conjunction with user subroutine *USDFLD* to redefine any field variables before they are passed in.

## User Subroutine Interface

```
      subroutine ucoupledconstresp(
     *     stress, dDStress_dDStrain, dDStress_dGradT, dDStress_dGradSol,
     *     dDStress_dT, dDStress_dSol,
     *     qHeat, dqHeat_dDStrain, dqHeat_dGradT, dqHeat_dGradSol,
     *     dqHeat_dT, dqHeat_dSol,
     *     fluxesSol, dfluxes_dDstrain, dfluxes_dGradT, dfluxes_dGradSol,
     *     dfluxes_dT, dfluxes_dSol,
     *     qLoss, dqLoss_dDStrain, dqLoss_dGradT, dqLoss_dGradSol,
     *     dqLoss_dT, dqLoss_dSol,
     *     iCoupledAssemble_GradDep,      iCoupledAssembleSoltnDep,
     *     iCoupledAssembleqLoss_GradDep, iCoupledAssembleqLoss_SoltnDep,
     *     nDim, nTens,  nSolutionVar,
     *     props, nprops, predef, dpredef, npredef,
     *     statev, statevOld, nstatv,
     *     temp, dtemp,
     *     solutionVar, dSolutionVar,
     *     strain, dStrain, GradT, GradSol,
     *     coords,
     *     energy, time, dtime, CMNAME, ndi, nshr,
     *     celent, dRot, dFgrd0, dFgrd1, noel,
     *     npt, kstep, lFlags, kinc,
     *     i_array, niarray, r_array, nrarray, c_array, ncarray)
C
      include 'aba_param.inc'
c Include below file to access property and parameter table.
      include 'aba_tcs_param.inc'
c Include below file to access universal constants
      include 'aba_phcon_param.inc'
```

```
c Include below file for parametrized location of various solution variables.

      include 'aba_solutionvarloc.inc'
C
      dimension strain(ntens),dStrain(ntens),
     *     stress(ntens),dDStress_dDStrain(ntens,ntens),
     *     dDStress_dGradT(ntens,ndim),
dDStress_dGradSol(ntens,ndim,nsolutionvar),
     *     dDStress_dT(ntens),dDStress_dSol(ntens,nsolutionvar),
     *    qHeat(ndim),dqHeat_dDStrain(ndim,ntens),dqHeat_dGradT(ndim,ndim),

     *     dqHeat_dGradSol(ndim,ndim,nsolutionvar),dqHeat_dT(ndim),
     *     dqHeat_dSol(ndim,nsolutionvar),
     *
fluxesSol(ndim,nsolutionvar),dfluxes_dDstrain(ndim,ntens,nsolutionvar),
     *     dfluxes_dGradT(ndim,ndim,nsolutionvar),
     *     dfluxes_dGradSol(ndim,ndim,nsolutionvar,nsolutionvar),
     *
dfluxes_dT(ndim,nsolutionvar),dfluxes_dSol(ndim,nsolutionvar,nsolutionvar),

     *     dqLoss_dDStrain(ntens),dqLoss_dGradT(ndim),
     *     dqLoss_dGradSol(ndim,nsolutionvar),dqLoss_dSol(nSolutionVar),
     *     iCoupledAssemble_GradDep(nAllFieldsMax,nAllFieldsMax),
     *     iCoupledAssembleSoltnDep(nAllFieldsMax,nAllFieldsMax),
     *     iCoupledAssembleqLoss_GradDep(nAllFieldsMax),
iCoupledAssembleqLoss_SoltnDep(nAllFieldsMax),
     *     props(nprops),
predef(npredef),dpredef(npredef),statev(nstatv),statevOld(nstatv),
     *     solutionvar(nsolutionvar),dsolutionvar(nsolutionvar),
     *     gradT(ndim),gradSol(ndim,nSolutionVar),coords(*),
     *     energy(*),time1(2),drot(3,3),dfgrd0(3,3),dfgrd1(3,3),
     *     i_array(*),r_array(*), lFlags(*)
      character*80   cmname, c_array(NCARRAY)

      user coding

      return
      end
```

## Variables to Be Defined

`stress`

This array is passed in as the stress tensor at the beginning of the increment and must be updated in this routine to be the stress tensor at the end of the increment. If you specified initial stresses (*Initial Conditions*), this array will contain the initial stresses at the start of the analysis. The size of this array depends on the value of `ntens` as defined below. In finite-strain problems the stress tensor has already been rotated to account for rigid body motion. The measure of stress used is “true” (Cauchy) stress.

`dDStress_dDStrain`

`dDStress_dDStrain(I,J)` defines the change in the Ith stress component at the end of the time increment caused by an infinitesimal perturbation of the Jth component of the strain increment array. Unless you invoke the unsymmetric equation solution capability for the user-defined material, Abaqus/Standard will use only the symmetric part of `dDStress_dDStrain`. The symmetric part of the matrix is calculated by taking one half the sum of the matrix and its transpose.

`dDStress_dGradT` `dDStress_dGradT` defines the variation of the stress components with respect to the gradient of temperature `GradT`.

`dDStress_dGradSol` `dDStress_dGradSol` defines the variation of the stress components with respect to the gradient of the various solution variables `GradSol`.

`dDStress_dT` `dDStress_dT` defines the variation of the stress components with respect to the temperature `Temp`.

`dDStress_dSol` `dDStress_dSol` defines the variation of the stress components with respect to the other physics fields as defined in the solution variables `solutionvar`.

`qHeat` Total heat flux from different fields.

`dqHeat_dDStrain` Variation of `qHeat` with respect to the strain increments.

`dqHeat_dGradT` Variation of `qHeat` with respect to the temperature gradients.

`dqHeat_dGradSol` Variation of `qHeat` with respect to the gradient of the solution variables.

`dqHeat_dT` Variation of `qHeat` with respect to temperature.

`dqHeat_dSol` Variation of `qHeat` with respect to the solution variables.

`fluxesSol` Fluxes for the different physics fields defined by the solution variables `solutionvar`.

`dfluxes_dDStrain` Variation of `fluxesSol` with respect to the strain increments.

`dfluxes_dGradT` Variation of `fluxesSol` with respect to the temperature gradients.

`dfluxes_dGradSol` Variation of `fluxesSol` with respect to the gradient of the solution variables.

`dfluxes_dT` Variation of `fluxesSol` with respect to temperature.

`dfluxes_dSol` Variation of `fluxesSol` with respect to the solution variables.

`qLoss` Volumetric heat generation per unit time at the end of the increment, which can be similar to `RPL` in *UMAT*.

`dqLoss_dDStrain` Variation of `qLoss` with respect to the strain increments.

`dqLoss_dGradT` Variation of `qLoss` with respect to the temperature gradients.

`dqLoss_dGradSol` Variation of `qLoss` with respect to the gradient of the solution variables.

`dqLoss_dT` Variation of `qLoss` with respect to temperature.

`dqLoss_dSol` Variation of `qLoss` with respect to the solution variables.

`iCoupledAssemble_GradDep` An array of integer flags of dimension (`nAllFieldsMax`, `nAllFieldsMax`) that indicates whether the variation of `fluxesSol` with respect to the gradients of the different fields must be assembled. These gradient-dependent flags are used for performance to avoid assembling zeros.

If you have gradient-dependent terms to be assembled, specify a value of 1 in the specific locations for the different fields. The default is set to 0.

The table below illustrates an example of the assembly flag set for assembling the gradient-dependent terms for the electric potential flux with respect to the gradient of other fields.

| | |
|---|---|
| `iCoupledAssemble_GradDep(i_sol_phiS, i_sol_phiS)` | If set to 1, assemble gradient dependence of electric potential flux with respect to the gradient of the electric potential field. |
| `iCoupledAssemble_GradDep(i_sol_phiS, i_sol_phiE)` | If set to 1, assemble gradient dependence of electric potential flux with respect to the gradient of electric potential in the electrolyte. |

| | |
|---|---|
| iCoupledAssemble_GradDep(i_sol_phiS, i_sol_Ce) | If set to 1, assemble gradient dependence of electric potential flux with respect to the gradient of ion concentration in the electrolyte. |
| iCoupledAssemble_GradDep(i_sol_phiS, i_sol_solidC) | If set to 1, assemble gradient dependence of electric potential flux with respect to the gradient of species concentration. |
| iCoupledAssemble_GradDep(i_sol_phiS, i_sol_porP) | If set to 1, assemble gradient dependence of electric potential flux with respect to the gradient of pore pressure. |
| iCoupledAssemble_GradDep(i_sol_phiS, i_sol_temp) | If set to 1, assemble gradient dependence of electric potential flux with respect to gradient of temperature. |
| iCoupledAssemble_GradDep(i_sol_phiS, i_sol_disp) | If set to 1, assemble gradient dependence of electric potential flux with respect to the gradient of displacement field. |

`iCoupledAssembleSoltnDep` An array of integer flags of dimension (`nAllFieldsMax` and `nAllFieldsMax`) that indicates whether the variation of `fluxesSol` with respect to the solution variables `solutionVar` for the different fields must be assembled.

If you have solution-dependent terms to be assembled, specify a value of 1 in the specific locations for the different fields. The default is set to 0.

The table below illustrates an example of assembling the solution dependence of ion concentration flux with respect to the other fields.

| | |
|---|---|
| iCoupledAssembleSoltnDep(i_sol_Ce, i_sol_phiS) | If set to 1, assemble dependence of ion concentration flux with respect to electric potential field. |
| iCoupledAssembleSoltnDep(i_sol_Ce, i_sol_phiE) | If set to 1, assemble dependence of ion concentration flux with respect to electric potential in the electrolyte. |
| iCoupledAssembleSoltnDep(i_sol_Ce, i_sol_Ce) | If set to 1, assemble dependence of ion concentration flux with respect to ion concentration in the electrolyte. |
| iCoupledAssembleSoltnDep(i_sol_Ce, i_sol_solidC) | If set to 1, assemble dependence of ion concentration flux with respect to species concentration. |
| iCoupledAssembleSoltnDep(i_sol_Ce, i_sol_porP) | If set to 1, assemble dependence of ion concentration flux with respect to pore pressure. |
| iCoupledAssembleSoltnDep(i_sol_Ce, i_sol_Temp) | If set to 1, assemble dependence of ion concentration flux with respect to temperature. |
| iCoupledAssembleSoltnDep(i_sol_Ce, i_sol_Disp) | If set to 1, assemble dependence of ion concentration flux with respect to displacement field. |

`iCoupledAssembleqLoss_GradDep` An array of integer flags of dimension (`nAllFieldsMax`) that indicates whether the variation of `qloss` with respect to the gradients of the different fields must be assembled.

If you have gradient-dependent terms to be assembled, specify a value of 1 in the specific locations for the different fields. The default is set to 0.

| | |
|---|---|
| iCoupledAssembleqLoss_GradDep(i_sol_phiS) | If set to 1, assemble dependence of heat loss with respect to the gradient in electric potential field. |
| iCoupledAssembleqLoss_GradDep(i_sol_phiE) | If set to 1, assemble dependence of heat loss with respect to the gradient in electric potential in the electrolyte. |
| iCoupledAssembleqLoss_GradDep(i_sol_Ce) | If set to 1, assemble dependence of heat loss with respect to gradient in ion concentration in the electrolyte. |
| iCoupledAssembleqLoss_GradDep(i_sol_solidC) | If set to 1, assemble dependence of heat loss with respect to the gradient in species concentration. |
| iCoupledAssembleqLoss_GradDep(i_sol_porP) | If set to 1, assemble dependence of heat loss with respect to the gradient in pore pressure. |
| iCoupledAssembleqLoss_GradDep(i_sol_Temp) | If set to 1, assemble dependence of heat loss with respect to the gradient in temperature. |
| iCoupledAssembleqLoss_GradDep(i_sol_Disp) | If set to 1, assemble dependence of heat loss with respect to the gradient in displacement field. |

`iCoupledAssembleqLoss_SoltnDep` An array of integer flags of dimension (`nAllFieldsMax`) that indicates whether the variation of `qloss` with respect to the solution variables of the different fields must be assembled.

If you have solution-dependent terms to be assembled, specify a value of 1 in the specific locations for the different fields. The default is set to 0.

| | |
|---|---|
| iCoupledAssembleqLoss_SoltnDep(i_sol_phiS) | If set to 1, assemble dependence of heat loss with respect to electric potential field. |
| iCoupledAssembleqLoss_SoltnDep(i_sol_phiE) | If set to 1, assemble dependence of heat loss with respect to electric potential in the electrolyte. |
| iCoupledAssembleqLoss_SoltnDep(i_sol_Ce) | If set to 1, assemble dependence of heat loss with respect to ion concentration in the electrolyte. |
| iCoupledAssembleqLoss_SoltnDep(i_sol_solidC) | If set to 1, assemble dependence of heat loss with respect to species concentration. |
| iCoupledAssembleqLoss_SoltnDep(i_sol_porP) | If set to 1, assemble dependence of heat loss with respect to pore pressure. |
| iCoupledAssembleqLoss_SoltnDep(i_sol_Temp) | If set to 1, assemble dependence of heat loss with respect to temperature. |
| iCoupledAssembleqLoss_SoltnDep(i_sol_Disp) | If set to 1, assemble dependence of heat loss with respect to displacement field. |

## Variables That Can Be Updated

`statev` Current value of the solution-dependent variables.

## Variables Passed in for Information

`ndim` Number of spatial dimensions.

`ntens` Size of the stress or strain component array.

`nsolutionVar` Number of solution variables.

`props` User-specified array of material constants associated with this user material.

`nprops` User-defined number of material constants associated with this user material.

`predef` Array of interpolated values of predefined field variables at this point at the start of the increment, based on the values read in at the nodes.

`dpredef` Array of increments of predefined field variables.

`npredef` Number of predefined field variables.

`statevold` Array containing the user-defined solution-dependent state variables at the end of the previous increment.

`nstatev` Number of solution-dependent state variables that are associated with this material type (defined as described in *Allocating Space for Solution-Dependent State Variables*).

`Temp` Current temperature.

`dTemp` Incremental temperature.

`solutionVar` Current value of the solution variables.

| | |
|---|---|
| `solutionVar(i_sol_phiS)` | Current value of the electric potential. |
| `solutionVar(i_sol_phiE)` | Current value of the electric potential in the electrolyte. |
| `solutionVar(i_sol_Ce)` | Current value of the ionic concentration in the electrolyte. |
| `solutionVar(i_sol_solidC)` | Current value of the species concentration. |
| `solutionVar(i_sol_porP)` | Current value of the pore pressure. |

`dsolutionVar` Incremental value of the solution variables.

| | |
|---|---|
| `dsolutionVar(i_sol_phiS)` | Incremental value of the electric potential. |
| `dsolutionVar(i_sol_phiE)` | Incremental value of the electric potential in the electrolyte. |
| `dsolutionVar(i_sol_Ce)` | Incremental value of the ionic concentration in the electrolyte. |
| `dsolutionVar(i_sol_solidC)` | Incremental value of the species concentration. |
| `dsolutionVar(i_sol_porP)` | Incremental value of the pore pressure. |

`strain` An array containing the total strains at the beginning of the increment. If thermal expansion is included in the same material definition, the strains passed into *UCOUPLEDCONSTRESP* are the mechanical strains only (that is, the thermal strains computed based on the thermal expansion coefficient have been subtracted from the total strains). These strains are available for output as the “elastic” strains.

In finite-strain problems the strain components have been rotated to account for rigid body motion in the increment before *UCOUPLEDCONSTRESP* is called and are approximations to logarithmic strain.

`dStrain` Array of strain increments. If thermal expansion is included in the same material definition, these are the mechanical strain increments (the total strain increments minus the thermal strain increments).

`gradT` Current values of the spatial gradients of temperature.

`gradSol` Current values of the spatial gradients of solution variables (see `solutionVar`).

`coords` An array containing the current coordinates of this integration point. These are the current coordinates if geometric nonlinearity is accounted for during the step; otherwise, the array contains the original coordinates of the point.

`time(1)` Current value of step time.

`time(2)` Current value of total time.

`dtime` Time increment.

`cmName` User-defined material name, left justified. Some internal material models are given names starting with the "ABQ_" character string. To avoid conflict, you should not use "ABQ_" as the leading string for `CMNAME`.

`ndi` Number of direct stress components at this point.

`nshr` Number of engineering shear stress components at this point.

`celent` Characteristic element length, which is a typical length of a line across an element for a first-order element.

`dRot` Rotation increment matrix. This matrix represents the increment of rigid body rotation of the basis system in which the components of stress and strain are stored. It is provided so that vector- or tensor-valued state variables can be rotated appropriately in this subroutine: stress and strain components are already rotated by this amount before *UCOUPLEDCONSTRESP* is called. This matrix is passed in as a unit matrix for small-displacement analysis and for large-displacement analysis if the basis system for the material point rotates with the material (when a local orientation is used).

`dFgrd0` Array containing the deformation gradient at the beginning of the increment. If a local orientation is defined at the material point, the deformation gradient components are expressed in the local coordinate system defined by the orientation at the beginning of the increment. For more information about the availability of the deformation gradient for various element types, see *Deformation Gradient*.

`dFgrd1` Array containing the deformation gradient at the end of the increment. If a local orientation is defined at the material point, the deformation gradient components are expressed in the local coordinate system defined by the orientation. This array is set to the identity matrix if nonlinear geometric effects are not included in the step definition associated with this increment. For more information about the availability of the deformation gradient for various element types, see *Deformation Gradient*.

`noel` User-assigned element number.

`npt` Integration point number.

`kstep` Current step number.

`lFlags` An array containing the flags that define the current solution procedure and analysis type.

| | |
|---|---|
| `lFlags(1)` | Defines the procedure type. See *Results File* for the key used for each procedure. |
| `lFlags(2)=0` | Small-displacement analysis. |

| | | |
|---|---|---|
| | `lFlags(2)=1` | Large-displacement analysis (nonlinear geometric effects included in the step). See *General and Perturbation Procedures*. |

| | |
|---|---|
| `kinc` | Current increment number. |
| `i_array` | Not used. |
| `niarray` | Size of array `i_array`. |
| `r_array` | Not used. |
| `nrarray` | Size of array `r_array`. |
| `c_array` | Not used. |
| `ncarray` | Size of array `c_array`. |

# UCREEPNETWORK: User subroutine to define time-dependent behavior (creep) for models defined within the parallel rheological framework.

**Products:** Abaqus/Standard

**References:**

- *Parallel Rheological Framework*
- *Nonlinear large-strain viscoelasticity with hyperelasticity*
- **VISCOELASTIC*

## Overview

User subroutine *UCREEPNETWORK*:

- is intended to provide creep laws for nonlinear viscoelastic networks for models defined using the parallel rheological framework (see *Parallel Rheological Framework*);
- can use and update solution-dependent state variables; and
- can be used in conjunction with user subroutine *USDFLD* to redefine any field variables before they are passed in.

## Model Description

The user subroutine allows a creep law of the following general form to be defined:

$$\dot{\bar{\varepsilon}}^{cr} = g^{cr}\left(\bar{\varepsilon}^{cr}, I_1^{cr}, \bar{I}_1, \bar{I}_2, J, p, \tilde{q}, t, \theta, FV\right),$$

where

$$I_1^{cr} = \mathbf{I} : \mathbf{C}^{cr},$$

and

$\mathbf{I}$

is the identity tensor,

$\mathbf{C}^{cr}$

is the right Cauchy-Green creep strain tensor,

$\dot{\bar{\varepsilon}}^{cr}$

is the equivalent creep strain rate,

$\bar{\varepsilon}^{cr}$

is the equivalent creep strain,

$\bar{I}_1$

is the first invariant of $\overline{\mathbf{B}}$,

$\bar{I}_2$

is the second invariant of $\overline{\mathbf{B}}$,

$J$

is the determinant of the deformation gradient, $\mathbf{F}$,

$p$

is the Kirchhoff pressure,

$\tilde{q}$

is the equivalent deviatoric Kirchhoff stress,

$t$

is the time,

$\theta$

is the temperature, and

$FV$

are field variables.

The left Cauchy-Green strain tensor, $\overline{\mathbf{B}}$, is defined as

$$\overline{\mathbf{B}} = \overline{\mathbf{F}}\overline{\mathbf{F}}^T,$$

where $\overline{\mathbf{F}}$ is the deformation gradient with volume change eliminated, which is computed using

$$\overline{\mathbf{F}} = J^{-\frac{1}{3}}\mathbf{F}.$$

The user subroutine must define the increment of creep equivalent strain, $\Delta\bar{\varepsilon}^{cr}$, as a function of the time increment, $\Delta t$, and the variables used in the definition of $g^{cr}$, as well as the derivatives of the equivalent creep strain increment with respect to those variables. If any solution-dependent state variables are included in the definition of $g^{cr}$, they must also be integrated forward in time in this user subroutine.

## User Subroutine Interface

```
      subroutine ucreepnetwork (
C Must be updated
     *   outputData,
C Can be updated
```

```
     *   statev,
C Information (Read only)
     *   nOutput,
     *   nstatv,
     *   networkid,
     *   coords,
     *   temp,
     *   dtemp,
     *   nfield,
     *   predef,
     *   dpred,
     *   nprops,
     *   props,
     *   i_array,
     *   niarray,
     *   r_array,
     *   nrarray,
     *   c_array,
     *   ncarray)
C
      include 'aba_param.inc'
C
      parameter( io_creep_equiv_creepinc       = 1,
     *           io_creep_deqcreepinc_deqcreep = 2,
     *           io_creep_deqcreepinc_dqtild   = 3,
     *           io_creep_deqcreepinc_dinv1crp = 4,
     *           io_creep_deqcreepinc_dinv1    = 5,
     *           io_creep_deqcreepinc_dinv2    = 6,
     *           io_creep_deqcreepinc_ddetf    = 7,
     *           io_creep_deqcreepinc_dpress   = 8 )
C
      parameter( i_creep_kstep   = 1,
     *           i_creep_kinc    = 2,
     *           i_creep_noel    = 3,
     *           i_creep_npt     = 4,
     *           i_creep_layer   = 5,
     *           i_creep_kspt    = 6,
     *           i_creep_lend    = 7 )
C
      parameter( ir_creep_step_time  = 1,
     *           ir_creep_total_time = 2,
     *           ir_creep_creep_time = 3,
     *           ir_creep_timeinc    = 4,
     *           ir_creep_equiv_creep_strain = 5,
     *           ir_creep_qtild      = 6,
     *           ir_creep_inv1crp    = 7,
     *           ir_creep_inv1       = 8,
     *           ir_creep_inv2       = 9,
     *           ir_creep_detf       = 10,
     *           ir_creep_press      = 11 )
C
      parameter( ic_creep_material_name = 1 )
C
      dimension
     *   statev(nstatv),
     *   predef(nfield),
     *   dpred(nfield),
     *   coords(*),
     *   props(nprops),
     *   outputData(nOutput),
     *   i_array(niarray),
```

```
     *   r_array(nrarray)

      character*80 c_array(ncarray)
C
      user coding to define outputData(io_creep_equiv_creepinc),
        outputData(io_creep_deqcreepinc_deqcreep),
        outputData(io_creep_deqcreepinc_dqtild),
        outputData(io_creep_deqcreepinc_dinv1crp),
        outputData(io_creep_deqcreepinc_dinv1),
        outputData(io_creep_deqcreepinc_dinv2),
        outputData(io_creep_deqcreepinc_ddetf) and
        outputData(io_creep_deqcreepinc_dpress)

      return
      end
```

## Variables to Be Defined

| | |
|---|---|
| `outputData(io_creep_equiv_creepinc)` | Equivalent creep strain increment, $\Delta\bar{\varepsilon}^{cr}$. |
| `outputData(io_creep_deqcreepinc_deqcreep)` | The derivative: $\partial\Delta\bar{\varepsilon}^{cr}/\partial\bar{\varepsilon}^{cr}$. |
| `outputData(io_creep_deqcreepinc_dqtild)` | The derivative: $\partial\Delta\bar{\varepsilon}^{cr}/\partial\tilde{q}$. |
| `outputData(io_creep_deqcreepinc_dinv1crp)` | The derivative: $\partial\Delta\bar{\varepsilon}^{cr}/\partial I_1^{cr}$. |
| `outputData(io_creep_deqcreepinc_dinv1)` | The derivative: $\partial\Delta\bar{\varepsilon}^{cr}/\partial\bar{I}_1$. |
| `outputData(io_creep_deqcreepinc_dinv2)` | The derivative: $\partial\Delta\bar{\varepsilon}^{cr}/\partial\bar{I}_2$. |
| `outputData(io_creep_deqcreepinc_ddetf)` | The derivative: $\partial\Delta\bar{\varepsilon}^{cr}/\partial J$. |
| `outputData(io_creep_deqcreepinc_dpress)` | The derivative: $\partial\Delta\bar{\varepsilon}^{cr}/\partial p$. |

## Variables That Can Be Updated

| | |
|---|---|
| `statev` | An array containing the user-defined solution-dependent state variables at this point. |

## Variables Passed in for Information

| | |
|---|---|
| `nOutput` | Size of array `outputData`. |
| `nstatv` | Number of solution-dependent state variables associated with this material. |
| `networkid` | Network identification number, which identifies the network for which creep is defined. |
| `coords` | An array containing the current coordinates at this point. |
| `temp` | Temperature at the end of the increment. |
| `dtemp` | Increment of temperature. |
| `nfield` | Number of field variables. |
| `predef` | An array of interpolated values of predefined field variables at this point at the end of the increment, based on the values read in at the nodes and, optionally, redefined in user subroutine *USDFLD*. |
| `dpred` | An array of increments of predefined field variables. |

| | |
|---|---|
| nprops | User-specified number of property values associated with this creep model. |
| props | An array of user-specified property values that are used to define the creep model. |
| i_array(i_creep_kstep) | Step number. |
| i_array(i_creep_kinc) | Increment number. |
| i_array(i_creep_noel) | Element number. |
| i_array(i_creep_npt) | Integration point. |
| i_array(i_creep_layer) | Layer number (for layered solids). |
| i_array(i_creep_kspt) | Section point number within the current layer. |
| i_array(i_creep_lend) | Start/end of increment flag. The value of 0 denotes the beginning of the increment, and the value of 1 denotes the end of the increment. |
| niarray | Size of array i_array. |
| r_array(ir_creep_step_time) | Value of step time at the end of the increment. |
| r_array(ir_creep_total_time) | Value of total time at the end of the increment. |
| r_array(ir_creep_creep_time) | Value of creep time at the end of the increment. |
| r_array(ir_creep_timeinc) | Time increment. |
| r_array(ir_creep_equiv_creep_strain) | Equivalent creep strain. |
| r_array(ir_creep_qtild) | Equivalent deviatoric Kirchhoff stress. |
| r_array(ir_creep_inv1crp) | The first invariant, $I_1^{cr}$, of the right Cauchy-Green creep strain tensor, $\mathbf{C}^{cr}$. |
| r_array(ir_creep_inv1) | The first invariant, $\overline{I_1}$, of the left Cauchy-Green strain tensor, $\overline{\mathbf{B}}$. |
| r_array(ir_creep_inv2) | The second invariant, $\overline{I_2}$, of the left Cauchy-Green strain tensor, $\overline{\mathbf{B}}$. |
| r_array(ir_creep_detf) | The determinant of the deformation gradient, $\mathbf{F}$. |
| r_array(ir_creep_press) | Kirchhoff pressure. |
| nrarray | Size of array r_array. |
| c_array(ic_creep_material_name) | User-specified material name, left justified. Some internal material models are given names starting with the "ABQ_" character string. To avoid conflict, you should not use "ABQ_" as the leading string for the material name. |
| ncarray | Size of array c_array. |

## Example: Bergstrom-Boyce Model

As an example of the coding of user subroutine *UCREEPNETWORK*, consider the Bergstrom-Boyce model. In this case the equivalent creep strain rate is expressed as (see *Parallel Rheological Framework*)

$$\dot{\tilde{\varepsilon}}^{cr} = A(\lambda^{cr} - 1 + E)^C(\tilde{q})^m,$$

where

$$\lambda^{cr} = \sqrt{\frac{1}{3}\mathbf{I} : \mathbf{C}^{cr}}$$

and

$\mathbf{C}^{cr}$

is the right Cauchy-Green creep strain tensor,

$\tilde{q}$

is the equivalent deviatoric Kirchhoff stress, and

$A$, $m$, $C$, and $E$

are material parameters.

The user subroutine would be coded as follows:

```
      subroutine ucreepnetwork (
C Must be updated
     *   outputData,
C Can be updated
     *   statev,
C Information (Read only)
     *   nOutput,
     *   nstatv,
     *   networkid,
     *   coords,
     *   temp,
     *   dtemp,
     *   nfield,
     *   predef,
     *   dpred,
     *   nprops,
     *   props,
     *   i_array,
     *   niarray,
     *   r_array,
     *   nrarray,
     *   c_array,
     *   ncarray)
C
      include 'aba_param.inc'
C
      parameter( io_creep_equiv_creepinc       = 1,
     *           io_creep_deqcreepinc_deqcreep = 2,
     *           io_creep_deqcreepinc_dqtild   = 3,
     *           io_creep_deqcreepinc_dinv1crp = 4,
     *           io_creep_deqcreepinc_dinv1    = 5,
     *           io_creep_deqcreepinc_dinv2    = 6,
     *           io_creep_deqcreepinc_ddetf    = 7,
     *           io_creep_deqcreepinc_dpress   = 8 )
C
      parameter( i_creep_kstep   = 1,
     *           i_creep_kinc    = 2,
     *           i_creep_noel    = 3,
     *           i_creep_npt     = 4,
     *           i_creep_layer   = 5,
     *           i_creep_kspt    = 6,
     *           i_creep_lend    = 7 )
C
```

```
      parameter( ir_creep_step_time  = 1,
     *           ir_creep_total_time = 2,
     *           ir_creep_creep_time = 3,
     *           ir_creep_timeinc    = 4,
     *           ir_creep_equiv_creep_strain = 5,
     *           ir_creep_qtild      = 6,
     *           ir_creep_inv1crp    = 7,
     *           ir_creep_inv1       = 8,
     *           ir_creep_inv2       = 9,
     *           ir_creep_detf       = 10,
     *           ir_creep_press      = 11 )
C
      parameter( ic_creep_material_name = 1 )
C
C model parameters
      parameter ( zero=0.0d0,half=0.5d0,one=1.0d0,two=2.0d0,
     &     three=3.0d0,five=5.0d0,six=6.0d0 )
C
      dimension
     *   statev(nstatv),
     *   predef(nfield),
     *   dpred(nfield),
     *   coords(*),
     *   props(nprops),
     *   outputData(nOutput),
     *   i_array(niarray),
     *   r_array(nrarray)

      character*80 c_array(ncarray)
C
C Bergstrom-Boyce Model
C
      A  = props(1)
      dm = props(2)
      C  = props(3)
      E  = props(4)
C
      dI1    = r_array(ir_creep_inv1crp)
      dLamb  = (dI1/three)**half
      sigmaB = r_array(ir_creep_qtild)
      dt     = r_array(ir_creep_timeinc)
C
C deq
      deq = dt*A*(dLamb-one+E)**C*sigmaB**dm
C
C d(deq)/(dI1crp)
      deqdi1 = deq*C/(dLamb-one+E)/dLamb/six
C
C d(eq)/d(eq)
      deqeq = zero
C
C d(eq)/d(q)
      deqdq = dm*dt*A*(dLamb-one+E)**C*sigmaB**(dm-one)
C
C set output
      outputData(io_creep_equiv_creepinc)       = deq
      outputData(io_creep_deqcreepinc_deqcreep) = deqeq
      outputData(io_creep_deqcreepinc_dqtild)   = deqdq
      outputData(io_creep_deqcreepinc_dinv1crp) = deqdi1
      outputData(io_creep_deqcreepinc_dinv1)    = zero
      outputData(io_creep_deqcreepinc_dinv2)    = zero
```

```
      outputData(io_creep_deqcreepinc_ddetf)    = zero
      outputData(io_creep_deqcreepinc_dpress)   = zero
C
      return
      end
```

# UCURE: User subroutine to define cure kinetics

**Products:** Abaqus/Standard

**References:**

- *Curing Processes in Polymers*
- **CURE GLASS TRANSITION TEMPERATURE*
- **CURE HEAT GENERATION*
- **CURE KINETICS*
- **CURE MAX CONVERSION*
- **CURE SHRINKAGE*

## Overview

User subroutine *UCURE*:

- can be used to define cure kinetics for the cure modeling capabilities;
- is called at all material points for elements for which the material definition includes user-defined cure kinetics;
- can use and update solution-dependent state variables; and
- requires that the degree of cure rate and its derivatives with respect to temperature and the degree of cure be defined.

## User Subroutine Interface

```
SUBROUTINE UCURE(
     *   DOC, DOCRATEARRAY, NOEL ,NPT, LAYER, KSPT, KSTEP, KINC, JPROC,
     *   CMNAME, NSTATV, STATEVOLD, STATEV, COORDS, TIME, DTIME, PNEWDT,
     *   TEMP, DTEMP, NFIELD, PREDEF, DPRED, NPROPS, PROPS,
     *   NIARRAY, I_ARRAY, NRARRAY, R_ARRAY, NCARRAY, C_ARRAY)
C
      INCLUDE 'ABA_PARAM.INC'
C
      DIMENSION
     *   DOCRATEARRAY(*),
     *   STATEVOLD(*),
     *   STATEV(*),
     *   COORDS(*),
     *   TIME(*),
     *   PREDEF(*),
     *   DPRED(*),
     *   PROPS(*),
     *   I_ARRAY(*),
     *   R_ARRAY(*)
C
      CHARACTER*80 C_ARRAY(*),CMNAME

      user coding to define DOCRATEARRAY, STATEV

      RETURN
      END
```

## Variables to Be Defined

DOCRATEARRAY(1) Rate of degree of cure, $\dot{\alpha}$.

DOCRATEARRAY(2) Derivative of the rate of degree of cure with respect to degree of cure, $\frac{\partial \dot{\alpha}}{\partial \alpha}$.

DOCRATEARRAY(3) Derivative of the rate of degree of cure with respect to temperature, $\frac{\partial \dot{\alpha}}{\partial \theta}$.

## Variables That Can Be Updated

STATEV Array containing the user-defined solution-dependent state variables at this point. These are supplied as values at the start of the increment or as values updated by other user subroutines (see *Allocating Space for Solution-Dependent State Variables*) and must be returned as values at the end of the increment.

## Variables Passed in for Information

DOC Degree of cure.

NOEL Element number.

NPT Integration point number.

LAYER Layer number (for composite shells and layered solids).

KSPT Section point number within the current layer.

KSTEP Step number.

KINC Increment number.

JPROC Procedure type key.

CMNAME User-specified material name, left justified.

NSTATV User-defined number of solution-dependent state variables associated with this material (see *Allocating Space for Solution-Dependent State Variables*).

STATEVOLD Array containing the user-defined solution-dependent state variables at the beginning of the increment.

COORDS Array containing the current coordinates at this point.

TIME(1) Value of step time at the end of the increment.

TIME(2) Value of total time at the end of the increment

TIME(3) Value of creep time at the end of the increment.

DTIME Time increment.

PNEWDT Ratio of suggested new time increment to the time increment being used (DTIME). This variable allows you to provide input to the automatic time incrementation algorithms in Abaqus/Standard if automatic time incrementation is chosen.

PNEWDT is set to a large value before each call to *UCURE*.

If PNEWDT is redefined to be less than 1.0, Abaqus/Standard must abandon the time increment and attempt it again with a smaller time increment. The suggested new time increment provided to the automatic time integration algorithms is PNEWDT × DTIME, where PNEWDT is the minimum value for all calls to user subroutines that allow redefinition of PNEWDT for this iteration.

| | |
|---|---|
| | If PNEWDT is given a value that is greater than 1.0 for all calls to user subroutines for this iteration and the increment converges in this iteration, Abaqus/Standard might increase the time increment. The suggested new time increment provided to the automatic time integration algorithms is PNEWDT × DTIME, where PNEWDT is the minimum value for all calls to user subroutines for this iteration.<br><br>If automatic time incrementation is not selected in the analysis procedure, values of PNEWDT that are greater than 1.0 will be ignored and values of PNEWDT that are less than 1.0 will cause the job to end. |
| TEMP | Temperature at the end of the increment. |
| DTEMP | Increment of temperature. |
| NFIELD | Number of field variables. |
| PREDEF | Array of interpolated values of predefined field variables at this material point at the end of the increment based on the values read in at the nodes (initial values at the beginning of the analysis and current values during the analysis). |
| DPRED | Array of increments of predefined field variables at this material point for this increment, including any values updated by user subroutine *USDFLD*. |
| NPROPS | Number of material properties entered for this user-defined cure kinetics model. |
| PROPS | Array of material properties entered for this user-defined cure kinetics model. |
| NIARRAY | Not used. |
| I_ARRAY | Not used. |
| NRARRAY | Not used. |
| | . |
| R_ARRAY | Not used. |
| NCARRAY | Not used. |
| C_ARRAY | Not used. |

## Example: Kamal Model

As an example of the coding of user subroutine *UCURE*, consider a special case of the Kamal model with one term and $m_1 = 0$, $n_1 = 1.0$, $b_1 = 0$, and $\alpha_{max} = 1.0$. In this case the Kamal equation has the form

$$\dot{\alpha} = Z_1 e^{-\frac{E_1}{R\left(\theta-\theta^Z\right)}} \left(1-\alpha\right),$$

where $\alpha$ denotes the degree of cure, $Z_1$ is a parameter, $E_1$ is the activation energy, $R$ is the universal gas constant, and $\theta^Z$ is the absolute zero temperature. The derivatives of the degree of cure rate with respect to temperature and with respect to degree of cure are given by the following relations:

$$\frac{\partial \dot{\alpha}}{\partial \theta} = \frac{E_1 \dot{\alpha}}{R\left(\theta-\theta^Z\right)^2}$$

and

$$\frac{\partial \dot{\alpha}}{\partial \alpha} = -Z_1 e^{-\frac{E_1}{R\left(\theta-\theta^Z\right)}}.$$

The user subroutine is coded as follows:

```
      SUBROUTINE UCURE(
     *   DOC, DOCRATEARRAY, NOEL, NPT, LAYER, KSPT, KSTEP, KINC, JPROC,
     *   CMNAME, NSTATV, STATEVOLD, STATEV,
```

```
     *   COORDS, TIME, DTIME, PNEWDT, TEMP, DTEMP, NFIELD, PREDEF,
     *   DPRED, NPROPS, PROPS, NIARRAY, I_ARRAY, NRARRAY, R_ARRAY, NCARRAY,
 C_ARRAY )
C
      INCLUDE 'ABA_PARAM.INC'
      INCLUDE 'ABA_PHCON_PARAM.INC'
C
      PARAMETER (SMALL=1.D-6, ZERO=0.0D0, HALF=0.5D0, ONE=1.0D0)
      DIMENSION
     *   DOCRATEARRAY(*),
     *   STATEVOLD(*),
     *   STATEV(*),
     *   COORDS(*),
     *   TIME(*),
     *   PREDEF(*),
     *   DPRED(*),
     *   PROPS(*),
     *   I_ARRAY(*),
     *   R_ARRAY(*)
C
      CHARACTER*80 C_ARRAY(*),CMNAME
C
c   Initialize variables
c
      jError = 0
      R = zero
      tempZero = zero
      alphaDot = zero
      dAlphaDotDT = zero
      dAlphaDotdAlpha = zero
C
C ... Read absolute zero temperature and universal gas constant
C
      call getphysicalconstant(j_phcon_Abs0, tempZero, jError)
      if (jError.ne.0) tempZero = zero
      call getphysicalconstant(j_phcon_UnivGas, R, jError)
      if (jError.ne.0) R = one
C
C ... Read material properties
C
      Zi = props(1)
      Ei = props(2)
      Ei = Ei/R
C
      tempDiff = max(zero,temp-tempZero)
      test = half - sign(half,tempDiff-small)
C
C ... Compute the degree of cure rate and its derivatives
C
      arrhenius = (one-test)*Zi*exp(-Ei/(test + tempDiff))
      alphaDot = arrhenius*(one-doc)
      dAlphaDotDT = alphaDot*Ei/(test + tempDiff)**2
      dAlphaDotdAlpha = -arrhenius
C
      DOCRATEARRAY(1) = alphaDot
      DOCRATEARRAY(2) = dAlphaDotdAlpha
      DOCRATEARRAY(3) = dAlphaDotDT
c
      RETURN
      END
```

# UDAMAGEMF: User subroutine to define damage evolution at the macro-level for a composite modeled with mean-field homogenization.

**Products:** Abaqus/Standard

**References:**

- *Obtaining Constituent-Level Material Point Information*
- **MEAN FIELD DAMAGE*
- *Composites with Damage*

## Overview

User subroutine UDAMAGEMF:

- can be used to define the overall damage variable, $D$, for a composite modeled with mean field homogenization;
- will be called at each integration point for elements with mean field damage defined as user-specified; and
- allows you to specify a separate damage variable, $d_i^{\text{int}}$, for each constituent, $i$, which is used to account for the additional contribution to the overall damage behavior due to interactions between the constituents.

## User Subroutine Interface

```
      SUBROUTINE UDAMAGEMF(SDEGMACRO,SDEGMICRO,NPHASES,CMNAME,CPHASENAME,
     1 NMICROI,NMICROR,IMICROPROPS,RMICROPROPS,STATEV,
     2 NSTATEV,PROPS,NPROPS,COORDS,NDI,NSHR,
     3 SECLAYO, JPHASE2MAT, JMAC, JMATYP, LACCFLG,
     4 NOEL, NPT, LAYER, KSPT,
     5 KSTEP, KINC, DT, TIME)
C
      INCLUDE 'ABA_PARAM.INC'
C
C     PROPERTIES IN IMICROPROPS
      PARAMETER (I_PHASETYPE=1, I_SHAPE=2)

C     PROPERTIES IN RMICROPROPS
      PARAMETER (I_VOLFRAC=1, I_AR=2)
C
C     PHASE TYPES
      PARAMETER (I_TYPE_MATRIX=1, I_TYPE_INCLUSION=2, I_TYPE_VOID=3)
C
C     SHAPE TYPES
      PARAMETER (I_SHAPE_SPHERE=1, I_SHAPE_CYLINDER=2, I_SHAPE_PENNY=3,
     1 I_SHAPE_OBLATE=4, I_SHAPE_PROLATE=5, I_SHAPE_ELLIPCYL=9)

      DIMENSION TIME(2), SDEGMICRO(NPHASES), JPHASE2MAT(NPHASES)
     DIMENSION IMICROPROPS(NMICROI, NPHASES), RMICROPROPS(NMICROR,NPHASES)

     1 STATEV(NSTATEV), PROPS(NPROPS), ARRAY(15), JARRAY(15), COORDS(*)
     2 JMAC(*), JMATYP(*)
      CHARACTER*3 FLGRAY(15)
      CHARACTER*80 CMNAME
      CHARACTER*80 CPHASENAME(NPHASES)
```

```
      user coding to define SDEGMICRO


      RETURN
      END
```

## Variables to Be Defined

| | |
|---|---|
| SDEGMACRO | The overall damage variable of the composite $\boldsymbol{D}$. |
| SDEGMICRO(NPHASES) | Array of damage variables $\boldsymbol{d}_i^{\mathbf{int}}$ in all constituents. The values are set to zero by default. |

## Variables That Can Be Updated

| | |
|---|---|
| STATEV(NSTATEV) | Array containing the solution-dependent state variables defined at the composite level. |

## Variables Passed in for Information

| | |
|---|---|
| NPHASES | Number of constituents in the composite. |
| CMNAME | Name of the composite material. |
| CPHASENAME | Character array containing the name of the constituents. |
| NMICROI | Number of integer value micro-level properties of each constituent, such as the shape of the constituent. |
| NMICROR | Number of real value micro-level properties of each constituent, such as the volume fraction of the constituent. |
| IMICROPROPS(NMICROI, NPHASES) | An integer array containing micro-level properties of the constituents. |
| RMICROPROPS(NMICROR, NPHASES) | A real array containing micro-level properties of the constituents. |
| NSTATEV | User-defined number of solution-dependent state variables. |
| PROPS(NPROPS) | User-specified array of parameters used to define macro-level damage. |
| COORDS | An array containing the coordinates of this point. These are the current coordinates if geometric nonlinearity is accounted for during the step; otherwise, the array contains the original coordinates of the point. |
| NDI | Number of direct stress components at this point. |
| NSHR | Number of shear stress components at this point. |
| SECLAYO | Variable that must be passed into the GETVRM_MF utility routine. The calling user subroutine provides this variable. |
| JPHASE2MAT | Variable that must be passed into the GETVRM_MF utility routine. The calling user subroutine provides this variable. |
| JMAC | Variable that must be passed into the GETVRM_MF utility routine. The calling user subroutine provides this variable. |
| JMATYP | Variable that must be passed into the GETVRM_MF utility routine. The calling user subroutine provides this variable. |
| SECLAYO | Variable that must be passed into the GETVRM_MF utility routine. The calling user subroutine provides this variable. |
| LACCFLG | Variable that must be passed into the GETVRM_MF utility routine. The calling user subroutine provides this variable. |

| | |
|---|---|
| NOEL | Element number. |
| NPT | Integration point number. |
| LAYER | Layer number (for composite shells and layered solids). |
| KSPT | Section point number within the current layer. |
| KSTEP | Step number. |
| KINC | Increment number. |
| DT | Time increment. |
| TIME(1) | Value of step time at the beginning of the current increment. |
| TIME(2) | Value of total time at the beginning of the current increment. |

## Example: User-Defined Damage Model

As an example of user subroutine *UDAMAGEMF*, consider a composite material with macro-level damage defined as follows (damage is also defined in the matrix material):

```
*MATERIAL, NAME=MAT
         *MEAN FIELD HOMOGENIZATION, FORMULATION=MT, UNIFORM MATRIX
STRAIN=YES
         *MEAN FIELD DAMAGE, DEGRADATION=USER, PROPERTIES=2
          0.2, 0.05
         *CONSTITUENT, MATERIAL=EPOXY, TYPE=MATRIX, NAME=MT_MATRIX
         *CONSTITUENT, MATERIAL=GLASS, TYPE=INCLUSION, NAME=MT1_GLASS,
SHAPE=CYLINDER
          0.3,
         *CONSTITUENT, MATERIAL=GLASS, TYPE=INCLUSION, NAME=MT2_GLASS,
SHAPE=SPHERE
          0.1,
```

A user-specified damage model is defined to take into account the additional damage in the inclusions caused by the damage in the matrix material. Twenty percent additional damage is added to the constituent with the cylinder shape, and five percent additional damage is added to the constituent with the name MT2_GLASS. These values are defined in the **MEAN FIELD DAMAGE* data line.

```
      SUBROUTINE UDAMAGEMF(SDEGMACRO,SDEGMICRO, NPHASES, CMNAME,CPHASENAME,

     1 NMICROI, NMICROR, IMICROPROPS, RMICROPROPS, STATEV,
     2 NSTATEV, PROPS, NPROPS, COORDS, NDI, NSHR,
     3 SECLAYO, JPHASE2MAT, JMAC, JMATYP, LACCFLG,
     4 NOEL, NPT, LAYER, KSPT,
     5 KSTEP, KINC, DT, TIME)
C
      INCLUDE 'ABA_PARAM.INC'
C
      PARAMETER (I_PHASETYPE=1, I_SHAPE=2)
      PARAMETER (I_VOLFRAC=1, I_AR=2)
C
C     PHASE TYPES
      PARAMETER (I_TYPE_MATRIX=1, I_TYPE_INCLUSION=2, I_TYPE_VOID=3)
C
C     SHAPE TYPES
      PARAMETER (I_SHAPE_SPHERE=1, I_SHAPE_CYLINDER=2, I_SHAPE_PENNY=3,
     1 ISHAPE_OBLATE=4, I_SHAPE_PROLATE=5 ,I_SHAPE_ELLIPCYL=9)
C
      DIMENSION TIME(2), SDEGMICRO(NPHASES), JPHASE2MAT(NPHASES)
     DIMENSION IMICROPROPS(NMICROI, NPHASES), RMICROPROPS(NMICROR, NPHASES)
```

```
     1 STATEV(NSTATEV), PROPS(NPROPS), ARRAY(15), JARRAY(15), COORDS(*)
     2 JMAC(*), JMATYP(*)
      CHARACTER*3 FLGRAY(15)
      CHARACTER*80 CMNAME
      CHARACTER*80 CPHASENAME(NPHASES)

      PARAMETER (ZERO=0.0)
C
C     GET OVERALL DAMAGE VARIABLE IN THE MATRIX
C
      SDEGMATRIX=ZERO
      DO KPHASE=1,NPHASES
         IF (IMICROPROPS(I_PHASETYPE,KPHASE).eq.I_TYPE_MATRIX) then
            CALL GETVRM_MF('SDEG', KPHASE, JPHASE2MAT,
     1           ARRAY, JARRAY, FLGRAY, JRCD,JMAC, JMATYP,
     2           SECLAYO, LACCFLG)
            IF(JRCD.EQ.0)SDEGMATRIX=ARRAY(1)
         END IF
      END DO
C
C     SPECIFY ADDITIONAL CONTRIBUTION FROM EACH CONSTITUENT
C     USING A FRACTION OF THE MATRIX DEGRADATION FACTOR.
C
      SC1=PROPS(1)
      SC2=PROPS(2)
      DO KPHASE=1, NPHASES
         SC=ZERO
         IF( IMICROPROPS(I_SHAPE,KPHASE).eq.I_SHAPE_CYLINDER ) THEN
             SC=SC1
         ELSE IF( CPHASENAME(KPHASE)(1:3).eq.'MT2' ) THEN
             SC=SC2
         END IF
         SDEGMICRO(KPHASE)=SC*SDEGMATRIX
      END DO

      RETURN
      END
```

# UDECURRENT: User subroutine to define nonuniform volume current density in an eddy current or magnetostatic analysis.

**Products:** Abaqus/Standard

**References:**

- *Eddy Current Analysis*
- *Magnetostatic Analysis*
- **DECURRENT*

## Overview

User subroutine *UDECURRENT*:

- can be used to define the variation of volume current density vector as a function of position, time, element number, etc. for a transient eddy current or magnetostatic analysis or as a function of position, excitation frequency, phase, element number, etc. for a time-harmonic eddy current analysis;
- will be called at each load integration point for each element-based nonuniform volume current density definition during eddy current or magnetostatic analysis; and
- ignores any amplitude references that may appear with the associated step definition or nonuniform distributed volume current density definition.

## User Subroutine Interface

```
      subroutine udecurrent (
C Write only -
     *   bodycurrent,
C Read only -
     *   predef, coords, nBlock,
     *   i_array, niarray,
     *   r_array, nrarray,
     *   c_array, ncarray )
C
      include 'aba_param.inc'
C
      dimension bodycurrent(nBlock,*),
     *   predef(nBlock,2,*),
     *   coords(nBlock,*),
     *   i_array(*),
     *   r_array(*)
C
      character*80 c_array(*)
C
      parameter( i_udecurr_kstep   = 1,
     *           i_udecurr_kinc    = 2,
     *           i_udecurr_noel    = 3,
     *           i_udecurr_npt     = 4,
     *           i_udecurr_jltyp   = 5,
     *           i_udecurr_phase   = 6,
     *           i_udecurr_proc    = 7,
     *           i_udecurr_nfld    = 8 )
C
```

```
      parameter( ir_udecurr_time_1 = 1,
     *           ir_udecurr_time_2 = 2,
     *           ir_udecurr_time_3 = 3 )
C
      parameter( i_jltyp_cj = 1 )
C
      parameter( i_proc_lf_th = 1,
     *           i_proc_lf_td = 2,
     *           i_proc_ms    = 3 )
C
      parameter( i_udecurr_phase_real   = 1,
     *           i_udecurr_phase_imag   = 2 )
C
      user coding to define bodycurrent
      return
      end
```

## Variables to Be Defined

`bodycurrent(nBlock,*)` Components of the body current density vector for a block of load integration points. The units are $CL^{-2}T^{-1}$. `bodycurrent` will be passed into the routine as the vector specified as part of the element-based distributed volume current density definition. If the vector is not defined, `bodycurrent` will be passed in as zero.

## Variables Passed in for Information

`predef(2,*)` An array containing values of temperature and all the predefined field variables at the current load integration point, based on interpolation from the values specified at the nodes. The first value in a pair, `predef(1,*)`, corresponds to initial values; the second value, `predef(2,*)`, corresponds to incremental values of the temperature and field variables. The first entry (for example, `predef(1,1)` or `predef(2,1)`) contains the temperature; the subsequent entries (for example, `predef(1,2)` or `predef(2,2)` onward) contain the field variables.

`coords(nBlock,*)` An array containing the coordinates of the load integration points.

`nBlock` Number of load integration points in this block. Currently equal to 1.

`i_array(i_udecurr_kstep)` Step number.

`i_array(i_udecurr_kinc)` Increment number.

`i_array(i_udecurr_noel)` Element number.

`i_array(i_udecurr_npt)` Load integration point number.

`i_array(i_udecurr_jltyp)` Currently equal to 1.

`i_array(i_udecurr_phase)` This value is relevant only for a time-harmonic eddy current analysis and is either 1 (`i_udecurr_phase_real`) or 2 (`i_udecurr_phase_imag`), depending on whether the current call to the user subroutine defines the real (in-phase) or the imaginary (out-of-phase) part of the volume current density vector.

`i_array(i_udecurr_proc)` Equal to 1 for a time-harmonic eddy current procedure, 2 for a transient eddy current procedure, and 3 for a magnetostatic procedure.

`i_array(i_udecurr_nfld)` Total number of predefined field variables.

`niarray` Size of array `i_array`. Currently equal to 8.

`r_array(ir_udecurr_time_1)` Excitation frequency in cycles/time for a time-harmonic eddy current analysis; alternatively, the value of step time at the beginning of the current increment for a transient eddy current or magnetostatic analysis.

`r_array(ir_udecurr_time_2)` Excitation frequency in radians/time for a time-harmonic eddy current analysis; alternatively, the value of total time at the beginning of the current increment for a transient eddy current or magnetostatic analysis.

`r_array(ir_udecurr_time_3)` Time increment for a transient eddy current or magnetostatic analysis.

`nrarray` Size of array `r_array`. Currently equal to 3.

`c_array(1)` Not used.

`ncarray` Size of array `c_array(1)`. Currently equal to 1.

# UDEMPOTENTIAL: User subroutine to define nonuniform magnetic vector potential on a surface in an eddy current or magnetostatic analysis.

**Products:** Abaqus/Standard

**References:**

- *Eddy Current Analysis*
- *Magnetostatic Analysis*
- **D EM POTENTIAL*

## Overview

User subroutine *UDEMPOTENTIAL*:

- can be used to define the variation of the magnetic vector potential as a function of position, time, element number, etc. for a transient eddy current or magnetostatic analysis or as a function of position, excitation frequency, phase, element number, etc. for a time-harmonic eddy current analysis;
- will be called for each surface-based nonuniform electromagnetic potential definition during eddy current or magnetostatic analysis; and
- ignores any amplitude references that may appear with the associated step definition or nonuniform distributed electromagnetic potential definition.

## User Subroutine Interface

```
      subroutine udempotential (
C Write only -
     *   vecPot,
C Read only -
     *   coords, nBlock,
     *   i_array, niarray,
     *   r_array, nrarray,
     *   c_array, ncarray )
C
      include 'aba_param.inc'
C
      dimension vecPot(nBlock,*),
     *   coords(nBlock,*),
     *   i_array(*),
     *   r_array(*)
C
      character*80 c_array(*)
C
      parameter( i_udempot_kstep   = 1,
     *           i_udempot_kinc    = 2,
     *           i_udempot_noel    = 3,
     *           i_udempot_currtyp = 4,
     *           i_udempot_phase   = 5,
     *           i_udempot_proc    = 6)
C
      parameter( ir_udempot_time_1 = 1,
     *           ir_udempot_time_2 = 2,
     *           ir_udempot_time_3 = 3 )
```

```
C
      parameter( ic_udempot_surf = 1)
C
      parameter( i_pottyp_mvp = 1 )
C
      parameter( i_proc_lf_th = 1,
     *           i_proc_lf_td = 2,
     *           i_proc_ms    = 3 )
C
      parameter( i_udempot_phase_real   = 1,
     *           i_udempot_phase_imag   = 2 )
C

      user coding to define vecPot

      return
      end
```

## Variables to Be Defined

`vecPot(nBlock,*)` Components of the magnetic vector potential at a block of surface points. `vecPot` will be passed into the routine as the vector specified as part of the surface-based nonuniform magnetic vector potential definition. If the vector is not defined, `vecPot` will be passed in as zero.

## Variables Passed in for Information

| | |
|---|---|
| `coords(nBlock,*)` | An array containing the coordinates of a block of surface points. |
| `nBlock` | Number of surface points in this block. Currently equal to 1. |
| `i_array(i_udempot_kstep)` | Step number. |
| `i_array(i_udempot_kinc)` | Increment number. |
| `i_array(i_udempot_noel)` | Element number. |
| `i_array(i_udempot_pottyp)` | Currently equal to 1. |
| `i_array(i_udempot_phase)` | This value is relevant only for a time-harmonic eddy current analysis and is either 1 (`i_udempot_phase_real`) or 2 (`i_udempot_phase_imag`), depending on whether the current call to the user subroutine defines the real (in-phase) or the imaginary (out-of-phase) part of the magnetic vector potential. |
| `i_array(i_udempot_proc)` | Equal to 1 for a time-harmonic eddy current procedure, 2 for a transient eddy current procedure, and 3 for a magnetostatic procedure. |
| `niarray` | Size of array `i_array`. Currently equal to 6. |
| `r_array(ir_udempot_time_1)` | Excitation frequency in cycles/time for a time-harmonic eddy current analysis; alternatively, the value of step time at the beginning of the current increment for a transient eddy current or magnetostatic analysis. |
| `r_array(ir_udempot_time_2)` | Excitation frequency in radians/time for a time-harmonic eddy current analysis; alternatively, the value of total time at the beginning of the current increment for a transient eddy current or magnetostatic analysis. |
| `r_array(ir_udempot_time_3)` | Time increment for a transient eddy current or magnetostatic analysis. |
| `nrarray` | Size of array `r_array`. Currently equal to 3. |
| `c_array(ic_udempot_surf)` | Surface name. |
| `ncarray` | Size of array `c_array`. Currently equal to 1. |

# UDMGINI: User subroutine to define the damage initiation criterion.

**Products:** Abaqus/Standard

**References:**

- *About Progressive Damage and Failure*
- *Modeling Discontinuities as an Enriched Feature Using the Extended Finite Element Method*
- **DAMAGE INITIATION*

## Overview

User subroutine *UDMGINI*:

- can be used to specify a user-defined damage initiation criterion;
- allows the specification of more than one failure mechanism in an element, with the most severe one governing the actual failure;
- can be used in combination with several Abaqus built-in damage evolution models, with each model corresponding to a particular failure mechanism;
- will be called at all integration points of elements for which the material definition contains user-defined damage initiation criterion;
- can call utility routine GETVRM to access material point data; and
- is currently available only for enriched elements.

## User Subroutine Interface

```
      SUBROUTINE UDMGINI(FINDEX,NFINDEX,FNORMAL,NDI,NSHR,NTENS,PROPS,
     1 NPROPS,STATEV,NSTATV,STRESS,STRAIN,STRAINEE,LXFEM,TIME,
     2 DTIME,TEMP,DTEMP,PREDEF,DPRED,NFIELD,COORDS,NOEL,NPT,LAYER,
     3 KSPT,KSTEP,KINC,KDIRCYC,KCYCLELCF,TIMECYC,SSE,SPD,SCD,SVD,
     4 SMD,JMAC,JMATYP,MATLAYO,LACCFLA,CELENT,DROT,ORI)
C
      INCLUDE 'ABA_PARAM.INC'

C
      DIMENSION FINDEX(NFINDEX),FNORMAL(NDI,NFINDEX),COORDS(*),
     1 STRESS(NTENS),STRAIN(NTENS),STRAINEE(NTENS),PROPS(NPROPS),
     2 STATEV(NSTATV),PREDEF(NFIELD),DPRED(NFIELD),TIME(2),JMAC(*),
     3 JMATYP(*),DROT(3,3),ORI(3,3),LXFEM(3)

      user coding to define FINDEX,  and  FNORMAL

      RETURN
      END
```

## Variables to Be Defined

| | |
|---|---|
| FINDEX(NFINDEX) | A Vector defining the indices for all the failure mechanisms. |
| FNORMAL(NDI, NFINDEX) | An Array defining the normal direction to the fracture plane (three dimensions) or line (two dimensions) for each failure mechanism. |

## Variables That Can Be Updated

`STATEV` An array containing the user-defined solution-dependent state variables at this point. This array will be passed in containing the values of these variables at the start of the increment unless the values are updated in user subroutine *USDFLD*. They can be updated in this subroutine to their values at the end of the increment. You define the size of this array by allocating space for it (see *Allocating Space for Solution-Dependent State Variables* for more information).

`SSE, SPD, SCD, SVD, SMD` Specific elastic strain energy, plastic dissipation, "creep" dissipation, viscous, and damage energy, respectively, passed in as the values at the start of the increment and should be updated to the corresponding specific energy values at the end of the increment. They have no effect on the solution, except that they are used for energy output.

## Variables Passed in for Information

`NFINDEX` Number of indices for all failure mechanisms.

`NDI` Number of direct stress components at this point.

`NSHR` Number of engineering shear stress components at this point.

`NTENS` Size of the stress or strain component array (`NRI` + `NSHR`).

`PROPS(NPROPS)` User-specified array of material constants associated with this user-defined failure criterion.

`NPROPS` User-defined number of material constants associated with this user-defined failure criterion.

`NSTATV` Number of solution-dependent state variables associated with this material (specified when space is allocated for the array; see *Allocating Space for Solution-Dependent State Variables*).

`STRESS(NTENS)` An Array passed in as the current stress tensor. If a local orientation is used at the same point as user subroutine *UDMGINI*, the stress components will be in the local orientation; in the case of finite-strain analysis, the basis system in which stress components are stored rotates with the material.

`STRAIN(NTENS)` An Array containing the current total strains. If a local orientation is used at the same point as user subroutine *UDMGINI*, the strain components will be in the local orientation; in the case of finite-strain analysis, the basis system in which strain components are stored rotates with the material.

`STRAINEE(NTENS)` An Array containing the current elastic strains. If a local orientation is used at the same point as user subroutine *UDMGINI*, the elastic strain components will be in the local orientation; in the case of finite-strain analysis, the basis system in which elastic strain components are stored rotates with the material.

`LXFEM(1)` An integer flag to indicate an enriched element.

`LXFEM(2)` An integer flag to indicate an enriched zone.

`LXFEM(3)` An integer flag to indicate the number of cracks initiated in an enriched zone.

`TIME(1)` Value of step time at the beginning of the current increment.

`TIME(2)` Value of total time at the beginning of the current increment.

`DTIME` Time increment.

`TEMP` Temperature at the start of the increment.

`DTEMP` Increment of temperature during the time increment.

| | |
|---|---|
| PREDEF | An array containing the values of all of the user-specified predefined variables at this point at the start of the increment. |
| DPRED | An array containing the increments of all of the predefined variables during the time increment. |
| NFIELD | Number of user-specified predefined variables. |
| COORDS | An array containing the current coordinates of this point. |
| NOEL | Element number. |
| NPT | Integration point number. |
| LAYER | Layer number (for composite shells and layered solids). |
| KSPT | Section point number within the current layer. |
| KSTEP | Step number. |
| KINC | Increment number. |
| KDIRCYC | Iteration number in a direct cyclic analysis. |
| KCYCLELCF | Cycle number in a direct cyclic low-cycle fatigue analysis. |
| TIMECYC | Time period in one loading cycle in a direct cyclic analysis. |
| JMAC | Variable that must be passed into the GETVRM utility routine to access a material point variable. |
| JMATYP | Variable that must be passed into the GETVRM utility routine to access a material point variable. |
| MATLAYO | Variable that must be passed into the GETVRM utility routine to access a material point variable. |
| LACCFLA | Variable that must be passed into the GETVRM utility routine to access a material point variable. |
| CELENT | Characteristic element length, which is a typical length of a line across an element for a first-order element; it is half of the same typical length for a second-order element. For beams and trusses it is a characteristic length along the element axis. For membranes and shells it is a characteristic length in the reference surface. For axisymmetric elements it is a characteristic length in the ($r$, $z$) plane only. For cohesive elements it is equal to the constitutive thickness. |
| DROT(3,3) | Rotation increment matrix. This matrix represents the increment of rigid body rotation of the basis system in which the components of stress (STRESS) and strain (STRAIN) are stored. It is provided so that vector- or tensor-valued state variables can be rotated appropriately in this subroutine: stress and strain components are already rotated by this amount before *UDMGINI* is called. This matrix is passed in as a unit matrix for small-displacement analysis and for large-displacement analysis if the basis system for the material point rotates with the material (as in a shell element or when a local orientation is used). |
| ORI(3,3) | Material orientation with respect to global basis. |

## Example: User-Defined Damage Initiation Criterion with Two Different Failure Mechanisms

As a simple example of the coding of user subroutine *UDMGINI*, consider a damage initiation criterion based on two different failure mechanisms: the maximum principal stress and the quadratic traction-interaction.

```
      SUBROUTINE UDMGINI(FINDEX,NFINDEX,FNORMAL,NDI,NSHR,NTENS,PROPS,
     1 NPROPS,STATEV,NSTATV,STRESS,STRAIN,STRAINEE,LXFEM,TIME,
     2 DTIME,TEMP,DTEMP,PREDEF,DPRED,NFIELD,COORDS,NOEL,NPT,
     3 KLAYER,KSPT,KSTEP,INC,KDIRCYC,KCYCLELCF,TIMECYC,SSE,SPD,
     4 SCD,SVD,SMD,JMAC,JMATYP,MATLAYO,LACCFLA,CELENT,DROT,ORI)
C
      INCLUDE 'ABA_PARAM.INC'
CC
```

```
      DIMENSION FINDEX(NFINDEX),FNORMAL(NDI,NFINDEX),COORDS(*),
     1 STRESS(NTENS),STRAIN(NTENS),STRAINEE(NTENS),PROPS(NPROPS),
     2 STATEV(NSTATV),PREDEF(NFIELD),DPRED(NFIELD),TIME(2),
     3 JMAC(*),JMATYP(*),DROT(3,3),ORI(3,3),LXFEM(3)

      DIMENSION PS(3), AN(3,3), WT(6)
      PS(1)=0.0
      PS(2)=0.0
      PS(3)=0.0
C
C ROTATE THE STRESS TO GLOBAL SYSTEM IF THERE IS ORIENTATION
C
      CALL ROTSIG(STRESS,ORI,WT,1,NDI,NSHR)
C
C MAXIMUM PRINCIPAL STRESS CRITERION
C
      CALL SPRIND(WT,PS,AN,1,NDI,NSHR)
      SIG1 = PS(1)
      KMAX=1
      DO K1 = 2, NDI
         IF(PS(K1).GT.SIG1) THEN
            SIG1 = PS(K1)
            KMAX = K1
         END IF
      END DO
      FINDEX(1) = SIG1/PROPS(1)
      DO K1=1, NDI
       FNORMAL(K1,1) = AN(KMAX,K1)
      END DO
C
C QUADRATIC TRACTION-INTERACTION CRITERION
c
      FINDEX(2)=(STRESS(1)/PROPS(2))**2.0+(STRESS(NDI+1)/
     $     PROPS(3))**2.0+(STRESS(NDI+2)/PROPS(4))**2.0
C
      FINDEX(2)=sqrt(FINDEX(2))
C
      DO K1=1, NDI
       FNORMAL(K1,2)=ORI(K1,1)
      END DO
      RETURN
      END
```

# UDSECURRENT: User subroutine to define nonuniform surface current density in an eddy current or magnetostatic analysis.

**Products:** Abaqus/Standard

**References:**

- *Eddy Current Analysis*
- *Magnetostatic Analysis*
- **DSECURRENT*

## Overview

User subroutine *UDSECURRENT*:

- can be used to define the variation of surface current density vector as a function of position, time, element number, load integration point number, etc. for a transient eddy current or magnetostatic analysis or as a function of position, excitation frequency, phase, element number, load integration point number, etc. for a time-harmonic eddy current analysis;
- will be called at each surface load integration point for each nonuniform surface current density definition during eddy current and magnetostatic analyses; and
- ignores any amplitude references that may appear with the associated step definition or nonuniform distributed surface current density definition.

## User Subroutine Interface

```
      subroutine udsecurrent (
C Write only -
     *   surfacecurrent,
C Read only -
     *   coords, nBlock,
     *   i_array, niarray,
     *   r_array, nrarray,
     *   c_array, ncarray )
C
      include 'aba_param.inc'
C
      dimension surfacecurrent(nBlock,*),
     *   coords(nBlock,*),
     *   i_array(*),
     *   r_array(*)
C
      character*80 c_array(*)
C
      parameter( i_udsecurr_kstep   = 1,
     *           i_udsecurr_kinc    = 2,
     *           i_udsecurr_noel    = 3,
     *           i_udsecurr_currtyp = 4,
     *           i_udsecurr_phase   = 5,
     *           i_udsecurr_proc    = 6)
C
      parameter( ir_udsecurr_time_1 = 1,
     *           ir_udsecurr_time_2 = 2,
```

```
     *           ir_udsecurr_time_3 = 3 )
C
      parameter( ic_udsecurr_surf = 1)
C
      parameter( i_currtyp_tangential = 1 )
C
      parameter( i_proc_lf_th = 1,
     *           i_proc_lf_td = 2,
     *           i_proc_ms    = 3 )
C
      parameter( i_udsecurr_phase_real   = 1,
     *           i_udsecurr_phase_imag   = 2 )
C

      user coding to define surfacecurrent

      return
      end
```

## Variables to Be Defined

`surfacecurrent(nBlock,*)` Components of the surface current density vector at a block of surface integration points. The units are $CL^{-1}T^{-1}$. `surfacecurrent` will be passed into the routine as the vector specified as part of the surface-based distributed surface current density definition. If the vector is not defined, `surfacecurrent` will be passed in as zero.

## Variables Passed in for Information

| | |
|---|---|
| `coords(nBlock,*)` | An array containing the coordinates of a block of surface load integration points. |
| `nBlock` | Number of surface integration points in this block. Currently equal to 1. |
| `i_array(i_udsecurr_kstep)` | Step number. |
| `i_array(i_udsecurr_kinc)` | Increment number. |
| `i_array(i_udsecurr_noel)` | Element number. |
| `i_array(i_udsecurr_currtyp)` | Currently equal to 1. |
| `i_array(i_udsecurr_phase)` | This value is relevant only for a time-harmonic eddy current analysis and is either 1 (`i_udsecurr_phase_real`) or 2 (`i_udsecurr_phase_imag`), depending on whether the current call to the user subroutine defines the real (in-phase) or the imaginary (out-of-phase) part of the surface current density vector. |
| `i_array(i_udsecurr_proc)` | Equal to 1 for a time-harmonic eddy current procedure, 2 for a transient eddy current procedure, and 3 for a magnetostatic procedure. |
| `niarray` | Size of array `i_array`. Currently equal to 6. |
| `r_array(ir_udsecurr_time_1)` | Excitation frequency in cycles/time for a time-harmonic eddy current analysis; alternatively, the value of step time at the beginning of the current increment for a transient eddy current or magnetostatic analysis. |
| `r_array(ir_udsecurr_time_2)` | Excitation frequency in radians/time for a time-harmonic eddy current analysis; alternatively, the value of total time at the beginning of the current increment for a transient eddy current or magnetostatic analysis. |
| `r_array(ir_udsecurr_time_3)` | Time increment for a transient eddy current or magnetostatic analysis. |

| | |
|---|---|
| `nrarray` | Size of array `r_array`. Currently equal to 3. |
| `c_array(ic_udsecurr_surf)` | Surface name. |
| `ncarray` | Size of array `c_array`. Currently equal to 1. |

# UECHEMECDBV: User subroutine to define the Butler-Volmer intercalation current to be used for elements that utilize porous electrode theory in an electrochemical analysis.

**Products:** Abaqus/Standard

**References:**

## Overview

User subroutine *UECHEMECDBV*:

- is called for each material integration point for each iteration;
- is called for each microscale particle that is defined at the active electrodes during an electrochemical analysis;
- can be used to define the Butler-Volmer intercalation current (see *Intercalation at the Solid-Liquid Interface*); and
- can be used only in procedures that support electrochemical analyses (see *Electrochemical Analysis*).

## User Subroutine Interface

```
 subroutine uechemecdbv(
     *     ecdbv,
     *     dEcdbv_dCSurfNorm, dEcdbv_dT, dEcdbv_dSol,dEcdbv_dPress,
     *     overpoteta_bv,
     *     RSEI_BV,chargeDischarge,
     *     jmac,jMatyp,matlayo,laccFlag,
     *     cmname, cParticleName,
     *     kstep,lFlags, kinc, time, dtime, noel, npt,
     *     ntens, ndi, nshr, props, nprops,
     *     predef, dpredef, npredef,
     *     statev, statevOld, nstatev,
     *     Temp, dTemp, Press, coords,
     *     solutionvar, dsolutionvar, nSolutionVar,
     *     cSurfNorm, cAvgNorm, OCPBV, dOCPBVdCsurfNorm, ecdTotalOld,
     *     SEILPLFilmThick,
     *     i_array,niarray,r_array,nrarray,c_array,ncarray)
C
      include 'aba_param.inc'
c Include below file to access property and parameter table.
      include 'aba_tcs_param.inc'
c Include below file to access universal constants
      include 'aba_phcon_param.inc'
c Include below file for parametrized location of various solution variables.

      include 'aba_solutionvarloc.inc'
C
      dimension i_array(*), r_array(*), jmatyp(*),jmac(*), lFlags(*)
      dimension time(2), props(nprops), predef(npredef), dpredef(npredef),

     *     statev(nstatev), statevold(nstatev), coords(*),
     *     solutionvar(nsolutionvar), dsolutionvar(nsolutionvar),
     *     dEcdbv_dSol(*)
      character*80   cmname, c_array(NCARRAY), cParticleName
```

```
      user coding

      return
      end
```

## Variables to Be Defined

| | |
|---|---|
| `ecdbv` | Value of the Butler-Volmer current density. |
| `dEcdbv_dCSurfNorm` | Variation of the Butler-Volmer current density `ecdbv` with respect to normalized particle surface concentration `CSurfNorm`. |
| `dEcdbv_dT` | Variation of the Butler-Volmer current density `ecdbv` with respect to temperature `temp`. |
| `dEcdbv_dSol` | Variation of the Butler-Volmer current density `ecdbv` with respect to the different solution variables `solutionvar`. |
| `dEcdbv_dPress` | Variation of the Butler-Volmer current density `ecdbv` with respect to the macroscale hydrostatic pressure `press`. |
| `overpoteta_bv` | Overpotential for Butler-Volmer current computation. |

## Variables That Can Be Updated

| | |
|---|---|
| `RSEI_bv` | Resistance of the solid electrolyte interface layer. |

## Variables Passed in for Information

| | |
|---|---|
| `chargeDischarge` | This variable will have a value that you can use to identify whether the current step is a charging step or a discharging step. The value that is passed in is defined in the property table *ABQ_EChemPET_Electrolyte_ChargeDischarge*. |
| `jMac` | Variable that must be passed into utility routine *GETVRM*. The calling user subroutine provides this variable. |
| `jMatyp` | Variable that must be passed into utility routine *GETVRM*. The calling user subroutine provides this variable. |
| `matlayo` | Variable that must be passed into utility routine *GETVRM*. The calling user subroutine provides this variable. |
| `lAccFlag` | Variable that must be passed into utility routine *GETVRM*. The calling user subroutine provides this variable. |
| `cmName` | User-defined material name, left justified. Some internal material models are given names starting with the “ABQ_” character string. To avoid conflict, you should not use “ABQ_” as the leading string for `CMNAME`. |
| `cParticleName` | User-defined particle name, left justified. |
| `kstep` | Current step number. |
| `lFlags` | An array containing the flags that define the current solution procedure and analysis type. |

| | |
|---|---|
| `lFlags(1)` | Defines the procedure type. See *Results File* for the key used for each procedure. |
| `lFlags(2)=0` | Small-displacement analysis. |
| `lFlags(2)=1` | Large-displacement analysis (nonlinear geometric effects included in the step; see *General and Perturbation Procedures*. |

| | |
|---|---|
| `kinc` | Current increment number. |

`time(1)` Current value of step time.

`time(2)` Current value of total time.

`dtime` Time increment.

`noel` User-assigned element number.

`npt` Integration point number.

`ntens` Size of the stress or strain component array.

`ndi` Number of direct stress components at this point.

`nshr` Number of engineering shear stress components at this point.

`props` User-specified array of material constants associated with this user material.

`npros` User-defined number of material constants associated with this user material.

`predef` Array of interpolated values of predefined field variables at this point at the start of the increment, based on the values read in at the nodes.

`dpredef` Array of increments of predefined field variables.

`npredef` Number of predefined field variables.

`statev` Current value of the solution-dependent variables

`statevold` Array containing the user-defined solution-dependent state variables at the end of the previous increment.

`nstatev` Number of solution-dependent state variables that are associated with this material type (defined as described in *Allocating Space for Solution-Dependent State Variables*) .

`Temp` Current temperature.

`dTemp` Incremental temperature.

`press` Current hydrostatic pressure.

`coords` An array containing the current coordinates of this integration point.

`solutionVar` Current value of the solution variables.

| | |
|---|---|
| `solutionVar(i_sol_phiS)` | Current value of the electric potential. |
| `solutionVar(i_sol_phiE)` | Current value of the electric potential in the electrolyte. |
| `solutionVar(i_sol_Ce)` | Current value of the ionic concentration in the electrolyte. |
| `solutionVar(i_sol_porP)` | Current value of the pore pressure. |

`dsolutionVar` Incremental value of the solution variables.

| | |
|---|---|
| `dsolutionVar(i_sol_phiS)` | Incremental value of the electric potential. |
| `dsolutionVar(i_sol_phiE)` | Incremental value of the electric potential in the electrolyte. |
| `dsolutionVar(i_sol_Ce)` | Incremental value of the ionic concentration in the electrolyte. |
| `dsolutionVar(i_sol_porP)` | Incremental value of the pore pressure. |

`nsolutionVar` Number of solution variables.

`cSurfNorm` Current normalized concentration on the surface of the particle.

`cAvgNorm` Current average normalized concentration in the particle.

`ocpbv` Current open circuit potential value for Butler-Volmer current computation.

| | |
|---|---|
| `dOcpbvDcsurfNorm` | Variation of the Open Circuit Potential OCP with respect to normalized particle surface concentration `CSurfNorm`. |
| `ecdTotalOld` | Value of the total electrical current density at the end of the previous increment. |
| `seilplfilmthick` | Thickness of the combined solid electrolyte interface and plated lithium film layer. |
| `i_array` | Not used. |
| `niarray` | Size of array `i_array`. |
| `r_array` | Not used. |
| `nrarray` | Size of array `r_array`. |
| `c_array` | Not used. |
| `ncarray` | Size of array `c_array`. |

# UECHEMECDDISS: User subroutine to define the electrode dissolution aging current to be used for elements that utilize porous electrode theory in an electrochemical analysis.

**Products:** Abaqus/Standard

## Overview

User subroutine *UECHEMECDDISS*:

- is called for each material integration point for each iteration;
- is called for each microscale particle that is defined at the active electrodes during an electrochemical analysis;
- can be used to define the dissolution current to model aging in batteries (see *Cathode Dissolution*); and
- can be used only in procedures that support electrochemical analyses (see *Electrochemical Analysis*).

## User Subroutine Interface

```
 subroutine uechemecddiss(
     *     ecddiss,
     *    dEcddiss_dCSurfNorm, dEcddiss_dT, dEcddiss_dSol,dEcddiss_dPress,

     *     overpoteta_diss, epsS_diss,
     *     ocpDiss,chargeDischarge,
     *     jmac,jMatyp,matlayo,laccFlag,
     *     cmname, cParticleName,
     *     kstep,lFlags, kinc, time, dtime, noel, npt,
     *     ntens, ndi, nshr, props, nprops,
     *     predef, dpredef, npredef,
     *     statev, statevOld, nstatev,
     *     Temp, dTemp, Press, coords,
     *     solutionvar, dsolutionvar, nSolutionVar,
     *     cSurfNorm, cAvgNorm, ecdTotalOld,
     *     i_array,niarray,r_array,nrarray,c_array,ncarray)
C
      include 'aba_param.inc'
c Include below file to access property and parameter table.
      include 'aba_tcs_param.inc'
c Include below file to access universal constants
      include 'aba_phcon_param.inc'
c Include below file for parametrized location of various solution variables.

      include 'aba_solutionvarloc.inc'
C
      dimension i_array(*), r_array(*), jmatyp(*),jmac(*), lFlags(*)
      dimension time(2), props(nprops), predef(npredef), dpredef(npredef),

     *     statev(nstatev), statevold(nstatev), coords(*),
     *     solutionvar(nsolutionvar), dsolutionvar(nsolutionvar),
     *     dEcdDiss_dSol(*)
      character*80   cmname, c_array(NCARRAY), cParticleName

      user coding

      return
      end
```

## Variables to Be Defined

| | |
|---|---|
| `ecddiss` | Value of the electrode dissolution current density. |
| `dEcddiss_dCSurfNorm` | Variation of the dissolution current density `ecddiss` with respect to normalized particle surface concentration `CSurfNorm`. |
| `dEcddiss_dT` | Variation of the dissolution current density `ecddiss` with respect to temperature `temp`. |
| `dEcddiss_dSol` | Variation of the dissolution current density `ecddiss` with respect to the different solution variables `solutionvar`. |
| `dEcddiss_dPress` | Variation of the dissolution current density `ecddiss` with respect to the macroscale hydrostatic pressure `press`. |
| `overpoteta_diss` | Overpotential for dissolution current computation. |

## Variables That Can Be Updated

| | |
|---|---|
| `ocpDiss` | Open circuit potential for dissolution. |

## Variables Passed in for Information

| | |
|---|---|
| `chargeDischarge` | This variable will have a value that you can use to identify whether the current step is a charging step or a discharging step. The value that is passed in is defined in the property table *ABQ_EChemPET_Electrolyte_ChargeDischarge*. |
| `jMac` | Variable that must be passed into utility routine *GETVRM*. The calling user subroutine provides this variable. |
| `jMatyp` | Variable that must be passed into utility routine *GETVRM*. The calling user subroutine provides this variable. |
| `matlayo` | Variable that must be passed into utility routine *GETVRM*. The calling user subroutine provides this variable. |
| `lAccFlag` | Variable that must be passed into utility routine *GETVRM*. The calling user subroutine provides this variable. |
| `cmName` | User-defined material name, left justified. Some internal material models are given names starting with the "ABQ_" character string. To avoid conflict, you should not use "ABQ_" as the leading string for `CMNAME`. |
| `cParticleName` | User-defined particle name, left justified. |
| `kstep` | Current step number. |
| `lFlags` | An array containing the flags that define the current solution procedure and analysis type. |

| | |
|---|---|
| `lFlags(1)` | Defines the procedure type. See *Results File* for the key used for each procedure. |
| `lFlags(2)=0` | Small-displacement analysis. |
| `lFlags(2)=1` | Large-displacement analysis (nonlinear geometric effects included in the step). See *General and Perturbation Procedures*. |

| | |
|---|---|
| `kinc` | Current increment number. |
| `time(1)` | Current value of step time. |
| `time(2)` | Current value of total time. |
| `dtime` | Time increment. |
| `noel` | User-assigned element number. |

`npt` Integration point number.

`ntens` Size of the stress or strain component array.

`ndi` Number of direct stress components at this point.

`nshr` Number of engineering shear stress components at this point.

`props` User-specified array of material constants associated with this user material.

`npros` User-defined number of material constants associated with this user material.

`predef` Array of interpolated values of predefined field variables at this point at the start of the increment, based on the values read in at the nodes.

`dpredef` Array of increments of predefined field variables.

`npredef` Number of predefined field variables.

`statev` Current value of the solution-dependent variables.

`statevold` Array containing the user-defined solution-dependent state variables at the end of the previous increment.

`nstatev` Number of solution-dependent state variables that are associated with this material type (defined as described in *Allocating Space for Solution-Dependent State Variables*).

`Temp` Current temperature.

`dTemp` Incremental temperature.

`press` Current hydrostatic pressure.

`coords` An array containing the current coordinates of this integration point.

`solutionVar` Current value of the solution variables.

| | |
|---|---|
| `solutionVar(i_sol_phiS)` | Current value of the electric potential. |
| `solutionVar(i_sol_phiE)` | Current value of the electric potential in the electrolyte. |
| `solutionVar(i_sol_Ce)` | Current value of the ionic concentration in the electrolyte. |
| `solutionVar(i_sol_porP)` | Current value of the pore pressure. |

`dsolutionVar` Incremental value of the solution variables.

| | |
|---|---|
| `dsolutionVar(i_sol_phiS)` | Incremental value of the electric potential. |
| `dsolutionVar(i_sol_phiE)` | Incremental value of the electric potential in the electrolyte. |
| `dsolutionVar(i_sol_Ce)` | Incremental value of the ionic concentration in the electrolyte. |
| `dsolutionVar(i_sol_porP)` | Incremental value of the pore pressure. |

`nsolutionVar` Number of solution variables.

`cSurfNorm` Current normalized concentration on the surface of the particle.

`cAvgNorm` Current average normalized concentration in the particle.

`ecdTotalOld` Value of the total electrical current density at the end of the previous increment.

`i_array` Not used.

`niarray` Size of array `i_array`.

`r_array` Not used.

`nrarray` Size of array `r_array`.

`c_array` Not used.

`ncarray` Size of array `c_array`.

# UECHEMECDLPL: User subroutine to define the lithium plating (LPL) aging current to be used for elements that utilize porous electrode theory in an electrochemical analysis.

**Products:** Abaqus/Standard

**References:**

## Overview

User subroutine *UECHEMECDLPL*:

- is called for each material integration point for each iteration;
- is called for each microscale particle that is defined at the active electrodes during an electrochemical analysis;
- can be used to define the LPL current to model aging in batteries (see *Lithium Plating*); and
- can be used only in procedures that support electrochemical analyses (see *Electrochemical Analysis*).

## User Subroutine Interface

```
 subroutine uechemecdlpl(
     *     ecdlpl,
     *     dEcdlpl_dCSurfNorm, dEcdlpl_dT, dEcdlpl_dSol,dEcdlpl_dPress,
     *     overpoteta_lpl,
     *     filmResis,chargeDischarge,
     *     jmac,jMatyp,matlayo,laccFlag,
     *     cmname, cParticleName,
     *     kstep,lFlags, kinc, time, dtime, noel, npt,
     *     ntens, ndi, nshr, props, nprops,
     *     predef, dpredef, npredef,
     *     statev, statevOld, nstatev,
     *     Temp, dTemp, Press, coords,
     *     solutionvar, dsolutionvar, nSolutionVar,
     *     cSurfNorm, cAvgNorm, ecdTotalOld,
     *     SEILPLFilmThick, conclpl,
     *     i_array,niarray,r_array,nrarray,c_array,ncarray)
C
      include 'aba_param.inc'
c Include below file to access property and parameter table.
      include 'aba_tcs_param.inc'
c Include below file to access universal constants
      include 'aba_phcon_param.inc'
c Include below file for parametrized location of various solution variables.

      include 'aba_solutionvarloc.inc'
C
      dimension i_array(*), r_array(*), jmatyp(*),jmac(*), lFlags(*)
      dimension time(2), props(nprops), predef(npredef), dpredef(npredef),

     *     statev(nstatev), statevold(nstatev), coords(*),
     *     solutionvar(nsolutionvar), dsolutionvar(nsolutionvar),
     *     dEcdlpl_dSol(*)
      character*80   cmname, c_array(NCARRAY), cParticleName
```

```
      user coding

      return
      end
```

## Variables to Be Defined

| | |
|---|---|
| `ecdlpl` | Value of the LPL current density. |
| `dEcdlpl_dCSurfNorm` | Variation of the LPL current density `ecdlpl` with respect to normalized particle surface concentration `CSurfNorm`. |
| `dEcdlpl_dT` | Variation of the LPL current density `ecdlpl` with respect to temperature `temp`.. |
| `dEcdlpl_dSol` | Variation of the LPL current density `ecdlpl` with respect to the different solution variables `solutionvar`. |
| `dEcdlpl_dPress` | Variation of the LPL current density `ecdlpl` with respect to macroscale hydrostatic pressure `press`. |
| `overpoteta_lpl` | Overpotential for LPL current computation. |

## Variables That Can Be Updated

| | |
|---|---|
| `filmResis` | Resistance of the interface film layer. |

## Variables Passed in for Information

| | |
|---|---|
| `chargeDischarge` | This variable will have a value that you can use to identify whether the current step is a charging step or a discharging step. The value that is passed in is defined in the property table *ABQ_EChemPET_Electrolyte_ChargeDischarge*. |
| `jMac` | Variable that must be passed into utility routine *GETVRM*. The calling user subroutine provides this variable. |
| `jMatyp` | Variable that must be passed into utility routine *GETVRM*. The calling user subroutine provides this variable. |
| `matlayo` | Variable that must be passed into utility routine *GETVRM*. The calling user subroutine provides this variable. |
| `lAccFlag` | Variable that must be passed into utility routine *GETVRM*. The calling user subroutine provides this variable. |
| `cmName` | User-defined material name, left justified. Some internal material models are given names starting with the "ABQ_" character string. To avoid conflict, you should not use "ABQ_" as the leading string for `CMNAME`. |
| `cParticleName` | User-defined particle name, left justified. |
| `kstep` | Current step number. |
| `lFlags` | An array containing the flags that define the current solution procedure and analysis type. |

| | |
|---|---|
| `lFlags(1)` | Defines the procedure type. See *Results File* for the key used for each procedure. |
| `lFlags(2)=0` | Small-displacement analysis. |
| `lFlags(2)=1` | Large-displacement analysis (nonlinear geometric effects included in the step). See *General and Perturbation Procedures*. |

| | |
|---|---|
| `kinc` | Current increment number. |

`time(1)` Current value of step time.

`time(2)` Current value of total time.

`dtime` Time increment.

`noel` User-assigned element number.

`npt` Integration point number.

`ntens` Size of the stress or strain component array.

`ndi` Number of direct stress components at this point.

`nshr` Number of engineering shear stress components at this point.

`props` User-specified array of material constants associated with this user material.

`npros` User-defined number of material constants associated with this user material.

`predef` Array of interpolated values of predefined field variables at this point at the start of the increment, based on the values read in at the nodes.

`dpredef` Array of increments of predefined field variables.

`npredef` Number of predefined field variables.

`statev` Current value of the solution-dependent variables.

`statevold` Array containing the user-defined solution-dependent state variables at the end of the previous increment.

`nstatev` Number of solution-dependent state variables that are associated with this material type (defined as described in *Allocating Space for Solution-Dependent State Variables*).

`Temp` Current temperature.

`dTemp` Incremental temperature.

`press` Current hydrostatic pressure.

`coords` An array containing the current coordinates of this integration point.

`solutionVar` Current value of the solution variables.

| | |
|---|---|
| `solutionVar(i_sol_phiS)` | Current value of the electric potential. |
| `solutionVar(i_sol_phiE)` | Current value of the electric potential in the electrolyte. |
| `solutionVar(i_sol_Ce)` | Current value of the ionic concentration in the electrolyte. |
| `solutionVar(i_sol_porP)` | Current value of the pore pressure. |

`dsolutionVar` Incremental value of the solution variables.

| | |
|---|---|
| `dsolutionVar(i_sol_phiS)` | Incremental value of the electric potential. |
| `dsolutionVar(i_sol_phiE)` | Incremental value of the electric potential in the electrolyte. |
| `dsolutionVar(i_sol_Ce)` | Incremental value of the ionic concentration in the electrolyte. |
| `dsolutionVar(i_sol_porP)` | Incremental value of the pore pressure. |

`nsolutionVar` Number of solution variables.

`cSurfNorm` Current normalized concentration on the surface of the particle.

`cAvgNorm` Current average normalized concentration in the particle.

`ecdTotalOld` Value of the total electrical current density at the end of the previous increment.

`seilplfilmthick` Thickness of the combined solid electrolyte interface and plated lithium film layer.

| | |
|---|---|
| `conclpl` | Concentration of plated lithium in the combined solid electrolyte interface and plated lithium film layer. |
| `i_array` | Not used. |
| `niarray` | Size of array `i_array`. |
| `r_array` | Not used. |
| `nrarray` | Size of array `r_array`. |
| `c_array` | Not used. |
| `ncarray` | Size of array `c_array`. |

# UECHEMECDSEI: User subroutine to define the Solid Electrolyte Interface (SEI) aging current to be used for elements that utilize porous electrode theory in an electrochemical analysis.

**Products:** Abaqus/Standard

**References:**

## Overview

User subroutine *UECHEMECDSEI*:

- is called for each material integration point for each iteration;
- is called for each microscale particle that is defined at the active electrodes during an electrochemical analysis;
- can be used to define the SEI current to model aging in batteries (see *Solid Electrolyte Interface*; and
- can be used only in procedures that support electrochemical analyses (see *Electrochemical Analysis*).

## User Subroutine Interface

```
 subroutine uechemecdsei(
     *     ecdsei,
     *     dEcdsei_dCSurfNorm, dEcdsei_dT, dEcdsei_dSol,dEcdsei_dPress,
     *     overpoteta_sei, concEC_bulk, concEC_surf,
     *     ocpSEI, SEIResis,chargeDischarge,
     *     jmac,jMatyp,matlayo,laccFlag,
     *     cmname, cParticleName,
     *     kstep,lFlags, kinc, time, dtime, noel, npt,
     *     ntens, ndi, nshr, props, nprops,
     *     predef, dpredef, npredef,
     *     statev, statevOld, nstatev,
     *     Temp, dTemp, Press, coords,
     *     solutionvar, dsolutionvar, nSolutionVar,
     *     cSurfNorm, cAvgNorm, ecdTotalOld,
     *     SEILPLFilmThick, concSEI,
     *     i_array,niarray,r_array,nrarray,c_array,ncarray)
C
      include 'aba_param.inc'
c Include below file to access property and parameter table.
      include 'aba_tcs_param.inc'
c Include below file to access universal constants
      include 'aba_phcon_param.inc'
c Include below file for parametrized location of various solution variables.

      include 'aba_solutionvarloc.inc'
C
      dimension i_array(*), r_array(*), jmatyp(*),jmac(*), lFlags(*)
      dimension time(2), props(nprops), predef(npredef), dpredef(npredef),

     *     statev(nstatev), statevold(nstatev), coords(*),
     *     solutionvar(nsolutionvar), dsolutionvar(nsolutionvar),
     *     dEcdsei_dSol(*)
      character*80   cmname, c_array(NCARRAY), cParticleName
```

```
      user coding

      return
      end
```

## Variables to Be Defined

| | |
|---|---|
| `ecdsei` | Value of the SEI current density. |
| `dEcdsei_dCSurfNorm` | Variation of the SEI current density `ecdsei` with respect to normalized particle surface concentration `CSurfNorm`. |
| `dEcdsei_dT` | Variation of the SEI current density `ecdsei` with respect to temperature `temp`. |
| `dEcdsei_dSol` | Variation of the SEI current density `ecdsei` with respect to the different solution variables `solutionvar`. |
| `dEcdsei_dPress` | Variation of the SEI current density `ecdsei` with respect to the macroscale hydrostatic pressure `press`. |
| `overpoteta_SEI` | Overpotential for SEI current computation. |
| `concEC_bulk` | Bulk concentration of Ethylene Carbonate EC in the electrolyte. |
| `concEC_surf` | Concentration of Ethylene Carbonate EC on the particle surface. |

## Variables That Can Be Updated

| | |
|---|---|
| `ocpSEI` | Current open circuit potential value for SEI current computation. |
| `SEIResis` | Resistance of the solid electrolyte interface layer. |

## Variables Passed in for Information

| | |
|---|---|
| `chargeDischarge` | This variable will have a value that you can use to identify whether the current step is a charging step or a discharging step. The value that is passed in is defined in the property table *ABQ_EChemPET_Electrolyte_ChargeDischarge*. |
| `jMac` | Variable that must be passed into utility routine *GETVRM*. The calling user subroutine provides this variable. |
| `jMatyp` | Variable that must be passed into utility routine *GETVRM*. The calling user subroutine provides this variable. |
| `matlayo` | Variable that must be passed into utility routine *GETVRM*. The calling user subroutine provides this variable. |
| `lAccFlag` | Variable that must be passed into utility routine *GETVRM*. The calling user subroutine provides this variable. |
| `cmname` | User-defined material name, left justified. Some internal material models are given names starting with the "ABQ_" character string. To avoid conflict, you should not use "ABQ_" as the leading string for `CMNAME`. |
| `cParticleName` | User-defined particle name, left justified. |
| `kstep` | Current step number. |
| `lFlags` | An array containing the flags that define the current solution procedure and analysis type. |

| | |
|---|---|
| `lFlags(1)` | Defines the procedure type. See *Results File* for the key used for each procedure. |
| `lFlags(2)=0` | Small-displacement analysis. |

| | | |
|---|---|---|
| | `lFlags(2)=1` | Large-displacement analysis (nonlinear geometric effects included in the step). See *General and Perturbation Procedures*. |

`kinc` Current increment number.

`time(1)` Current value of step time.

`time(2)` Current value of total time.

`dtime` Time increment.

`noel` User-assigned element number.

`npt` Integration point number.

`ntens` Size of the stress or strain component array.

`ndi` Number of direct stress components at this point.

`nshr` Number of engineering shear stress components at this point.

`props` User-specified array of material constants associated with this user material.

`npros` User-defined number of material constants associated with this user material.

`predef` Array of interpolated values of predefined field variables at this point at the start of the increment, based on the values read in at the nodes.

`dpredef` Array of increments of predefined field variables.

`npredef` Number of predefined field variables.

`statev` Current value of the solution-dependent variables.

`statevold` Array containing the user-defined solution-dependent state variables at the end of the previous increment.

`nstatev` Number of solution-dependent state variables that are associated with this material type (defined as described in *Allocating Space for Solution-Dependent State Variables*).

`Temp` Current temperature.

`dTemp` Incremental temperature.

`press` Current hydrostatic pressure.

`coords` An array containing the current coordinates of this integration point.

`solutionVar` Current value of the solution variables.

| | |
|---|---|
| `solutionVar(i_sol_phiS)` | Current value of the electric potential. |
| `solutionVar(i_sol_phiE)` | Current value of the electric potential in the electrolyte. |
| `solutionVar(i_sol_Ce)` | Current value of the ionic concentration in the electrolyte. |
| `solutionVar(i_sol_porP)` | Current value of the pore pressure. |

`dsolutionVar` Incremental value of the solution variables.

| | |
|---|---|
| `dsolutionVar(i_sol_phiS)` | Incremental value of the electric potential. |
| `dsolutionVar(i_sol_phiE)` | Incremental value of the electric potential in the electrolyte. |
| `dsolutionVar(i_sol_Ce)` | Incremental value of the ionic concentration in the electrolyte. |
| `dsolutionVar(i_sol_porP)` | Incremental value of the pore pressure. |

`nsolutionVar` Number of solution variables.

| | |
|---|---|
| `cSurfNorm` | Current normalized concentration on the surface of the particle. |
| `cAvgNorm` | Current average normalized concentration in the particle. |
| `ecdTotalOld` | Value of the total electrical current density at the end of the previous increment. |
| `seilplfilmthick` | Thickness of the combined solid electrolyte interface and plated lithium film layer. |
| `concSEI` | Concentration of SEI in the combined solid electrolyte interface and plated lithium film layer. |
| `i_array` | Not used. |
| `niarray` | Size of array `i_array`. |
| `r_array` | Not used. |
| `nrarray` | Size of array `r_array`. |
| `c_array` | Not used. |
| `ncarray` | Size of array `c_array`. |

# UECHEMECDSEIREFORM: User subroutine to define the Solid Electrolyte Interface (SEI) reformation current to be used for elements that utilize porous electrode theory in an electrochemical analysis.

**Products:** Abaqus/Standard

**References:**

## Overview

User subroutine *UECHEMECDSEIREFORM*:

- is called for each material integration point for each iteration;
- is called for each microscale particle that is defined at the active electrodes during an electrochemical analysis;
- can be used to define the SEI reformation current to model aging in batteries (see *SEI Reformation*); and
- can be used only in procedures that support electrochemical analyses (see *Electrochemical Analysis*).

## User Subroutine Interface

```
 subroutine uechemecdseireform(
     *     ecdseir,
     *    dEcdseir_dCSurfNorm, dEcdseir_dT, dEcdseir_dSol,dEcdseir_dPress,

     *     overpoteta_seir,
     *     ocpSEI,chargeDischarge,
     *     jmac,jMatyp,matlayo,laccFlag,
     *     cmname, cParticleName,
     *     kstep,lFlags, kinc, time, dtime, noel, npt,
     *     ntens, ndi, nshr, props, nprops,
     *     predef, dpredef, npredef,
     *     statev, statevOld, nstatev,
     *     Temp, dTemp, Press, coords,
     *     solutionvar, dsolutionvar, nSolutionVar,
     *     cSurfNorm, cAvgNorm, ecdTotalOld, ecdbvold,
     *     SEILPLFilmThick, concEC_surf,
     *     i_array,niarray,r_array,nrarray,c_array,ncarray)
C
      include 'aba_param.inc'
c Include below file to access property and parameter table.
      include 'aba_tcs_param.inc'
c Include below file to access universal constants
      include 'aba_phcon_param.inc'
c Include below file for parametrized location of various solution variables.

      include 'aba_solutionvarloc.inc'
C
      dimension i_array(*), r_array(*), jmatyp(*),jmac(*), lFlags(*)
      dimension time(2), props(nprops), predef(npredef), dpredef(npredef),

     *     statev(nstatev), statevold(nstatev), coords(*),
     *     solutionvar(nsolutionvar), dsolutionvar(nsolutionvar),
     *     dEcdseir_dSol(*)
      character*80   cmname, c_array(NCARRAY), cParticleName
```

```
    user coding

    return
    end
```

## Variables to Be Defined

| | |
|---|---|
| `ecdseir` | Value of the SEI reformation current density. |
| `dEcdseir_dCSurfNorm` | Variation of the SEI reformation current density `ecdseir` with respect to normalized particle surface concentration `CSurfNorm`. |
| `dEcdseir_dT` | Variation of the SEI reformation current density `ecdseir` with respect to temperature `temp`. |
| `dEcdseir_dSol` | Variation of the SEI reformation current density `ecdseir` with respect to the different solution variables `solutionvar`. |
| `dEcdseir_dPress` | Variation of the SEI reformation current density `ecdseir` with respect to the macroscale hydrostatic pressure `press`. |
| `overpoteta_SEIr` | Overpotential for SEI reformation current computation. |

## Variables That Can Be Updated

| | |
|---|---|
| `ocpSEI` | Current open circuit potential value for SEI current computation. |

## Variables Passed in for Information

| | |
|---|---|
| `chargeDischarge` | This variable will have a value that you can use to identify whether the current step is a charging step or a discharging step. The value that is passed in is defined in the property table *ABQ_EChemPET_Electrolyte_ChargeDischarge*. |
| `jMac` | Variable that must be passed into utility routine *GETVRM*. The calling user subroutine provides this variable. |
| `jMatyp` | Variable that must be passed into utility routine *GETVRM*. The calling user subroutine provides this variable. |
| `matlayo` | Variable that must be passed into utility routine *GETVRM*. The calling user subroutine provides this variable. |
| `lAccFlag` | Variable that must be passed into utility routine *GETVRM*. The calling user subroutine provides this variable. |
| `cmName` | User-defined material name, left justified. Some internal material models are given names starting with the "ABQ_" character string. To avoid conflict, you should not use "ABQ_" as the leading string for `CMNAME`. |
| `cParticleName` | User-defined particle name, left justified. |
| `kstep` | Current step number. |
| `lFlags` | An array containing the flags that define the current solution procedure and analysis type. |

| | |
|---|---|
| `lFlags(1)` | Defines the procedure type. See *Results File* for the key used for each procedure. |
| `lFlags(2)=0` | Small-displacement analysis. |
| `lFlags(2)=1` | Large-displacement analysis (nonlinear geometric effects included in the step; see *General and Perturbation Procedures*. |

| | |
|---|---|
| `kinc` | Current increment number. |

`time(1)` Current value of step time.

`time(2)` Current value of total time.

`dtime` Time increment.

`noel` User-assigned element number.

`npt` Integration point number.

`ntens` Size of the stress or strain component array.

`ndi` Number of direct stress components at this point.

`nshr` Number of engineering shear stress components at this point.

`props` User-specified array of material constants associated with this user material.

`npros` User-defined number of material constants associated with this user material.

`predef` Array of interpolated values of predefined field variables at this point at the start of the increment, based on the values read in at the nodes.

`dpredef` Array of increments of predefined field variables.

`npredef` Number of predefined field variables.

`statev` Current value of the solution-dependent variables.

`statevold` Array containing the user-defined solution-dependent state variables at the end of the previous increment.

`nstatev` Number of solution-dependent state variables that are associated with this material type (defined as described in *Allocating Space for Solution-Dependent State Variables*).

`Temp` Current temperature.

`dTemp` Incremental temperature.

`press` Current hydrostatic pressure.

`coords` An array containing the current coordinates of this integration point.

`solutionVar` Current value of the solution variables.

| | |
|---|---|
| `solutionVar(i_sol_phiS)` | Current value of the electric potential. |
| `solutionVar(i_sol_phiE)` | Current value of the electric potential in the electrolyte. |
| `solutionVar(i_sol_Ce)` | Current value of the ionic concentration in the electrolyte. |
| `solutionVar(i_sol_porP)` | Current value of the pore pressure. |

`dsolutionVar` Incremental value of the solution variables.

| | |
|---|---|
| `dsolutionVar(i_sol_phiS)` | Incremental value of the electric potential. |
| `dsolutionVar(i_sol_phiE)` | Incremental value of the electric potential in the electrolyte. |
| `dsolutionVar(i_sol_Ce)` | Incremental value of the ionic concentration in the electrolyte. |
| `dsolutionVar(i_sol_porP)` | Incremental value of the pore pressure. |

`nsolutionVar` Number of solution variables.

`cSurfNorm` Current normalized concentration on the surface of the particle.

`cAvgNorm` Current average normalized concentration in the particle.

`ecdTotalOld` Value of the total electrical current density at the end of the previous increment.

| | |
|---|---|
| `ecdBVold` | Value of the Butler-Volmer electrical current density at the end of the previous increment. |
| `seilplfilmthick` | Thickness of the combined solid electrolyte interface and plated lithium film layer. |
| `concEC_surf` | Concentration of Ethylene Carbonate EC on the particle surface. |
| `i_array` | Not used. |
| `niarray` | Size of array `i_array`. |
| `r_array` | Not used. |
| `nrarray` | Size of array `r_array`. |
| `c_array` | Not used. |
| `ncarray` | Size of array `c_array`. |

# UEIGENSTRAIN: User subroutine to define the eigenstrain in a material from different physics fields.

**Products:** Abaqus/Standard

**References:**

## Overview



**Warning:** The use of this subroutine generally requires considerable expertise. You are cautioned that the implementation of any realistic constitutive model requires extensive development and testing. Initial testing on a single-element model with prescribed loading in different fields is strongly recommended.

User subroutine *UEIGENSTRAIN*:

- is called for each material integration point for each iteration;
- can be used to define an eigenstrain tensor;
- can be used to account for eigenstrains due to other fields such as electrical potentials and ion concentrations in a fully coupled manner;
- can be used in conjunction with user subroutine *USDFLD* if it is desired to redefine any field variables before they are passed in; and
- can be used only in procedures that support electrochemical analyses (see *Electrochemical Analysis*).

## User Subroutine Interface

```
 subroutine ueigenstrain(
     *     dEigStrain, dEigStrain_dT, dEigStrain_dSol, iEigStrainAssemble,

     *     cmname, kstep, lFlags, kinc, time1, dtime1, noel, npt,
     *     ntens, ndi, nshr, props, nprops, predef, dpredef, npredef,
     *     statev, statevold, nstatv, temp, dtemp, coords,
     *     solutionvar, dsolutionvar, nSolutionVar,
     *     i_array, niarray, r_array, nrarray, c_array, ncarray)
C
      include 'aba_param.inc'
c Include below file to access property and parameter table.
      include 'aba_tcs_param.inc'
c Include below file to access universal constants
      include 'aba_phcon_param.inc'
c Include below file for parametrized location of various solution variables.

      include 'aba_solutionvarloc.inc'
C
      dimension dEigStrain(ntens),  dEigStrain_dT(ntens),
     *     dEigStrain_dSol(ntens, nSolutionVar),
     *     iEigStrainAssemble(nAllFieldsMax),
     *     time(2),props(nprops),predef(npredef), dpredef(npredef),
     *     statev(nstatev), statevOld(nstatev),
     *     solutionvar(nsolutionvar), dsolutionvar(nsolutionvar),
     *     coords(*),i_array(*), r_array(*), lFlags(*)
      character*80   cmname, c_array(NCARRAY)
```

```
      user coding

      return
      end
```

## Variables to Be Defined

`dEigStrain` Incremental eigenstrain.

`dEigStrain_dT` Variation of `dEigStrain` with respect to temperature `Temp`.

`dEigStrain_dSol` Variation of `dEigStrain` with respect to the different solution variables `solutionvar`.

`iEigStrainAssemble` An array of integer flags of dimension `nAllFieldsMax` that identify which linearization terms in the different fields should be assembled. The default value is 0, which means no sources are present. You must set the values of the corresponding entries to 1 to ensure proper assembly of the source terms from the different fields.

| | |
|---|---|
| `iEigStrainAssemble(i_sol_phiS)` | If set to 1, assemble variation of eigenstrains with respect to electric potential. |
| `iEigStrainAssemble(i_sol_phiE)` | If set to 1, assemble variation of eigenstrains with respect to electric potential in the electrolyte. |
| `iEigStrainAssemble(i_sol_Ce)` | If set to 1, assemble variation of eigenstrains with respect to ion concentration in the electrolyte. |
| `iEigStrainAssemble(i_sol_solidC)` | If set to 1, assemble variation of eigenstrains with respect to species concentration in the electrolyte. |
| `iEigStrainAssemble(i_sol_porP)` | If set to 1, assemble variation of eigenstrains with respect to pore pressure in the electrolyte. |
| `iEigStrainAssemble(i_sol_Temp)` | If set to 1, assemble variation of eigenstrains with respect to temperature. |

## Variables That Can Be Updated

`statev` Current value of the solution-dependent variables.

## Variables Passed in for Information

`cmName` User-defined material name, left justified. Some internal material models are given names starting with the “ABQ_” character string. To avoid conflict, you should not use “ABQ_” as the leading string for `CMNAME`.

`kstep` Current step number.

`lFlags` An array containing the flags that define the current solution procedure and analysis type.

| | |
|---|---|
| `lFlags(1)` | Defines the procedure type. See *Results File* for the key used for each procedure. |
| `lFlags(2)=0` | Small-displacement analysis. |
| `lFlags(2)=1` | Large-displacement analysis (nonlinear geometric effects included in the step). See *General and Perturbation Procedures*. |

`kinc` Current increment number.

`time(1)` Current value of step time.

`time(2)` Current value of total time.

`dtime` Time increment.

`noel` User-assigned element number.

`npt` Integration point number.

`ntens` Size of the stress or strain component array.

`ndi` Number of direct stress components at this point.

`nshr` Number of engineering shear stress components at this point.

`props` User-specified array of material constants associated with this user material.

`npros` User-defined number of material constants associated with this user material.

`predef` Array of interpolated values of predefined field variables at this point at the start of the increment, based on the values read in at the nodes.

`dpredef` Array of increments of predefined field variables.

`npredef` Number of predefined field variables.

`statevold` Array containing the user-defined solution-dependent state variables at the end of the previous increment.

`nstatev` Number of solution-dependent state variables that are associated with this material type (defined as described in *Allocating Space for Solution-Dependent State Variables*).

`Temp` Current temperature.

`dTemp` Incremental temperature.

`coords` An array containing the current coordinates of this integration point.

`solutionVar` Current value of the solution variables.

| | |
|---|---|
| `solutionVar(i_sol_phiS)` | Current value of the electric potential. |
| `solutionVar(i_sol_phiE)` | Current value of the electric potential in the electrolyte. |
| `solutionVar(i_sol_Ce)` | Current value of the ionic concentration in the electrolyte. |
| `solutionVar(i_sol_solidC)` | Current value of the species concentration. |
| `solutionVar(i_sol_porP)` | Current value of the pore pressure. |

`dsolutionVar` Incremental value of the solution variables.

| | |
|---|---|
| `dsolutionVar(i_sol_phiS)` | Incremental value of the electric potential. |
| `dsolutionVar(i_sol_phiE)` | Incremental value of the electric potential in the electrolyte. |
| `dsolutionVar(i_sol_Ce)` | Incremental value of the ionic concentration in the electrolyte. |
| `dsolutionVar(i_sol_solidC)` | Incremental value of the species concentration. |
| `dsolutionVar(i_sol_porP)` | Incremental value of the pore pressure. |

`nsolutionVar` Number of solution variables.

`i_array` Not used.

`niarray` Size of array `i_array`.

`r_array` Not used.

`nrarray` Size of array `r_array`.

`c_array` Not used.

`ncarray` Size of array `c_array`.

# UEL: User subroutine to define an element.

**Products:** Abaqus/Standard

**References:**

- *User-Defined Elements*
- **UEL PROPERTY*
- **USER ELEMENT*

## Overview



**Warning:** This feature is intended for advanced users only. Its use in all but the simplest test examples will require considerable coding by the user/developer. *User-Defined Elements* should be read before proceeding.

User subroutine UEL:

- will be called for each element that is of a general user-defined element type (that is, not defined by a linear stiffness, damping, or mass matrix read either directly or from results file data) each time element calculations are required; and
- (or subroutines called by user subroutine UEL) must perform, depending on the analysis type, all or most of the calculations for the element, appropriate to the current activity in the analysis.

## Wave Kinematic Data

For Abaqus/Aqua applications four utility routines—GETWAVE, GETWAVEVEL, GETWINDVEL, and GETCURRVEL—are provided to access the fluid kinematic data. These routines are used from within user subroutine UEL and are discussed in detail in *Obtaining Wave Kinematic Data in an Abaqus/Aqua Analysis*.

## User Subroutine Interface

```
      SUBROUTINE UEL(RHS,AMATRX,SVARS,ENERGY,NDOFEL,NRHS,NSVARS,
     1 PROPS,NPROPS,COORDS,MCRD,NNODE,U,DU,V,A,JTYPE,TIME,DTIME,
     2 KSTEP,KINC,JELEM,PARAMS,NDLOAD,JDLTYP,ADLMAG,PREDEF,NPREDF,
     3 LFLAGS,MLVARX,DDLMAG,MDLOAD,PNEWDT,JPROPS,NJPROP,PERIOD)
C
      INCLUDE 'ABA_PARAM.INC'
C
      DIMENSION RHS(MLVARX,*),AMATRX(NDOFEL,NDOFEL),PROPS(*),
     1 SVARS(*),ENERGY(8),COORDS(MCRD,NNODE),U(NDOFEL),
     2 DU(MLVARX,*),V(NDOFEL),A(NDOFEL),TIME(2),PARAMS(*),
     3 JDLTYP(MDLOAD,*),ADLMAG(MDLOAD,*),DDLMAG(MDLOAD,*),
     4 PREDEF(2,NPREDF,NNODE),LFLAGS(*),JPROPS(*)


      user coding to define RHS, AMATRX, SVARS, ENERGY, and PNEWDT


      RETURN
      END
```

## Variables to Be Defined

These arrays depend on the value of the LFLAGS array.

RHS An array containing the contributions of this element to the right-hand-side vectors of the overall system of equations. For most nonlinear analysis procedures, NRHS=1 and RHS should contain the residual vector (external forces minus internal forces). The exception is the modified Riks static procedure (*Static Stress Analysis*), for which NRHS=2 and the first column in RHS should contain the residual vector and the second column should contain the increments of external load on the element. RHS(K1,K2) is the entry for the K1th degree of freedom of the element in the K2 the right-hand-side vector. For direct steady-state analyses, set NRHS=2 in the recovery path for the reaction force to define real and imaginary parts of the vectors. RHS(K1,K2) is the entry for the K1th degree of freedom, K2=1 is the entry for the real part, and K2=2 is the entry for the imaginary part of the vector. For mode-based procedures, this user subroutine is called only to form the left-side matrices: stiffness, damping, and mass. Right-side vectors, as well as the reaction force vector, are calculated outside the user subroutine automatically. The reaction force calculation takes the inertia force and the forces due to the damping into account.

AMATRX An array containing the contribution of this element to the Jacobian (stiffness) or other matrix of the overall system of equations. The particular matrix required at any time depends on the entries in the LFLAGS array (see below).

All nonzero entries in AMATRX should be defined, even if the matrix is symmetric. If you do not specify that the matrix is unsymmetric when you define the user element, Abaqus/Standard will use the symmetric matrix defined by $\frac{1}{2}\left([A]+[A]^T\right)$, where $[A]$ is the matrix defined as AMATRX in this subroutine. If you specify that the matrix is unsymmetric when you define the user element, Abaqus/Standard will use AMATRX directly.

SVARS An array containing the values of the solution-dependent state variables associated with this element. The number of such variables is NSVARS (see below). You define the meaning of these variables.

For general nonlinear steps this array is passed into *UEL* containing the values of these variables at the start of the current increment. They should be updated to be the values at the end of the increment, unless the procedure during which *UEL* is being called does not require such an update. This depends on the entries in the LFLAGS array (see below). For linear perturbation steps this array is passed into *UEL* containing the values of these variables in the base state. They should be returned containing perturbation values if you want to output such quantities.

When KINC is equal to zero, the call to *UEL* is made for zero increment output (see *About Output*). In this case the values returned will be used only for output purposes and are not updated permanently.

ENERGY For general nonlinear steps array ENERGY contains the values of the energy quantities associated with the element. The values in this array when *UEL* is called are the element energy quantities at the start of the current increment. They should be updated to the values at the end of the current increment. For linear perturbation steps the array is passed into *UEL* containing the energy in the base state. They should be returned containing perturbation values if you wish to output such quantities. They are not available for updates for mode-based procedures. The entries in the array are as follows:

| | |
|---|---|
| ENERGY(1) | Kinetic energy. |
| ENERGY(2) | Elastic strain energy. |
| ENERGY(3) | Creep dissipation. |
| ENERGY(4) | Plastic dissipation. |
| ENERGY(5) | Viscous dissipation. |
| ENERGY(6) | “Artificial strain energy” associated with such effects as artificial stiffness introduced to control hourglassing or other singular modes in the element. |

| | |
|---|---|
| ENERGY(7) | Electrostatic energy. |
| ENERGY(8) | Incremental work done by loads applied within the user element. |

When KINC is equal to zero, the call to *UEL* is made for zero increment output (see *About Output*). In this case the energy values returned will be used only for output purposes and are not updated permanently.

## Variables That Can Be Updated

PNEWDT Ratio of suggested new time increment to the time increment currently being used (DTIME, see below). This variable allows you to provide input to the automatic time incrementation algorithms in Abaqus/Standard (if automatic time incrementation is chosen). It is useful only during equilibrium iterations with the normal time incrementation, as indicated by LFLAGS(3)=1. During a severe discontinuity iteration (such as contact changes), PNEWDT is ignored unless CONVERT SDI=YES is specified for this step. The usage of PNEWDT is discussed below.

PNEWDT is set to a large value before each call to *UEL*.

If PNEWDT is redefined to be less than 1.0, Abaqus/Standard must abandon the time increment and attempt it again with a smaller time increment. The suggested new time increment provided to the automatic time integration algorithms is PNEWDT × DTIME, where the PNEWDT used is the minimum value for all calls to user subroutines that allow redefinition of PNEWDT for this iteration.

If PNEWDT is given a value that is greater than 1.0 for all calls to user subroutines for this iteration and the increment converges in this iteration, Abaqus/Standard may increase the time increment. The suggested new time increment provided to the automatic time integration algorithms is PNEWDT × DTIME, where the PNEWDT used is the minimum value for all calls to user subroutines for this iteration.

If automatic time incrementation is not selected in the analysis procedure, values of PNEWDT that are greater than 1.0 will be ignored and values of PNEWDT that are less than 1.0 will cause the job to terminate.

## Variables Passed in for Information

**Arrays:**

PROPS A floating point array containing the NPROPS real property values defined for use with this element. NPROPS is the user-specified number of real property values. See *Defining the Element Properties*.

JPROPS An integer array containing the NJPROP integer property values defined for use with this element. NJPROP is the user-specified number of integer property values. See *Defining the Element Properties*.

COORDS An array containing the original coordinates of the nodes of the element. COORDS(K1,K2) is the K1th coordinate of the K2th node of the element.

U, DU, V, A Arrays containing the current estimates of the basic solution variables (displacements, rotations, temperatures, depending on the degree of freedom) at the nodes of the element at the end of the current increment. Values are provided as follows:

| | |
|---|---|
| U(K1) | Total values of the variables. If this is a linear perturbation step, it is the value in the base state. |
| DU(K1,KRHS) | Incremental values of the variables for the current increment for right-hand-side KRHS. If this is an eigenvalue extraction step, this is the eigenvector magnitude for eigenvector KRHS. For steady-state dynamics, KRHS $= 1$ denotes real components of perturbation displacement and KRHS $= 2$ denotes imaginary components of perturbation displacement. |

**Arrays:**

| | |
|---|---|
| `V(K1)` | Time rate of change of the variables (velocities, rates of rotation). Defined for implicit dynamics only (`LFLAGS(1)` = 11 or 12). |
| `A(K1)` | Accelerations of the variables. Defined for implicit dynamics only (`LFLAGS(1)` = 11 or 12). |

`JDLTYP` An array containing the integers used to define distributed load types for the element. Loads of type U*n* are identified by the integer value *n* in `JDLTYP`; loads of type U*n*NU are identified by the negative integer value $-n$ in `JDLTYP`. `JDLTYP(K1,K2)` is the identifier of the `K1`th distributed load in the `K2`th load case. For general nonlinear steps `K2` is always 1.

`ADLMAG` For general nonlinear steps `ADLMAG(K1,1)` is the total load magnitude of the `K1`th distributed load at the end of the current increment for distributed loads of type U*n*. For distributed loads of type U*n*NU, the load magnitude is defined in *UEL*; therefore, the corresponding entries in `ADLMAG` are zero. For linear perturbation steps `ADLMAG(K1,1)` contains the total load magnitude of the `K1`th distributed load of type U*n* applied in the base state. Base state loading of type U*n*NU must be dealt with inside *UEL*. `ADLMAG(K1,2)`, `ADLMAG(K1,3)`, etc. are currently not used.

`DDLMAG` For general nonlinear steps `DDLMAG` contains the increments in the magnitudes of the distributed loads that are currently active on this element for distributed loads of type U*n*. `DDLMAG(K1,1)` is the increment of magnitude of the load for the current time increment. The increment of load magnitude is required to compute the external work contribution. For distributed loads of type U*n*NU, the load magnitude is defined in *UEL*; therefore, the corresponding entries in `DDLMAG` are zero. For linear perturbation steps `DDLMAG(K1,K2)` contains the perturbation in the magnitudes of the distributed loads that are currently active on this element for distributed loads of type U*n*. `K1` denotes the `K1`th perturbation load active on the element. `K2` is always 1, except for steady-state dynamics, where `K2`=1 for real loads and `K2`=2 for imaginary loads. Perturbation loads of type U*n*NU must be dealt with inside *UEL*.

`PREDEF` An array containing the values of predefined field variables, such as temperature in an uncoupled stress/displacement analysis, at the nodes of the element (*Predefined Fields*).

The first index of the array, `K1`, is either 1 or 2, with 1 indicating the value of the field variable at the end of the increment and 2 indicating the increment in the field variable. The second index, `K2`, indicates the variable: the temperature corresponds to index 1, and the predefined field variables correspond to indices 2 and above. In cases where temperature is not defined, the predefined field variables begin with index 1. The third index, `K3`, indicates the local node number on the element.

| | |
|---|---|
| `PREDEF(K1,1,K3)` | Temperature. |
| `PREDEF(K1,2,K3)` | First predefined field variable. |
| `PREDEF(K1,3,K3)` | Second predefined field variable. |
| Etc. | Any other predefined field variable. |
| `PREDEF(K1,K2,K3)` | Total or incremental value of the `K2`th predefined field variable at the `K3`th node of the element. |
| `PREDEF(1,K2,K3)` | Values of the variables at the end of the current increment. |
| `PREDEF(2,K2,K3)` | Incremental values corresponding to the current time increment. |

`PARAMS` An array containing the parameters associated with the solution procedure. The entries in this array depend on the solution procedure currently being used when *UEL* is called, as indicated by the entries in the `LFLAGS` array (see below).

For implicit dynamics (`LFLAGS(1)` = 11 or 12) `PARAMS` contains the integration operator values, as:

| | |
|---|---|
| `PARAMS(1)` | $\alpha$ |

**Arrays:**

PARAMS(2) $\beta$

PARAMS(3) $\gamma$

LFLAGS An array containing the flags that define the current solution procedure and requirements for element calculations. Detailed requirements for the various Abaqus/Standard procedures are defined earlier in this section.

| | |
|---|---|
| LFLAGS(1) | Defines the procedure type. See *Results File* for the key used for each procedure. |
| LFLAGS(2)=0 | Small-displacement analysis. |
| LFLAGS(2)=1 | Large-displacement analysis (nonlinear geometric effects included in the step; see *General and Perturbation Procedures*). |
| LFLAGS(3)=1 | Normal implicit time incrementation procedure. User subroutine *UEL* must define the residual vector in RHS and the Jacobian matrix in AMATRX. |
| LFLAGS(3)=2 | Define the current stiffness matrix (AMATRX$= K^{NM} = -\partial F^N/\partial u^M$ or $-\partial G^N/\partial u^M$) only. |
| LFLAGS(3)=3 | Define the current damping matrix (AMATRX$= C^{NM} = -\partial F^N/\partial \dot{u}^M$ or $-\partial G^N/\partial \dot{u}^M$) only. To declare the type of damping matrix as viscous or structural, use LFLAGS(7) as well (see below). |
| LFLAGS(3)=4 | Define the current mass matrix (AMATRX$= M^{NM} = -\partial F^N/\partial \ddot{u}^M$) only. Abaqus/Standard always requests an initial mass matrix at the start of the analysis. |
| LFLAGS(3)=5 | Define the current residual or load vector (RHS $= F^N$) only. |
| LFLAGS(3)=6 | Define the current mass matrix and the residual vector for the initial acceleration calculation (or the calculation of accelerations after impact). |
| LFLAGS(3)=100 | Define perturbation quantities for output. Not available for direct steady-state dynamic and mode-based procedures. |
| LFLAGS(4)=0 | The step is a general step. |
| LFLAGS(4)=1 | The step is a linear perturbation step. |
| LFLAGS(5)=0 | The current approximations to $u^M$, etc. were based on Newton corrections. |
| LFLAGS(5)=1 | The current approximations were found by extrapolation from the previous increment. |
| LFLAGS(7)=1 | When the damping matrix flag is set, the viscous damping matrix is defined. |
| LFLAGS(7)=2 | When the damping matrix flag is set, the structural damping matrix is defined. |

TIME(1) Current value of step time or frequency.

TIME(2) Current value of total time.

**Scalar parameters:**

| | |
|---|---|
| DTIME | Time increment. |
| PERIOD | Time period of the current step. |
| NDOFEL | Number of degrees of freedom in the element. |
| MLVARX | Dimensioning parameter used when several displacement or right-hand-side vectors are used. |
| NRHS | Number of load vectors. NRHS is 1 in most nonlinear problems: it is 2 for the modified Riks static procedure (*Static Stress Analysis*), and it is greater than 1 in some linear analysis procedures and during substructure generation. For example, in the recovery path for the |

**Scalar parameters:**

| | |
|---|---|
| | direct steady-state procedure, it is 2 to accommodate the real and imaginary parts of the vectors. |
| NSVARS | User-defined number of solution-dependent state variables associated with the element (*Defining the Number of Solution-Dependent Variables That Must Be Stored within the Element*). |
| NPROPS | User-defined number of real property values associated with the element (*Defining the Element Properties*). |
| NJPROP | User-defined number of integer property values associated with the element (*Defining the Element Properties*). |
| MCRD | MCRD is defined as the maximum of the user-defined maximum number of coordinates required at any node point (*Defining the Maximum Number of Coordinates Needed at Any Nodal Point*) and the value of the largest active degree of freedom of the user element that is less than or equal to 3. For example, if you specify that the maximum number of coordinates is 1 and the active degrees of freedom of the user element are 2, 3, and 6, MCRD will be 3. If you specify that the maximum number of coordinates is 2 and the active degrees of freedom of the user element are 11 and 12, MCRD will be 2. |
| NNODE | User-defined number of nodes on the element (*Defining the Number of Nodes Associated with the Element*). |
| JTYPE | Integer defining the element type. This is the user-defined integer value *n* in element type U*n* (*Assigning an Element Type Key to a User-Defined Element*). |
| KSTEP | Current step number. |
| KINC | Current increment number. |
| JELEM | User-assigned element number. |
| NDLOAD | Identification number of the distributed load or flux currently active on this element. |
| MDLOAD | Total number of distributed loads and/or fluxes defined on this element. |
| NPREDF | Number of predefined field variables, including temperature. For user elements Abaqus/Standard uses one value for each field variable per node. |

## UEL Conventions

The solution variables (displacement, velocity, etc.) are arranged on a node/degree of freedom basis. The degrees of freedom of the first node are first, followed by the degrees of freedom of the second node, etc.

## Usage with General Nonlinear Procedures

The values of $\boldsymbol{u^N}$ (and, in direct-integration dynamic steps, $\dot{\boldsymbol{u}}^{\boldsymbol{N}}$ and $\ddot{\boldsymbol{u}}^{\boldsymbol{N}}$) enter user subroutine *UEL* as their latest approximations at the end of the time increment; that is, at time $\boldsymbol{t + \Delta t}$.

The values of $\boldsymbol{H^\alpha}$ enter the subroutine as their values at the beginning of the time increment; that is, at time *t*. It is your responsibility to define suitable time integration schemes to update $\boldsymbol{H^\alpha}$. To ensure accurate, stable integration of internal state variables, you can control the time incrementation via PNEWDT.

The values of $\boldsymbol{p^\beta}$ enter the subroutine as the values of the total load magnitude for the $\boldsymbol{\beta}$th distributed load at the end of the increment. Increments in the load magnitudes are also available.

In the following descriptions of the user element's requirements, it will be assumed that LFLAGS(3)=1 unless otherwise stated.

### Static Analysis (LFLAGS(1)=1, 2)

- $F^N = F^N\left(u^M, H^\alpha, p^\beta, t\right)$, where the residual force is the external force minus the internal force.
- Automatic convergence checks are applied to the force residuals corresponding to degrees of freedom 1–7.
- You must define AMATRX$= K^{NM} = -\partial F^N/\partial u^M$ and RHS$= F^N$ and update the state variables, $H^\alpha$.

### Modified Riks Static Analysis (LFLAGS(1)=1) and (NRHS=2)

- $F^N = F^N\left(u^M, H^\alpha, p^\beta\right)$, where $p^\beta = p_0^\beta + \lambda q^\beta$, $p_0^\beta$ and $q^\beta$ are fixed load parameters, and $\lambda$ is the Riks (scalar) load parameter.
- Automatic convergence checks are applied to the force residuals corresponding to degrees of freedom 1–7.
- You must define AMATRX$= K^{NM} = -\partial F^N/\partial u^M$, RHS(1) $= F^N$, and RHS(2) $= \Delta\lambda\left(\partial F^N/\partial\lambda\right)$ and update the state variables, $H^\alpha$. RHS(2) is the incremental load vector.

### Direct-Integration Dynamic Analysis (LFLAGS(1)=11, 12)

- Automatic convergence checks are applied to the force residuals corresponding to degrees of freedom 1–7.
- LFLAGS(3)=1: Normal time increment. Either the Hilber-Hughes-Taylor or the backward Euler time integration scheme will be used. With $\alpha$ set to zero for the backward Euler, both schemes imply

  $$F^N = -M^{NM}\ddot{u}_{t+\Delta t} + (1+\alpha)\, G^N{}_{t+\Delta t} - \alpha\, G^N_t,$$

  where $M^{NM} = M^{NM}\left(u^M, \dot{u}^M, H^\alpha, p^\beta, t, \ldots\right)$ and $G^N = G^N\left(u^M, \dot{u}^M, H^\alpha, p^\beta, t, \ldots\right)$; that is, the highest time derivative of $u^M$ in $M^{NM}$ and $G^N$ is $\dot{u}^M$, so that

  $$-\frac{\partial F^N}{\partial \ddot{u}^M{}_{t+\Delta t}} = M^{NM}.$$

  Therefore, you must store $G^N_t$ as an internal state vector. If half-increment residual calculations are required, you must also store $G^N_{t^-}$ as an internal state vector, where $t^-$ indicates the time at the beginning of the previous increment. For $\alpha = 0$, $F^N = -M^{NM}\ddot{u}_{t+\Delta t} + G^N{}_{t+\Delta t}$ and $G^N_t$ is not needed. You must define AMATRX $= M^{NM}\left(d\ddot{u}/du\right) + (1+\alpha)\, C^{NM}\left(d\dot{u}/du\right) + (1+\alpha)\, K^{NM}$, where $C^{NM} = -\partial G^N{}_{t+\Delta t}/\partial\dot{u}^M$ and $K^{NM} = -\partial G^N{}_{t+\Delta t}/\partial u^M$. RHS $= F^N$ must also be defined and the state variables, $H^\alpha$, updated. Although the value of $\alpha$ given in the dynamic step definition is passed into *UEL*, the value of $\alpha$ can vary from element to element. For example, $\alpha$ can be set to zero for some elements in the model where numerical dissipation is not desired.

- LFLAGS(3)=5: Half-increment residual ($F^N_{1/2}$) calculation. Abaqus/Standard will adjust the time increment so that $\max\left|F^N_{1/2}\right| < tolerance$ (where $tolerance$ is specified in the dynamic step definition). The half-increment residual is defined as

  $$F^N_{1/2} = -M^{NM}\ddot{u}_{t+\Delta t/2} + (1+\alpha)\, G^N{}_{t+\Delta t/2} - \tfrac{\alpha}{2}\left(G^N_t + G^N_{t^-}\right),$$

where $t^-$ indicates the time at the beginning of the previous increment ($\alpha$ is a parameter of the Hilber-Hughes-Taylor time integration operator and will be set to zero if the backward Euler time integration operator is used). You must define RHS $= F^N_{1/2}$. To evaluate $M^{NM}$ and ${G^N}_{t+\Delta t/2}$, you must calculate ${H^\alpha}_{t+\Delta t/2}$. These half-increment values will not be saved. DTIME will still contain $\Delta t$ (not $\Delta t/2$). The values contained in U, V, A, and DU are half-increment values.

- LFLAGS(3)=4: Velocity jump calculation. Abaqus/Standard solves $-M^{NM}\,\Delta\dot{u}^M = 0$ for $\Delta\dot{u}^M$, so you must define AMATRX $= M^{NM}$.
- LFLAGS(3)=6: Initial acceleration calculation. Abaqus/Standard solves $-M^{NM}\ddot{u}^M + G^N = 0$ for $\ddot{u}^M$, so you must define AMATRX $= M^{NM}$ and RHS $= G^N$.

### Subspace-Based Dynamic Analysis (LFLAGS(1)=13)

- The requirements are identical to those of static analysis, except that the Jacobian (stiffness), AMATRX, is not required. No convergence checks are performed in this case.

### Quasi-Static Analysis (LFLAGS(1)=21)

- The requirements are identical to those of static analysis.

### Steady-State Heat Transfer Analysis (LFLAGS(1)=31)

- The requirements are identical to those of static analysis, except that the automatic convergence checks are applied to the heat flux residuals corresponding to degrees of freedom 11, 12, …

### Transient Heat Transfer Analysis ($\Delta\theta_{max}$) (LFLAGS(1)=32, 33)

- Automatic convergence checks are applied to the heat flux residuals corresponding to degrees of freedom 11, 12, …
- The backward difference scheme is always used for time integration; that is, Abaqus/Standard assumes that $\dot{u}_{t+\Delta t} = \Delta u/\Delta t$, where $\Delta u = u_{t+\Delta t} - u_t$ and so $d\dot{u}/du = 1/\Delta t$ always. For degrees of freedom 11, 12, …, $\max|\Delta u|$ will be compared against the user-prescribed maximum allowable nodal temperature change in an increment, $\Delta\theta_{max}$, for controlling the time integration accuracy.
- You need to define AMATRX $= K^{NM} + (1/\Delta t)\ C^{NM}$, where $C^{NM}$ is the heat capacity matrix and RHS $= F^N$, and must update the state variables, $H^\alpha$.

### Geostatic Analysis (LFLAGS(1)=61)

- Identical to static analysis, except that the automatic convergence checks are applied to the residuals corresponding to degrees of freedom 1–8.

### Steady-State Coupled Pore Fluid Diffusion/Stress Analysis (`LFLAGS(1)=62, 63`)

- Identical to static analysis, except that the automatic convergence checks are applied to the residuals corresponding to degrees of freedom 1–8.

### Transient Coupled Pore Fluid Diffusion/Stress (Consolidation) Analysis ($\Delta u_w^{max}$) (`LFLAGS(1)=64, 65`)

- Automatic convergence checks are applied to the residuals corresponding to degrees of freedom 1–8.
- The backward difference scheme is used for time integration; that is, $\dot{u}^M_{t+\Delta t} = \Delta u^M / \Delta t$, where $\Delta u^M = u^M_{t+\Delta t} - u^M_t$.
- For degree of freedom 8, $\max|\Delta u^M|$ will be compared against the user-prescribed maximum wetting liquid pore pressure change, $\Delta u_w^{max}$, for automatic control of the time integration accuracy.
- You must define AMATRX $= K^{NM} + (1/\Delta t)\, C^{NM}$, where $C^{NM}$ is the pore fluid capacity matrix and RHS $= F^N$, and must update the state variables, $H^\alpha$.

### Steady-State Fully Coupled Thermal-Stress Analysis (`LFLAGS(1)=71`)

- Identical to static analysis, except that the automatic convergence checks are applied to the residuals corresponding to degrees of freedom 1–7 and 11, 12, …

### Transient Fully Coupled Thermal-Stress Analysis ($\Delta\theta_{max}$) (`LFLAGS(1)=72,73`)

- Automatic convergence checks are applied to the residuals corresponding to degrees of freedom 1–7 and 11, 12, …
- The backward difference scheme is used for time integration; that is, $\dot{u}^M_{t+\Delta t} = \Delta u^M / \Delta t$, where $\Delta u^M = u^M_{t+\Delta t} - u^M_t$.
- For degrees of freedom 11, 12, …, $\max|\Delta u^M|$ will be compared against the user-prescribed maximum allowable nodal temperature change in an increment, $\Delta\theta_{max}$, for automatic control of the time integration accuracy.
- You must define AMATRX $= K^{NM} + (1/\Delta t)\, C^{NM}$, where $C^{NM}$ is the heat capacity matrix and RHS $= F^N$, and must update the state variables, $H^\alpha$.

### Steady-State Coupled Thermal-Electrical Analysis (`LFLAGS(1)=75`)

- The requirements are identical to those of static analysis, except that the automatic convergence checks are applied to the current density residuals corresponding to degree of freedom 9, in addition to the heat flux residuals.

### Transient Coupled Thermal-Electrical Analysis ($\Delta\theta_{max}$) (LFLAGS(1)=76, 77)

- Automatic convergence checks are applied to the current density residuals corresponding to degree of freedom 9 and to the heat flux residuals corresponding to degree of freedom 11.
- The backward difference scheme is always used for time integration; that is, Abaqus/Standard assumes that $\dot{u}_{t+\Delta t} = \Delta u/\Delta t$, where $\Delta u = u_{t+\Delta t} - u_t$. Therefore, $d\dot{u}/du = 1/\Delta t$ always. For degree of freedom 11 $\max|\Delta u|$ will be compared against the user-prescribed maximum allowable nodal temperature change in an increment, $\Delta\theta_{max}$, for controlling the time integration accuracy.
- You must define AMATRX $= K^{NM} + (1/\Delta t)\ C^{NM}$, where $C^{NM}$ is the heat capacity matrix and RHS $= F^N$, and must update the state variables, $H^\alpha$.

### Steady-State Coupled Thermal-Electrical-Structural Analysis (LFLAGS(1)=102)

- Identical to static analysis, except that the automatic convergence checks are applied to the residuals corresponding to degrees of freedom 1–7, 9, and 11.

### Transient Coupled Thermal-Electrical-Structural Analysis ($\Delta\theta_{max}$) (LFLAGS(1)=103,104)

- Automatic convergence checks are applied to the residuals corresponding to degrees of freedom 1–7, 9, and 11.
- The backward difference scheme is always used for time integration; that is, Abaqus/Standard assumes that $\dot{u}_{t+\Delta t} = \Delta u/\Delta t$, where $\Delta u = u_{t+\Delta t} - u_t$. Therefore, $d\dot{u}/du = 1/\Delta t$ always. For degree of freedom 11 $\max|\Delta u|$ will be compared against the user-prescribed maximum allowable nodal temperature change in an increment, $\Delta\theta_{max}$, for controlling the time integration accuracy.
- You must define AMATRX $= K^{NM} + (1/\Delta t)\ C^{NM}$, where $C^{NM}$ is the heat capacity matrix and RHS $= F^N$; and you must update the state variables, $H^\alpha$.

## Usage with Linear Perturbation Procedures

*General and Perturbation Procedures* describes the linear perturbation capabilities in Abaqus/Standard. Here, base state values of variables will be denoted by $u^M$, $H^\alpha$, etc. Perturbation values will be denoted by $\tilde{u}^M$, $\widetilde{H}^\alpha$, etc.

Abaqus/Standard will not call user subroutine *UEL* for the eigenvalue buckling prediction procedure.

For mode-based procedures, user subroutine *UEL* is called in a prior natural frequency extraction analysis for mass, stiffness, and nondiagonal damping contributions due to damping elements and damping material properties. It is also called to form the left-hand-side matrices and, in some cases, the recovery path.

In addition, for direct-solution and mode-based steady-state dynamic, complex eigenvalue extraction, matrix generation, and substructure generation procedures, Abaqus/Standard calls user subroutine *UEL* for mass, stiffness, and nondiagonal damping contributions due to damping elements and damping material properties.

### Static Analysis (LFLAGS(1)=1, 2)

- Abaqus/Standard will solve $K^{NM}\tilde{u}^{M} = \tilde{P}^{N}$ for $\tilde{u}^{M}$, where $K^{NM}$ is the base state stiffness matrix and the perturbation load vector, $\tilde{P}^{N}$, is a linear function of the perturbation loads, $\tilde{p}$; that is, $\tilde{P}^{N} = (\partial F / \partial \tilde{p})\tilde{p}$.
- LFLAGS(3)=1: You must define AMATRX $= K^{NM}$ and RHS $= \tilde{P}^{N}$.
- LFLAGS(3)=100: You must compute perturbations of the internal variables, $\widetilde{H}^{\alpha}$, and define RHS $= \tilde{P}^{N} - K^{NM}\tilde{u}^{M}$ for output purposes.

### Eigenfrequency Extraction Analysis (LFLAGS(1)=41)

- $F^{N} = -M^{NM}\ddot{\tilde{u}} + G^{N}\left(u^{M} + \tilde{u}^{M}, \ldots\right) = -M^{NM}\ddot{\tilde{u}} + \left(\partial G^{N} / \partial u^{M}\right)\tilde{u}^{M}$.
- Abaqus/Standard will solve $K^{NM}\phi_{i}^{M} = \omega_{i}^{2} M^{NM}\phi_{i}^{M}$ for $\phi_{i}^{N}$ and $\omega_{i}$, where $K^{NM} = -\partial F^{N} / \partial u^{M}$ is the base state stiffness matrix and $M^{NM} = -\partial F^{NM} / \partial \ddot{u}^{M}$ is the base state mass matrix.
- LFLAGS(3)=2: Define AMATRX $= K^{NM}$.
- LFLAGS(3)=4: Define AMATRX $= M^{NM}$.

### Direct Steady-State Analysis (LFLAGS(1)=95)

- LFLAGS(3)=2: Define stiffness matrix AMATRX $= K^{NM}$.
- LFLAGS(3)=4: Define mass matrix AMATRX $= M^{NM}$.
- LFLAGS(3)=3: Define damping matrix AMATR $= C^{NM}$.
- LFLAGS(7)=1: Define viscous damping matrix AMATRX $= {C_{viscous}}^{NM}$.
- LFLAGS(7)=2: Define structural damping matrix AMATRX $= {C_{structural}}^{NM}$.
- Reaction force (the right-hand vector in a recovery path) is calculated outside the user subroutine. It includes inertia force and forces due to damping. However, you can augment the reaction force with additional contributions when LFLAGS(3)=2 and NRHS=2 (denoting the recovery path).

### Mode-Based Dynamic Analysis (LFLAGS(1)=91, 92, 93, 94)

- LFLAGS(3)=2: Define stiffness matrix AMATRX $= K^{NM}$.
- LFLAGS(3)=4: Define mass matrix AMATRX $= M^{NM}$.
- LFLAGS(3)=3: Define damping matrix AMATR $= C^{NM}$.
- LFLAGS(7)=1: Define viscous damping matrix AMATRX $= {C_{viscous}}^{NM}$.
- LFLAGS(7)=2: Define structural damping matrix AMATRX $= {C_{structural}}^{NM}$.
- For these procedures, the user subroutine defines only the left-hand matrices. Right-hand vectors are calculated outside this user subroutine.

## Nondiagonal Damping in Linear Perturbation Procedures

- For all Abaqus/Standard procedures in which nondiagonal damping can be used, you must set the flag LFLAGS(3)=3 to indicate that the damping matrix will be provided. To provide the viscous damping matrix, you must also set LFLAGS(7)=1 and define AMATRX= $C_{viscous}^{NM}$. To provide structural damping matrix you must also set LFLAGS(7)=2 and define AMATRX= $C_{structural}^{NM}$.

## Example: Structural and Heat Transfer User Element

Both a structural and a heat transfer user element have been created to demonstrate the usage of subroutine *UEL*. These user-defined elements are applied in a number of analyses. The following excerpt is from the verification problem that invokes the structural user element in an implicit dynamics procedure:

```
*USER ELEMENT, NODES=2, TYPE=U1, PROPERTIES=4, COORDINATES=3,
VARIABLES=12
1, 2, 3
*ELEMENT, TYPE=U1
101, 101, 102
*ELGEN, ELSET=UTRUSS
101, 5
*UEL PROPERTY, ELSET=UTRUSS
0.002, 2.1E11, 0.3, 7200.
```

The user element consists of two nodes that are assumed to lie parallel to the *x*-axis. The element behaves like a linear truss element. The supplied element properties are the cross-sectional area, Young's modulus, Poisson's ratio, and density, respectively.

The next excerpt shows the listing of the subroutine. The user subroutine has been coded for use in a perturbation static analysis; general static analysis, including Riks analysis with load incrementation defined by the subroutine; eigenfrequency extraction analysis; and direct-integration dynamic analysis. The names of the verification input files associated with the subroutine and these procedures can be found in *UEL*. The subroutine performs all calculations required for the relevant procedures as described earlier in this section. The flags passed in through the LFLAGS array are used to associate particular calculations with solution procedures.

During a modified Riks analysis all force loads must be passed into UEL by means of distributed load definitions such that they are available for the definition of incremental load vectors; the load keys U*n* and U*n*NU must be used properly, as discussed in *User-Defined Elements*. The coding in subroutine *UEL* must distribute the loads into consistent equivalent nodal forces and account for them in the calculation of the RHS and ENERGY arrays.

```
      SUBROUTINE UEL(RHS,AMATRX,SVARS,ENERGY,NDOFEL,NRHS,NSVARS,
     1     PROPS,NPROPS,COORDS,MCRD,NNODE,U,DU,V,A,JTYPE,TIME,
     2     DTIME,KSTEP,KINC,JELEM,PARAMS,NDLOAD,JDLTYP,ADLMAG,
     3     PREDEF,NPREDF,LFLAGS,MLVARX,DDLMAG,MDLOAD,PNEWDT,
     4     JPROPS,NJPROP,PERIOD)
C
C#include <cdp.cmn>
      INCLUDE 'ABA_PARAM.INC'
      PARAMETER ( ZERO = 0.D0, HALF = 0.5D0, ONE = 1.D0)
      PARAMETER ( alfdyn = 0.1D0, betdyn=0.02d0, s=0.0D0)
C
      DIMENSION RHS(MLVARX,*),AMATRX(NDOFEL,NDOFEL),
     1     SVARS(NSVARS),ENERGY(8),PROPS(*),COORDS(MCRD,NNODE),
     2     U(NDOFEL),DU(MLVARX,*),V(NDOFEL),A(NDOFEL),TIME(2),
     3     PARAMS(3),JDLTYP(MDLOAD,*),ADLMAG(MDLOAD,*),
     4     DDLMAG(MDLOAD,*),PREDEF(2,NPREDF,NNODE),LFLAGS(*),
     5     JPROPS(*)
      DIMENSION SRESID(6)
```

```
C
C UEL SUBROUTINE FOR A HORIZONTAL TRUSS ELEMENT
C
C     SRESID - stores the static residual at time t+dt
C     SVARS  - In 1-6, contains the static residual at time t
C              upon entering the routine. SRESID is copied to
C              SVARS(1-6) after the dynamic residual has been
C              calculated.
C            - For half-increment residual calculations: In 7-12,
C              contains the static residual at the beginning
C              of the previous increment. SVARS(1-6) are copied
C              into SVARS(7-12) after the dynamic residual has
C              been calculated.
C
      AREA = PROPS(1)
      E    = PROPS(2)
      ANU  = PROPS(3)
      RHO  = PROPS(4)
C
      ALEN = ABS(COORDS(1,2)-COORDS(1,1))
      AK   = AREA*E/ALEN
      AM   = HALF*AREA*RHO*ALEN
C
      DO K1 = 1, NDOFEL
        SRESID(K1) = ZERO
        DO KRHS = 1, NRHS
          RHS(K1,KRHS) = ZERO
        END DO
        DO K2 = 1, NDOFEL
          AMATRX(K2,K1) = ZERO
        END DO
      END DO
C
      IF (LFLAGS(3).EQ.1) THEN
C       Static or nonlinear dynamic analysis
C       Normal incrementation
        IF (LFLAGS(1).EQ.1 .OR. LFLAGS(1).EQ.2) THEN
C         *STATIC
          AMATRX(1,1) =  AK
          AMATRX(4,4) =  AK
          AMATRX(1,4) = -AK
          AMATRX(4,1) = -AK
          IF (LFLAGS(4).NE.0) THEN
            FORCE  = AK*(U(4)-U(1))
            DFORCE = AK*(DU(4,1)-DU(1,1))
            SRESID(1) = -DFORCE
            SRESID(4) =  DFORCE
            RHS(1,1) = RHS(1,1)-SRESID(1)
            RHS(4,1) = RHS(4,1)-SRESID(4)
            ENERGY(2) = HALF*FORCE*(DU(4,1)-DU(1,1))
     *           + HALF*DFORCE*(U(4)-U(1))
     *           + HALF*DFORCE*(DU(4,1)-DU(1,1))
          ELSE
            FORCE = AK*(U(4)-U(1))
            SRESID(1) = -FORCE
            SRESID(4) =  FORCE
            RHS(1,1) = RHS(1,1)-SRESID(1)
            RHS(4,1) = RHS(4,1)-SRESID(4)
            DO KDLOAD = 1, NDLOAD
              IF (JDLTYP(KDLOAD,1).EQ.1001) THEN
                RHS(4,1)  = RHS(4,1)+ADLMAG(KDLOAD,1)
```

```
                ENERGY(8) = ENERGY(8)+(ADLMAG(KDLOAD,1)
     *             - HALF*DDLMAG(KDLOAD,1))*DU(4,1)
                IF (NRHS.EQ.2) THEN
C                 Riks
                  RHS(4,2) = RHS(4,2)+DDLMAG(KDLOAD,1)
                END IF
              END IF
            END DO
            ENERGY(2) = HALF*FORCE*(U(4)-U(1))
          END IF
        ELSE IF (LFLAGS(1).EQ.11 .OR. LFLAGS(1).EQ.12) THEN
C         *DYNAMIC
          ALPHA = PARAMS(1)
          BETA  = PARAMS(2)
          GAMMA = PARAMS(3)
C
          DADU = ONE/(BETA*DTIME**2)
          DVDU = GAMMA/(BETA*DTIME)
c
          dynt6 = GAMMA/(BETA*DTIME)
          dynt4 = ONE + ALPHA
C
C         LHS operator and RHS: Mass related terms
          DO K1 = 1, NDOFEL
            AMATRX(K1,K1) = AM*DADU + dynt6*dynt4*am*alfdyn
            RHS(K1,1) = RHS(K1,1)-AM*A(K1)
          END DO
C
C         LHS operator: stiffness related terms
          AMATRX(1,1) = AMATRX(1,1)+(ONE+ALPHA)*AK+dynt6*dynt4*betdyn*AK

          AMATRX(4,4) = AMATRX(4,4)+(ONE+ALPHA)*AK+dynt6*dynt4*betdyn*AK

          AMATRX(1,4) = AMATRX(1,4)-(ONE+ALPHA)*AK-dynt6*dynt4*betdyn*AK

          AMATRX(4,1) = AMATRX(4,1)-(ONE+ALPHA)*AK-dynt6*dynt4*betdyn*AK
C         Force
          FORCE = AK*(U(4)-U(1))+(betdyn*AK)*(v(4)-v(1))
          SRESID(1) = -FORCE+alfdyn*AM*v(1)
          SRESID(4) =  FORCE+alfdyn*AM*v(4)

C         RHS due to stiffnes and damping
          RHS(1,1) = RHS(1,1) -
     *         ((ONE+ALPHA)*SRESID(1)-ALPHA*SVARS(1))
          RHS(4,1) = RHS(4,1) -
     *         ((ONE+ALPHA)*SRESID(4)-ALPHA*SVARS(4))
          ENERGY(1) = ZERO
          DO K1 = 1, NDOFEL
            SVARS(K1+6) = SVARS(k1)
            SVARS(K1)   = SRESID(K1)
            ENERGY(1)   = ENERGY(1)+HALF*V(K1)*AM*V(K1)
          END DO
C
          ENERGY(2) = HALF*AK*(U(4)-U(1))*(U(4)-U(1))
        END IF
C

       ELSE IF (LFLAGS(3).EQ.2) THEN
C          Stiffness matrix requested
           DO K1 = 1, NDOFEL
            AMATRX(K1,K1) = AK
```

```
           END DO
           AMATRX(1,4) = -AK
           AMATRX(4,1) = -AK
      ELSE IF (LFLAGS(3).EQ.3) THEN
C       Damping requested
        if(lflags(7).eq.1) then
C          Viscous damping matrix requested
C          Mass matrix diagonal only
           DO K1 = 1, NDOFEL
            AMATRX(K1,K1) = betdyn*AK+alfdyn*AM
           END DO
           AMATRX(1,4) = -betdyn*AK
           AMATRX(4,1) = -betdyn*AK
        else if(lflags(7).eq.2) then
C          Structural damping matrix requested
           DO K1 = 1, NDOFEL
            AMATRX(K1,K1) = s*AK
           END DO
           AMATRX(1,4) = -s*AK
           AMATRX(4,1) = -s*AK
        end if
      ELSE IF (LFLAGS(3).EQ.4) THEN
C       Mass matrix
        DO K1 = 1, NDOFEL
          AMATRX(K1,K1) = AM
        END DO
      ELSE IF (LFLAGS(3).EQ.5) THEN
C       Half-increment residual calculation
        ALPHA = PARAMS(1)
        FORCE = AK*(U(4)-U(1))
        SRESID(1) = -FORCE
        SRESID(4) =  FORCE
        RHS(1,1) = RHS(1,1)-AM*A(1)-(ONE+ALPHA)*SRESID(1)
     *       + HALF*ALPHA*( SVARS(1)+SVARS(7) )
        RHS(4,1) = RHS(4,1)-AM*A(4)-(ONE+ALPHA)*SRESID(4)
     *       + HALF*ALPHA*( SVARS(4)+SVARS(10) )
      ELSE IF (LFLAGS(3).EQ.6) THEN
C       Initial acceleration calculation
        DO K1 = 1, NDOFEL
          AMATRX(K1,K1) = AM
        END DO
        FORCE = AK*(U(4)-U(1))
        SRESID(1) = -FORCE
        SRESID(4) =  FORCE
        RHS(1,1) = RHS(1,1)-SRESID(1)
        RHS(4,1) = RHS(4,1)-SRESID(4)
        ENERGY(1) = ZERO
        DO K1 = 1, NDOFEL
          SVARS(K1) = SRESID(K1)
          ENERGY(1) = ENERGY(1)+HALF*V(K1)*AM*V(K1)
        END DO
        ENERGY(2) = HALF*FORCE*(U(4)-U(1))
      ELSE IF (LFLAGS(3).EQ.100) THEN
C       Output for perturbations
        IF (LFLAGS(1).EQ.1 .OR. LFLAGS(1).EQ.2) THEN
C         *STATIC
          FORCE  = AK*(U(4)-U(1))
          DFORCE = AK*(DU(4,1)-DU(1,1))
          SRESID(1) = -DFORCE
          SRESID(4) =  DFORCE
          RHS(1,1) = RHS(1,1)-SRESID(1)
```

```
          RHS(4,1) = RHS(4,1)-SRESID(4)
          ENERGY(2) = HALF*FORCE*(DU(4,1)-DU(1,1))
     *          + HALF*DFORCE*(U(4)-U(1))
     *          + HALF*DFORCE*(DU(4,1)-DU(1,1))
          DO KVAR = 1, NSVARS
            SVARS(KVAR) = ZERO
          END DO
          SVARS(1) = RHS(1,1)
          SVARS(4) = RHS(4,1)
        ELSE IF (LFLAGS(1).EQ.41.or.LFLAGS(1).EQ.47) THEN
C         *FREQUENCY or *COMPLEX FREQUENCY
          DO KRHS = 1, NRHS
            DFORCE = AK*(DU(4,KRHS)-DU(1,KRHS))
            SRESID(1) = -DFORCE
            SRESID(4) =  DFORCE
            RHS(1,KRHS) = RHS(1,KRHS)-SRESID(1)
            RHS(4,KRHS) = RHS(4,KRHS)-SRESID(4)
          END DO
          DO KVAR = 1, NSVARS
            SVARS(KVAR) = ZERO
          END DO
          SVARS(1) = RHS(1,1)
          SVARS(4) = RHS(4,1)
        END IF
      END IF
C
      RETURN
      END
```

# UELEMDRESP: User subroutine to define an element design response and its derivatives.

**Products:** Abaqus/Standard

**References:**

- *Adjoint Design Sensitivity Analysis Overview*
- **ELEMENT RESPONSE*

## Overview

User subroutine *UELEMDRESP*:

- can be used to define an element design response that depends on either the stress tensor, the strain tensor, the logarithmic strain tensor, the nominal strain tensor, or the plastic strain tensor;
- is called at all material calculation points of elements that are members of the specified element set over which the design response is computed; and
- must provide the value of the design response function as well as its derivatives.

## User-Defined Element Design Response

The adjoint design sensitivity capabilities in Abaqus/Standard support the following element design responses:

- stress (MISES, S_MISES, SP3, SP1),
- principal strain (EP3, EP1),
- principal logarithmic strain (LEP3, LEP1),
- principal nominal strain (NEP3, NEP1),
- and plastic strain magnitude (PEMAG),

which are well-known invariants of the primal Abaqus variables stress (S), strain (E), logarithmic strain (LE), nominal strain (NE), and plastic strain (PE), respectively. If you want to use a different invariant or failure criterion that is based on the stress, strain, logarithmic strain, nominal strain, or the plastic strain tensor, you can use user subroutine *UELEMDRESP* to define this function and its derivatives.

User subroutine *UELEMDRESP* is called at all material points for all the elements of the element set over which the element response is calculated.

## User Subroutine Interface

```
     Subroutine uelemdresp (nDi, nShr, nTens, lFlag,  nOutput, index_output,
 primal_var,
     1 dresp_id, output_id, dresp_derivatives, temp, field, statev, direct,

      2 t, cElent, time, coord, cMName, orName, nField, nStatv, noel, nPt,
 layer, kSpt,
      3 kStep, kInc, i_array, niarray, r_array, nrarray, c_array, ncarray)
C
      include 'aba_param.inc'
C
      dimension primal_var (*) , dresp_derivatives (*),
index_output(nOutput,2)
```

```
     dimension field(nField), statev(nStatv), direct(3,3), t(3,3), time(2),
coord(*)
      dimension i_array(niarray), r_array(nrarray), c_array(ncarray)
      character*80 dresp_id, output_id(noutput)

      User code to define dresp_value and dresp_derivatives (*)

      return
      end
```

## Variables to Be Defined

| | |
|---|---|
| `dresp_value` | Value of the design response. |
| `dresp_derivatives (*)` | An array containing the derivatives of the design response with respect to the individual components of the tensor output variable S, E, LE, NE, or PE. |

## Variables Passed in for Information

| | |
|---|---|
| `nDi` | Number of direct tensor components. |
| `nShr` | Number of shear tensor components. |
| `nTens` | Total number of tensor components. |
| `lFlag` | When `lFlag=1`, define the design response only. When `lFlag=2`, define the design response derivatives only. |
| `nOutput` | Number of primal output variables on which the design response depends. This is the number of output variables specified in the data line under **ELEMENT RESPONSE*. `nOutput` can be set only to 1. |
| `index_output (nOutput, 2)` | An integer array with `nOutput` rows. `nOutput` can be set only to 1. The first column is an index into the `primal_var` array for the first component of the output variable, and the second column contains the size (number of components) of the output variable. For example, if you specify S as the primal output variable, `index_output` will be `(1,6)`. |
| `primal_var (*)` | Components of the basic output variable specified in the data line following **ELEMENT RESPONSE* (that is, components of the output variable S, E, LE, NE, or PE). For example, if you specified S as the basic output variable, the entries in `primal_var` are `(S11, S22, S33, S12, S13, S23)`. |
| `dresp_id` | Design response identifier label. |
| `output_id(*)` | Output variable identifier (either S, E, LE, NE, or PE). |
| `temp` | Current temperature. |
| `field (nField)` | Current values of all field variables. |
| `statv (nStatv)` | Current value of all solution-dependent state variables. |
| `direct (3,3)` | An array containing the direction cosines of the material directions in terms of the global basis directions. For example, `direct(1,1)`, `direct(2,1)`, `direct(3,1)` give the `(1, 2, 3)` components of the first material direction and `direct(1,2)`, `direct(2,2)`, `direct(3,2)` give the second material direction. For shell and membrane elements, the first two directions are in the plane of the element and the third direction is the normal. This information is not available for beam elements. |
| `t (3,3)` | An array containing the direction cosines of the material orientation components relative to the element basis directions. This is the orientation that defines the material directions `(direct)` in terms of the element basis directions. For continuum elements, `t` and `DIRECT` are identical. For shell and membrane elements, `t(1,1)=`, `t(1,2)=`, `t(2,1)=`, `t(2,2)=`, `t(3,3)=`, and all other components are zero, where the counterclockwise rotation around the normal vector defines the |

| | |
|---|---|
| | orientation. If no orientation is used, `t` is an identity matrix. The orientation is not available for beam elements. |
| `cElent` | Characteristic element length. This is a typical length of a line across an element for a first-order element; it is half of the same typical length for a second-order element. For beams and trusses, it is a characteristic length along the element axis. For membranes and shells, it is a characteristic length in the reference surface. For axisymmetric elements, it is a characteristic length in the (r, z) plane only. |
| `time (1)` | Value of step time. |
| `time (2)` | Value of total time. |
| `coord` | Coordinates at this material point. |
| `cMName` | Encoded user-specified material name that can be passed to the `getMaterialName` utility to decode it. |
| `orName` | Encoded user-specified local orientation name that can be passed to the `getOrientationName` utility to decode it. |
| `nField` | Number of field variables defined at this material point. |
| `nStatv` | User-defined number of solution-dependent state variables. |
| `noel` | Element number. |
| `nPt` | Integration point number. |
| `layer` | Layer number (for composite shells and layered solids). |
| `kSpt` | Section point number within the current layer. |
| `kStep` | Step number. |
| `kInc` | Increment number. |
| `i_array(niarray)` | Array of integers. `i_array(1)=klc`, the current load case number, while `i_array(2)=nlc`, the total number of load cases. |
| `niarray` | Size of `i_array`, currently equal to 2. |
| `r_array(nrarray)` | Not used. |
| `nrarray` | Size of `r_array`. |
| `c_array(ncarray)` | Not used. |
| `ncarray` | Size of `c_array`. |

# UELMAT: User subroutine to define an element with access to Abaqus materials.

**Products:** Abaqus/Standard

**References:**

- *User-Defined Elements*
- **UEL PROPERTY*
- **USER ELEMENT*
- *Accessing Abaqus Materials*
- *Accessing Abaqus Thermal Materials*

## Overview



**Warning:** This feature is intended for advanced users only. Its use in all but the simplest test examples will require considerable coding by the user/developer. *User-Defined Elements* should be read before proceeding.

User subroutine UELMAT:

- will be called for each element that is of a general user-defined element type (i.e., not defined by a linear stiffness or mass matrix read either directly or from results file data) each time element calculations are required;
- (or subroutines called by user subroutine UELMAT) must perform all of the calculations for the element, appropriate to the current activity in the analysis;
- can access some of the Abaqus materials through utility routines MATERIAL_LIB_MECH and MATERIAL_LIB_HT;
- is available for a subset of the procedures supported for user subroutine UEL (see *User-Defined Elements*); and
- is available for plane stress and three-dimensional element types in a stress/displacement analysis and for two-dimensional and three-dimensional element types in a heat transfer analysis (see *User-Defined Elements*).

## User Subroutine Interface

```
      SUBROUTINE UELMAT(RHS,AMATRX,SVARS,ENERGY,NDOFEL,NRHS,NSVARS,
     1 PROPS,NPROPS,COORDS,MCRD,NNODE,U,DU,V,A,JTYPE,TIME,DTIME,
     2 KSTEP,KINC,JELEM,PARAMS,NDLOAD,JDLTYP,ADLMAG,PREDEF,NPREDF,
     3 LFLAGS,MLVARX,DDLMAG,MDLOAD,PNEWDT,JPROPS,NJPROP,PERIOD,
     4 MATERIALLIB)
C
      INCLUDE 'ABA_PARAM.INC'
C
      DIMENSION RHS(MLVARX,*),AMATRX(NDOFEL,NDOFEL),PROPS(*),
     1 SVARS(*),ENERGY(8),COORDS(MCRD,NNODE),U(NDOFEL),
     2 DU(MLVARX,*),V(NDOFEL),A(NDOFEL),TIME(2),PARAMS(*),
     3 JDLTYP(MDLOAD,*),ADLMAG(MDLOAD,*),DDLMAG(MDLOAD,*),
     4 PREDEF(2,NPREDF,NNODE),LFLAGS(*),JPROPS(*)


      user coding to define RHS, AMATRX, SVARS, ENERGY, and PNEWDT
```

```
      RETURN
      END
```

## Variables to Be Defined

These arrays depend on the value of the LFLAGS array.

RHS An array containing the contributions of this element to the right-hand-side vectors of the overall system of equations. For most nonlinear analysis procedures, NRHS=1 and RHS should contain the residual vector. The exception is the modified Riks static procedure (*Static Stress Analysis*), for which NRHS=2 and the first column in RHS should contain the residual vector and the second column should contain the increments of external load on the element. RHS(K1,K2) is the entry for the K1th degree of freedom of the element in the K2th right-hand-side vector.

AMATRX An array containing the contribution of this element to the Jacobian (stiffness) or other matrix of the overall system of equations. The particular matrix required at any time depends on the entries in the LFLAGS array (see below).

All nonzero entries in AMATRX should be defined, even if the matrix is symmetric. If you do not specify that the matrix is unsymmetric when you define the user element, Abaqus/Standard will use the symmetric matrix defined by $\frac{1}{2}\left([A]+[A]^T\right)$, where $[A]$ is the matrix defined as AMATRX in this subroutine. If you specify that the matrix is unsymmetric when you define the user element, Abaqus/Standard will use AMATRX directly.

SVARS An array containing the values of the solution-dependent state variables associated with this element. The number of such variables is NSVARS (see below). You define the meaning of these variables.

For general nonlinear steps this array is passed into *UELMAT* containing the values of these variables at the start of the current increment. They should be updated to be the values at the end of the increment, unless the procedure during which *UELMAT* is being called does not require such an update; this requirement depends on the entries in the LFLAGS array (see below). For linear perturbation steps this array is passed into *UELMAT* containing the values of these variables in the base state. They should be returned containing perturbation values if you wish to output such quantities.

When KINC is equal to zero, the call to *UELMAT* is made for zero increment output (see *About Output*). In this case the values returned will be used only for output purposes and are not updated permanently.

ENERGY For general nonlinear steps array ENERGY contains the values of the energy quantities associated with the element. The values in this array when *UELMAT* is called are the element energy quantities at the start of the current increment. They should be updated to the values at the end of the current increment. For linear perturbation steps the array is passed into *UELMAT* containing the energy in the base state. They should be returned containing perturbation values if you wish to output such quantities. The entries in the array are as follows:

| | |
|---|---|
| ENERGY(1) | Kinetic energy. |
| ENERGY(2) | Elastic strain energy. |
| ENERGY(3) | Creep dissipation. |
| ENERGY(4) | Plastic dissipation. |
| ENERGY(5) | Viscous dissipation. |
| ENERGY(6) | “Artificial strain energy” associated with such effects as artificial stiffness introduced to control hourglassing or other singular modes in the element. |
| ENERGY(7) | Electrostatic energy. |
| ENERGY(8) | Incremental work done by loads applied within the user element. |

When KINC is equal to zero, the call to *UELMAT* is made for zero increment output (see *About Output*). In this case the energy values returned will be used only for output purposes and are not updated permanently.

## Variables That Can Be Updated

PNEWDT Ratio of suggested new time increment to the time increment currently being used (DTIME, see below). This variable allows you to provide input to the automatic time incrementation algorithms in Abaqus/Standard (if automatic time incrementation is chosen). It is useful only during equilibrium iterations with the normal time incrementation, as indicated by LFLAGS(3)=1. During a severe discontinuity iteration (such as contact changes), PNEWDT is ignored unless CONVERT SDI=YES is specified for this step. The usage of PNEWDT is discussed below.

PNEWDT is set to a large value before each call to *UELMAT*.

If PNEWDT is redefined to be less than 1.0, Abaqus/Standard must abandon the time increment and attempt it again with a smaller time increment. The suggested new time increment provided to the automatic time integration algorithms is PNEWDT × DTIME, where the PNEWDT used is the minimum value for all calls to user subroutines that allow redefinition of PNEWDT for this iteration.

If PNEWDT is given a value that is greater than 1.0 for all calls to user subroutines for this iteration and the increment converges in this iteration, Abaqus/Standard may increase the time increment. The suggested new time increment provided to the automatic time integration algorithms is PNEWDT × DTIME, where the PNEWDT used is the minimum value for all calls to user subroutines for this iteration.

If automatic time incrementation is not selected in the analysis procedure, values of PNEWDT that are greater than 1.0 will be ignored and values of PNEWDT that are less than 1.0 will cause the job to terminate.

## Variables Passed in for Information

**Arrays:**

PROPS A floating point array containing the NPROPS real property values defined for use with this element. NPROPS is the user-specified number of real property values. See *Defining the Element Properties*.

JPROPS An integer array containing the NJPROP integer property values defined for use with this element. NJPROP is the user-specified number of integer property values. See *Defining the Element Properties*.

COORDS An array containing the original coordinates of the nodes of the element. COORDS(K1,K2) is the K1th coordinate of the K2th node of the element.

U, DU, V, A Arrays containing the current estimates of the basic solution variables (displacements, rotations, temperatures, depending on the degree of freedom) at the nodes of the element at the end of the current increment. Values are provided as follows:

| | |
|---|---|
| U(K1) | Total values of the variables. If this is a linear perturbation step, it is the value in the base state. |
| DU(K1,KRHS) | Incremental values of the variables for the current increment for right-hand-side KRHS. If this is an eigenvalue extraction step, this is the eigenvector magnitude for eigenvector KRHS. For steady-state dynamics KRHS = 1 denotes real components of perturbation displacement and KRHS = 2 denotes imaginary components of perturbation displacement. |
| V(K1) | Time rate of change of the variables (velocities, rates of rotation). Defined for implicit dynamics only (LFLAGS(1) = 11 or 12). |
| A(K1) | Accelerations of the variables. Defined for implicit dynamics only (LFLAGS(1) = 11 or 12). |

**Arrays:**

JDLTYP An array containing the integers used to define distributed load types for the element. Loads of type U*n* are identified by the integer value *n* in JDLTYP; loads of type U*n*NU are identified by the negative integer value $-n$ in JDLTYP. JDLTYP(K1,K2) is the identifier of the K1th distributed load in the K2th load case. For general nonlinear steps K2 is always 1.

ADLMAG For general nonlinear steps ADLMAG(K1,1) is the total load magnitude of the K1th distributed load at the end of the current increment for distributed loads of type U*n*. For distributed loads of type U*n*NU, the load magnitude is defined in *UELMAT*; therefore, the corresponding entries in ADLMAG are zero. For linear perturbation steps ADLMAG(K1,1) contains the total load magnitude of the K1th distributed load of type U*n* applied in the base state. Base state loading of type U*n*NU must be dealt with inside *UELMAT*. ADLMAG(K1,2), ADLMAG(K1,3), etc. are currently not used.

DDLMAG For general nonlinear steps DDLMAG contains the increments in the magnitudes of the distributed loads that are currently active on this element for distributed loads of type U*n*. DDLMAG(K1,1) is the increment of magnitude of the load for the current time increment. The increment of load magnitude is needed to compute the external work contribution. For distributed loads of type U*n*NU the load magnitude is defined in *UELMAT*; therefore, the corresponding entries in DDLMAG are zero. For linear perturbation steps DDLMAG(K1,K2) contains the perturbation in the magnitudes of the distributed loads that are currently active on this element for distributed loads of type U*n*. K1 denotes the K1th perturbation load active on the element. K2 is always 1, except for steady-state dynamics, where K2=1 for real loads and K2=2 for imaginary loads. Perturbation loads of type U*n*NU must be dealt with inside *UELMAT*.

PREDEF An array containing the values of predefined field variables, such as temperature in an uncoupled stress/displacement analysis, at the nodes of the element (*Predefined Fields*).

The first index of the array, K1, is either 1 or 2, with 1 indicating the value of the field variable at the end of the increment and 2 indicating the increment in the field variable. The second index, K2, indicates the variable: the temperature corresponds to index 1, and the predefined field variables correspond to indices 2 and above. In cases where temperature is not defined, the predefined field variables begin with index 1. The third index, K3, indicates the local node number on the element.

| | |
|---|---|
| PREDEF(K1,1,K3) | Temperature. |
| PREDEF(K1,2,K3) | First predefined field variable. |
| PREDEF(K1,3,K3) | Second predefined field variable. |
| Etc. | Any other predefined field variable. |
| PREDEF(K1,K2,K3) | Total or incremental value of the K2th predefined field variable at the K3th node of the element. |
| PREDEF(1,K2,K3) | Values of the variables at the end of the current increment. |
| PREDEF(2,K2,K3) | Incremental values corresponding to the current time increment. |

PARAMS An array containing the parameters associated with the solution procedure. The entries in this array depend on the solution procedure currently being used when *UELMAT* is called, as indicated by the entries in the LFLAGS array (see below).

For implicit dynamics (LFLAGS(1) = 11 or 12) PARAMS contains the integration operator values, as:

| | |
|---|---|
| PARAMS(1) | $\alpha$ |
| PARAMS(2) | $\beta$ |
| PARAMS(3) | $\gamma$ |

**Arrays:**

LFLAGS An array containing the flags that define the current solution procedure and requirements for element calculations. Detailed requirements for the various Abaqus/Standard procedures are defined earlier in this section.

| | |
|---|---|
| LFLAGS(1) | Defines the procedure type. See *Results File* for the key used for each procedure. |
| LFLAGS(2)=0 | Small-displacement analysis. |
| LFLAGS(2)=1 | Large-displacement analysis (nonlinear geometric effects included in the step; see *General and Perturbation Procedures*). |
| LFLAGS(3)=1 | Normal implicit time incrementation procedure. User subroutine *UELMAT* must define the residual vector in RHS and the Jacobian matrix in AMATRX. |
| LFLAGS(3)=2 | Define the current stiffness matrix (AMATRX$= K^{NM} = -\partial F^N/\partial u^M$ or $-\partial G^N/\partial u^M$) only. |
| LFLAGS(3)=3 | Define the current damping matrix (AMATRX$= C^{NM} = -\partial F^N/\partial \dot{u}^M$ or $-\partial G^N/\partial \dot{u}^M$) only. |
| LFLAGS(3)=4 | Define the current mass matrix (AMATRX$= M^{NM} = -\partial F^N/\partial \ddot{u}^M$) only. Abaqus/Standard always requests an initial mass matrix at the start of the analysis. |
| LFLAGS(3)=5 | Define the current residual or load vector (RHS $= F^N$) only. |
| LFLAGS(3)=6 | Define the current mass matrix and the residual vector for the initial acceleration calculation (or the calculation of accelerations after impact). |
| LFLAGS(3)=100 | Define perturbation quantities for output. |
| LFLAGS(4)=0 | The step is a general step. |
| LFLAGS(4)=1 | The step is a linear perturbation step. |
| LFLAGS(5)=0 | The current approximations to $u^M$, etc. were based on Newton corrections. |
| LFLAGS(5)=1 | The current approximations were found by extrapolation from the previous increment. |

TIME(1) Current value of step time or frequency.

TIME(2) Current value of total time.

**Scalar parameters:**

| | |
|---|---|
| DTIME | Time increment. |
| PERIOD | Time period of the current step. |
| NDOFEL | Number of degrees of freedom in the element. |
| MLVARX | Dimensioning parameter used when several displacement or right-hand-side vectors are used. |
| NRHS | Number of load vectors. NRHS is 1 in most nonlinear problems: it is 2 for the modified Riks static procedure (*Static Stress Analysis*), and it is greater than 1 in some linear analysis procedures and during substructure generation. |
| NSVARS | User-defined number of solution-dependent state variables associated with the element (*Defining the Number of Solution-Dependent Variables That Must Be Stored within the Element*). |
| NPROPS | User-defined number of real property values associated with the element (*Defining the Element Properties*). |
| NJPROP | User-defined number of integer property values associated with the element (*Defining the Element Properties*). |

**Scalar parameters:**

| | |
|---|---|
| MCRD | MCRD is defined as the maximum of the user-defined maximum number of coordinates needed at any node point (*Defining the Maximum Number of Coordinates Needed at Any Nodal Point*) and the value of the largest active degree of freedom of the user element that is less than or equal to 3. For example, if you specify that the maximum number of coordinates is 1 and the active degrees of freedom of the user element are 2, 3, and 6, MCRD will be 3. If you specify that the maximum number of coordinates is 2 and the active degrees of freedom of the user element are 11 and 12, MCRD will be 2. |
| NNODE | User-defined number of nodes on the element (*Defining the Number of Nodes Associated with the Element*). |
| JTYPE | Integer defining the element type. This is the user-defined integer value *n* in element type U*n* (*Assigning an Element Type Key to a User-Defined Element*). |
| KSTEP | Current step number. |
| KINC | Current increment number. |
| JELEM | User-assigned element number. |
| NDLOAD | Identification number of the distributed load or flux currently active on this element. |
| MDLOAD | Total number of distributed loads and/or fluxes defined on this element. |
| NPREDF | Number of predefined field variables, including temperature. For user elements Abaqus/Standard uses one value for each field variable per node. |
| MATERIALLIB | A variable that must be passed to the utility routines performing material point computations. |

## UELMAT Conventions

The solution variables (displacement, velocity, etc.) are arranged on a node/degree of freedom basis. The degrees of freedom of the first node are first, followed by the degrees of freedom of the second node, etc.

## Usage with General Nonlinear Procedures

The values of $u^N$ (and, in direct-integration dynamic steps, $\dot{u}^N$ and $\ddot{u}^N$) enter user subroutine *UELMAT* as their latest approximations at the end of the time increment; that is, at time $t + \Delta t$.

The values of $H^\alpha$ enter the subroutine as their values at the beginning of the time increment; that is, at time *t*. It is your responsibility to define suitable time integration schemes to update $H^\alpha$. To ensure accurate, stable integration of internal state variables, you can control the time incrementation via PNEWDT.

The values of $p^\beta$ enter the subroutine as the values of the total load magnitude for the $\beta$th distributed load at the end of the increment. Increments in the load magnitudes are also available.

In the following descriptions of the user element's requirements, it will be assumed that LFLAGS(3)=1 unless otherwise stated.

### Static Analysis (LFLAGS(1)=1, 2)

- $F^N = F^N\left(u^M, H^\alpha, p^\beta, t\right)$.
- Automatic convergence checks are applied to the force residuals corresponding to degrees of freedom 1–7.
- You must define AMATRX$= K^{NM} = -\partial F^N / \partial u^M$ and RHS$= F^N$ and update the state variables, $H^\alpha$.

## Direct-Integration Dynamic Analysis (LFLAGS(1)=11, 12)

- Automatic convergence checks are applied to the force residuals corresponding to degrees of freedom 1–7.
- LFLAGS(3)=1: Normal time increment. Either the Hilber-Hughes-Taylor or the backward Euler time integration scheme will be used. With $\alpha$ set to zero for the backward Euler, both schemes imply

$$F^N = -M^{NM}\ddot{u}_{t+\Delta t} + (1+\alpha)\, G^N - \alpha\, G^N_t,$$

where $M^{NM} = M^{NM}\left(u^M, \dot{u}^M, H^\alpha, p^\beta, t, \ldots\right)$ and $G^N = G^N\left(u^M, \dot{u}^M, H^\alpha, p^\beta, t, \ldots\right)$; that is, the highest time derivative of $u^M$ in $M^{NM}$ and $G^N$ is $\dot{u}^M$, so that

$$-\frac{\partial F^N}{\partial \ddot{u}^M{}_{t+\Delta t}} = M^{NM}.$$

Therefore, you must store $G^N_t$ as an internal state vector. If half-increment residual calculations are required, you must also store $G^N_{t^-}$ as an internal state vector, where $t^-$ indicates the time at the beginning of the previous increment. For $\alpha = 0$, $F^N = -M^{NM}\ddot{u}_{t+\Delta t} + G^N{}_{t+\Delta t}$ and $G^N_t$ is not needed. You must define AMATRX $= M^{NM}\,(d\ddot{u}/du) + (1+\alpha)\,C^{NM}\,(d\dot{u}/du) + (1+\alpha)\,K^{NM}$, where $C^{NM} = -\partial G^N{}_{t+\Delta t}/\partial \dot{u}^M$ and $K^{NM} = -\partial G^N{}_{t+\Delta t}/\partial u^M$. RHS $= F^N$ must also be defined and the state variables, $H^\alpha$, updated. Although the value of $\alpha$ given in the dynamic step definition is passed into *UELMAT*, the value of $\alpha$ can vary from element to element. For example, $\alpha$ can be set to zero for some elements in the model where numerical dissipation is not desired.

- LFLAGS(3)=5: Half-increment residual ($F^N_{1/2}$) calculation. Abaqus/Standard will adjust the time increment so that $\max\left|F^N_{1/2}\right| < tolerance$ (where $tolerance$ is specified in the dynamic step definition). The half-increment residual is defined as

$$F^N_{1/2} = -M^{NM}\ddot{u}_{t+\Delta t/2} + (1+\alpha)\, G^N{}_{t+\Delta t/2} - \tfrac{\alpha}{2}\left(G^N_t + G^N_{t^-}\right),$$

where $t^-$ indicates the time at the beginning of the previous increment ($\alpha$ is a parameter of the Hilber-Hughes-Taylor time integration operator and will be set to zero if the backward Euler time integration operator is used). You must define RHS $= F^N_{1/2}$. To evaluate $M^{NM}$ and $G^N{}_{t+\Delta t/2}$, you must calculate $H^\alpha{}_{t+\Delta t/2}$. These half-increment values will not be saved. DTIME will still contain $\Delta t$ (not $\Delta t/2$). The values contained in U, V, A, and DU are half-increment values.

- LFLAGS(3)=4: Velocity jump calculation. Abaqus/Standard solves $-M^{NM}\,\Delta\dot{u}^M = 0$ for $\Delta\dot{u}^M$, so you must define AMATRX $= M^{NM}$.
- LFLAGS(3)=6: Initial acceleration calculation. Abaqus/Standard solves $-M^{NM}\ddot{u}^M + G^N = 0$ for $\ddot{u}^M$, so you must define AMATRX $= M^{NM}$ and RHS $= G^N$.

## Quasi-Static Analysis (LFLAGS(1)=21)

- The requirements are identical to those of static analysis.

### Steady-State Heat Transfer Analysis (`LFLAGS(1)=31`)

- The requirements are identical to those of static analysis, except that the automatic convergence checks are applied to the heat flux residuals corresponding to degrees of freedom 11, 12, …

### Transient Heat Transfer Analysis ( $_{max}$) (`LFLAGS(1)=32, 33`)

- Automatic convergence checks are applied to the heat flux residuals corresponding to degrees of freedom 11, 12, …
- The backward difference scheme is always used for time integration; that is, Abaqus/Standard assumes that $\dot{u}_{t+\Delta t} = \Delta u/\Delta t$, where $\Delta u = u_{t+\Delta t} - u_t$ and so $d\dot{u}/du = 1/\Delta t$ always. For degrees of freedom 11, 12, …, $\mathbf{max}|\Delta u|$ will be compared against the user-prescribed maximum allowable nodal temperature change in an increment, $\Delta\theta_{max}$, for controlling the time integration accuracy.
- You need to define AMATRX $= K^{NM} + (1/\Delta t)\, C^{NM}$, where $C^{NM}$ is the heat capacity matrix and RHS $= F^N$, and must update the state variables, $H^\alpha$.

## Usage with Linear Perturbation Procedures

*General and Perturbation Procedures* describes the linear perturbation capabilities in Abaqus/Standard. Here, base state values of variables will be denoted by $u^M$, $H^\alpha$, etc. Perturbation values will be denoted by $\tilde{u}^M$, $\widetilde{H}^\alpha$, etc.

Abaqus/Standard will not call user subroutine *UELMAT* for the following procedures: eigenvalue buckling prediction, response spectrum, transient modal dynamic, steady-state dynamic (modal and direct), and random response.

### Static Analysis (`LFLAGS(1)=1, 2`)

- Abaqus/Standard will solve $K^{NM}\,\tilde{u}^M = \tilde{P}^N$ for $\tilde{u}^M$, where $K^{NM}$ is the base state stiffness matrix and the perturbation load vector, $\tilde{P}^N$, is a linear function of the perturbation loads, $\tilde{p}$; that is, $\tilde{P}^N = (\partial F/\partial \tilde{p})\,\tilde{p}$.
- `LFLAGS(3)=1`: You must define AMATRX $= K^{NM}$ and RHS $= \tilde{P}^N$.
- `LFLAGS(3)=100`: You must compute perturbations of the internal variables, $\widetilde{H}^\alpha$, and define RHS $= \tilde{P}^N - K^{NM}\,\tilde{u}^M$ for output purposes.

### Eigenfrequency Extraction Analysis (`LFLAGS(1)=41`)

- $F^N = -M^{NM}\ddot{\tilde{u}} + G^N\left(u^M + \tilde{u}^M, \ldots\right) = -M^{NM}\ddot{\tilde{u}} + \left(\partial G^N/\partial u^M\right)\,\tilde{u}^M$.
- Abaqus/Standard will solve $K^{NM}\,\phi_i^M = \omega_i^2\, M^{NM}\,\phi_i^M$ for $\phi_i^N$ and $\omega_i$, where $K^{NM} = -\partial F^N/\partial u^M$ is the base state stiffness matrix and $M^{NM} = -\partial F^{NM}/\partial \ddot{u}^M$ is the base state mass matrix.
- `LFLAGS(3)=2`: Define AMATRX $= K^{NM}$.

- LFLAGS(3)=4: Define AMATRX $= \boldsymbol{M^{NM}}$.

## Example: Structural User Element with Abaqus Isotropic Linearly Elastic Material

Both a structural and a heat transfer user element have been created to demonstrate the usage of subroutine *UELMAT*. These user-defined elements are applied in a number of analyses. The following excerpt illustrates how the linearly elastic isotropic material available in Abaqus can be accessed from user subroutine *UELMAT*:

```
...
*USER ELEMENT, TYPE=U1, NODES=4, COORDINATES=2, VAR=16,
   INTEGRATION=4, TENSOR=PSTRAIN
   1,2
*ELEMENT, TYPE=U1, ELSET=SOLID
   1, 1,2,3,4
...
*UEL PROPERTY, ELSET=SOLID, MATERIAL=MAT
...
*MATERIAL, NAME=MAT
*ELASTIC
 7.00E+010, 0.33
```

The user element defined above is a 4-node, fully integrated plane strain element, similar to the AbaqusCPE4 element.

The next excerpt shows the listing of the user subroutine. Inside the subroutine, a loop over the integration points is performed. For each integration point the utility routine MATERIAL_LIB_MECH is called, which returns stress and Jacobian at the integration point. These quantities are used to compute the right-hand-side vector and the element Jacobian.

```
c**********************************************************
      subroutine uelmat(rhs,amatrx,svars,energy,ndofel,nrhs,
     1     nsvars,props,nprops,coords,mcrd,nnode,u,du,
     2     v,a,jtype,time,dtime,kstep,kinc,jelem,params,
     3     ndload,jdltyp,adlmag,predef,npredf,lflags,mlvarx,
     4     ddlmag,mdload,pnewdt,jprops,njpro,period,
     5     materiallib)
c
      include 'aba_param.inc'
C
      dimension rhs(mlvarx,*), amatrx(ndofel, ndofel), props(*),
     1  svars(*), energy(*), coords(mcrd, nnode), u(ndofel),
     2  du(mlvarx,*), v(ndofel), a(ndofel), time(2), params(*),
     3  jdltyp(mdload,*), adlmag(mdload,*), ddlmag(mdload,*),
     4  predef(2, npredf, nnode), lflags(*), jprops(*)
      parameter (zero=0.d0, dmone=-1.0d0, one=1.d0, four=4.0d0,
     1     fourth=0.25d0,gaussCoord=0.577350269d0)
      parameter (ndim=2, ndof=2, nshr=1,nnodemax=4,
     1     ntens=4, ninpt=4, nsvint=4)
c
c      ndim  ... number of spatial dimensions
c      ndof  ... number of degrees of freedom per node
c      nshr  ... number of shear stress component
c      ntens ... total number of stress tensor components
c                (=ndi+nshr)
c      ninpt ... number of integration points
c      nsvint... number of state variables per integration pt
c                (strain)
c
      dimension  stiff(ndof*nnodemax,ndof*nnodemax),
     1 force(ndof*nnodemax), shape(nnodemax), dshape(ndim,nnodemax),
```

```
     2 xjac(ndim,ndim),xjaci(ndim,ndim), bmat(nnodemax*ndim),
     3 statevLocal(nsvint),stress(ntens), ddsdde(ntens, ntens),
     4 stran(ntens), dstran(ntens), wght(ninpt)
c
      dimension predef_loc(npredf),dpredef_loc(npredf),
     1     defGrad(3,3),utmp(3),xdu(3),stiff_p(3,3),force_p(3)
      dimension coord24(2,4),coords_ip(3)
      data  coord24 /dmone, dmone,
     2                one, dmone,
     3                one,   one,
     4               dmone,   one/
c
      data wght /one, one, one, one/
c
c************************************************************
c
c     U1 = first-order, plane strain, full integration
c
c     State variables: each integration point has nsvint SDVs
c
c       isvinc=(npt-1)*nsvint    ... integration point counter
c       statev(1+isvinc        ) ... strain
c
c************************************************************
      if (lflags(3).eq.4) then
        do i=1, ndofel
          do j=1, ndofel
            amatrx(i,j) = zero
          end do
          amatrx(i,i) = one
        end do
        goto 999
      end if
c
c     PRELIMINARIES
c
      pnewdtLocal = pnewdt
      if(jtype .ne. 1) then
        write(7,*)'Incorrect element type'
        call xit
      endif
      if(nsvars .lt. ninpt*nsvint) then
        write(7,*)'Increase the number of SDVs to', ninpt*nsvint
        call xit
      endif
      thickness = 0.1d0
c
c     INITIALIZE RHS AND LHS
c
      do k1=1, ndof*nnode
        rhs(k1, 1)= zero
        do k2=1, ndof*nnode
          amatrx(k1, k2)= zero
        end do
      end do
c
c     LOOP OVER INTEGRATION POINTS
c
      do kintk = 1, ninpt
c
c       EVALUATE SHAPE FUNCTIONS AND THEIR DERIVATIVES
```

```
c
c       determine (g,h)
c
        g = coord24(1,kintk)*gaussCoord
        h = coord24(2,kintk)*gaussCoord
c
c       shape functions
        shape(1) = (one - g)*(one - h)/four;
        shape(2) = (one + g)*(one - h)/four;
        shape(3) = (one + g)*(one + h)/four;
        shape(4) = (one - g)*(one + h)/four;
c
c       derivative d(Ni)/d(g)
        dshape(1,1) = -(one - h)/four;
        dshape(1,2) =  (one - h)/four;
        dshape(1,3) =  (one + h)/four;
        dshape(1,4) = -(one + h)/four;
c
c       derivative d(Ni)/d(h)
        dshape(2,1) = -(one - g)/four;
        dshape(2,2) = -(one + g)/four;
        dshape(2,3) =  (one + g)/four;
        dshape(2,4) =  (one - g)/four;
c
c       compute coordinates at the integration point
c
        do k1=1, 3
          coords_ip(k1) = zero
        end do
        do k1=1,nnode
          do k2=1,mcrd
            coords_ip(k2)=coords_ip(k2)+shape(k1)*coords(k2,k1)
          end do
        end do
c
c       INTERPOLATE FIELD VARIABLES
c
        if(npredf.gt.0) then

          do k1=1,npredf
            predef_loc(k1) = zero
            dpredef_loc(k1) = zero
            do k2=1,nnode
              predef_loc(k1) =
     &             predef_loc(k1)+
     &             (predef(1,k1,k2)-predef(2,k1,k2))*shape(k2)
             dpredef_loc(k1) =
     &             dpredef_loc(k1)+predef(2,k1,k2)*shape(k2)
            end do
          end do
        end if
c
c       FORM B-MATRIX
c
        djac = one
c
        do i = 1, ndim
          do j = 1, ndim
            xjac(i,j)  = zero
            xjaci(i,j) = zero
          end do
```

```
        end do
c
        do inod= 1, nnode
          do idim = 1, ndim
            do jdim = 1, ndim
              xjac(jdim,idim) = xjac(jdim,idim) +
     1             dshape(jdim,inod)*coords(idim,inod)
            end do
          end do
        end do
        djac = xjac(1,1)*xjac(2,2) - xjac(1,2)*xjac(2,1)
        if (djac .gt. zero) then
        ! jacobian is positive - o.k.
          xjaci(1,1) =  xjac(2,2)/djac
          xjaci(2,2) =  xjac(1,1)/djac
          xjaci(1,2) = -xjac(1,2)/djac
          xjaci(2,1) = -xjac(2,1)/djac
        else
          ! negative or zero jacobian
          write(7,*)'WARNING: element',jelem,'has neg.
     1         Jacobian'
          pnewdt = fourth
        endif


        if (pnewdt .lt. pnewdtLocal) pnewdtLocal = pnewdt
c
        do i = 1, nnode*ndim
          bmat(i) = zero
        end do

        do inod = 1, nnode
          do ider = 1, ndim
            do idim = 1, ndim
              irow = idim + (inod - 1)*ndim
              bmat(irow) = bmat(irow) +
     1             xjaci(idim,ider)*dshape(ider,inod)
            end do
          end do
        end do

c
c       CALCULATE INCREMENTAL STRAINS
c
        do i = 1, ntens
          dstran(i) = zero
        end do
        !
        ! set deformation gradient  to Identity matrix
        do k1=1,3
          do k2=1,3
            defGrad(k1,k2) = zero
          end do
          defGrad(k1,k1) = one
        end do
c
c       COMPUTE INCREMENTAL STRAINS
c
        do nodi = 1, nnode

          incr_row = (nodi - 1)*ndof
```

```
          do i = 1, ndof
            xdu(i)= du(i + incr_row,1)
            utmp(i) = u(i + incr_row)
          end do

          dNidx = bmat(1 + (nodi-1)*ndim)
          dNidy = bmat(2 + (nodi-1)*ndim)

          dstran(1) = dstran(1) + dNidx*xdu(1)
          dstran(2) = dstran(2) + dNidy*xdu(2)
          dstran(4) = dstran(4) +
     1         dNidy*xdu(1) +
     2         dNidx*xdu(2)

c        deformation gradient

          defGrad(1,1) = defGrad(1,1) + dNidx*utmp(1)
          defGrad(1,2) = defGrad(1,2) + dNidy*utmp(1)
          defGrad(2,1) = defGrad(2,1) + dNidx*utmp(2)
          defGrad(2,2) = defGrad(2,2) + dNidy*utmp(2)
        end do

c
c       CALL CONSTITUTIVE ROUTINE
c

        isvinc= (kintk-1)*nsvint  ! integration point increment
c
c       prepare arrays for entry into material routines
c
        do i = 1, nsvint
          statevLocal(i)=svars(i+isvinc)
        end do
c
c       state variables
c
!DEC$ NOVECTOR
        do k1=1,ntens
           stran(k1) = statevLocal(k1)
           stress(k1) = zero
        end do
c
        do i=1, ntens
!DEC$ NOVECTOR
          do j=1, ntens
            ddsdde(i,j) = zero
          end do
          ddsdde(i,j) = one
        enddo
c
c       compute characteristic element length
c
        celent = sqrt(djac*dble(ninpt))
        dvmat  = djac*thickness
c
        dvdv0 = one
        call material_lib_mech(materiallib,stress,ddsdde,
     1       stran,dstran,kintk,dvdv0,dvmat,defGrad,
     2       predef_loc,dpredef_loc,npredf,celent,coords_ip)
c
```

```
        do k1=1,ntens
           statevLocal(k1) = stran(k1) + dstran(k1)
        end do
c
        isvinc= (kintk-1)*nsvint  ! integration point increment
c
c       update element state variables
c
        do i = 1, nsvint
          svars(i+isvinc)=statevLocal(i)
        end do
c
c       form stiffness matrix and internal force vector
c
        dNjdx = zero
        dNjdy = zero
        do i = 1, ndof*nnode
          force(i) = zero
          do j = 1, ndof*nnode
            stiff(j,i) = zero
          end do
        end do

        dvol= wght(kintk)*djac
        do nodj = 1, nnode

          incr_col = (nodj - 1)*ndof

          dNjdx = bmat(1+(nodj-1)*ndim)
          dNjdy = bmat(2+(nodj-1)*ndim)
          force_p(1) = dNjdx*stress(1) + dNjdy*stress(4)
          force_p(2) = dNjdy*stress(2) + dNjdx*stress(4)
          do jdof = 1, ndof

            jcol = jdof + incr_col

            force(jcol) = force(jcol) +
     &           force_p(jdof)*dvol

          end do

          do nodi = 1, nnode

            incr_row = (nodi -1)*ndof

            dNidx = bmat(1+(nodi-1)*ndim)
            dNidy = bmat(2+(nodi-1)*ndim)
            stiff_p(1,1) = dNidx*ddsdde(1,1)*dNjdx
     &           + dNidy*ddsdde(4,4)*dNjdy
     &           + dNidx*ddsdde(1,4)*dNjdy
     &           + dNidy*ddsdde(4,1)*dNjdx

            stiff_p(1,2) = dNidx*ddsdde(1,2)*dNjdy
     &           + dNidy*ddsdde(4,4)*dNjdx
     &           + dNidx*ddsdde(1,4)*dNjdx
     &           + dNidy*ddsdde(4,2)*dNjdy

            stiff_p(2,1) = dNidy*ddsdde(2,1)*dNjdx
     &           + dNidx*ddsdde(4,4)*dNjdy
     &           + dNidy*ddsdde(2,4)*dNjdy
     &           + dNidx*ddsdde(4,1)*dNjdx
```

```
            stiff_p(2,2) = dNidy*ddsdde(2,2)*dNjdy
     &           + dNidx*ddsdde(4,4)*dNjdx
     &           + dNidy*ddsdde(2,4)*dNjdx
     &           + dNidx*ddsdde(4,2)*dNjdy

            do jdof = 1, ndof
              icol = jdof + incr_col
              do idof = 1, ndof
                irow = idof + incr_row
                stiff(irow,icol) = stiff(irow,icol) +
     &               stiff_p(idof,jdof)*dvol
              end do
            end do
          end do
        end do

c
c       assemble rhs and lhs
c
        do k1=1, ndof*nnode
            rhs(k1, 1) = rhs(k1, 1) - force(k1)
          do k2=1, ndof*nnode
            amatrx(k1, k2) = amatrx(k1, k2) + stiff(k1,k2)
          end do
        end do
      end do       ! end loop on material integration points
      pnewdt = pnewdtLocal
c
 999  continue
c
      return
      end
```

# UEPACTIVATIONFACET: User subroutine to specify the area fraction over which convective or radiative cooling is applied when using element activation.

**Products:** Abaqus/Standard

**References:**

- **ACTIVATE ELEMENTS*
- **ELEMENT PROGRESSIVE ACTIVATION*
- *Progressive Element Activation*
- *Specifying Element-Based Film Conditions on Evolving Faces of an Element in Abaqus/Standard*
- *Specifying Element-Based Radiation Conditions on Evolving Faces of an Element in Abaqus/Standard*

## Overview

User subroutine *UEPACTIVATIONFACET*:

- can be used to specify the facet area of an element to apply film or radiation during element progressive activation;
- can use and update element solution-dependent variables;
- will be called at the beginning of each increment in a step in which element activation is specified; and
- can be used only in heat transfer procedures.

## User Subroutine Interface

```
      SUBROUTINE UEPACTIVATIONFACET(
     * LFLAG, EPANAME, NOEL, NELEMNODES, IELEMNODES,
     * MCRD, COORDNODES, UNODES, KSTEP, KINC,
     * TIME, DT, TEMP, NPREDEF, PREDEF,
     * NSVARS, SVARS, SOL, SOLINC,
     * NELEMFACETS, NFACETNODES, JFACETNODES,
     * VOLFRACT, VOLFRACTADDED, AREAFACETOLD, AREACUTOLD,
     * NEVENTS,
     * DELTAAREAFACET, DELTAAREAFACETCSI, DELTAAREAFACETTIME,
     * AREACUT, AREACUTCSI, AREACUTTIME)
C
      INCLUDE 'ABA_PARAM.INC'
C
      PARAMETER (NEVENTSMAX = 1000)
      CHARACTER*80 EPANAME
      DIMENSION IELEMNODES(NELEMNODES),
     * COORDNODES(MCRD,NELEMNODES), UNODES(MCRD,NELEMNODES),
     * TIME(2), SVARS(NSVARS,2), TEMP(2,NELEMNODES),
     * PREDEF(2,NPREDEF,NELEMNODES), SOL(NELEMNODES),
     * SOLINC(NELEMNODES), JFACETNODES(NFACETNODES,*),
     * AREAFACETOLD(NELEMFACETS),
     * DELTAAREAFACET(NELEMFACETS,NEVENTSMAX),
     * DELTAAREAFACETCSI(MCRD,NELEMFACETS,NEVENTSMAX),
     * DELTAAREAFACETTIME(NELEMFACETS,NEVENTSMAX),
     * AREACUT(NEVENTSMAX), AREACUTCSI(MCRD,NEVENTSMAX),
     * AREACUTTIME(NEVENTSMAX)
```

```
      CHARACTER*80 EPANAME

      user coding to define NEVENTS, DELTAAREAFACET,
      DELTAAREAFACETCSI, DELTAAREAFACETTIME, AREACUT, AREACUTCSI,
      and possibly update SVARS

      RETURN
      END
```

## Variables to Be Defined

`NEVENTS` The number of times facet areas are changed in this increment.

`DELTAAREAFACET(NELEMFACETS,NEVENTS)` An array of area fractions that are added per element facet per event.

`DELTAAREAFACETCSI(MCRD,NELEMFACETS,NEVENTS)` An array of integration points at which the cooling loads are computed per area fraction that is added per element facet per event.

`DELTAAREAFACETTIME(NELEMFACETS,NEVENTS)` An array of the total time over the increment for which each of the element facet area fractions are exposed for cooling.

`AREACUT(NEVENTS)` An array of total areas of the internal free surface that is exposed per event.

`AREACUTCSI(MCRD,NEVENTS)` An array of integration points at which the cooling loads are computed for each of the internal free cut areas per event.

`AREACUTTIME(NEVENTS)` An array of total time over the increment for which each of the internal free surfaces is exposed per event.

## Variables That Can Be Updated

`SVARS(NSVARS,1)` An array for defining new values of element solution-dependent variables at the end of the increment. These are passed in as the values at the beginning of the increment unless they are updated by other user subroutines, in which case the updated values are passed in. The size of the array is `NSVARS` (see below), which can be defined as described in *Allocating Space for Element Solution-Dependent Variables*.

## Variables Passed in for Information

`LFLAG` An array containing the flags that define the current free surface definition type.

| | |
|---|---|
| `LFLAG=2` | Defines free surface as facet. |

`EPANAME` The name of the element progressive activation. It is the name specified on the **ELEMENT PROGRESSIVE ACTIVATION* option.

`NOEL` User-assigned element number.

`NELEMNODES` Number of element nodes.

`IELEMNODES(NELEMNODES)` An array containing user-assigned node numbers of the element.

| | |
|---|---|
| MCRD | Maximum number of coordinates at any node. Typically has a value of 3. |
| COORDNODES(MCRD,NELEMNODES) | An array containing the coordinates of the nodes of the element. |
| UNODES(MCRD,NELEMNODES) | An array containing displacements at the element nodes. |
| KSTEP | Current step number. |
| KINC | Current increment number. |
| TIME(1) | Current value of step time. |
| TIME(2) | Current value of total time. |
| DTIME | Time increment. |
| TEMP(2,NELEMNODES) | An array containing pairs of values of predefined temperature at the element nodes. The first value in a pair, TEMP(1,NELEMNODES), corresponds to the value at the end of the increment; and the second value, TEMP(2,NELEMNODES), corresponds to the increment in temperature. |
| NPREDEF | Number of field variables. |
| PREDEF(2,NPREDEF,NELEMNODES) | An array containing pairs of values of predefined field variables at the element nodes. The first value in a pair, PREDEF(1,NPREDEF,NELEMNODES), corresponds to the value at the end of the increment; and the second value, PREDEF(2,NPREDEF,NELEMNODES), corresponds to the increment in the field variables. |
| NSVARS | Number of element solution-dependent variables. |
| SVARS(NSVARS,2) | An array containing values of element solution-dependent variables at the beginning of the increment. |
| SOL(NELEMNODES) | An array of the solution variables (temperature in a heat transfer analysis) at the beginning of the increment. |
| SOLINC(NELEMNODES) | An array of estimated values of the increment of the solution variables (temperature in a heat transfer analysis). The values are always set equal to zero. |
| NELEMFACETS | Number of element facets. |
| NFACETNODES | Number of nodes per facet of the element. |
| JFACETNODES(NFACETNODES,*) | An array of nodes for each facet based on the facet connectivity. |
| VOLFRACT(*) | Total volume fraction of the material at the previous increment. |
| VOLFRACTADDED(*) | The volume fraction of material added to the element in the current increment |
| AREAFACETOLD(NELEMFACETS) | An array of total area fractions for each facet at the previous increment. |
| AREACUTOLD | The total area of the internal cut facet of the element at the previous increment when partial element activation is used. |

# UEPACTIVATIONSETUP: User subroutine to control interaction with the toolpath-mesh intersection module and to specify the list of elements that can be activated.

**Products:** Abaqus/Standard

**References:**

- **ACTIVATE ELEMENTS*
- **ELEMENT PROGRESSIVE ACTIVATION*
- *Progressive Element Activation*

## Overview

User subroutine *UEPACTIVATIONSETUP*:

- is called for each progressive activation once at the start of the step and again at the start of each increment;
- can be used to identify potentially active elements at the start of an increment;
- can be used only in heat transfer and static analyses;
- can be use to modify the time increment at the start of an increment; and
- can be used to communicate data between external programs and user subroutines.

## User Subroutine Interface

```
            subroutine uepactivationsetup(
     * lop,
     * epaName,
     * epaElsetName,
     * jActivatingElems,
     * nActivatingElems,
     * kstep,
     * kinc,
     * time,
     * dtime)
C
      include 'aba_param.inc'
C
      dimension
     * jActivatingElems(nActivatingElems),
     * time(2),
     * dtime(2)

      character*80
     * epaName,
     * epaElsetName

      user coding

      return
      end
```

## Variables to Be Defined

None.

## Variables That Can Be Updated

`jActivatingElems(nActivatingElems)` An array containing elements that can be activated in this increment. This array contains all the elements of the element set `epaElsetName` that was specified on the **ELEMENT PROGRESSIVE ACTIVATION* option. It can be modified by removing elements that will not be activated in the increment, which may improve performance for models with large number of elements that potentially can be activated. It is used only when `lop=2`.

`nActivatingElems` Number of elements that can be activated in this increment. It must be modified if the array `jActivatingElems` is modified and its size changes. It is used only when `lop=2`.

`dtime(2)` Ratio of your defined time increment to the time increment currently being used [`dtime(1)`, see below]. This variable allows you to provide input to the automatic time incrementation algorithms if automatic time incrementation is chosen.

If `dtime(2)` is redefined to be other than 1.0, Abaqus/Standard must abandon the time increment and attempt it again with a new time increment. The suggested new time increment provided to the automatic time integration algorithms is `dtime(2) × dtime(1)`, where the `dtime(2)` used is the minimum value for all calls to user subroutine *UEPACTIVATIONSETUP* that allows redefinition of `dtime(2)` for this iteration.

If automatic time incrementation is not selected in the analysis procedure, values of `dtime(2)` greater than 1.0 will be ignored and values of `dtime(2)` less than 1.0 will cause the job to terminate.

## Variables Passed in for Information

`lop` `lop`=1 indicates that the user subroutine is called at the start of a step.

`lop`=2 indicates that the user subroutine is called at the start of an increment.

`epaName` The name of the element progressive activation. It is the name specified on the **ELEMENT PROGRESSIVE ACTIVATION* option.

`epaElsetName` The name of the element set specified on the **ELEMENT PROGRESSIVE ACTIVATION* option.

`kstep` Current step number.

`kinc` Current increment number.

`time(1)` Current value of step time.

`time(2)` Current value of total time.

`dtime(1)` Time increment.

# UEPACTIVATIONVOL: User subroutine to specify material volume fraction added during element activation.

**Products:** Abaqus/Standard

**References:**

- **ACTIVATE ELEMENTS*
- **ELEMENT PROGRESSIVE ACTIVATION*
- *Progressive Element Activation*

## Overview

User subroutine *UEPACTIVATIONVOL*:

- can be used to add a volume fraction of material to an element during element activation;
- can be used to prescribe eigenstrains to material added to an element during element activation;
- can be used to update the orientation for solid elements;
- can use and update element solution-dependent variables;
- will be called at the beginning of each increment in a step in which element activation is active; and
- can be used only with the heat transfer and static procedures.

## User Subroutine Interface

```
      subroutine uepactivationvol(lFlags, epaName,
     * noel, nElemNodes, iElemNodes, mcrd, coordNodes, uNodes,
     * kstep, kinc, time, dtime, temp,
     * npredef, predef, nsvars, svars,
     * sol, solinc, volFract, nVolumeAddEvents, volFractAdded,
     * csiAdded, ori, eigenstrains)
C
      include 'aba_param.inc'
C
      dimension
     * lFlags(*), iElemNodes(nElemNodes),
     * coordNodes(mcrd, nElemNodes),
     * uNodes(mcrd, nElemNodes),
     * time(2), svars(nsvars, 2),
     * temp(2, nElemNodes),
     * predef(2, npredef, nElemNodes),
     * sol(nElemNodes), solinc(nElemNodes),
     * volFract(*), volFractAdded(*),
     * csiAdded(3,*), ori(3,3), eigenstrains(*)

      character*80  epaName


      user coding to define volFractAdded
      and possibly update eigenstrains, lFlags(4), ori, svars
```

```
      RETURN
      END
```

## Variables to Be Defined

`volFractAdded(1)` The volume fraction of material added to the element in the current increment.

## Variables That Can Be Updated

`eigenstrains(*)` An array of eigenstrain components. If a local orientation is defined (`lFlags(3)=1`), the eigenstrain components must be specified in the local orientation. If the local orientation is updated, the eigenstrain components must be specified in the new orientation. Direct components are stored first, followed by shear components.

For a solid element, provide the eigenstrain components in the following order:

$\varepsilon^*_{11}$, $\varepsilon^*_{22}$, $\varepsilon^*_{33}$, $\varepsilon^*_{12}$, $\varepsilon^*_{13}$, $\varepsilon^*_{23}$. For a shell element, provide the eigenstrain components in the following order: $\varepsilon^*_{11}$, $\varepsilon^*_{22}$, $\varepsilon^*_{12}$. You need to provide a single set of values for the element. These values will be used for all integration points.

`lFlags(4)` A flag to indicate if the orientation should be updated.

| | |
|---|---|
| `lFlags(4)=0` | The orientation should not be updated. |
| `lFlags(4)=1` | The orientation should be updated. This flag is relevant only for solid elements (`lFlags(5)=1`) when local orientation is defined (`lFlags(3)=1`) and volume fraction of material is added to the element for the first time (`volFract(1)=0` and `volFractAdded(1)` > 0). In all other cases this flag is ignored. |

`ori(3, 3)` If a local orientation is not defined (`lFlags(3)=0`), this array is set to 0.

If a local orientation is specified (`lFlags(3)=1`), this array contains the direction cosines of the material directions in terms of the global basis directions at the first material integration point of the element. For example, `ori(1, 1)`, `ori(2, 1)`, `ori(3, 1)` give the (1, 2, 3) components of the first material direction; `ori(1, 2)`, `ori(2, 2)`, `ori(3, 2)` give the components of the second material direction, etc.

For shell elements, the first two directions are in the plane of the element and the third direction is the normal.

For solid elements, if a local orientation is defined (`lFlags(3)=1`), this orientation can be updated when the material is added to the element for the first time.

`svars(nsvars, 1)` An array for defining new values of element solution-dependent variables at the end of the increment. These are passed in as the values at the beginning of the increment unless they are updated by other user subroutines, in which case the updated values are passed in. The size of the array is `nsvars` (see below), which can be defined as described in *Allocating Space for Element Solution-Dependent Variables*.

## Variables Passed in for Information

`lFlags` An array containing the flags that define the current solution procedure, analysis type, and element type.

| | |
|---|---|
| `lFlags(1)` | Defines the procedure type. See *Results File* for the key used for each procedure. |

| | |
|---|---|
| `lFlags(2)=0` | Small-displacement analysis. |
| `lFlags(2)=1` | Large-displacement analysis (nonlinear geometric effects included in the step; see *General and Perturbation Procedures*). |
| `lFlags(3)=0` | Local orientation is not defined. |
| `lFlags(3)=1` | Local orientation is defined. |
| `lFlags(5)=1` | Solid element. |
| `lFlags(5)=2` | Shell element. |
| `lFlags(5)=3` | All other element types. |

`epaName` The name of the progressive element activation. It is the name specified on the **ELEMENT PROGRESSIVE ACTIVATION* option.

`noel` User-assigned element number.

`nElemNodes` Number of element nodes.

`iElemNodes(nElemNodes)` An array containing user-assigned node numbers of the element.

`mcrd` Defined as the maximum of the user-defined maximum number of coordinates needed at any node point (*Defining the Maximum Number of Coordinates Needed at Any Nodal Point*) and the value of the largest active degree of freedom of the user element that is less than or equal to 3. For example, if you specify that the maximum number of coordinates is 1 and the active degrees of freedom of the user element are 2, 3, and 6, `mcrd` will be 3. If you specify that the maximum number of coordinates is 2 and the active degrees of freedom of the user element are 11 and 12, `mcrd` will be 2.

`coordNodes(mcrd, nElemNodes)` An array containing the coordinates of the nodes of the element.

`uNodes(mcrd, nElemNodes)` An array containing displacements at the element nodes.

`kstep` Current step number.

`kinc` Current increment number.

`time(1)` Current value of step time.

`time(2)` Current value of total time.

`dtime` Time increment.

`temp(2, nElemNodes)` An array containing pairs of values of predefined temperature at the element nodes. The first value in a pair, `temp(1, nElemNodes)`, corresponds to the value at the end of the increment, and the second value, `temp(2, nElemNodes)`, corresponds to the increment in temperature.

`npredef` Number of field variables.

`predef(2, npredef, nElemNodes)` An array containing pairs of values of predefined field variables at the element nodes. The first value in a pair, `predef(1, npredef, nElemNodes)`, corresponds to the value at the end of the increment, and the second value, `predef(2, npredef,nElemNodes)`, corresponds to the increment in the field variables.

`nsvars` Number of element solution-dependent variables.

`svars(nsvars, 2)` An array containing values of element solution-dependent variables at the beginning of the increment.

`sol(nElemNodes)` An array of the solution variables (temperature in a heat transfer analysis) at the beginning of the increment.

| | |
|---|---|
| `solinc(nElemNodes)` | An array of estimated values of the increment of the solution variables (temperature in a heat transfer analysis). The values are always set equal to zero. |
| `volFract(1)` | Total volume fraction of the material at the previous increment. |
| `nVolumeAddEvents` | Not used. |
| `csiAdded(3, *)` | Not used. |

# UEREF: User subroutine to define the Open Circuit Potential (OCP) to be used for elements that utilize porous electrode theory in an electrochemical analysis.

**Products:** Abaqus/Standard

**References:**

## Overview

User subroutine *UEREF*:

- is called for each material integration point for each iteration;
- is called for each microscale particle that is defined at the active electrodes during an electrochemical analysis;
- can be used to define the OCP for overpotentials in Butler-Volmer computation (see *Intercalation at the Solid-Liquid Interface*); and
- can be used only in procedures that support electrochemical analyses (see *Electrochemical Analysis*).

## User Subroutine Interface

```
 subroutine ueref(
     *     ocp, docp_dCSurfNorm, docp_dT,
     *     chargeDischarge,
     *     jmac,jMatyp,matlayo,laccFlag,
     *     cmname, cParticleName,
     *     kstep,lFlags, kinc, time, dtime, noel, npt,
     *     ntens, ndi, nshr, props, nprops,
     *     predef, dpredef, npredef,
     *     statev, statevOld, nstatev,
     *     Temp, dTemp, coords,
     *     solutionvar, dsolutionvar, nSolutionVar,
     *     cSurfNorm, cAvgNorm, ecdBVOld,
     *     i_array,niarray,r_array,nrarray,c_array,ncarray)
C
      include 'aba_param.inc'
c Include below file to access property and parameter table.
      include 'aba_tcs_param.inc'
c Include below file to access universal constants
      include 'aba_phcon_param.inc'
c Include below file for parametrized location of various solution variables.

      include 'aba_solutionvarloc.inc'
C
      dimension i_array(*), r_array(*), jmatyp(*),jmac(*), lFlags(*)
      dimension time(2), props(nprops), predef(npredef), dpredef(npredef),

     *     statev(nstatev), statevold(nstatev), coords(*),
     *     solutionvar(nsolutionvar), dsolutionvar(nsolutionvar)
      character*80   cmname, c_array(NCARRAY), cParticleName

      user coding
```

```
      return
      end
```

## Variables to Be Defined

| | |
|---|---|
| `ocp` | Value of the open circuit potential. |
| `docp_dCSurfNorm` | Variation of the `OCP` with respect to normalized particle surface concentration `CSurfNorm`. |
| `docp_dT` | Variation of the `ocp` with respect to temperature `temp`. |

## Variables Passed in for Information

| | |
|---|---|
| `chargeDischarge` | This variable will have a value that you can use to identify whether the current step is a charging step or a discharging step. The value that is passed in is defined in the property table *ABQ_EChemPET_Electrolyte_ChargeDischarge*. |
| `jMac` | Variable that must be passed into utility routine *GETVRM*. The calling user subroutine provides this variable. |
| `jMatyp` | Variable that must be passed into utility routine *GETVRM*. The calling user subroutine provides this variable. |
| `matlayo` | Variable that must be passed into utility routine *GETVRM*. The calling user subroutine provides this variable. |
| `lAccFlag` | Variable that must be passed into utility routine *GETVRM*. The calling user subroutine provides this variable. |
| `cmName` | User-defined material name, left justified. Some internal material models are given names starting with the "ABQ_" character string. To avoid conflict, you should not use "ABQ_" as the leading string for `CMNAME`. |
| `cParticleName` | User-defined particle name, left justified. |
| `kstep` | Current step number. |
| `lFlags` | An array containing the flags that define the current solution procedure and analysis type. |

| | |
|---|---|
| `lFlags(1)` | Defines the procedure type. See *Results File* for the key used for each procedure. |
| `lFlags(2)=0` | Small-displacement analysis. |
| `lFlags(2)=1` | Large-displacement analysis (nonlinear geometric effects included in the step). See *General and Perturbation Procedures*. |

| | |
|---|---|
| `kinc` | Current increment number. |
| `time(1)` | Current value of step time. |
| `time(2)` | Current value of total time. |
| `dtime` | Time increment. |
| `noel` | User-assigned element number. |
| `npt` | Integration point number. |
| `ntens` | Size of the stress or strain component array. |
| `ndi` | Number of direct stress components at this point. |
| `nshr` | Number of engineering shear stress components at this point. |
| `props` | User-specified array of material constants associated with this user material. |
| `npros` | User-defined number of material constants associated with this user material. |

`predef` Array of interpolated values of predefined field variables at this point at the start of the increment, based on the values read in at the nodes.

`dpredef` Array of increments of predefined field variables.

`npredef` Number of predefined field variables.

`statev` Current value of the solution-dependent variables.

`statevold` Array containing the user-defined solution-dependent state variables at the end of the previous increment.

`nstatev` Number of solution-dependent state variables that are associated with this material type (defined as described in *Allocating Space for Solution-Dependent State Variables*).

`Temp` Current temperature.

`dTemp` Incremental temperature.

`coords` An array containing the current coordinates of this integration point.

`solutionVar` Current value of the solution variables.

| | |
|---|---|
| `solutionVar(i_sol_phiS)` | Current value of the electric potential. |
| `solutionVar(i_sol_phiE)` | Current value of the electric potential in the electrolyte. |
| `solutionVar(i_sol_Ce)` | Current value of the ionic concentration in the electrolyte. |
| `solutionVar(i_sol_porP)` | Current value of the pore pressure. |

`dsolutionVar` Incremental value of the solution variables.

| | |
|---|---|
| `dsolutionVar(i_sol_phiS)` | Incremental value of the electric potential. |
| `dsolutionVar(i_sol_phiE)` | Incremental value of the electric potential in the electrolyte. |
| `dsolutionVar(i_sol_Ce)` | Incremental value of the ionic concentration in the electrolyte. |
| `dsolutionVar(i_sol_porP)` | Incremental value of the pore pressure. |

`nsolutionVar` Number of solution variables.

`cSurfNorm` Current normalized concentration on the surface of the particle.

`cAvgNorm` Current average normalized concentration in the particle.

`ecdBVOld` Value of the Butler-Volmer electrical current density at the end of the previous increment.

`i_array` Not used.

`niarray` Size of array `i_array`.

`r_array` Not used.

`nrarray` Size of array `r_array`.

`c_array` Not used.

`ncarray` Size of array `c_array`.

# UEXPAN: User subroutine to define incremental thermal or field strains.

**Products:** Abaqus/Standard

**References:**

- *Thermal Expansion*
- **EXPANSION*
- *UEXPAN*

## Overview

User subroutine *UEXPAN*:

- can be used to define incremental thermal and/or field expansion strains as functions of temperature, predefined field variables, and state variables;
- is intended for models in which the thermal and/or field expansion strains depend on temperature and/or predefined field variables in complex ways or depend on state variables, which can be used and updated in this routine;
- is called at all integration points of elements for which the material or gasket behavior definition contains user-subroutine-defined thermal expansion and/or field expansion;
- is called once at each integration point for each source of expansion defined in the model;
- is called twice per material point in each iteration during coupled temperature-displacement and coupled thermal-electrical-structural analyses for each source of expansion defined in the model; and
- can be used to define the thermal and/or field expansion strains in the solid grains and/or the pore fluid in a porous medium.

## User Subroutine Interface

```
      SUBROUTINE UEXPAN(EXPAN,DEXPANDT,TEMP,TIME,DTIME,PREDEF,
     1 DPRED,STATEV,CMNAME,NSTATV,IARRAY)
C
      INCLUDE 'ABA_PARAM.INC'
C
      CHARACTER*80 CMNAME
C
      DIMENSION EXPAN(*),DEXPANDT(*),TEMP(2),TIME(2),PREDEF(*),
     1 DPRED(*),STATEV(NSTATV),IARRAY(3)


      user coding to define EXPAN, DEXPANDT and update
      STATEV if necessary.


      RETURN
      END
```

## Variables to Be Defined

`EXPAN(*)` Increments of thermal strain. The number of values to be defined and the order in which they are arranged depend on the type of thermal expansion being defined.

- For isotropic expansion give the isotropic thermal strain increment as the first and only component of the matrix.
- For orthotropic expansion give $\Delta\epsilon^{th}_{11}$, $\Delta\epsilon^{th}_{22}$, and $\Delta\epsilon^{th}_{33}$ as the first, second, and third components of the matrix, respectively. For plane stress only two components of the matrix are needed; give $\Delta\epsilon^{th}_{11}$ and $\Delta\epsilon^{th}_{22}$ as the first and second components.
- For anisotropic expansion give $\Delta\epsilon^{th}_{11}$, $\Delta\epsilon^{th}_{22}$, $\Delta\epsilon^{th}_{33}$, $\Delta\epsilon^{th}_{12}$, $\Delta\epsilon^{th}_{13}$, and $\Delta\epsilon^{th}_{23}$. Direct components are stored first, followed by shear components in the order presented here. For plane stress only three components of the matrix are needed; give $\Delta\epsilon^{th}_{11}$, $\Delta\epsilon^{th}_{22}$, and $\Delta\epsilon^{th}_{12}$ as the first, second, and third components, respectively.

`DEXPANDT(*)` Variation of thermal strains with respect to temperature, $\partial\epsilon^{th}/\partial\theta$. The number of values and the order in which they are arranged depend on the type of thermal expansion being defined. For field expansion strains, this parameter is scalar and should be defined as zero.

- For isotropic expansion give the variation of the isotropic thermal strain with respect to temperature as the first and only component of the matrix.
- For orthotropic expansion give $\partial\epsilon^{th}_{11}/\partial\theta$, $\partial\epsilon^{th}_{22}/\partial\theta$, and $\partial\epsilon^{th}_{33}/\partial\theta$ as the first, second, and third components of the matrix, respectively. For plane stress only two components of the matrix are needed; give $\partial\epsilon^{th}_{11}/\partial\theta$ and $\partial\epsilon^{th}_{22}/\partial\theta$ as the first and second components.
- For anisotropic expansion give $\partial\epsilon^{th}_{11}/\partial\theta$, $\partial\epsilon^{th}_{22}/\partial\theta$, $\partial\epsilon^{th}_{33}/\partial\theta$, $\partial\epsilon^{th}_{12}/\partial\theta$, $\partial\epsilon^{th}_{13}/\partial\theta$, and $\partial\epsilon^{th}_{23}/\partial\theta$. Direct components are stored first, followed by shear components in the order presented here. For plane stress only three components of the matrix are needed; give $\partial\epsilon^{th}_{11}/\partial\theta$, $\partial\epsilon^{th}_{22}/\partial\theta$, and $\partial\epsilon^{th}_{12}/\partial\theta$, as the first, second, and third components, respectively.

## Variables That Can Be Updated

`STATEV(NSTATV)` Array containing the user-defined solution-dependent state variables at this point. Except for coupled temperature-displacement and coupled thermal-electrical-structural analyses, these are supplied as values at the start of the increment and can be updated to their values at the end of the increment. For coupled temperature-displacement and coupled thermal-electrical-structural analyses, *UEXPAN* is called twice per material point per iteration. In the first call for a given material point and iteration, the values supplied are those at the start of the increment and can be updated. In the second call for the same material point and iteration, the values supplied are those returned from the first call, and they can be updated again to their values at the end of the increment.

User subroutine *UEXPAN* allows for the incremental thermal strains to be only weakly dependent on the state variables. The Jacobian terms arising from the derivatives of the thermal strains with respect to the state variables are not taken into account.

## Variables Passed in for Information

| | |
|---|---|
| `TEMP(1)` | Current temperature (at the end of the increment). |
| `TEMP(2)` | Temperature increment. |
| `TIME(1)` | Step time at the end of the increment. |

`TIME(2)` Total time at the end of the increment.

`DTIME` Time increment.

`PREDEF(*)` Array containing the values of all the user-specified predefined field variables at this point (initial values at the beginning of the analysis and current values during the analysis).

`DPRED(*)` Array of increments of predefined field variables.

`CMNAME` User-specified material name or gasket behavior name, left justified.

`NSTATV` Number of solution-dependent state variables associated with this material or gasket behavior type (specified when space is allocated for the array; see *Allocating Space for Solution-Dependent State Variables*).

`IARRAY`

| | |
|---|---|
| `IARRAY(1)` | User-defined element number. |
| `IARRAY(2)` | Pore fluid flag that is set to 1 if the pore fluid expansion strain must be defined for this call to the user subroutine. It is set to 0 otherwise. |
| IARRAY(3) | Field variable number for which the field expansion strain must be defined for this call to the user subroutine. It is set to 0 if the thermal strain must be defined for this call to the user subroutine. |

# UEXTERNALDB: User subroutine to manage user-defined external databases and calculate model-independent history information.

**Products:** Abaqus/Standard

**References:**

- *About User Subroutines and Utilities*

## Overview

User subroutine *UEXTERNALDB*:

- is called once each at the beginning of the analysis, at the beginning of each increment, at the end of each increment, and at the end of the analysis (in addition, the user subroutine is also called once at the beginning of a restart analysis);
- can be used to communicate between other software and user subroutines within Abaqus/Standard;
- can be used to open external files needed for other user subroutines at the beginning of the analysis and to close those files at the end of the analysis;
- can be used to calculate or read history information at the beginning of each increment. This information can be written to user-defined COMMON block variables or external files for use during the analysis by other user subroutines; and
- can be used to write the current values of the user-calculated history information to external files.

## User Subroutine Interface

```
      SUBROUTINE UEXTERNALDB(LOP,LRESTART,TIME,DTIME,KSTEP,KINC)
C
      INCLUDE 'ABA_PARAM.INC'
C
      DIMENSION TIME(2)
C
      user coding to set up the Fortran environment, open files, close files,
      calculate user-defined model-independent history information,
      write history information to external files,
      recover history information during restart analyses, etc.
      do not include calls to utility routine XIT


      RETURN
      END
```

## Variables to Be Defined

None.

## Variables Passed in for Information

LOP LOP=0 indicates that the user subroutine is being called at the start of the analysis.

LOP=1 indicates that the user subroutine is being called at the start of the current analysis increment. The user subroutine can be called multiple times at the beginning of an analysis increment if the increment fails to converge and a smaller time increment is required.

LOP=2 indicates that the user subroutine is being called at the end of the current analysis increment. When LOP=2, all information that you need to restart the analysis should be written to external files.

LOP=3 indicates that the user subroutine is being called at the end of the analysis.

LOP=4 indicates that the user subroutine is being called at the beginning of a restart analysis. When LOP=4, all necessary external files should be opened and properly positioned and all information required for the restart should be read from the external files.

LOP=5 indicates that the user subroutine is being called at the start of a step. The KSTEP argument contains the current step number.

LOP=6 indicates that the user subroutine is being called at the end of a step. The KSTEP argument contains the current step number.

LRESTART — LRESTART=0 indicates that an analysis restart file is not being written for this increment.

LRESTART=1 indicates that an analysis restart file is being written for this increment.

LRESTART=2 indicates that an analysis restart file is being written for this increment and that only one increment is being retained per step so that the current increment overwrites the previous increment in the restart file (see *Restarting an Analysis*).

TIME(1) — Value of current step time.

TIME(2) — Value of current total time.

DTIME — Time increment.

KSTEP — Current step number. For LOP=0 or LOP=4, KSTEP is 0.

KINC — Current increment number. For LOP=0 or LOP=4, KINC is 0.

# UFIELD: User subroutine to specify predefined field variables.

**Products:** Abaqus/Standard

**References:**

- *USDFLD*
- *Beam Elements*
- *Shell Elements*
- *Predefined Fields*
- *FIELD
- *UTEMP, VUTEMP, UFIELD, UMASFL, and UPRESS*

## Overview

User subroutine *UFIELD*:

- allows you to prescribe predefined field variables at the nodes of a model—the predefined field variables at a node can be updated individually, or a number of field variables at the node can be updated simultaneously;
- is called whenever a user-subroutine-defined field appears;
- ignores any field variable values specified directly;
- can be used to modify field variable values read from a results file; and
- can be used in conjunction with user subroutine *USDFLD* such that the field variables that are passed in from *UFIELD* and interpolated to the material points can be modified (such changes are local to material point values, and nodal field variable values remain unaffected).

## Updating Field Variables

Two different methods are provided for updating field variables.

### Individual Variable Updates

By default, only one field variable at a time can be updated in user subroutine *UFIELD*. In this case the user subroutine will be called whenever a current value of a field variable is needed for a node that is listed in the specified field variable definition. This method can be used only for cases in which the field variables are independent of each other.

### Simultaneous Variable Updates

For cases in which the field variables depend on each other, multiple (possibly all) field variables at a point can be updated simultaneously in user subroutine *UFIELD*. In this case you must specify the number of field variables to be updated simultaneously at a point, and the user subroutine will be called each time the current field variable values are needed.

## User Subroutine Interface

```
      SUBROUTINE UFIELD(FIELD,KFIELD,NSECPT,KSTEP,KINC,TIME,NODE,
     1 COORDS,TEMP,DTEMP,NFIELD)
```

```
C
      INCLUDE 'ABA_PARAM.INC'
C
      DIMENSION FIELD(NSECPT,NFIELD), TIME(2), COORDS(3),
     1 TEMP(NSECPT), DTEMP(NSECPT)
C


      user coding to define FIELD


      RETURN
      END
```

## Variables to Be Defined

`FIELD(NSECPT,NFIELD)` Array of predefined field variable values at node number `NODE`. When updating only one field variable at a time, only the value of the specified field variable (see `KFIELD` below) must be returned. In this case `NFIELD` is passed into user subroutine *UFIELD* with a value of 1, and `FIELD` is thus dimensioned as `FIELD(NSECPT,1)`. When updating all field variables simultaneously, the values of the specified number of field variables at the point must be returned. In this case `FIELD` is dimensioned as `FIELD(NSECPT,NFIELD)`, where `NFIELD` is the number of field variables specified and `KFIELD` has no meaning.

If `NODE` is part of any element other than a beam or shell, only one value of each field variable must be returned (`NSECPT`=1). Otherwise, the number of values to be returned depends on the mode of temperature and field variable input selected for the beam or shell section. The following cases are possible:

1. Temperatures and field variables for a beam section are given as values at the points shown in the beam section descriptions. The number of values required, `NSECPT`, is determined by the particular section type specified, as described in *Beam Cross-Section Library*.
2. Temperatures and field variables are given as values at *n* equally spaced points through each layer of a shell section. The number of values required, `NSECPT`, is equal to *n*.
3. Temperatures and field variables for a beam section are given as values at the origin of the cross-section together with gradients with respect to the 2-direction and, for three-dimensional beams, the 1-direction of the section; or temperatures and field variables for a shell section are given as values at the reference surface together with gradients through the thickness. The number of values required, `NSECPT`, is 3 for three-dimensional beams, 2 for two-dimensional beams, and 2 for shells. Give the midsurface value first, followed by the first and (if necessary) second gradients, as described in *Beam Elements* and *Shell Elements*.

Since field variables can also be defined directly, it is important to understand the hierarchy used in situations of conflicting information (see *Predefined Fields*).

When the array `FIELD` is passed into user subroutine *UFIELD*, it will contain either the field variable values from the previous increment or those values obtained from the results file if this method was used. You are then free to modify these values within this subroutine.

## Variables Passed in for Information

| | |
|---|---|
| KFIELD | User-specified field variable number. This variable is meaningful only when updating individual field variables at a time. |
| NFIELD | User-specified number of field variables to be updated. This variable is meaningful only when updating multiple field variables simultaneously. |
| NSECPT | Maximum number of section values required for any node in the model. The NSECPT can be 2 when only one field variable is specified at some non-beam or non-shell nodes in the model with contact. |
| KSTEP | Step number. |
| KINC | Increment number. |
| TIME(1) | Current value of step time. |
| TIME(2) | Current total time. |
| NODE | Node number. |
| COORDS | An array containing the coordinates of this node. These are the current coordinates if geometric nonlinearity is accounted for during the step (see *Defining an Analysis*); otherwise, the array contains the original coordinates of the node. |
| TEMP(NSECPT) | Current temperature at the node. If user subroutines *UTEMP* and *UFIELD* are both used, user subroutine *UTEMP* is processed before user subroutine *UFIELD*. |
| DTEMP(NSECPT) | Temperature increment at the node. |

# UFLUID: User subroutine to define fluid density and fluid compliance for hydrostatic fluid elements.

**Products:** Abaqus/Standard

**References:**

- *Fluid Cavity Definition*
- **FLUID BEHAVIOR*
- *UFLUID*

## Overview

User subroutine *UFLUID*:

- is called for each cavity for which a user-defined fluid constitutive model is being specified;
- is called for every fluid element (*About Surface-Based Fluid Cavities*) and for every fluid exchange definition (*Fluid Exchange Definition*) connected to a cavity reference node;
- requires that the fluid density, $\rho(p,\theta)$, and the fluid pressure compliance, $C_p$, be defined;
- requires that the fluid temperature compliance, $C_\theta$, be defined if the routine is to be used in a linear perturbation step and the fluid is subjected to a temperature excursion; and
- ignores any data specified for the fluid constitutive model outside the user subroutine.

## Density and Fluid Mass

At the start of the analysis (prior to the first iteration) the density calculated in user subroutine *UFLUID* (for the initial pressure, $p_I$, and temperature, $\theta_I$) is used to calculate the fluid mass from the initial cavity volume. During the analysis the expected cavity volume is calculated from the fluid mass and the density.

## User Subroutine Interface

```
      SUBROUTINE UFLUID(RHO,CP,CT,PNEWDT,ENER,PRESS,DPRESS,PRESSI,
     1 TEMP,DTEMP,TEMPI,TIME,DTIME,KSTEP,KINC,NONUM,FLNAME,LFLAG)
C
      INCLUDE 'ABA_PARAM.INC'
C
      CHARACTER*80 FLNAME
      DIMENSION TIME(2)

      user coding to define RHO, CP, and CT

      RETURN
      END
```

## Variables to Be Defined

RHO — Fluid density, $\rho$, at the end of the increment.

CP Fluid pressure compliance, $C_p$, at the end of the increment. For a linear perturbation step this is the base state compliance. Fluid pressure compliance is defined as

$$C_p = \frac{d\rho^{-1}}{dp} = -\rho^{-2}\frac{d\rho}{dp},$$

where $p$ is the fluid cavity pressure.

CT Fluid temperature compliance, $C_\theta$. This variable is needed only if a fluid temperature excursion occurs in a linear perturbation step and is the base state compliance. Fluid temperature compliance is defined as

$$C_\theta = \frac{d\rho^{-1}}{d\theta} = -\rho^{-2}\frac{d\rho}{d\theta},$$

where $\theta$ is the fluid cavity temperature.

## Variables That Can Be Updated

PNEWDT Ratio of suggested new time increment to the time increment being used (DTIME, see below). This variable allows you to provide input to the automatic time incrementation algorithms in Abaqus/Standard (if automatic time incrementation is chosen).

PNEWDT is set to a large value before each call to *UFLUID*.

If PNEWDT is redefined to be less than 1.0, Abaqus/Standard must abandon the time increment and attempt it again with a smaller time increment. The suggested new time increment provided to the automatic time integration algorithms is PNEWDT × DTIME, where the PNEWDT used is the minimum value for all calls to user subroutines that allow redefinition of PNEWDT for this iteration.

If PNEWDT is given a value that is greater than 1.0 for all calls to user subroutines for this iteration and the increment converges in this iteration, Abaqus/Standard may increase the time increment. The suggested new time increment provided to the automatic time integration algorithms is PNEWDT × DTIME, where the PNEWDT used is the minimum value for all calls to user subroutines for this iteration.

If automatic time incrementation is not selected in the analysis procedure, values of PNEWDT that are greater than 1.0 will be ignored and values of PNEWDT that are less than 1.0 will cause the job to terminate.

ENER Energy per unit mass stored in the fluid. This variable is used for energy output only and has no effect on the solution.

## Variables Passed in for Information

| | |
|---|---|
| PRESS | Fluid cavity pressure at the end of the increment. For a linear perturbation step this is the base state pressure. |
| DPRESS | Fluid cavity pressure increment. For a linear perturbation step this value is zero. |
| PRESSI | Fluid cavity pressure at the beginning of the analysis. |
| TEMP | Fluid cavity temperature at the end of the increment. For a linear perturbation step this is the base state temperature. |
| DTEMP | Fluid cavity temperature increment. For a linear perturbation step this value is zero. |
| TEMPI | Fluid cavity temperature at the beginning of the analysis. |
| TIME(1) | Current value of step time at the start of the increment. |
| TIME(2) | Current value of total time at the start of the increment. |

| | |
|---|---|
| DTIME | Time increment. |
| KSTEP | Step number. |
| KINC | Increment number. |
| NONUM | Cavity reference node number. |
| FLNAME | User-specified fluid property name, left justified. |
| LFLAG | Linear perturbation flag for the step. If this is a linear perturbation step, LFLAG=1. For a general analysis step LFLAG=0. |

# UFLUIDCONNECTORLOSS: User subroutine to define the loss coefficient for fluid flow in fluid pipe connector elements.

**Products:** Abaqus/Standard

**References:**

- **FLUID PIPE CONNECTOR LOSS*

## Overview

User subroutine *UFLUIDCONNECTORLOSS*:

- can be used to define the loss coefficient in a fluid pipe connector element;
- corresponds to the Darcy-Weisbach equation for pressure loss; and
- can be used with the fluid pipe connector elements.

## User Subroutine Interface

```
      subroutine ufluidconnectorloss (
C Write only -
     *   ak1, ak2,
C Read only -
     *   coords, flow, rho, visc,
     *   dia, area,
     *   ndim, jelno, kStep, kInc,
     *   time,
     *   nIarray,
     *   i_array,
     *   nRarray,
     *   r_array,
     *   ncarray,
     *   c_array)
C
      include 'aba_param.inc'
C
      dimension time(2),
     *   coords(2*ndim),
     *   i_array(nIarray),
     *   r_array(nRarray)
C
      character*80 c_array(ncarray)
C
      user coding to define connector friction values
      return
      end
```

## Variables to Be Defined

`ak1` This connector loss coefficient must be updated and is used when the flow is from node 1 to node 2.

| | |
|---|---|
| `ak2` | This connector loss coefficient must be updated and is used when the flow is from node 2 to node 1. |

## Variables Passed in for Information

| | |
|---|---|
| `coords(2*ndim)` | Array containing original coordinates of the element. `coords(1:ndim)` is the coordinate of the first node, and `coords(ndim+1:2*ndim)` is the coordinate of the second node. |
| `flow` | Current flow rate through the element. |
| `rho` | Current density of fluid flowing through the pipe. |
| `visc` | Current viscosity of fluid flowing through the pipe. |
| `dia` | User-specified hydraulic diameter. |
| `area` | User-specified hydraulic area. |
| `ndim` | Dimension of the element. |
| `jelno` | User element number for which friction coefficient is required. |
| `kStep` | Step number. |
| `kInc` | Increment number. |
| `time(1)` | Current step time. |
| `time(2)` | Total time. |
| `nIarray` | Size of array `i_array`. |
| `i_array` | Not used. |
| `nRarray` | Size of array `r_array`. |
| `r_array` | Not used. |
| `ncarray` | Size of array `c_array`. |
| `c_array` | Not used. |

# UFLUIDCONNECTORVALVE: User subroutine to define the valve opening to control flow in fluid pipe connector elements.

**Products:** Abaqus/Standard

**References:**

- **FLUID PIPE CONNECTOR LOSS*

## Overview

User subroutine *UFLUIDCONNECTORVALVE*:

- can be used to control the valve opening to turn off or turn on fluid flow; and
- can be used with fluid pipe connector elements.

## User Subroutine Interface

```
      subroutine ufluidconnectorvalve (
C Write only -
     *   valveOpening,
C Read only -
     *   coords, flow, rho, visc,
     *   dia, area,
     *   ndim, jelno, kStep, kInc,
     *   time,
     *   nIarray,
     *   i_array,
     *   nRarray,
     *   r_array,
     *   ncarray,
     *   c_array)
C
      include 'aba_param.inc'
C
      dimension time(2),
     *   coords(2*ndim),
     *   i_array(nIarray),
     *   r_array(nRarray)
C
      character*80 c_array(ncarray)
C
      user coding to define control valve opening
      return
      end
```

## Variables to Be Defined

`valveOpening` The value of this variable must be set between 0.0 (closed/shut-off) and 1.0 (fully open) to determine whether the valve is fully or partially open or closed.

## Variables Passed in for Information

| | |
|---|---|
| `coords(2*ndim)` | Array containing the original coordinates of the element. `acoords(1:ndim)` is the coordinate of the first node, and `acoords(ndim+1:2*ndim)` is the coordinate of the second node. |
| `flow` | Current flow rate through the element. |
| `rho` | Current density of fluid flowing through the pipe. |
| `visc` | Current viscosity of fluid flowing through the pipe. |
| `dia` | User-specified hydraulic diameter. |
| `area` | User-specified hydraulic area. |
| `ndim` | Dimension of the element. |
| `jelno` | User element number for which a friction coefficient is required. |
| `kStep` | Step number. |
| `kInc` | Increment number. |
| `time(1)` | Current step time. |
| `time(2)` | Total time. |
| `nIarray` | Size of array `i_array`. |
| `i_array` | Not used. |
| `nRarray` | Size of array `r_array`. |
| `r_array` | Not used. |
| `ncarray` | Size of array `c_array`. |
| `c_array` | Not used. |

# UFLUIDLEAKOFF: User subroutine to define the fluid leak-off coefficients for pore pressure cohesive elements.

**Products:** Abaqus/Standard

**References:**

- *Defining the Constitutive Response of Fluid within the Cohesive Element Gap*
- **FLUID LEAKOFF*
- *Propagation of hydraulically driven fracture*

## Overview

User subroutine *UFLUIDLEAKOFF*:

- can be used to define the fluid leak-off coefficients for pore pressure cohesive elements and for slurry cohesive elements;
- is called at all material calculation points of elements for which the material definition contains user-defined leak-off coefficients; and
- can include material behavior dependent on field variables or state variables.

## Leak-Off Models

Abaqus provides two methods for coding user subroutine *UFLUIDLEAKOFF*, corresponding to the two models you can use to describe leak-off behavior:

- For the simple leak-off model, you define the coefficients that govern the pressure-flow relationship.
- For the physics-based leak-off model, you define the leak-off flow rate directly.

Only the simple leak-off model is available in cases without degree of freedom of slurry concentration. You can use both leak-off models for slurry transport analyses (see *Defining Slurry Transport and Placement within the Cohesive Element Gap*).

## Simple Leak-Off Model

You can define the fluid leak-off coefficient as a function of the gap opening, fracture fluid pressure, pore pressure on the top and bottom surfaces, temperature, and field variables.

### User Subroutine Interface

```
      SUBROUTINE UFLUIDLEAKOFF( PERM, PGRAD, DN, P_INT, P_BOT, P_TOP,
     1 ANM, TANG, TIME, DTIME, TEMP, DTEMP, PREDEF, DPRED, C_BOT, C_TOP,
     2 DC_BOT, DC_TOP, STATEV, NSTATV, NOEL, NPT, KSTEP, KINC )
C
      INCLUDE 'ABA_PARAM.INC'
C
      CHARACTER*80 CMNAME
      DIMENSION PERM(2), PGRAD(2), ANM(3), TANG(3,2), TIME(2), PREDEF(1),
     1 DPRED(1), DC_BOT(3), DC_TOP(3), STATEV(NSTATV)
```

```
      user coding to define C_BOT, C_TOP, DC_BOT, and DC_TOP

      RETURN
      END
```

## Variables to Be Defined

| | |
|---|---|
| C_BOT | $C_{bot}$, fluid leak-off coefficient on the bottom side of a pore pressure cohesive element. |
| C_TOP | $C_{top}$, fluid leak-off coefficient on the top side of a pore pressure cohesive element. |
| DC_BOT(1) | $\partial C_{bot} / \partial d$, where $d$=DN. |
| DC_BOT(2) | $\partial C_{bot} / \partial p_{int}$, where $p_{int}$=P_INT. |
| DC_BOT(3) | $\partial C_{bot} / \partial p_{bot}$, where $p_{bot}$=P_BOT. |
| DC_TOP(1) | $\partial C_{top} / \partial d$, where $d$=DN. |
| DC_TOP(2) | $\partial C_{top} / \partial p_{int}$, where $p_{int}$=P_INT. |
| DC_TOP(3) | $\partial C_{top} / \partial p_{top}$, where $p_{top}$=P_TOP. |
| STATEV(NSTATV) | An array containing the values of the solution-dependent state variables. You define the meaning of these variables. These are passed in as the values at the beginning of the increment and must be returned as the values at the end of the increment. The size of the array is defined as described in . |

## Variables Passed in for Information

| | |
|---|---|
| PERM(1) | Fluid permeability. |
| PERM(2) | Derivative of fluid permeability with regard to the opening. |
| PGRAD(1) | First component of internal pressure gradient. |
| PGRAD(2) | Second component of internal pressure gradient. |
| DN | Relative opening of the element. |
| P_INT | Internal pressure. |
| P_BOT | Bottom pressure. |
| P_TOP | Top pressure. |
| ANM | Normal vector directed from the bottom face toward the top face. |
| TANG | Tangent direction vectors. |
| TIME(1) | Value of step time at the beginning of the current increment. |
| TIME(2) | Value of total time at the beginning of the current increment. |
| DTIME | Time increment. |
| TEMP | Temperature at the start of the increment. |
| DTEMP | Increment of temperature. |
| PREDEF | Array of interpolated values of predefined field variables at this point at the start of the increment, based on the values read in at the nodes. |
| DPRED | Array of increments of predefined field variables. |

| | |
|---|---|
| NSTATV | Number of solution-dependent state variables that are associated with this material type (defined as described in *Allocating Space for Solution-Dependent State Variables*). |
| NOEL | Element number. |
| NPT | Integration point number. |
| KSTEP | Step number. |
| KINC | Increment number. |

## Physics-Based Leak-Off Model

You can define the leak-off flux rate as a function of the gap opening, fracture fluid pressure, pore pressure on the top and bottom surfaces, slurry concentration, pressure gradients, temperature, and field variables.

### User Subroutine Interface

```
      SUBROUTINE UFLUIDLEAKOFF( PERM, PGRAD, DN, P_INT, P_BOT, P_TOP,
     1 ANM, TANG, TIME, DTIME, TEMP, DTEMP, PREDEF, DPRED, Q_BOT, Q_TOP,
     2 DQ_BOT, DQ_TOP, STATEV, NSTATV, NOEL, NPT, KSTEP, KINC )
C
      INCLUDE 'ABA_PARAM.INC'
C
      CHARACTER*80 CMNAME
      DIMENSION PERM(2), PGRAD(2), DN(2), ANM(3), TANG(3,2), TIME(2),
     1 PREDEF(1), DPRED(1), DQ_BOT(6), DQ_TOP(6), STATEV(NSTATV)


      user coding to define C_BOT, C_TOP, DC_BOT, and DC_TOP

      RETURN
      END
```

### Variables to Be Defined

| | |
|---|---|
| Q_BOT | $q_{bot}$, leak-off flow rate into the bottom crack surface. |
| Q_TOP | $q_{top}$, leak-off flow rate into the top crack surface. |
| DQ_BOT(1) | $\partial q_{bot}/\partial d$, where $d$=DN(1). |
| DQ_BOT(2) | $\partial q_{bot}/\partial p_{int}$, where $p_{int}$=P_INT. |
| DQ_BOT(3) | $\partial q_{bot}/\partial p_{bot}$, where $p_{bot}$=P_BOT. |
| DQ_BOT(4) | $\partial q_{bot}/\partial C_v$, where $C_v$=DN(2). |
| DQ_BOT(5) | $\partial q_{bot}/(\partial p_{int}/\partial x)$, where $\partial p_{int}/\partial x$=PGRAD(1). |
| DQ_BOT(6) | $\partial q_{bot}/(\partial p_{int}/\partial y)$, where $\partial p_{int}/\partial y$=PGRAD(2). |
| DQ_TOP(1) | $\partial q_{top}/\partial d$, where $d$=DN. |
| DQ_TOP(2) | $\partial q_{top}/\partial p_{int}$, where $p_{int}$=P_INT. |
| DQ_TOP(3) | $\partial q_{top}/\partial p_{top}$, where $p_{top}$=P_TOP. |
| DQ_TOP(4) | $\partial q_{top}/\partial C_v$, where $C_v$=DN(2). |

DQ_BOT(5)
: $\partial q_{top} / (\partial p_{int} / \partial x)$, where $\partial p_{int} / \partial x$=PGRAD(1).

DQ_TOP(6)
: $\partial q_{top} / (\partial p_{int} / \partial y)$, where $\partial p_{int} / \partial y$=PGRAD(2).

STATEV(NSTATV)
: An array containing the values of the solution-dependent state variables. You define the meaning of these variables, which are passed in as the values at the beginning of the increment and must be returned as the values at the end of the increment. The size of the array is defined as described in *Allocating Space for Solution-Dependent State Variables*.

## Variables Passed in for Information

PERM(1)
: Fluid permeability.

PERM(2)
: Derivative of fluid permeability with regard to the opening.

PGRAD(1)
: First component of internal pressure gradient.

PGRAD(2)
: Second component of internal pressure gradient.

DN(1)
: Relative opening of the element.

DN(2)
: Slurry concentration.

P_INT
: Internal pressure.

P_BOT
: Bottom pressure.

P_TOP
: Top pressure.

ANM
: Normal vector directed from the bottom face toward the top face.

TANG
: Tangent direction vectors.

TIME(1)
: Value of step time at the beginning of the current increment.

TIME(2)
: Value of total time at the beginning of the current increment.

DTIME
: Time increment.

TEMP
: Temperature at the start of the increment.

DTEMP
: Increment of temperature.

PREDEF
: Array of interpolated values of predefined field variables at this point at the start of the increment, based on the values read in at the nodes.

DPRED
: Array of increments of predefined field variables.

NSTATV
: Number of solution-dependent state variables that are associated with this material type (defined as described in *Allocating Space for Solution-Dependent State Variables*).

NOEL
: Element number.

NPT
: Integration point number.

KSTEP
: Step number.

KINC
: Increment number.

# UFLUIDPIPEFRICTION: User subroutine to define the frictional coefficient for fluid flow in fluid pipe elements.

**Products:** Abaqus/Standard

**References:**

- **FLUID PIPE FLOW LOSS*

## Overview

User subroutine *UFLUIDPIPEFRICTION*:

- can be used to define the frictional coefficient for fluid flow to determine the pipe loss;
- corresponds to the Darcy-Weisbach equation for pressure loss; and
- can be used with the fluid pipe elements.

## User Subroutine Interface

```
      subroutine ufluidpipefriction (
C Write only -
     *   friction,
C Read only -
     *   flow, rho, visc, rough,
     *   dia, area,
     *   ndim, jelno, kstep, kinc,
     *   time, coords,
     *   niarray,
     *   i_array,
     *   nrarray,
     *   r_array,
     *   ncarray,
     *   c_array)
      include 'aba_param.inc'
C
      dimension time(2),
     *   coords(2*ndim),
     *   i_array(niarray),
     *   r_array(nrarray)
C
      character*80 c_array(ncarray)
C
      user coding to define friction
      return
      end
```

## Variables to Be Defined

| | |
|---|---|
| `friction` | This value must be updated to the current value of the friction coefficient. |

## Variables Passed in for Information

| | |
|---|---|
| `flow` | Current flow rate through the element. |
| `rho` | Current density of fluid flowing through the pipe. |
| `visc` | Current viscosity of fluid flowing through the pipe. |
| `rough` | User-specified pipe roughness. |
| `dia` | User-specified hydraulic diameter. |
| `area` | User-specified hydraulic area. |
| `ndim` | Dimension of the element. |
| `jelno` | User element number for which friction coefficient is required. |
| `kstep` | Step number. |
| `kinc` | Increment number. |
| `time(1)` | Current step time. |
| `time(2)` | Total time. |
| `coords(2*ndim)` | Array containing original coordinates of the element. `coords(1:ndim)` is the coordinate of the first node, and `coords(ndim+1:2*ndim)` is the coordinate of the second node. |
| `niarray` | Size of array `i_array`. |
| `i_array` | Not used. |
| `nrarray` | Size of array `r_array`. |
| `r_array` | Not used. |
| `ncarray` | Size of array `c_array`. |
| `c_array` | Not used. |

# UGENS: User subroutine to define the mechanical behavior of a shell section.

**Products:** Abaqus/Standard

**References:**

- *Using a General Shell Section to Define the Section Behavior*
- **SHELL GENERAL SECTION*

## Overview

User subroutine *UGENS*:

- is used to define the (nonlinear) mechanical behavior of a shell section directly in terms of generalized section quantities;
- requires you to define the section behavior of the shell directly in terms of membrane strains and curvature changes;
- is called at all integration points in all shell elements with a general, arbitrary, elastic shell section and a user-subroutine-defined shell section stiffness; and
- can be used with all static or dynamic procedures other than the quasi-static procedure, since that procedure uses automatic time stepping based on the techniques used by Abaqus/Standard to integrate standard creep laws.

## Storage of Membrane and Bending Components

In the force and strain arrays and in the matrix DDNDDE, direct membrane terms are stored first, followed by the shear membrane term, and then the direct and shear bending terms. Only active components are stored, so the number of entries depends on the element type (see *Table 1*).

**Table 1: Active section force/moment components.**

| Element type | Force and moment components |
|---|---|
| Three-dimensional shells (S4R, S8R, S8R5, etc.) and axisymmetric shells with asymmetric deformation (SAXA1*N*, SAXA2*N*) | $N_{11}, N_{22}, N_{12}, M_{11}, M_{22}, M_{12}$ |
| Axisymmetric shells (SAX1, SAX2, etc) | $N_{11}, N_{22}, M_{11}, M_{22}$ |

There are NDI direct membrane and NSHR shear membrane components and NDI direct bending and NSHR shear bending components: a total of NSECV components. The order of the components is defined in *Using a General Shell Section to Define the Section Behavior*.

Engineering measures of shear membrane strain ($\boldsymbol{\gamma_{12}}$) and twist ($\mathbf{K_{12}}$) are used.

## Increments for Which Only the Section Stiffness Can Be Defined

Abaqus/Standard passes zero strain increments into user subroutine *UGENS* to start the first increment of all the steps and all increments of steps for which you have suppressed extrapolation in time from the previous incremental solution (*Defining an Analysis*). In this case you can define only the section stiffness (DDNDDE).

## Stability

You should ensure that the integration scheme coded in this routine is stable—no direct provision is made to include a stability limit in the time stepping scheme based on the calculations in *UGENS*.

## Convergence Rate

DDNDDE must be defined accurately if rapid convergence of the overall Newton scheme is to be achieved. In most cases the accuracy of this definition is the most important factor governing the convergence rate. Unsymmetric equation solution is as much as four times as expensive as the corresponding symmetric system. Therefore, if the section stiffness matrix (DDNDDE) is only slightly unsymmetric, it might be computationally less expensive to use a symmetric approximation and accept a slightly slower rate of convergence.

## Use with Shells That Have Transverse Shear and/or Hourglass Stiffness

If user subroutine *UGENS* is used to describe the section behavior of shells with transverse shear, you must define the transverse shear stiffness (see *Defining the Transverse Shear Stiffness*).

If user subroutine *UGENS* is used to describe the section behavior of shells with hourglass stiffness, you must define the hourglass stiffness parameter for hourglass control based on total stiffness (see *Specifying Nondefault Hourglass Control Parameters for Reduced-Integration Shell Elements*). The hourglass stiffness parameter is not required for enhanced hourglass control, but you can define a scaling factor for the stiffness associated with the drill degree of freedom (rotation about the surface normal).

## Use with Continuum Shell Elements

User subroutine *UGENS* cannot be used to describe the section behavior of continuum shell elements.

## User Subroutine Interface

```
      SUBROUTINE UGENS(DDNDDE,FORCE,STATEV,SSE,SPD,PNEWDT,STRAN,
     1 DSTRAN,TSS,TIME,DTIME,TEMP,DTEMP,PREDEF,DPRED,CENAME,NDI,
     2 NSHR,NSECV,NSTATV,PROPS,JPROPS,NPROPS,NJPROP,COORDS,CELENT,
     3 THICK,DFGRD,CURV,BASIS,NOEL,NPT,KSTEP,KINC,NIT,LINPER)
C
      INCLUDE 'ABA_PARAM.INC'
C
      CHARACTER*80 CENAME
      DIMENSION DDNDDE(NSECV,NSECV),FORCE(NSECV),STATEV(NSTATV),
     1 STRAN(NSECV),DSTRAN(NSECV),TSS(2),TIME(2),PREDEF(*),
     2 DPRED(*),PROPS(*),JPROPS(*),COORDS(3),DFGRD(3,3),
     3 CURV(2,2),BASIS(3,3)

      user coding to define DDNDDE, FORCE, STATEV, SSE, PNEWDT

      RETURN
      END
```

## Variables to Be Defined

DDNDDE(NSECV,NSECV) Section stiffness matrix of the shell section, $\partial \mathbf{N}/\partial \mathbf{E}$, where $\mathbf{N}$ are the section forces and moments on the shell section and $\mathbf{E}$ are the generalized section strains in the shell. DDNDDE(I,J) defines the change in the Ith force component at the end of the time increment caused by an infinitesimal perturbation of the Jth

component of the section strain increment array. The size of this matrix depends on the values of NSECV (see below for details).

Unless you invoke the unsymmetric equation solution capability in the general shell section definition (*Defining Whether or Not the Section Stiffness Matrices Are Symmetric in Abaqus/Standard*), Abaqus/Standard will use only the symmetric part of DDNDDE. The symmetric part of the matrix is calculated by taking one half the sum of the matrix and its transpose.

FORCE(NSECV) This array is passed in as the forces and moments per unit length on the shell surface at the beginning of the increment and must be updated in this routine to be the forces and moments at the end of the increment.

STATEV(NSTATV) An array containing the solution-dependent state variables. These are passed in as the values at the beginning of the increment and must be returned as the values at the end of the increment.

SSE, SPD Elastic strain energy and plastic dissipation, respectively. These are passed in as the values at the beginning of the increment and should be updated to the corresponding energy values at the end of the increment. These values have no effect on the solution; they are used for the energy output.

## Variables That Can Be Updated

PNEWDT Ratio of suggested new time increment to the time increment being used (DTIME, see below). This variable allows you to provide input to the automatic time incrementation algorithms in Abaqus/Standard (if automatic time incrementation is chosen).

PNEWDT is set to a large value before each call to *UGENS*.

If PNEWDT is redefined to be less than 1.0, Abaqus/Standard must abandon the time increment and attempt it again with a smaller time increment. The suggested new time increment provided to the automatic time integration algorithms is PNEWDT × DTIME, where the PNEWDT used is the minimum value for all calls to user subroutines that allow redefinition of PNEWDT for this iteration.

If PNEWDT is given a value that is greater than 1.0 for all calls to user subroutines for this iteration and the increment converges in this iteration, Abaqus/Standard might increase the time increment. The suggested new time increment provided to the automatic time integration algorithms is PNEWDT × DTIME, where the PNEWDT used is the minimum value for all calls to user subroutines for this iteration.

If automatic time incrementation is not selected in the analysis procedure, values of PNEWDT that are greater than 1.0 will be ignored and values of PNEWDT that are less than 1.0 will cause the job to terminate.

## Variables Passed in for Information

STRAN(NSECV) An array containing the generalized section strains (membrane strains and curvature changes) at the beginning of the increment. The size of this array depends on the value of NSECV (see below for details).

DSTRAN(NSECV) Array of generalized section strain increments.

TSS(2) Array containing the transverse shear strains.

TIME(1) Value of step time at the beginning of the current increment.

TIME(2) Value of total time at the beginning of the current increment.

DTIME Time increment.

TEMP Temperature at the start of the increment.

DTEMP Increment of temperature.

`PREDEF` Array of interpolated values of predefined field variables at this point at the start of the increment, based on the values read in at the nodes.

`DPRED` Array of increments of predefined field variables.

`CENAME` User-specified element set name associated with this section, left justified.

`NDI` Number of direct force components at this point.

`NSHR` Number of shear force components at this point.

`NSECV` Size of the force and strain component arrays.

`NSTATV` User-defined number of solution-dependent state variables associated with this section (*Defining the Number of Solution-Dependent Variables That Must Be Stored for the Section*).

`PROPS(NPROPS)` A floating point array containing the `NPROPS` real property values defined for use with this section.

`JPROPS(NJPROP)` An integer array containing the `NJPROP` integer property values defined for use with this section.

`NPROPS` User-defined number of real property values associated with this section (*Defining the Section Properties*).

`NJPROP` User-defined number of integer property values associated with the element (*Defining the Section Properties*).

`COORDS` An array containing the current coordinates of this integration point.

`CELENT` Characteristic element length in the reference surface.

`THICK` Original section thickness.

`DFGRD(3,3)`

An array containing the components of the midsurface deformation gradient, $\overline{f}_{ij}$. The deformation gradient curvature tensor is available for finite-strain shells (S3/S3R, S4, S4R, SAXs, and SAXAs); it is not available for small-strain shells.

The deformation gradient is stored as a 3 × 3 matrix with component equivalence `DFGRD(I,J)` $\Leftrightarrow \overline{f}_{ij}$. $\overline{f}_{\alpha\beta}$ (Greek subscripts range from 1 to 2) are the in-plane components of the deformation gradient, and $\overline{f}_{33}$ is the thickness change component.

The components, $\overline{f}_{\alpha 3}$, are the transverse shear strains scaled by $\overline{f}_{33}$. The remaining components, $\overline{f}_{3\beta}$, are all zero.

The tensor is provided in the local shell coordinate system.

`CURV(2,2)`

An array containing the midsurface curvature tensor, $b_{\alpha\beta}$. The curvature tensor is available for finite-strain shells (S3/S3R, S4, S4R, SAXs, and SAXAs); it is not available for small-strain shells. The initial curvature, $b^{\circ}_{\alpha\beta}$, does not affect the formulation of these elements (see *Curvature increment*) and can be assumed to be zero, $b^{\circ}_{\alpha\beta} = \mathbf{0}$.

The curvature tensor is stored as a 2 × 2 matrix with component equivalence `CURV(I,J)` $\Leftrightarrow \quad b_{\alpha\beta}$.

The tensor is provided in the local shell coordinate system.

`BASIS(3,3)` An array containing the direction cosines of the shell local surface coordinate system. `BASIS(1,1)`, `BASIS(2,1)`, and `BASIS(3,1)` give the (1, 2, 3) components of the first local direction, etc. The first two directions are in the plane of the element surface, and the third direction is the normal. The conventions for local directions on shell surfaces are defined in *Conventions*. You can redefine the local system; see *Orientations*.

`NOEL` Element number.

| | |
|---|---|
| NPT | Integration point number. |
| KSTEP | Step number. |
| KINC | Increment number. |
| NIT | Iteration number. NIT=0 during the first assembly of the system matrix in any increment. |
| LINPER | Linear perturbation flag. LINPER=1 if the step is a linear perturbation step. LINPER=0 if the step is a general step. |

# UHARD: User subroutine to define the yield surface size and hardening parameters for isotropic plasticity or combined hardening models.

**Products:** Abaqus/Standard

**References:**

- *Classical Metal Plasticity*
- *Models for Metals Subjected to Cyclic Loading*
- **CYCLIC HARDENING*
- **PLASTIC*
- *UHARD*

## Overview

User subroutine *UHARD*:

- is called at all material calculation points of elements for which the material definition includes user-defined isotropic hardening or cyclic hardening for metal plasticity;
- can be used to define a material's isotropic yield behavior;
- can be used to define the size of the yield surface in a combined hardening model;
- can include material behavior dependent on field variables or state variables; and
- requires, when appropriate, that the values of the derivatives of the yield stress (or yield surface size in combined hardening models) be defined with respect to the strain, strain rate, and temperature.

## User Subroutine Interface

```
      SUBROUTINE UHARD(SYIELD,HARD,EQPLAS,EQPLASRT,TIME,DTIME,TEMP,
     1     DTEMP,NOEL,NPT,LAYER,KSPT,KSTEP,KINC,CMNAME,NSTATV,
     2     STATEV,NUMFIELDV,PREDEF,DPRED,NUMPROPS,PROPS)
C
      INCLUDE 'ABA_PARAM.INC'
C
      CHARACTER*80 CMNAME
      DIMENSION HARD(3),STATEV(NSTATV),TIME(*),
     $          PREDEF(NUMFIELDV),DPRED(*),PROPS(*)

      user coding to define SYIELD,HARD(1),HARD(2),HARD(3)

      RETURN
      END
```

## Variables to Be Defined

`SYIELD` — $\sigma^0$. Yield stress for isotropic plasticity. Yield surface size for combined hardening.

`HARD(1)` — Variation of `SYIELD` with respect to the equivalent plastic strain, $\partial \sigma^0 / \partial \bar{\varepsilon}^{pl}$.

`HARD(2)` — Variation of `SYIELD` with respect to the equivalent plastic strain rate, $\partial \sigma^0 / \partial \dot{\bar{\varepsilon}}^{pl}$.

| | |
|---|---|
| HARD(3) | Variation of SYIELD with respect to temperature, $\partial\sigma^0/\partial\theta$. This quantity is required only in adiabatic, fully coupled temperature-displacement, and thermal-electrical-structural analyses. |
| STATEV(NSTATV) | Array containing the user-defined solution-dependent state variables at this point. These are supplied as values at the beginning of the increment or as values updated by other user subroutines (see *Allocating Space for Solution-Dependent State Variables*) and must be returned as values at the end of the increment. |

## Variables Passed in for Information

| | |
|---|---|
| EQPLAS | Equivalent plastic strain, $\bar{\varepsilon}^{pl}$. |
| EQPLASRT | Equivalent plastic strain rate, $\dot{\bar{\varepsilon}}^{pl}$. |
| TIME(1) | Value of step time at the beginning of the current increment. |
| TIME(2) | Value of total time at the beginning of the current increment. |
| DTIME | Time increment. |
| TEMP | Temperature at the beginning of the increment. |
| DTEMP | Increment of temperature. |
| NOEL | Element number. |
| NPT | Integration point number. |
| LAYER | Layer number (for composite shells and layered solids). |
| KSPT | Section point number within the current layer. |
| KSTEP | Step number. |
| KINC | Increment number. |
| CMNAME | User-specified material name, left justified. |
| NSTATV | User-specified number of solution-dependent state variables associated with this material (*Allocating Space for Solution-Dependent State Variables*). |
| NUMFIELDV | Number of field variables. |
| PREDEF(NUMFIELDV) | Array of interpolated values of predefined field variables at this material point at the start of the increment based on the values read in at the nodes (initial values at the beginning of the analysis and current values during the analysis). |
| DPRED(NUMFIELDV) | Array of increments of predefined field variables at this material point for this increment; this includes any values updated by user subroutine *USDFLD*. |
| NPROPS | Number of hardening properties entered for this user-defined hardening definition. |
| PROPS(NPROPS) | Array of hardening properties entered for this user-defined hardening definition. |

# UHYPEL: User subroutine to define a hypoelastic stress-strain relation.

**Products:** Abaqus/Standard

**References:**

- *Hypoelastic Behavior*
- **HYPOELASTIC*

## Overview

User subroutine *UHYPEL*:

- can be used to define isotropic hypoelastic material behavior, thus requiring the definition of Young's modulus, $E$, and Poisson's ratio, $\nu$;
- is called at all material calculation points of elements for which the material definition contains user-defined hypoelastic behavior;
- can be used in conjunction with user subroutine *USDFLD* to redefine any field variables that are passed in (see *USDFLD*); and
- ignores any data specified outside the user subroutine for the associated hypoelastic material definition.

## Special Considerations for Various Element Types

There are several special considerations that need to be noted.

### Beams and Shells That Calculate Transverse Shear Energy

When *UHYPEL* is used to define the material response of shell or beam elements that calculate transverse shear energy, Abaqus/Standard cannot calculate a default value for the transverse shear stiffness of the element. Hence, you must define the material transverse shear modulus (see *Defining the Elastic Transverse Shear Modulus*) or the element's transverse shear stiffness (see *Shell Section Behavior*).

### Elements with Hourglassing Modes

If this capability is used to describe the material of elements with hourglassing modes, you must define the hourglass stiffness for hourglass control based on the total stiffness approach. The hourglass stiffness is not required for enhanced hourglass control, but you can define a scaling factor for the stiffness associated with the drill degree of freedom (rotation about the surface normal). See *Section Controls*.

## User Subroutine Interface

```
      SUBROUTINE UHYPEL(E,GNU,STRAIN,NDI,NSHR,EINV1,EINV2,EINV3,
     1 COORDS,NOEL,TEMP,PREDEF,CMNAME)
C
      INCLUDE 'ABA_PARAM.INC'
C
      CHARACTER*80 CMNAME
C
      DIMENSION STRAIN(*),COORDS(3),PREDEF(*)
```

```
      user coding to define E and GNU

      RETURN
      END
```

## Variables to Be Defined

| | |
|---|---|
| E | Young's modulus. |
| GNU | Poisson's ratio. |

## Variables Passed in for Information

| | |
|---|---|
| STRAIN | Array containing the total (elastic) strains, $(\boldsymbol{\varepsilon})$. |
| NDI | Number of direct strain components at this point. |
| NSHR | Number of shear strain components at this point. |
| EINV1 | $I_1 = \mathbf{trace}\ (\boldsymbol{\varepsilon})$, the first strain invariant. |
| EINV2 | $I_2 = 1/2\left(\boldsymbol{\varepsilon} : \boldsymbol{\varepsilon} - {I_1}^2\right)$, the second strain invariant. |
| EINV3 | $I_3 = \mathbf{det}(\boldsymbol{\varepsilon})$, the third strain invariant. |
| COORDS | An array containing the coordinates of the material point. These are the current coordinates if geometric nonlinearity is accounted for during the step (see *Defining an Analysis*); otherwise, the array contains the original coordinates of the point. |
| NOEL | Element number. |
| TEMP | Current temperature at this point. |
| PREDEF | An array containing current values of the predefined field variables at this point (initial values at the beginning of the analysis and current values during the analysis). |
| CMNAME | User-specified material name, left justified. |

# UHYPER: User subroutine to define a hyperelastic material.

**Products:** Abaqus/Standard

**References:**

- *Hyperelastic Behavior of Rubberlike Materials*
- **HYPERELASTIC*
- *UMAT and UHYPER*

## Overview

User subroutine *UHYPER*:

- can be used to define the strain energy potential for isotropic hyperelastic material behavior;
- is called at all material calculation points of elements for which the material definition contains user-defined hyperelastic behavior;
- can include material behavior dependent on field variables or state variables; and
- requires that the values of the derivatives of the strain energy density function of the hyperelastic material be defined with respect to the strain invariants.

## Special Considerations for Various Element Types

There are several special considerations that need to be noted.

### Shells That Calculate Transverse Shear Energy

When *UHYPER* is used to define the material response of shell elements that calculate transverse shear energy, Abaqus/Standard cannot calculate a default value for the transverse shear stiffness of the element. Hence, you must define the material transverse shear modulus (see *Defining the Elastic Transverse Shear Modulus*) or the element's transverse shear stiffness (see *Shell Section Behavior*).

### Elements with Hourglassing Modes

If this capability is used to describe the material of elements with hourglassing modes, you must define the hourglass stiffness for hourglass control based on the total stiffness approach. The hourglass stiffness is not required for enhanced hourglass control, but you can define a scaling factor for the stiffness associated with the drill degree of freedom (rotation about the surface normal). See *Section Controls*.

## User Subroutine Interface

```
      SUBROUTINE UHYPER(BI1,BI2,AJ,U,UI1,UI2,UI3,TEMP,NOEL,
     1 CMNAME,JFLAGS,NUMSTATEV,STATEV,NUMFIELDV,FIELDV,
     2 FIELDVINC,NUMPROPS,PROPS)
C
      INCLUDE 'ABA_PARAM.INC'
C
      CHARACTER*80 CMNAME
      DIMENSION U(2),UI1(3),UI2(6),UI3(6),STATEV(NUMSTATEV,2),FIELDV(*),
     2 FIELDVINC(*),PROPS(*),JFLAGS(2)
```

```
      user coding to define U,UI1,UI2,UI3,STATEV(*,1)

      RETURN
      END
```

## Variables to Be Defined

`U(1)` $U$, strain energy density function. For a compressible material, at least one derivative involving $J$ should be nonzero. For an incompressible material, all derivatives involving $J$ will be ignored. The strain invariants—$\bar{I}_1$, $\bar{I}_2$, and $J$—are defined in *Hyperelastic Behavior of Rubberlike Materials*.

`U(2)` $\tilde{U}_{dev}$, the deviatoric part of the strain energy density of the primary material response. This quantity is needed only if the current material definition also includes Mullins effect (see *Stress Softening in Rubberlike Materials*).

`UI1(1)` $\partial U/\partial \bar{I}_1$.

`UI1(2)` $\partial U/\partial \bar{I}_2$.

`UI1(3)` $\partial U/\partial J$.

`UI2(1)` $\partial^2 U/\partial \bar{I}_1^2$.

`UI2(2)` $\partial^2 U/\partial \bar{I}_2^2$.

`UI2(3)` $\partial^2 U/\partial J^2$.

`UI2(4)` $\partial^2 U/\partial \bar{I}_1 \partial \bar{I}_2$.

`UI2(5)` $\partial^2 U/\partial \bar{I}_1 \partial J$.

`UI2(6)` $\partial^2 U/\partial \bar{I}_2 \partial J$.

`UI3(1)` $\partial^3 U/\partial \bar{I}_1^2 \partial J$.

`UI3(2)` $\partial^3 U/\partial \bar{I}_2^2 \partial J$.

`UI3(3)` $\partial^3 U/\partial \bar{I}_1 \partial \bar{I}_2 \partial J$.

`UI3(4)` $\partial^3 U/\partial \bar{I}_1 \partial J^2$.

`UI3(5)` $\partial^3 U/\partial \bar{I}_2 \partial J^2$.

`UI3(6)` $\partial^3 U/\partial J^3$.

`STATEV(*,1)` Array containing the user-defined solution-dependent state variables at this point. These are supplied as values at the start of the increment or as values updated by other user subroutines (see *About User Subroutines and Utilities*) and must be returned as values at the end of the increment.

## Variables Passed in for Information

| | |
|---|---|
| BI1 | $\bar{I}_1$. |
| BI2 | $\bar{I}_2$. |
| AJ | $J$. |
| TEMP | Current temperature at this point. |
| NOEL | Element number. |
| CMNAME | User-specified material name, left justified. |
| JFLAGS(1) | Incompressibility flag defined to be 1 if the material is specified as incompressible or 0 if the material is specified as compressible. |
| JFLAGS(2) | Network identification number, which indicates the network for which the strain energy potential is defined when *UHYPER* is used in combination with the parallel rheological framework (see *Parallel Rheological Framework*). You can define different strain energy functions for different networks. A value of 0 indicates the equilibrium network, and a value of N indicates the $n^{th}$ network. |
| NUMSTATEV | User-defined number of solution-dependent state variables associated with this material (see *Allocating Space for Solution-Dependent State Variables*). |
| STATEV(*,2) | Array containing the user-defined solution-dependent state variables at this point. These are supplied as values at the start of the increment. |
| NUMFIELDV | Number of field variables. |
| FIELDV | Array of interpolated values of predefined field variables at this material point at the end of the increment based on the values read in at the nodes (initial values at the beginning of the analysis and current values during the analysis). |
| FIELDVINC | Array of increments of predefined field variables at this material point for this increment; this includes any values updated by the user subroutine USDFLD. |
| NUMPROPS | Number of material properties entered for this user-defined hyperelastic material. |
| PROPS | Array of material properties entered for this user-defined hyperelastic material. |

# UHYPER_STRETCH: User subroutine to define a hyperelastic material in terms of principal stretches.

**Products:** Abaqus/Standard

**References:**

- *Hyperelastic Behavior of Rubberlike Materials*
- **HYPERELASTIC*

## Overview

User subroutine *UHYPER_STRETCH*:

- can be used to define the strain energy potential for isotropic hyperelastic material behavior;
- is called at all material calculation points of elements for which the material definition contains user-defined hyperelastic behavior;
- can include material behavior dependent on field variables or state variables;
- requires that the strain energy density function has the Valanis-Landel form (see *Valanis-Landel Form*); and
- requires that the values of the derivatives of the strain energy density function of the hyperelastic material be defined with respect to the principal deviatoric stretches and the elastic volume ratio (see *Valanis-Landel Form*).

## Shells That Calculate Transverse Shear Energy

When you use *UHYPER_STRETCH* to define the material response of shell elements that calculate transverse shear energy, Abaqus/Standard cannot calculate a default value for the transverse shear stiffness of the element.Therefore, you must define the material transverse shear modulus (see *Defining the Elastic Transverse Shear Modulus*) or the element's transverse shear stiffness (see *Shell Section Behavior*).

## Elements with Hourglassing Modes

If you use *UHYPER_STRETCH* to describe the material of elements with hourglassing modes, you must define the hourglass stiffness for hourglass control based on the total stiffness approach. The hourglass stiffness is not required for enhanced hourglass control, but you can define a scaling factor for the stiffness associated with the drill degree of freedom (rotation about the surface normal) [see *Section Controls*].

## User Subroutine Interface

```
      SUBROUTINE UHYPER_STRETCH(DLAMBDA,AJ,U,U1,U2,U3,U4,TEMP,NOEL,
     1 CMNAME,INCMPFLAG,NUMSTATEV,STATEVOLD,STATEV,NUMFIELDV,FIELDV,
     2
FIELDVINC,NUMPROPS,PROPS,I_ARRAY,NIARRAY,R_ARRAY,NRARRAY,C_ARRAY,NCARRAY)
C
      INCLUDE 'ABA_PARAM.INC'
C
      CHARACTER*80 CMNAME,C_ARRAY(*)
      DIMENSION
DLAMBDA(*),U(2),U1(4),U2(4),U3(4),U4(3),STATEVOLD(*),STATEV(*),FIELDV(*),
     2 FIELDVINC(*),PROPS(*),I_ARRAY(),R_ARRAY(*)
```

```
      user coding to define U,U1,U2,U3,U4,STATEV


      RETURN
      END
```

## Variables to Be Defined

`U(1)` $U$, strain energy density function. For a compressible material, at least one derivative involving $J$ must be nonzero. For an incompressible material, all derivatives involving $J$ are ignored. The principal deviatoric stretches ($\overline{\lambda}_1$, $\overline{\lambda}_2$, $\overline{\lambda}_3$, and $J$) are defined in *Hyperelastic Behavior of Rubberlike Materials*.

`U(2)` $\widetilde{U}_{dev}$, the deviatoric part of the strain energy density of the primary material response. This quantity is needed only if the current material definition includes the Mullins effect (see *Stress Softening in Rubberlike Materials*).

`U1(1)` $\partial U/\partial \overline{\lambda}_1$.

`U1(2)` $\partial U/\partial \overline{\lambda}_2$.

`U1(3)` $\partial U/\partial \overline{\lambda}_3$.

`U1(4)` $\partial U/\partial J$.

`U2(1)` $\partial^2 U/\partial \overline{\lambda}_1^2$.

`U2(2)` $\partial^2 U/\partial \overline{\lambda}_2^2$.

`U2(3)` $\partial^2 U/\partial \overline{\lambda}_3^2$.

`U2(4)` $\partial^2 U/\partial J^2$.

`U3(1)` $\partial^3 U/\partial \overline{\lambda}_1^3$.

`U3(2)` $\partial^3 U/\partial \overline{\lambda}_2^3$.

`U3(3)` $\partial^3 U/\partial \overline{\lambda}_3^3$.

`U3(4)` $\partial^3 U/\partial J^3$.

`U4(1)` $\partial^4 U/\partial \overline{\lambda}_1^4$.

`U4(2)` $\partial^4 U/\partial \overline{\lambda}_2^4$.

`U4(3)` $\partial^4 U/\partial \overline{\lambda}_3^4$.

`STATEV` Array containing the user-defined solution-dependent state variables at this point. These are supplied as values at the start of the increment or as values updated by other user subroutines (see *About User Subroutines and Utilities*) and must be returned as values at the end of the increment.

## Variables Passed in for Information

`DLAMBDA(1)` $\overline{\lambda}_1$.

| | |
|---|---|
| DLAMBDA(2) | $\bar{\lambda}_2$. |
| DLAMBDA(3) | $\bar{\lambda}_3$. |
| AJ | $J$. |
| TEMP | Current temperature at this point. |
| NOEL | Element number. |
| CMNAME | User-specified material name, left justified. |
| INCMPFLAG | Incompressibility flag defined to be 1 if the material is specified as incompressible or 0 if the material is specified as compressible. |
| NUMSTATEV | User-defined number of solution-dependent state variables associated with this material (see *Allocating Space for Solution-Dependent State Variables*). |
| STATEVOLD | Array containing the user-defined solution-dependent state variables at this material point at the start of the increment. |
| NUMFIELDV | Number of field variables. |
| FIELDV | Array of interpolated values of predefined field variables at this material point at the end of the increment based on the values read in at the nodes (initial values at the beginning of the analysis and current values during the analysis). |
| FIELDVINC | Array of increments of predefined field variables at this material point for this increment; this includes any values updated by user subroutine USDFLD. |
| NUMPROPS | Number of material properties entered for this user-defined hyperelastic material. |
| PROPS | Array of material properties entered for this user-defined hyperelastic material. |
| I_ARRAY(1) | Network identification number, which indicates the network for which the strain energy potential is defined when *UHYPER_STRETCH* is used in combination with the parallel rheological framework (see *Parallel Rheological Framework*). You can define different strain energy functions for different networks. A value of 0 indicates the equilibrium network, and a value of N indicates the $n^{th}$ network. |
| NIARRAY | Size of array I_ARRAY. |
| R_ARRAY | Not used. |
| NRARRAY | Size of array R_ARRAY. |
| C_ARRAY | Not used. |
| NCARRAY | Size of array C_ARRAY. |

# UINTER: User subroutine to define surface interaction behavior for contact surfaces.

**Products:** Abaqus/Standard

**References:**

- *User-Defined Interfacial Constitutive Behavior*
- **SURFACE INTERACTION*
- *UINTER*

## Overview

User subroutine *UINTER*:

- is called at points on the secondary surface of a contact pair with a user-defined constitutive model defining the interaction between the surfaces;
- can be used to define the mechanical (normal and shear) and thermal (heat flux) interactions between surfaces;
- can be used when the normal surface behavior (contact pressure versus overclosure) models (*Contact Pressure-Overclosure Relationships*) or the extended versions of the classical Coulomb friction model (*Frictional Behavior*) are too restrictive and a more complex definition of normal and shear transmission between contacting surfaces, including damping properties, are required;
- must provide the entire definition of the mechanical and the thermal interaction between the contacting surfaces (hence, no additional surface behaviors can be specified in conjunction with this capability);
- can provide the entire definition of viscous and structural damping for interactions between the contacting surfaces for direct and mode-based steady-state dynamic analysis (including the subspace projection method), transient mode-based analysis, complex eigenvalue extraction, matrix generation, and substructure generation;
- only accounts for element damping in mode-based procedures if the SIM architecture is used;
- can use and update solution-dependent state variables; and
- is not available for contact elements.

## User Subroutine Interface

```
      SUBROUTINE UINTER(STRESS,DDSDDR,DVISCOUS,DSTRUCTURAL,FLUX,DDFDDT,
     1 DDSDDT,DDFDDR,STATEV,SED,SFD,SPD,SVD,SCD,PNEWDT,RDISP,
     2 DRDISP,
     3 TEMP,DTEMP,PREDEF,DPRED,TIME,DTIME,FREQR,CINAME,SECNAME,
     4 MAINNAME,
     5 PROPS,COORDS,ALOCALDIR,DROT,AREA,CHRLNGTH,NODE,NDIR,NSTATV,
     6 NPRED,NPROPS,MCRD,KSTEP,KINC,KIT,LINPER,LOPENCLOSE,LSTATE,
     7 LSDI,LPRINT)
C
      INCLUDE 'ABA_PARAM.INC'
C
      CHARACTER*80 CINAME,SECNAME,MAINNAME
      DIMENSION STRESS(NDIR),DDSDDR(NDIR,NDIR),FLUX(2),DDFDDT(2,2),
     1 DDSDDT(NDIR,2),DDFDDR(2,NDIR),STATEV(NSTATV),
     2 RDISP(NDIR),DRDISP(NDIR),TEMP(2),DTEMP(2),PREDEF(2,NPRED),
     3 DPRED(2,NPRED),TIME(2),PROPS(NPROPS),COORDS(MCRD),
```

```
     4 ALOCALDIR(3,3),DROT(2,2),DVISCOUS(NDIR,NDIR),
     5 DSTRUCTURAL(NDIR,NDIR)

      user coding to define STRESS, DDSDDR, FLUX, DDFDDT,
                            DDSDDT, DDFDDR,


      and, optionally, STATEV, SED, SFD, SPD, SVD, SCD, PNEWDT,
                            LOPENCLOSE, LSTATE, LSDI, DVISCOUS, DSTRUCTURAL

      RETURN

 END
```

## Variables to Be Defined

`STRESS(NDIR)` This array is passed in as the stress between the secondary and main surfaces at the beginning of the increment and must be updated in this routine to be the stress at the end of the increment. The stress must be defined in a local coordinate system (see `ALOCDIR`). This variable must be defined for a stress/displacement, a fully coupled temperature-displacement, or a coupled thermal-electrical-structural analysis. The sign convention for stresses is that a positive stress indicates compression across contact surfaces, while a negative stress indicates tension.

`DDSDDR(NDIR,NDIR)` Interface stiffness matrix. `DDSDDR(I,J)` defines the change in the Ith stress component at the end of the time increment caused by an infinitesimal perturbation of the Jth component of the relative displacement increment array. Unless you invoke the unsymmetric equation solution capability in the contact property model definition (*Use with the Unsymmetric Equation Solver in Abaqus/Standard*), Abaqus/Standard will use only the symmetric part of `DDSDDR`. For a particular off-diagonal `(I,J)` entry, the symmetrization is done by halving the sum of `(I,J)` and `(J,I)` components. `DDSDDR` must be defined for a stress/displacement, a fully coupled temperature-displacement, or a coupled thermal-electrical-structural analysis to ensure proper convergence characteristics.

`FLUX(2)` Magnitude of the heat flux flowing into the secondary and main surfaces, respectively. This array is passed in as the value at the beginning of the increment and must be updated to the flux at the end of the increment. The convention for defining the flux is that a positive flux indicates heat flowing into a surface, while a negative flux indicates heat flowing out of the surface. This variable must be defined for a heat transfer, a fully coupled temperature-displacement, or a coupled thermal-electrical-structural analysis. The sum of these two flux terms represents the heat generated in the interface, and the difference in these flux terms represents the heat conducted through the interface.

`DDFDDT(2,2)` The negative of the variation of the flux at the two surfaces with respect to their respective temperatures, for a fixed relative displacement. This variable must be defined for a heat transfer, a fully coupled temperature-displacement, or a coupled thermal-electrical-structural analysis to ensure proper convergence characteristics. The entries in the first row contain the negatives of the derivatives of `FLUX(1)` with respect to `TEMP(1)` and `TEMP(2)`, respectively. The entries in the second row contain the negatives of the corresponding derivatives of `FLUX(2)`.

`DDSDDT(NDIR,2)` Variation of the stress with respect to the temperatures of the two surfaces for a fixed relative displacement. This variable is required only for thermally coupled elements (in a fully coupled temperature-displacement or a coupled thermal-electrical-structural analysis), in which the stress is a function of the surface temperatures.

`DDSDDT(NDIR,1)` corresponds to the secondary surface, and `DDSDDT(NDIR,2)` corresponds to the main surface.

`DDFDDR(2,NDIR)` Variation of the flux with respect to the relative displacement between the two surfaces. This variable is required only for thermally coupled elements (in a fully coupled temperature-displacement or a coupled thermal-electrical-structural analysis), in which the flux is a function of the relative displacement. `DDFDDR(1,NDIR)` corresponds to the secondary surface, and `DDFDDR(2,NDIR)` corresponds to the main surface.

## Variables That Can Be Updated

`DVISCOUS(NDIR,NDIR)` Interface viscous damping matrix that can be used in direct steady-state dynamic analysis and transient and steady-state mode-based dynamic analysis (including the subspace projection method), as well as in complex eigenvalue extraction, matrix generation, and substructure generation. `DVISCOUS(I,J)` defines an element in the material viscous damping matrix at the current frequency. Abaqus/Standard requires that this element is defined as a damping value for each `(I,J)` entry in the damping matrix.

Unless you invoke the unsymmetric equation solution capability in the contact property model definition (*Use with the Unsymmetric Equation Solver in Abaqus/Standard*), Abaqus/Standard uses only the symmetric part of `DVISCOUS`. For a particular off-diagonal `(I,J)` entry the symmetrization is done by halving the sum of the `(I,J)` and `(J,I)` components.

`DSTRUCTURAL(NDIR,NDIR)` Interface structural damping matrix that can be used in direct steady-state dynamic analysis and steady-state mode-based dynamic analysis (including the subspace projection method), as well as in complex eigenvalue extraction, matrix generation, and substructure generation. `DSTRUCTURAL(I,J)` defines an element in the material structural damping matrix.

Unless you invoke the unsymmetric equation solution capability in the contact property model definition (*Use with the Unsymmetric Equation Solver in Abaqus/Standard*), Abaqus/Standard uses only the symmetric part of `DSTRUCTURAL`. For a particular off-diagonal `(I,J)` entry the symmetrization is done by halving the sum of the `(I,J)` and `(J,I)` components.

`STATEV(NSTATV)` An array containing the solution-dependent state variables. These are passed in as values at the beginning of the increment and must be returned as values at the end of the increment. You define the number of available state variables as described in *Allocating Space for Solution-Dependent State Variables*.

`SED` This variable is passed in as the value of the elastic energy density at the start of the increment and should be updated to the elastic energy density at the end of the increment. This variable is used for output only and has no effect on other solution variables. It contributes to the output variable ALLSE.

`SFD` This variable should be defined as the incremental frictional dissipation. The units are energy per unit area. This variable is used for output only and has no effect on other solution variables. It contributes to the output variables ALLFD and SFDR (and related variables). For computing its contribution to SFDR, `SFD` is divided by the time increment.

`SPD` This variable should be defined as the incremental dissipation due to plasticity effects in the interfacial constitutive behavior. The units are energy per unit area. This variable is used for output only and has no effect on other solution variables. It contributes to the output variable ALLPD.

SVD

This variable should be defined as the incremental dissipation due to viscous effects in the interfacial constitutive behavior. The units are energy per unit area. This variable is used for output only and has no effect on other solution variables. It contributes to the output variable ALLVD.

SCD

This variable should be defined as the incremental dissipation due to creep effects in the interfacial constitutive behavior. The units are energy per unit area. This variable is used for output only and has no effect on other solution variables. It contributes to the output variable ALLCD.

PNEWDT

Ratio of suggested new time increment to the time increment currently being used (DTIME, see below). This variable allows you to provide input to the automatic time incrementation algorithms in Abaqus/Standard (if automatic time incrementation is chosen).

PNEWDT is set to a large value before each call to *UINTER*.

If PNEWDT is redefined to be less than 1.0, Abaqus/Standard must abandon the time increment and attempt it again with a smaller time increment. The suggested new time increment provided to the automatic time integration algorithms is PNEWDT × DTIME, where the PNEWDT used is the minimum value for all calls to user subroutines that allow redefinition of PNEWDT for this iteration.

If PNEWDT is given a value that is greater than 1.0 for all calls to user subroutines for this iteration and the increment converges in this iteration, Abaqus/Standard may increase the time increment. The suggested new time increment provided to the automatic time integration algorithms is PNEWDT × DTIME, where the PNEWDT used is the minimum value for all calls to user subroutines for this iteration.

If automatic time incrementation is not selected in the analysis procedure, values of PNEWDT greater than 1.0 will be ignored and values of PNEWDT less than 1.0 will cause the job to terminate.

LOPENCLOSE

An integer flag that is used to track the contact status in situations where user subroutine *UINTER* is used to model standard contact between two surfaces, like the default hard contact model in Abaqus/Standard. It comes in as the value at the beginning of the current iteration and should be set to the value at the end of the current iteration. It is set to −1 at the beginning of the analysis before *UINTER* is called. You should set it to 0 to indicate an open status and to 1 to indicate a closed status. A change in this flag from one iteration to the next will have two effects. It will result in output related to a change in contact status if you request a detailed contact printout in the message file (*The Abaqus/Standard Message File*). In addition, it will also trigger a severe discontinuity iteration. Any time this flag is reset to a value of −1, Abaqus/Standard assumes that the flag is not being used. A change in this flag from −1 to another value or vice versa will not have any of the above effects.

LSTATE

An integer flag that should be used in non-standard contact situations where a simple open/close status is not appropriate or enough to describe the state. It comes in as the value at the beginning of the current iteration and should be set to the value at the end of the current iteration. It is set to −1 at the beginning of the analysis before *UINTER* is called. It can be assigned any user-defined integer value, each corresponding to a different state. You can track changes in the value of this flag and use it to output appropriate diagnostic messages to the message file (unit 7). You may choose to output diagnostic messages only when a detailed contact printout is requested (*The Abaqus/Standard Message File*). In the latter case, the LPRINT parameter is useful. In conjunction with

the LSTATE flag, you may also utilize the LSDI flag to trigger a severe discontinuity iteration any time the state changes from one iteration to the next. Any time this flag is reset to a value of −1, Abaqus/Standard assumes that the flag is not being used.

LSDI — This flag is set to 0 before each call to *UINTER* and should be set to 1 if the current iteration should be treated as a severe discontinuity iteration. This would typically be done in non-standard contact situations based on a change in the value of the LSTATE flag from one iteration to the next. The use of this flag has no effect when the LOPENCLOSE flag is also used. In that case, severe discontinuity iterations are determined based on changes in the value of LOPENCLOSE alone.

## Variables Passed in for Information

RDISP(NDIR) — An array containing the current relative positions between the two surfaces at the end of the increment. The first component is the relative position of the point on the secondary surface, with respect to the main surface, in the normal direction. The second and third components, if applicable, are the accumulated incremental relative tangential displacements, measured from the beginning of the analysis. For the relative position in the normal direction a negative quantity represents an open status, while a positive quantity indicates penetration into the main surface. For open points on the secondary surface for which no pairing main is found, the first component is a very large negative number ($-1 \times 10^{36}$). The local directions in which the relative displacements are defined are stored in ALOCALDIR.

DRDISP(NDIR) — An array containing the increments in relative positions between the two surfaces.

TEMP(2) — Temperature at the end of the increment at a point on the secondary surface and the opposing main surface, respectively.

DTEMP(2) — Increment in temperature at the point on the secondary surface and the opposing main surface, respectively.

PREDEF(2,NPRED) — An array containing pairs of values of all the predefined field variables at the end of the current increment (initial values at the beginning of the analysis and current values during the analysis). The first value in a pair, PREDEF(1,NPRED), corresponds to the value at the point on the secondary surface, and the second value, PFREDEF(2,NPRED), corresponds to the value of the field variable at the nearest point on the opposing surface.

DPRED(2,NPRED) — Array of increments in predefined field variables.

TIME(1) — Value of step time at the end of the increment.

TIME(2) — Value of total time at the end of the increment.

DTIME — Current increment in time.

FREQR — Current frequency for steady-state dynamic analysis in rad/time.

CINAME — User-specified surface interaction name, left justified.

SECNAME — Secondary surface name.

MAINNAME — Main surface name.

PROPS(NPROPS) — User-specified array of property values to define the interfacial constitutive behavior between the contacting surfaces.

COORDS(MCRD) — An array containing the current coordinates of this point.

ALOCALDIR(3,3) — An array containing the direction cosines of the local surface coordinate system. The directions are stored in columns. For example, ALOCALDIR(1,1), ALOCALDIR(2,1), and ALOCALDIR(3,1) give the (1, 2, 3) components of the normal direction. Thus, the first direction is the normal direction to the surface, and the remaining two directions are

| | |
|---|---|
| | the local tangent directions in the plane of the surface. The local system is defined by the geometry of the main surface. The convention for the local directions is the same as the convention in situations where the model uses the built-in contact capabilities in Abaqus/Standard (described in *Contact Formulations in Abaqus/Standard* for the tangential directions). |
| `DROT(2,2)` | Rotation increment matrix. For contact with a three-dimensional rigid surface, this matrix represents the incremental rotation of the surface directions relative to the rigid surface. It is provided so that vector- or tensor-valued state variables can be rotated appropriately in this subroutine. Relative displacement components are already rotated by this amount before *UINTER* is called. This matrix is passed in as a unit matrix for two-dimensional and axisymmetric contact problems. |
| `AREA` | Surface area associated with the contact point. |
| `CHRLNGTH` | Characteristic contact surface face dimension. |
| `NODE` | User-defined global secondary node number (or internal node number for models defined in terms of an assembly of part instances) involved with this contact point. Corresponds to the predominant secondary node of the constraint if the surface-to-surface contact formulation is used. |
| `NDIR` | Number of force components at this point. |
| `NSTATV` | Number of solution-dependent state variables. |
| `NPRED` | Number of predefined field variables. |
| `NPROPS` | User-defined number of property values associated with this interfacial constitutive model (*Interfacial Constants*). |
| `MCRD` | Number of coordinate directions at the contact point. |
| `KSTEP` | Step number. |
| `KINC` | Increment number. |
| `KIT` | Iteration number. `KIT`=1 for the first assembly, `KIT`=2 for the first recovery/second assembly, `KIT`=3 for the second recovery/third assembly, and so on. |
| `LINPER` | Linear perturbation flag. `LINPER`=1 if the step is a linear perturbation step. `LINPER`=0 if the step is a general step. For a linear perturbation step, the inputs to user subroutine *UINTER* represent perturbation quantities about the base state. The user-defined quantities in *UINTER* are also perturbation quantities. The Jacobian terms should be based on the base state. No change in contact status should occur during a linear perturbation step. |
| `LPRINT` | This flag is equal to 1 if a detailed contact printout to the message file is requested and 0 otherwise (*The Abaqus/Standard Message File*). This flag can be used to print out diagnostic messages regarding changes in contact status selectively only when a detailed contact printout is requested. |

# UINTFLUXGEN: User subroutine to define the internal heat generation in a material from different physics fields.

**Products:** Abaqus/Standard

**References:**

## Overview



**Warning:** The use of this subroutine generally requires considerable expertise. You are cautioned that the implementation of any realistic constitutive model requires extensive development and testing. Initial testing on a single-element model with prescribed loading in different fields is strongly recommended.

User subroutine *UINTFLUXGEN*:

- is called for each material integration point for each iteration;
- can be used to define a heat flux due to internal heat generation in a material (for example, as might be associated with phase changes occurring during the solution);
- can be used to define the heat flux due to Butler-Volmer currents in a coupled thermal-electrochemical analysis (see *Coupled Thermal-Electrochemical Analysis*);
- can be used to account for not only the heat fluxes but also fluxes due to other fields such as electrical potentials and ion concentrations in a fully coupled manner;
- can be used in conjunction with user subroutine *USDFLD* if it is desired to redefine any field variables before they are passed in; and
- can be used only in procedures that support electrochemical analyses (see *Electrochemical Analysis*).

## User Subroutine Interface

```
 subroutine uintfluxgen(
     *   fMech, dfMech_dT, dfMech_dSol,
     *   qHeat, dqHeat_dT, dqHeat_dSol,
     *   sources, dSources_dT, dSources_dSol,
     *   iSourceAssemble,
     *   cmname, kstep, lFlags, kinc,
     *   time, dtime, noel, npt,
     *   nDim, ntens, ndi, nshr,
     *   props, nprops, predef, dpredef, npredef,
     *   statev, stateOld, nstatev,
     *   Temp, dTemp, coords,
     *   solutionvar, dsolutionvar, nSolutionVar,
     *   i_array, niarray, r_array, nrarray, c_array, ncarray)
C
      include 'aba_param.inc'
c Include below file to access property and parameter table.
      include 'aba_tcs_param.inc'
c Include below file to access universal constants
      include 'aba_phcon_param.inc'
c Include below file for parametrized location of various solution variables.

      include 'aba_solutionvarloc.inc'
C
```

```
      dimension fMech(nDim),  dfMech_dT(nDim),
     *     dfMech_dSol(nDim, nSolutionVar),qHeat(2),
     *     dqHeat_dSol(nSolutionVar),
     *     sources(nSolutionVar),
     *     dSources_dT(nSolutionVar), dSources_dSol(nSolutionVar,
     *     nSolutionVar),
     *     isourceAssemble(nnAllFieldsMax),
     *     time(2),props(nprops),predef(npredef), dpredef(npredef),
     *     statev(nstatev), statevOld(nstatev),
     *     solutionvar(nsolutionvar), dsolutionvar(nsolutionvar),
     *     coords(*),i_array(*), r_array(*), lFlags(*)
      character*80   cmname, c_array(NCARRAY)

      user coding

      return
      end
```

## Variables to Be Defined

`fMech` Volumetric forces.

`dfMech_dT` Variation of `fMech` with respect to temperature `Temp`.

`dfMech_dSol` Variation of `fMech` with respect to the different solution variables `solutionvar`.

`qHeat` Volumetric heat generation.

`dqHeat_dT` Variation of `qHeat` with respect to temperature `Temp`.

`dqHeat_dSol` Variation of `qHeat` with respect to the different solution variables `solutionvar`.

`sources` Volumetric sources from the different fields as described in `solutionVar`.

`dSources_dT` Variation of `sources` with respect to temperature `Temp`.

`dSources_dSol` Variation of `sources` with respect to solution variables `solutionvar`.

`iSourceAssemble` An array of integer flags of dimension `nAllFieldsMax` that identify whether the source terms in the different fields should be assembled. The default value is 0, which means no sources are present. You must set the values of the corresponding entries to 1 to ensure proper assembly of the source terms from the different fields.

| | |
|---|---|
| `iSourceAssemble(i_sol_phiS)` | If set to 1, assemble the source term for electric potential field. |
| `iSourceAssemble(i_sol_phiE)` | If set to 1, assemble the source term for electric potential in the electrolyte. |
| `iSourceAssemble(i_sol_Ce)` | If set to 1, assemble the source term for ion concentration in the electrolyte. |
| `iSourceAssemble(i_sol_solidC)` | If set to 1, assemble the source term for species concentration. |
| `iSourceAssemble(i_sol_porP)` | If set to 1, assemble the source term for pore pressure field. |
| `iSourceAssemble(i_sol_Temp)` | If set to 1, assemble the source term for temperature. |
| `iSourceAssemble(i_sol_Disp)` | If set to 1, assemble the source term for displacement field. |

## Variables That Can Be Updated

`statev` Current value of the solution-dependent variables.

## Variables Passed in for Information

`cmName` User-defined material name, left justified. Some internal material models are given names starting with the "ABQ_" character string. To avoid conflict, you should not use "ABQ_" as the leading string for `CMNAME`.

`kstep` Current step number.

`lFlags` An array containing the flags that define the current solution procedure and analysis type.

| | |
|---|---|
| `lFlags(1)` | Defines the procedure type. See *Results File* for the key used for each procedure. |
| `lFlags(2)=0` | Small-displacement analysis. |
| `lFlags(2)=1` | Large-displacement analysis (nonlinear geometric effects included in the step). See *General and Perturbation Procedures*. |

`kinc` Current increment number.

`time(1)` Current value of step time.

`time(2)` Current value of total time.

`dtime` Time increment.

`noel` User-assigned element number.

`npt` Integration point number.

`nDim` Number of spatial dimensions.

`ntens` Size of the stress or strain component array.

`ndi` Number of direct stress components at this point.

`nshr` Number of engineering shear stress components at this point.

`props` User-specified array of material constants associated with this user material.

`npros` User-defined number of material constants associated with this user material.

`predef` Array of interpolated values of predefined field variables at this point at the start of the increment, based on the values read in at the nodes.

`dpredef` Array of increments of predefined field variables.

`npredef` Number of predefined field variables.

`statevold` Array containing the user-defined solution-dependent state variables at the end of the previous increment.

`nstatev` Number of solution-dependent state variables that are associated with this material type (defined as described in *Allocating Space for Solution-Dependent State Variables*).

`Temp` Current temperature.

`dTemp` Incremental temperature.

`coords` An array containing the current coordinates of this integration point.

`solutionVar` Current value of the solution variables.

| | |
|---|---|
| `solutionVar(i_sol_phiS)` | Current value of electric potential. |
| `solutionVar(i_sol_phiE)` | Current value of electric potential in the electrolyte. |
| `solutionVar(i_sol_Ce)` | Current value of ionic concentration in the electrolyte. |

| | |
|---|---|
| `solutionVar(i_sol_solidC)` | Current value of species concentration. |
| `solutionVar(i_sol_porP)` | Current value of the pore pressure. |

`dsolutionVar` Incremental value of the solution variables.

| | |
|---|---|
| `dsolutionVar(i_sol_phiS)` | Incremental value of the electric potential. |
| `dsolutionVar(i_sol_phiE)` | Incremental value of the electric potential in the electrolyte. |
| `dsolutionVar(i_sol_Ce)` | Incremental value of the ionic concentration in the electrolyte. |
| `dsolutionVar(i_sol_solidC)` | Incremental value of the species concentration. |
| `dsolutionVar(i_sol_porP)` | Incremental value of the pore pressure. |

`nsolutionVar` Number of solution variables.

`i_array` Not used.

`niarray` Size of array `i_array`.

`r_array` Not used.

`nrarray` Size of array `r_array`.

`c_array` Not used.

`ncarray` Size of array `c_array`.

# UMASFL: User subroutine to specify prescribed mass flow rate conditions for a convection/diffusion heat transfer analysis.

**Products:** Abaqus/Standard

**References:**

- *Uncoupled Heat Transfer Analysis*
- **MASS FLOW RATE*
- *UTEMP, VUTEMP, UFIELD, UMASFL, and UPRESS*

## Overview

User subroutine *UMASFL*:

- can be used to prescribe the mass flow rate vector at the nodes of a model as a function of position and time;
- will be called whenever a current value of mass flow rate (per unit area) is needed for a node listed in a user-subroutine-defined mass flow rate definition (the node should belong to one or more convection/diffusion elements); and
- will overwrite any flow rate data specified for the associated mass flow rate definition outside the user subroutine.

## User Subroutine Interface

```
      SUBROUTINE UMASFL(FLOW,KFLOW,KSTEP,KINC,TIME,NODE,COORDS)
C
      INCLUDE 'ABA_PARAM.INC'
C
      DIMENSION FLOW(KFLOW), TIME(2), COORDS(3)
C


      user coding to define FLOW


      RETURN
      END
```

## Variables to Be Defined

`FLOW` Total value of the mass flow rate vector at this point. The number of components in this vector is `KFLOW`. If `KFLOW`=1, give the total mass flow rate through the cross-section (for one-dimensional elements). If `KFLOW`=2, give the *x*-component and *y*-component of the flow rate vector as `FLOW(1)` and `FLOW(2)`. If `KFLOW`=3, give the *x*-component, *y*-component, and *z*-component as `FLOW(1)`, `FLOW(2)`, and `FLOW(3)`.

## Variables Passed in for Information

`KFLOW` The number of components in the mass flow rate vector. *UMASFL* will be called only once per node.

`KSTEP` Step number.

| | |
|---|---|
| KINC | Increment number. |
| TIME(1) | Current value of step time. |
| TIME(2) | Current value of total time. |
| NODE | Node number. |
| COORDS | An array containing the coordinates of this node. |

# UMAT: User subroutine to define a material's mechanical behavior.

**Products:** Abaqus/Standard

**References:**

- *User-Defined Mechanical Material Behavior*
- *User-Defined Thermal Material Behavior*
- **USER MATERIAL*
- *SDVINI*
- *UMAT and UHYPER*

## Overview



**Warning:** The use of this subroutine generally requires considerable expertise. You are cautioned that the implementation of any realistic constitutive model requires extensive development and testing. Initial testing on a single-element model with prescribed traction loading is strongly recommended.

User subroutine *UMAT*:

- can be used to define the mechanical constitutive behavior of a material;
- will be called at all material calculation points of elements for which the material definition includes a user-defined material behavior;
- can be used with any procedure that includes mechanical behavior except the steady-state transport procedure;
- can use solution-dependent state variables;
- must update the stresses and solution-dependent state variables to their values at the end of the increment for which it is called;
- must provide the material Jacobian matrix, $\mathbf{C} = (1/J)\,\partial\Delta(J\boldsymbol{\sigma})/\partial\Delta\boldsymbol{\varepsilon}$, for the mechanical constitutive model;
- can be used in conjunction with user subroutine *USDFLD* to redefine any field variables before they are passed in; and
- is described further in *User-Defined Mechanical Material Behavior*.

## Storage of Stress and Strain Components

In the stress and strain arrays and in the matrices DDSDDE, DDSDDT, and DRPLDE, direct components are stored first, followed by shear components. There are NDI direct and NSHR engineering shear components. The order of the components is defined in *Conventions*. Since the number of active stress and strain components varies between element types, the routine must be coded to provide for all element types with which it will be used.

## Defining Local Orientations

If a local orientation (*Orientations*) is used at the same point as user subroutine UMAT, the stress and strain components will be in the local orientation; and, in the case of finite-strain analysis, the basis system in which stress and strain components are stored rotates with the material.

## Stability

You should ensure that the integration scheme coded in this routine is stable—no direct provision is made to include a stability limit in the time stepping scheme based on the calculations in UMAT.

## Convergence Rate

DDSDDE and—for coupled temperature-displacement and coupled thermal-electrical-structural analyses—DDSDDT, DRPLDE, and DRPLDT must be defined accurately if rapid convergence of the overall Newton scheme is to be achieved. In most cases the accuracy of this definition is the most important factor governing the convergence rate. Since nonsymmetric equation solution is as much as four times as expensive as the corresponding symmetric system, if the constitutive Jacobian (DDSDDE) is only slightly nonsymmetric (for example, a frictional material with a small friction angle), it might be less expensive computationally to use a symmetric approximation and accept a slower convergence rate.

An incorrect definition of the material Jacobian affects only the convergence rate; the results (if obtained) are unaffected.

## Viscoelastic Behavior in Frequency Domain

The constitutive Jacobian (DDSDDE) must provide both the stiffness (storage modulus) and damping (loss modulus) for modeling frequency domain viscoelastic behavior.

## Special Considerations for Various Element Types

There are several special considerations that need to be noted.

### Deformation Gradient

The deformation gradient is available for solid (continuum) elements, membranes, and finite-strain shells (S3/S3R, S4, S4R, SAXs, and SAXAs). It is not available for beams or small-strain shells. It is stored as a $3 \times 3$ matrix with component equivalence DFGRD0(I,J) $\Leftrightarrow \boldsymbol{F_{IJ}}$. For fully integrated first-order isoparametric elements (4-node quadrilaterals in two dimensions and 8-node hexahedra in three dimensions) the selectively reduced integration technique is used (also known as the $\overline{\boldsymbol{B}}$ technique). Thus, a modified deformation gradient

$$\overline{\mathbf{F}} = \mathbf{F}\left(\frac{\bar{J}}{J}\right)^{\frac{1}{n}}$$

is passed into user subroutine *UMAT*. For more details, see *Solid isoparametric quadrilaterals and hexahedra*.

The deformation gradient, which is passed to the user subroutine, is computed with respect to the initial configuration. If a local orientation is not specified, the components of the deformation gradient are expressed in the global coordinate system. If a local orientation is used, the components of the same deformation gradient are expressed in the local coordinate system; in the case of finite-strain analysis, the basis system rotates with the material.

### Beams and Shells That Calculate Transverse Shear Energy

If user subroutine *UMAT* is used to describe the material of beams or shells that calculate transverse shear energy, you must specify the transverse shear modulus as part of the material definition (see *Defining the Elastic Transverse Shear Modulus*) or the transverse shear stiffness as part of the beam or shell section definition (see *Shell Section Behavior* and *Choosing a Beam Element*) to define the transverse shear behavior. If transverse shear modulus is specified for a user-defined material that associates with shells, Abaqus/Standard calls user

subroutine *UMAT* during the preprocessing phase of the analysis (where KINC=0) to obtain the initial plane-stress constitutive Jacobian matrix and computes the transverse shear stiffness by matching the shear response for the case of the shell bending about one axis (see *Transverse shear stiffness in composite shells and offsets from the midsurface*).

### Open-Section Beam Elements

When user subroutine *UMAT* is used to describe the material response of beams with open sections (for example, an I-section), the torsional stiffness is obtained as

$$GJ = \frac{(K_{13}+K_{23})J}{2kA},$$

where $J$ is the torsional constant, $A$ is the section area, $k$ is a shear factor, and $K_{\alpha 3}$ is the user-specified transverse shear stiffness (see *Transverse Shear Stiffness Definition*).

### Elements with Hourglassing Modes

If this capability is used to describe the material of elements with hourglassing modes, you must define the hourglass stiffness factor for hourglass control based on the total stiffness approach as part of the element section definition. The hourglass stiffness factor is not required for enhanced hourglass control, but you can define a scaling factor for the stiffness associated with the drill degree of freedom (rotation about the surface normal). See *Section Controls* for information on specifying the stiffness factor.

### Pipe-Soil Interaction Elements

The constitutive behavior of the pipe-soil interaction elements (see *Pipe-Soil Interaction Elements*) is defined by the force per unit length caused by relative displacement between two edges of the element. The relative-displacements are available as "strains" (STRAN and DSTRAN). The corresponding forces per unit length must be defined in the STRESS array. The Jacobian matrix defines the variation of force per unit length with respect to relative displacement.

For two-dimensional elements two in-plane components of "stress" and "strain" exist (NTENS=NDI=2, and NSHR=0). For three-dimensional elements three components of "stress" and "strain" exist (NTENS=NDI=3, and NSHR=0).

## Large Volume Changes with Geometric Nonlinearity

If the material model allows large volume changes and geometric nonlinearity is considered, the exact definition of the consistent Jacobian should be used to ensure rapid convergence. These conditions are most commonly encountered when considering either large elastic strains or pressure-dependent plasticity. In the former case, total-form constitutive equations relating the Cauchy stress to the deformation gradient are commonly used; in the latter case, rate-form constitutive laws are generally used.

For total-form constitutive laws, the exact consistent Jacobian $\mathbf{C}$ is defined through the variation in Kirchhoff stress:

$$\delta\left(J\boldsymbol{\sigma}\right) = J\left(\mathbf{C} : \delta\mathbf{D} + \delta\mathbf{W}\cdot\boldsymbol{\sigma} - \boldsymbol{\sigma}\cdot\delta\mathbf{W}\right)$$

Here, $J$ is the determinant of the deformation gradient, $\boldsymbol{\sigma}$ is the Cauchy stress, $\delta\mathbf{D}$ is the virtual rate of deformation, and $\delta\mathbf{W}$ is the virtual spin tensor, defined as

$$\delta\mathbf{D} \overset{\text{def}}{=} \operatorname{sym}\left(\delta\mathbf{F}\cdot\mathbf{F}^{-1}\right)$$

and

$$\delta \mathbf{W} \stackrel{\text{def}}{=} \operatorname{asym}\left(\delta \mathbf{F} \cdot \mathbf{F}^{-1}\right).$$

For rate-form constitutive laws, the exact consistent Jacobian is given by

$$\mathbf{C} = \frac{1}{J}\frac{\partial \Delta(J\boldsymbol{\sigma})}{\partial \Delta \boldsymbol{\varepsilon}}.$$

## Use with Almost Incompressible or Fully Incompressible Elastic Materials

For user-defined almost incompressible or incompressible elastic materials, a few different options are available depending on whether hybrid or nonhybrid elements are used. For all cases the first option should be to use user subroutine *UHYPER* instead of user subroutine *UMAT* when it is possible to do so. In user subroutine *UMAT* incompressible materials can be modeled via a penalty method; that is, you ensure that a finite bulk modulus is used. The bulk modulus should be large enough to model incompressibility sufficiently but small enough to avoid loss of precision. As a general guideline, the bulk modulus should be about $10^4$–$10^6$ times the shear modulus. The tangent bulk modulus $K^t$ can be calculated from

$$K^t = \frac{1}{9}\sum_{I=1}^{3}\sum_{J=1}^{3}\text{DDSDDE}(I,J).$$

If a hybrid element is used with user subroutine *UMAT*, Abaqus/Standard, by default, replaces the pressure stress calculated from your definition of STRESS with that derived from the Lagrange multiplier and modifies the Jacobian appropriately (*Hybrid incompressible solid element formulation*). This approach is suitable for material models that use an incremental formulation (for example, metal plasticity) but is not consistent with a total formulation that is commonly used for hyperelastic materials. In the latter situation, the default formulation might lead to convergence problems. Such convergence problems might be observed, for example, when an almost incompressible nonlinear elastic user material is subjected to large deformations. Abaqus/Standard provides an alternate total formulation when user materials are used with hybrid elements (see *User-Defined Mechanical Material Behavior*). This formulation is consistent with the native almost incompressible formulation used by Abaqus for hyperelastic materials (*Hyperelastic material behavior*) and works better than the default formulation for such cases.

The total hybrid formulation assumes that the response of the material can be written as the sum of its deviatoric and volumetric parts and that these parts are decoupled from each other. In particular, the volumetric response is assumed to be defined in terms of a strain energy potential, $U\left(\widehat{J}\right)$, which is a function of an alternate variable, $\widehat{J}$ in place of the actual volume change $J$. The alternate variable is made available inside user subroutine *UMAT* by extending the STRESS array beyond NTENS, with the NTENS+1 entry providing read access to $\widehat{J}$. You must define the hydrostatic part of the stress tensor as $\hat{p} = -\frac{\partial U}{\partial \widehat{J}}$. The formulation also requires the additional derivatives $\widehat{K} = J\frac{\partial^2 U}{\partial \widehat{J}^2}$ and $\frac{\partial \widehat{K}}{\partial \widehat{J}} = J\frac{\partial^3 U}{\partial \widehat{J}^3}$. You must define these additional derivatives inside user subroutine *UMAT* as the NTENS+2 and the NTENS+3 entry, respectively, of the STRESS array. In addition, the bulk modulus of the material (contributes toward the material Jacobian matrix, DDSDDE) must be defined as $\widehat{K}$.

Abaqus/Standard also provides a fully incompressible user material formulation for use with hybrid elements to define a fully incompressible user material response. This formulation is consistent with the native formulation used by Abaqus for incompressible hyperelastic materials and assumes that the deviatoric stress can be derived from a strain energy potential function. You need define only the deviatoric stress and Jacobian to define a fully incompressible material response through user subroutine *UMAT*.

For incompressible pressure-sensitive materials the element choice is particularly important when using user subroutine *UMAT*. In particular, first-order wedge elements should be avoided. For these elements the $\overline{B}$ technique

is not used to alter the deformation gradient that is passed into user subroutine *UMAT*, which increases the risk of volumetric locking.

## Increments for Which Only the Jacobian Can Be Defined

Abaqus/Standard passes zero strain increments into user subroutine *UMAT* to start the first increment of all the steps and all increments of steps for which you have suppressed extrapolation (see *Defining an Analysis*). In this case you can define only the Jacobian (DDSDDE).

## Utility Routines

Several utility routines might help in coding user subroutine *UMAT*. Their functions include determining stress invariants for a stress tensor and calculating principal values and directions for stress or strain tensors. These utility routines are discussed in detail in *Obtaining Stress Invariants, Principal Stress/Strain Values and Directions, and Rotating Tensors*.

## User Subroutine Interface

```
      SUBROUTINE UMAT(STRESS,STATEV,DDSDDE,SSE,SPD,SCD,
     1 RPL,DDSDDT,DRPLDE,DRPLDT,
     2 STRAN,DSTRAN,TIME,DTIME,TEMP,DTEMP,PREDEF,DPRED,CMNAME,
     3 NDI,NSHR,NTENS,NSTATV,PROPS,NPROPS,COORDS,DROT,PNEWDT,
     4 CELENT,DFGRD0,DFGRD1,NOEL,NPT,LAYER,KSPT,JSTEP,KINC)
C
      INCLUDE 'ABA_PARAM.INC'
C
      CHARACTER*80 CMNAME
      DIMENSION STRESS(NTENS),STATEV(NSTATV),
     1 DDSDDE(NTENS,NTENS),DDSDDT(NTENS),DRPLDE(NTENS),
     2 STRAN(NTENS),DSTRAN(NTENS),TIME(2),PREDEF(1),DPRED(1),
     3 PROPS(NPROPS),COORDS(3),DROT(3,3),DFGRD0(3,3),DFGRD1(3,3),
     4 JSTEP(4)


      user coding to define DDSDDE, STRESS, STATEV, SSE, SPD, SCD
      and, if necessary, RPL, DDSDDT, DRPLDE, DRPLDT, PNEWDT


      RETURN
      END
```

## Variables to Be Defined

**In all situations**

`DDSDDE(NTENS,NTENS)` Jacobian matrix of the constitutive model, $\mathbf{C} = (1/J)\,\partial\Delta(J\boldsymbol{\sigma})/\partial\Delta\boldsymbol{\varepsilon}$, where $\Delta(J\boldsymbol{\sigma})$ are the Kirchhoff stress increments, $J$ is the determinant of the deformation gradient representing the volume change from the reference configuration, $\boldsymbol{\sigma}$ is the Cauchy stress, and $\Delta\boldsymbol{\varepsilon}$ are the strain increments. If the volume change is small, the Jacobian matrix can be approximated as $\partial\Delta\boldsymbol{\sigma}/\partial\Delta\boldsymbol{\varepsilon}$, where $\Delta\boldsymbol{\sigma}$ are the Cauchy stress increments. `DDSDDE(I,J)` defines the change in the `I`th stress component at the end of the time increment caused by an infinitesimal perturbation of the `J`th component of the strain increment array. Unless you invoke the unsymmetric equation solution capability for the user-defined material, Abaqus/Standard will use

**In all situations**

only the symmetric part of DDSDDE. The symmetric part of the matrix is calculated by taking one half the sum of the matrix and its transpose.

For viscoelastic behavior in the frequency domain, the Jacobian matrix must be dimensioned as DDSDDE(NTENS,NTENS,2). The stiffness contribution (storage modulus) must be provided in DDSDDE(NTENS,NTENS,1), while the damping contribution (loss modulus) must be provided in DDSDDE(NTENS,NTENS,2).

STRESS(NTENS) This array is passed in as the stress tensor at the beginning of the increment and must be updated in this routine to be the stress tensor at the end of the increment. If you specified initial stresses (*Initial Conditions*), this array will contain the initial stresses at the start of the analysis. The size of this array depends on the value of NTENS as defined below. In finite-strain problems the stress tensor has already been rotated to account for rigid body motion in the increment before *UMAT* is called, so that only the corotational part of the stress integration should be done in *UMAT*. The measure of stress used is "true" (Cauchy) stress.

If the *UMAT* utilizes a hybrid formulation that is total (as opposed to the default incremental behavior), the stress array is extended beyond NTENS. The first NTENS entries of the array contain the stresses, as described above. The additional quantities are as follows:

STRESS(NTENS+1)

Read only: $\widehat{J}$,

STRESS(NTENS+2)

Write only: $\widehat{K} = J\frac{\partial^2 U}{\partial \widehat{J}^2}$, and

STRESS(NTENS+3)

Write only: $\frac{\partial \widehat{K}}{\partial \widehat{J}} = J\frac{\partial^3 U}{\partial \widehat{J}^3}$, where $U$ is the volumetric part of the strain energy density potential.

STATEV(NSTATV) An array containing the solution-dependent state variables. These are passed in as the values at the beginning of the increment unless they are updated in user subroutines *USDFLD* or *UEXPAN*, in which case the updated values are passed in. In all cases STATEV must be returned as the values at the end of the increment. The size of the array is defined as described in *Allocating Space for Solution-Dependent State Variables*.

In finite-strain problems any vector-valued or tensor-valued state variables must be rotated to account for rigid body motion of the material, in addition to any update in the values associated with constitutive behavior. The rotation increment matrix, DROT, is provided for this purpose.

SSE, SPD, SCD Specific elastic strain energy, plastic dissipation, and "creep" dissipation, respectively. These are passed in as the values at the start of the increment and should be updated to the corresponding specific energy values at the end of the increment. They have no effect on the solution, except that they are used for energy output.

**Only in a fully coupled thermal-stress or a coupled thermal-electrical-structural analysis**

| | |
|---|---|
| RPL | Volumetric heat generation per unit time at the end of the increment caused by mechanical working of the material. |
| DDSDDT(NTENS) | Variation of the stress increments with respect to the temperature. |
| DRPLDE(NTENS) | Variation of RPL with respect to the strain increments. |
| DRPLDT | Variation of RPL with respect to the temperature. |

**Only in a geostatic stress procedure or a coupled pore fluid diffusion/stress analysis for pore pressure cohesive elements**

| | |
|---|---|
| RPL | RPL is used to indicate whether or not a cohesive element is open to the tangential flow of pore fluid. Set RPL equal to 0 if there is no tangential flow; otherwise, assign a nonzero value to RPL if an element is open. Once opened, a cohesive element will remain open to the fluid flow. |

## Variables That Can Be Updated

PNEWDT Ratio of suggested new time increment to the time increment being used (DTIME, see discussion later in this section). This variable allows you to provide input to the automatic time incrementation algorithms in Abaqus/Standard (if automatic time incrementation is chosen). For a quasi-static procedure the automatic time stepping that Abaqus/Standard uses, which is based on techniques for integrating standard creep laws (see *Quasi-Static Analysis*), cannot be controlled from within the *UMAT* subroutine.

PNEWDT is set to a large value before each call to *UMAT*.

If PNEWDT is redefined to be less than 1.0, Abaqus/Standard must abandon the time increment and attempt it again with a smaller time increment. The suggested new time increment provided to the automatic time integration algorithms is PNEWDT × DTIME, where the PNEWDT used is the minimum value for all calls to user subroutines that allow redefinition of PNEWDT for this iteration.

If PNEWDT is given a value that is greater than 1.0 for all calls to user subroutines for this iteration and the increment converges in this iteration, Abaqus/Standard might increase the time increment. The suggested new time increment provided to the automatic time integration algorithms is PNEWDT × DTIME, where the PNEWDT used is the minimum value for all calls to user subroutines for this iteration.

If automatic time incrementation is not selected in the analysis procedure, values of PNEWDT that are greater than 1.0 will be ignored and values of PNEWDT that are less than 1.0 will cause the job to terminate.

## Variables Passed in for Information

STRAN(NTENS) An array containing the total strains at the beginning of the increment. If thermal expansion is included in the same material definition, the strains passed into *UMAT* are the mechanical strains only (that is, the thermal strains computed based on the thermal expansion coefficient have been subtracted from the total strains). These strains are available for output as the “elastic” strains.

In finite-strain problems the strain components have been rotated to account for rigid body motion in the increment before *UMAT* is called and are approximations to logarithmic strain.

| | |
|---|---|
| DSTRAN(NTENS) | Array of strain increments. If thermal expansion is included in the same material definition, these are the mechanical strain increments (the total strain increments minus the thermal strain increments). |
| TIME(1) | Value of step time at the beginning of the current increment or frequency. |
| TIME(2) | Value of total time at the beginning of the current increment. |
| DTIME | Time increment. |
| TEMP | Temperature at the start of the increment. |
| DTEMP | Increment of temperature. |
| PREDEF | Array of interpolated values of predefined field variables at this point at the start of the increment, based on the values read in at the nodes. |
| DPRED | Array of increments of predefined field variables. |
| CMNAME | User-defined material name, left justified. Some internal material models are given names starting with the "ABQ_" character string. To avoid conflict, you should not use "ABQ_" as the leading string for CMNAME. |
| NDI | Number of direct stress components at this point. |
| NSHR | Number of engineering shear stress components at this point. |
| NTENS | Size of the stress or strain component array (NDI + NSHR). |
| NSTATV | Number of solution-dependent state variables that are associated with this material type (defined as described in *Allocating Space for Solution-Dependent State Variables*). |
| PROPS(NPROPS) | User-specified array of material constants associated with this user material. |
| NPROPS | User-defined number of material constants associated with this user material. |
| COORDS | An array containing the coordinates of this point. These are the current coordinates if geometric nonlinearity is accounted for during the step (see *Defining an Analysis*); otherwise, the array contains the original coordinates of the point. |
| DROT(3,3) | Rotation increment matrix. This matrix represents the increment of rigid body rotation of the basis system in which the components of stress (STRESS) and strain (STRAN) are stored. It is provided so that vector- or tensor-valued state variables can be rotated appropriately in this subroutine: stress and strain components are already rotated by this amount before *UMAT* is called. This matrix is passed in as a unit matrix for small-displacement analysis and for large-displacement analysis if the basis system for the material point rotates with the material (as in a shell element or when a local orientation is used). |
| CELENT | Characteristic element length, which is a typical length of a line across an element for a first-order element; it is half of the same typical length for a second-order element. For beams and trusses it is a characteristic length along the element axis. For membranes and shells it is a characteristic length in the reference surface. For axisymmetric elements it is a characteristic length in the $(r, z)$ plane only. For cohesive elements it is equal to the constitutive thickness. |
| DFGRD0(3,3) | Array containing the deformation gradient at the beginning of the increment. If a local orientation is defined at the material point, the deformation gradient components are expressed in the local coordinate system defined by the orientation at the beginning of the increment. For a discussion regarding the availability of the deformation gradient for various element types, see *Deformation Gradient*. |
| DFGRD1(3,3) | Array containing the deformation gradient at the end of the increment. If a local orientation is defined at the material point, the deformation gradient components are expressed in the local coordinate system defined by the orientation. This array is set to the identity matrix if nonlinear geometric effects are not included in the step definition associated |

| | |
|---|---|
| | with this increment. For a discussion regarding the availability of the deformation gradient for various element types, see *Deformation Gradient*. |
| NOEL | Element number. |
| NPT | Integration point number. |
| LAYER | Layer number (for composite shells and layered solids). |
| KSPT | Section point number within the current layer. |
| JSTEP(1) | Step number. |
| JSTEP(2) | Procedure type key (see *Results File*). |
| JSTEP(3) | 1 if NLGEOM=YES for the current step; 0 otherwise. |
| JSTEP(4) | 1 if current step is a linear perturbation procedure; 0 otherwise. |
| KINC | Increment number. |

## Example: Using More than One User-Defined Mechanical Material Model

To use more than one user-defined mechanical material model, the variable CMNAME can be tested for different material names inside user subroutine *UMAT* as illustrated below:

```
IF (CMNAME(1:4) .EQ. 'MAT1') THEN
   CALL UMAT_MAT1(argument_list)
ELSE IF(CMNAME(1:4) .EQ. 'MAT2') THEN
   CALL UMAT_MAT2(argument_list)
END IF
```

UMAT_MAT1 and UMAT_MAT2 are the actual user material subroutines containing the constitutive material models for each material MAT1 and MAT2, respectively. Subroutine *UMAT* merely acts as a directory here. The argument list might be the same as that used in subroutine *UMAT*.

## Example: Simple Linear Viscoelastic Material

As a simple example of the coding of user subroutine *UMAT*, consider the linear, viscoelastic model shown in *Figure 1*. Although this is not a very useful model for real materials, it serves to illustrate how to code the routine.

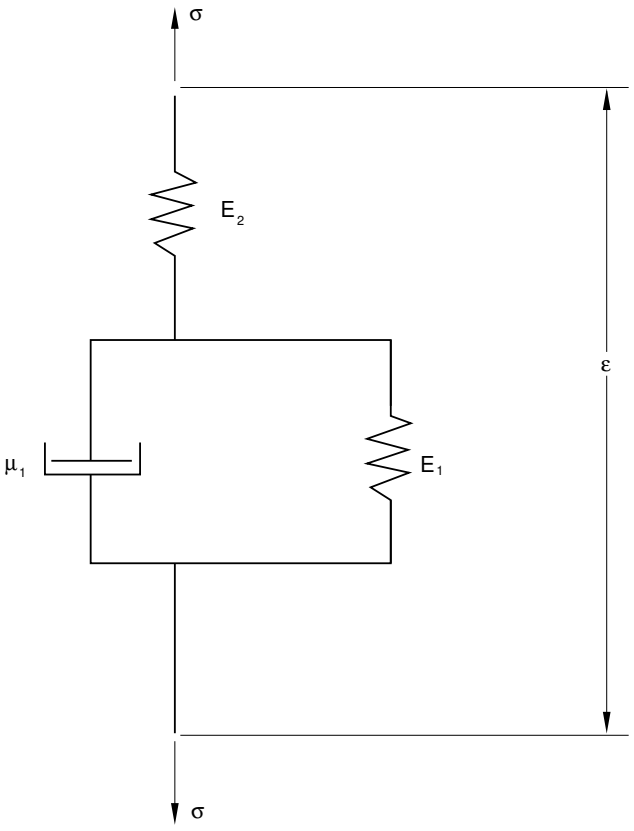



**Figure 1: Simple linear viscoelastic model.**

The behavior of the one-dimensional model shown in the figure is

$$\sigma + \frac{\mu_1}{(E_1+E_2)}\dot{\sigma} = \frac{\mu_1}{(1+E_1/E_2)}\dot{\varepsilon} + \frac{1}{(1/E_1+1/E_2)}\varepsilon,$$

where $\dot{\boldsymbol{\sigma}}$ and $\dot{\boldsymbol{\varepsilon}}$ are the time rates of change of stress and strain. This can be generalized for small straining of an isotropic solid as

$$\sigma_{xx} + \tilde{\nu}\dot{\sigma}_{xx} = \lambda\varepsilon_V + 2\mu\varepsilon_{xx} + \tilde{\lambda}\dot{\varepsilon}_V + 2\tilde{\mu}\dot{\varepsilon}_{xx}, \text{ etc.},$$

and

$$\sigma_{xy} + \tilde{\nu}\dot{\sigma}_{xy} = \mu\gamma_{xy} + \tilde{\mu}\dot{\gamma}_{xy}, \text{ etc.},$$

where

$$\varepsilon_V = \varepsilon_{xx} + \varepsilon_{yy} + \varepsilon_{zz},$$

and $\tilde{\nu}$, $\lambda$, $\mu$, $\tilde{\lambda}$, and $\tilde{\mu}$ are material constants ($\lambda$ and $\mu$ are the Lamé constants).

A simple, stable integration operator for this equation is the central difference operator:

$$\dot{f}_{t+\frac{1}{2}\Delta t} = \frac{\Delta f}{\Delta t},$$

$$f_{t+\frac{1}{2}\Delta t} = f_t + \frac{\Delta f}{2},$$

where $f$ is some function, $f_t$ is its value at the beginning of the increment, $\Delta f$ is the change in the function over the increment, and $\Delta t$ is the time increment.

Applying this to the rate constitutive equations above gives

$$\left(\frac{\Delta t}{2}+\tilde{\nu}\right)\Delta\sigma_{xx}=\left(\Delta t\frac{\lambda}{2}+\tilde{\lambda}\right)\Delta\varepsilon_V+(\Delta t\mu+2\tilde{\mu})\,\Delta\varepsilon_{xx}+\Delta t(\lambda\varepsilon_V+2\mu\varepsilon_{xx}-\sigma_{xx})_t,\ \text{etc.},$$

and

$$\left(\frac{\Delta t}{2}+\tilde{\nu}\right)\Delta\sigma_{xy}=\left(\Delta t\frac{\mu}{2}+\tilde{\mu}\right)\Delta\gamma_{xy}+\Delta t(\mu\gamma_{xy}-\sigma_{xy})_t,\ \text{etc.},$$

so that the Jacobian matrix has the terms

$$\frac{\partial\Delta\sigma_{xx}}{\partial\Delta\varepsilon_{xx}}=\frac{1}{\left(\frac{\Delta t}{2}+\tilde{\nu}\right)}\left[\Delta t\left(\frac{\lambda}{2}+\mu\right)+\tilde{\lambda}+2\tilde{\mu}\right],$$

$$\frac{\partial\Delta\sigma_{xx}}{\partial\Delta\varepsilon_{yy}}=\frac{1}{\left(\frac{\Delta t}{2}+\tilde{\nu}\right)}\left[\Delta t\frac{\lambda}{2}+\tilde{\lambda}\right],$$

and

$$\frac{\partial\Delta\sigma_{xy}}{\partial\Delta\gamma_{xy}}=\frac{1}{\left(\frac{\Delta t}{2}+\tilde{\nu}\right)}\left[\Delta t\frac{\mu}{2}+\tilde{\mu}\right].$$

The total change in specific energy in an increment for this material is

$$\left(\sigma_{ij}+\tfrac{1}{2}\Delta\sigma_{ij}\right)\Delta\varepsilon_{ij},$$

while the change in specific elastic strain energy is

$$\left(\varepsilon_{ij}+\tfrac{1}{2}\Delta\varepsilon_{ij}\right)D_{ijkl}\Delta\varepsilon_{kl},$$

where $D$ is the elasticity matrix:

$$\begin{bmatrix}\lambda+2\mu & \lambda & \lambda & 0 & 0 & 0\\ \lambda & \lambda+2\mu & \lambda & 0 & 0 & 0\\ \lambda & \lambda & \lambda+2\mu & 0 & 0 & 0\\ 0 & 0 & 0 & \mu & 0 & 0\\ 0 & 0 & 0 & 0 & \mu & 0\\ 0 & 0 & 0 & 0 & 0 & \mu\end{bmatrix}.$$

No state variables are needed for this material, so the allocation of space for them is not necessary. In a more realistic case a set of parallel models of this type might be used, and the stress components in each model might be stored as state variables.

For our simple case a user material definition can be used to read in the five constants in the order $\lambda$, $\mu$, $\tilde{\lambda}$, $\tilde{\mu}$, and $\tilde{\nu}$ so that

$$\begin{aligned}\texttt{PROPS(1)}&=\lambda,\\ \texttt{PROPS(2)}&=\mu,\\ \texttt{PROPS(3)}&=\tilde{\lambda},\\ \texttt{PROPS(4)}&=\tilde{\mu},\\ \texttt{PROPS(5)}&=\tilde{\nu}.\end{aligned}$$

The routine can then be coded as follows:

```
      SUBROUTINE UMAT(STRESS,STATEV,DDSDDE,SSE,SPD,SCD,
     1 RPL,DDSDDT,DRPLDE,DRPLDT,
     2 STRAN,DSTRAN,TIME,DTIME,TEMP,DTEMP,PREDEF,DPRED,CMNAME,
     3 NDI,NSHR,NTENS,NSTATV,PROPS,NPROPS,COORDS,DROT,PNEWDT,
     4 CELENT,DFGRD0,DFGRD1,NOEL,NPT,LAYER,KSPT,JSTEP,KINC)
C
      INCLUDE 'ABA_PARAM.INC'
C
      CHARACTER*80 CMNAME
      DIMENSION STRESS(NTENS),STATEV(NSTATV),
     1 DDSDDE(NTENS,NTENS),
     2 DDSDDT(NTENS),DRPLDE(NTENS),
     3 STRAN(NTENS),DSTRAN(NTENS),TIME(2),PREDEF(1),DPRED(1),
     4 PROPS(NPROPS),COORDS(3),DROT(3,3),DFGRD0(3,3),DFGRD1(3,3),
     5 JSTEP(4)
      DIMENSION DSTRES(6),D(3,3)
C
C  EVALUATE NEW STRESS TENSOR
C
      EV = 0.
      DEV = 0.
      DO K1=1,NDI
         EV = EV + STRAN(K1)
         DEV = DEV + DSTRAN(K1)
      END DO
C
      TERM1 = .5*DTIME + PROPS(5)
      TERM1I = 1./TERM1
      TERM2 = (.5*DTIME*PROPS(1)+PROPS(3))*TERM1I*DEV
      TERM3 = (DTIME*PROPS(2)+2.*PROPS(4))*TERM1I
C
      DO K1=1,NDI
         DSTRES(K1) = TERM2+TERM3*DSTRAN(K1)
     1     +DTIME*TERM1I*(PROPS(1)*EV
     2     +2.*PROPS(2)*STRAN(K1)-STRESS(K1))
         STRESS(K1) = STRESS(K1) + DSTRES(K1)
      END DO
C
      TERM2 = (.5*DTIME*PROPS(2) + PROPS(4))*TERM1I
      I1 = NDI
      DO K1=1,NSHR
         I1 = I1+1
         DSTRES(I1) = TERM2*DSTRAN(I1)+
     1     DTIME*TERM1I*(PROPS(2)*STRAN(I1)-STRESS(I1))
         STRESS(I1) = STRESS(I1)+DSTRES(I1)
      END DO
C
C  CREATE NEW JACOBIAN
C
      TERM2 = (DTIME*(.5*PROPS(1)+PROPS(2))+PROPS(3)+
     1  2.*PROPS(4))*TERM1I
      TERM3 = (.5*DTIME*PROPS(1)+PROPS(3))*TERM1I
      DO K1=1,NTENS
         DO K2=1,NTENS
            DDSDDE(K2,K1) = 0.
         END DO
      END DO
C
      DO K1=1,NDI
```

```
         DDSDDE(K1,K1) = TERM2
      END DO
C
      DO K1=2,NDI
         N2 = K1-1
         DO K2=1,N2
            DDSDDE(K2,K1) = TERM3
            DDSDDE(K1,K2) = TERM3
         END DO
      END DO
      TERM2 = (.5*DTIME*PROPS(2)+PROPS(4))*TERM1I
      I1 = NDI
      DO K1=1,NSHR
         I1 = I1+1
         DDSDDE(I1,I1) = TERM2
      END DO
C
C  TOTAL CHANGE IN SPECIFIC ENERGY
C
      TDE = 0.
      DO K1=1,NTENS
         TDE = TDE + (STRESS(K1)-.5*DSTRES(K1))*DSTRAN(K1)
      END DO
C
C  CHANGE IN SPECIFIC ELASTIC STRAIN ENERGY
C
      TERM1 = PROPS(1) + 2.*PROPS(2)
      DO  K1=1,NDI
         D(K1,K1) = TERM1
      END DO
      DO K1=2,NDI
         N2 = K1-1
         DO K2=1,N2
            D(K1,K2) = PROPS(1)
            D(K2,K1) = PROPS(1)
         END DO
      END DO
      DEE = 0.
      DO K1=1,NDI
         TERM1 = 0.
         TERM2 = 0.
         DO K2=1,NDI
            TERM1 = TERM1 + D(K1,K2)*STRAN(K2)
            TERM2 = TERM2 + D(K1,K2)*DSTRAN(K2)
         END DO
         DEE = DEE + (TERM1+.5*TERM2)*DSTRAN(K1)
      END DO
      I1 = NDI
      DO K1=1,NSHR
         I1 = I1+1
         DEE = DEE + PROPS(2)*(STRAN(I1)+.5*DSTRAN(I1))*DSTRAN(I1)
      END DO
      SSE = SSE + DEE
      SCD = SCD + TDE - DEE
      RETURN
      END
```

# UMATHT: User subroutine to define a material's thermal behavior.

**Products:** Abaqus/Standard

**References:**

- *User-Defined Thermal Material Behavior*
- **USER MATERIAL*
- *Freezing of a square solid: the two-dimensional Stefan problem*
- *UMATHT*

## Overview



**Warning:** The use of this subroutine generally requires considerable expertise. You are cautioned that the implementation of any realistic thermal model requires significant development and testing. Initial testing on models with few elements under a variety of boundary conditions is strongly recommended.

User subroutine *UMATHT*:

- can be used to define the thermal constitutive behavior of the material as well as internal heat generation during heat transfer processes;
- will be called at all material calculation points of elements for which the material definition includes a user-defined thermal material behavior;
- can be used with the procedures discussed in *About Heat Transfer Analysis Procedures*;
- can use solution-dependent state variables;
- must define the internal energy per unit mass and its variation with respect to temperature and to spatial gradients of temperature;
- must define the heat flux vector and its variation with respect to temperature and to gradients of temperature;
- must update the solution-dependent state variables to their values at the end of the increment;
- can be used in conjunction with user subroutine *USDFLD* to redefine any field variables before they are passed in; and
- is described further in *User-Defined Thermal Material Behavior*.

## Use of Subroutine UMATHT with Coupled Temperature-Displacement and Coupled Thermal-Electrical-Structural Elements

User subroutine *UMATHT* should be used only with reduced-integration or modified coupled temperature-displacement and coupled thermal-electrical-structural elements if the mechanical and thermal fields are not coupled through plastic dissipation. No such restriction exists with fully integrated coupled temperature-displacement and coupled thermal-electrical-structural elements.

## User Subroutine Interface

```
      SUBROUTINE UMATHT(U,DUDT,DUDG,FLUX,DFDT,DFDG,
     1 STATEV,TEMP,DTEMP,DTEMDX,TIME,DTIME,PREDEF,DPRED,
     2 CMNAME,NTGRD,NSTATV,PROPS,NPROPS,COORDS,PNEWDT,
     3 NOEL,NPT,LAYER,KSPT,KSTEP,KINC)
```

```
C
      INCLUDE 'ABA_PARAM.INC'
C
      CHARACTER*80 CMNAME
      DIMENSION DUDG(NTGRD),FLUX(NTGRD),DFDT(NTGRD),
     1 DFDG(NTGRD,NTGRD),STATEV(NSTATV),DTEMDX(NTGRD),
     2 TIME(2),PREDEF(1),DPRED(1),PROPS(NPROPS),COORDS(3)


      user coding to define  U,DUDT,DUDG,FLUX,DFDT,DFDG,
      and possibly update  STATEV,  PNEWDT


      RETURN
      END
```

## Variables to Be Defined

`U` Internal thermal energy per unit mass, $U$, at the end of increment. This variable is passed in as the value at the start of the increment and must be updated to its value at the end of the increment.

`DUDT` Variation of internal thermal energy per unit mass with respect to temperature, $\partial U/\partial\theta$, evaluated at the end of the increment.

`DUDG(NTGRD)` Variation of internal thermal energy per unit mass with respect to the spatial gradients of temperature, $\partial U/\partial(\partial\theta/\partial\mathbf{x})$, at the end of the increment. The size of this array depends on the value of `NTGRD` as defined below. This term is typically zero in classical heat transfer analysis.

`FLUX(NTGRD)` Heat flux vector, $\mathbf{f}$, at the end of the increment. This variable is passed in with the values at the beginning of the increment and must be updated to the values at the end of the increment.

`DFDT(NTGRD)` Variation of the heat flux vector with respect to temperature, $\partial\mathbf{f}/\partial\theta$, evaluated at the end of the increment.

`DFDG(NTGRD,NTGRD)` Variation of the heat flux vector with respect to the spatial gradients of temperature, $\partial\mathbf{f}/\partial(\partial\theta/\partial\mathbf{x})$, at the end of the increment. The size of this array depends on the value of `NTGRD` as defined below.

## Variables That Can Be Updated

`STATEV(NSTATV)` An array containing the solution-dependent state variables.

In an uncoupled heat transfer analysis `STATEV` is passed into *UMATHT* with the values of these variables at the beginning of the increment. However, any changes in `STATEV` made in user subroutine *USDFLD* will be included in the values passed into *UMATHT*, since *USDFLD* is called before *UMATHT*. In addition, if *UMATHT* is being used in a fully coupled temperature-displacement or coupled thermal-electrical-structural analysis and user subroutine *CREEP*, user subroutine *UEXPAN*, user subroutine *UMAT*, or user subroutine *UTRS* is used to define the mechanical behavior of the material, those routines are called before this routine; therefore, any updating of `STATEV` done in *CREEP*, *UEXPAN*, *UMAT*, or *UTRS* will be included in the values passed into *UMATHT*.

In all cases `STATEV` should be passed back from *UMATHT* as the values of the state variables at the end of the current increment.

PNEWDT — Ratio of suggested new time increment to the time increment being used (DTIME, see below). This variable allows you to provide input to the automatic time incrementation algorithms in Abaqus/Standard (if automatic time incrementation is chosen).

PNEWDT is set to a large value before each call to *UMATHT*.

If PNEWDT is redefined to be less than 1.0, Abaqus/Standard must abandon the time increment and attempt it again with a smaller time increment. The suggested new time increment provided to the automatic time integration algorithms is PNEWDT × DTIME, where the PNEWDT used is the minimum value for all calls to user subroutines that allow redefinition of PNEWDT for this iteration.

If PNEWDT is given a value that is greater than 1.0 for all calls to user subroutines for this iteration and the increment converges in this iteration, Abaqus/Standard may increase the time increment. The suggested new time increment provided to the automatic time integration algorithms is PNEWDT × DTIME, where the PNEWDT used is the minimum value for all calls to user subroutines for this iteration.

If automatic time incrementation is not selected in the analysis procedure, values of PNEWDT that are greater than 1.0 will be ignored and values of PNEWDT that are less than 1.0 will cause the job to terminate.

## Variables Passed in for Information

| | |
|---|---|
| TEMP | Temperature at the start of the increment. |
| DTEMP | Increment of temperature. |
| DTEMDX(NTGRD) | Current values of the spatial gradients of temperature, $\partial\theta/\partial\mathbf{x}$. |
| TIME(1) | Value of step time at the beginning of the current increment. |
| TIME(2) | Value of total time at the beginning of the current increment. |
| DTIME | Time increment. |
| PREDEF | Array of interpolated values of predefined field variables at this point at the start of the increment, based on the values read in at the nodes. |
| DPRED | Array of increments of predefined field variables. |
| CMNAME | User-defined material name, left justified. |
| NTGRD | Number of spatial gradients of temperature. |
| NSTATV | Number of solution-dependent state variables associated with this material type (defined as described in *Allocating Space for Solution-Dependent State Variables*). |
| PROPS(NPROPS) | User-specified array of material constants associated with this user material. |
| NPROPS | User-defined number of material constants associated with this user material. |
| COORDS | An array containing the coordinates of this point. These are the current coordinates in a fully coupled temperature-displacement or coupled thermal-electrical-structural analysis if geometric nonlinearity is accounted for during the step (see *Defining an Analysis*); otherwise, the array contains the original coordinates of the point. |
| NOEL | Element number. |
| NPT | Integration point number. |
| LAYER | Layer number (for composite shells and layered solids). |
| KSPT | Section point number within the current layer. |
| KSTEP | Step number. |
| KINC | Increment number. |

## Example: Using More than One User-Defined Thermal Material Model

To use more than one user-defined thermal material model, the variable CMNAME can be tested for different material names inside user subroutine *UMATHT*, as illustrated below:

```
IF (CMNAME(1:4) .EQ. 'MAT1') THEN
   CALL UMATHT_MAT1(argument_list)
ELSE IF(CMNAME(1:4) .EQ. 'MAT2') THEN
    CALL UMATHT_MAT2(argument_list)
END IF
```

UMATHT_MAT1 and UMATHT_MAT2 are the actual user material subroutines containing the constitutive material models for each material MAT1 and MAT2, respectively. Subroutine *UMATHT* merely acts as a directory here. The argument list can be the same as that used in subroutine *UMATHT*.

## Example: Uncoupled Heat Transfer

As a simple example of the coding of user subroutine *UMATHT*, consider uncoupled heat transfer analysis in a material. The equations for this case are developed here, and the corresponding *UMATHT* is given. This problem can also be solved by specifying thermal conductivity, specific heat, density, and internal heat generation directly.

First, the equations for an uncoupled heat transfer analysis are outlined.

The basic energy balance is

$$\int_V \rho \dot{U} dV = \int_S q\, dS + \int_V r\, dV,$$

where $V$ is the volume of solid material with surface area $S$, $\rho$ is the density of the material, $\dot{U}$ is the material time rate of the internal thermal energy, $q$ is the heat flux per unit area of the body flowing into the body, and $r$ is the heat supplied externally into the body per unit volume.

A heat flux vector $\mathbf{f}$ is defined such that

$$q = -\mathbf{f} \cdot \mathbf{n},$$

where $\mathbf{n}$ is the unit outward normal to the surface $S$. Introducing the above relation into the energy balance equation and using the divergence theorem, the following relation is obtained:

$$\int_V \rho \dot{U} dV = -\int_V \frac{\partial}{\partial \mathbf{x}} \cdot \mathbf{f} dV + \int_V r\, dV.$$

The corresponding weak form is given by

$$\int_V \delta\theta\, \rho \dot{U}\, dV - \int_V \delta\mathbf{g} \cdot \mathbf{f} dV = \int_V \delta\theta\, r\, dV + \int_S \delta\theta\, q\, dS,$$

where

$$\mathbf{g} = \frac{\partial \theta}{\partial \mathbf{x}}$$

is the temperature gradient and $\delta\theta$ is an arbitrary variational field satisfying the essential boundary conditions.

Introducing the backward difference integration algorithm:

$$\dot{U}_{t+\Delta t} = (U_{t+\Delta t} - U_t)\ (1/\Delta t),$$

the weak form of the energy balance equation becomes

$$\frac{1}{\Delta t} \int_v \delta\theta\, \rho\, (U_{t+\Delta t} - U_t)\, dV = \int_V \delta\mathbf{g} \cdot \mathbf{f} dV + \int_V \delta\theta\, r\, dV + \int_S \delta\theta\, q\, dS.$$

This nonlinear system is solved using Newton's method.

In the above equations the thermal constitutive behavior of the material is given by

$$U = U\left(\theta, t, \partial\theta/\partial\mathbf{x}, s^i, \ldots\right) U = U\left(\theta, \mathbf{g}, t, s^i, \ldots\right)$$

and

$$\mathbf{f} = \mathbf{f}\left(\theta, t, \partial\theta/\partial\mathbf{x}, s^i, \ldots\right), \mathbf{f} = \mathbf{f}\left(\theta, \mathbf{g}, t, s^i, \ldots\right),$$

where $s^i$ are state variables.

The Jacobian for Newton's method is given by (after dropping the subscripts $t + \Delta t$ on $U$)

$$\begin{aligned}\frac{1}{\Delta t}\int_V \delta\theta\rho\frac{\partial U}{\partial\theta}d\theta dV + \frac{1}{\Delta t}\int_V \delta\theta\rho\frac{\partial U}{\partial\mathbf{g}}\cdot d\mathbf{g}dV \\ -\int_V \delta\mathbf{g}\cdot\frac{\partial\mathbf{f}}{\partial\theta}d\theta dV - \int_V \delta\mathbf{g}\cdot\frac{\partial\mathbf{f}}{\partial\mathbf{g}}\cdot d\mathbf{g}\,dV \\ -\int_V \delta\theta\frac{\partial r}{\partial\theta}d\theta dV - \int_S \delta\theta\frac{\partial q}{\partial\theta}d\theta dS.\end{aligned}$$

The thermal constitutive behavior for this example is now defined. We assume a constant specific heat for the material. The heat conduction in the material is assumed to be governed by Fourier's law.

The internal thermal energy per unit mass is defined as

$$U = U(\theta),$$

with

$$\frac{\partial U}{\partial\theta} = c,$$

where $c$ is the specific heat of the material and

$$\frac{\partial U}{\partial\mathbf{g}} = 0.$$

Fourier's law for heat conduction is given as

$$\mathbf{f} = -\mathbf{k}\cdot\mathbf{g},$$

where $\mathbf{k}$ is the thermal conductivity matrix and $\mathbf{x}$ is position, so that

$$\frac{\partial\mathbf{f}}{\partial\mathbf{g}} = -\mathbf{k}$$

and

$$\frac{\partial\mathbf{f}}{\partial\theta} = -\frac{\partial\mathbf{k}}{\partial\theta}\cdot\mathbf{g}.$$

The assumption of conductivity without any temperature dependence implies that

$$\frac{\partial\mathbf{f}}{\partial\theta} = 0.$$

No state variables are needed for this material, so the allocation of space for them is not necessary.

A thermal user material definition can be used to read in the two constants for our simple case, namely the specific heat, $c$, and the coefficient of thermal conductivity, $k$, so that

$$\mathbf{PROPS(1)} = k,$$
$$\mathbf{PROPS(2)} = c.$$

```
      SUBROUTINE UMATHT(U,DUDT,DUDG,FLUX,DFDT,DFDG,
     1 STATEV,TEMP,DTEMP,DTEMDX,TIME,DTIME,PREDEF,DPRED,
```

```
     2 CMNAME,NTGRD,NSTATV,PROPS,NPROPS,COORDS,PNEWDT,
     3 NOEL,NPT,LAYER,KSPT,KSTEP,KINC)
C
      INCLUDE 'ABA_PARAM.INC'
C
      CHARACTER*80 CMNAME
      DIMENSION DUDG(NTGRD),FLUX(NTGRD),DFDT(NTGRD),
     1 DFDG(NTGRD,NTGRD),STATEV(NSTATV),DTEMDX(NTGRD),
     2 TIME(2),PREDEF(1),DPRED(1),PROPS(NPROPS),COORDS(3)
C
      COND = PROPS(1)
      SPECHT = PROPS(2)
C
      DUDT = SPECHT
      DU = DUDT*DTEMP
      U = U+DU
C
      DO I=1, NTGRD
         FLUX(I) = -COND*DTEMDX(I)
         DFDG(I,I) = -COND
      END DO
C
      RETURN
      END
```

# UMDFLUX: User subroutine for specifying moving or stationary nonuniform heat flux in a heat transfer analysis.

**Products:** Abaqus/Standard

**References:**

- **DFLUX*
- *DFLUX*

## Overview

User subroutine *UMDFLUX*:

- can be used to prescribe multiple moving or stationary heat fluxes as a function of position, time, temperature, etc.;
- can use and update element solution-dependent variables;
- will be called once for each element to prescribe heat moving sources between start points and end points inside the element; and
- applies heat fluxes as point heat sources, which will have the units of power ($JT^{-1}$); no volumetric integration is performed outside the user subroutine.

## User Subroutine Interface

```
      subroutine umdflux(
     *     jFlags, amplitude, noel, nElemNodes, iElemNodes,
     *     mcrd, coordNodes, uNodes, kstep, kinc, time, dt, jlTyp,
     *     temp, npredef, predef, nsvars, svars, sol, dsol,
     *     nIntp, volElm, volInt,
     *     nHeatEvents, flux, dfluxdT, csiStart, csiEnd)
C
      include 'aba_param.inc'
C
      dimension jFlags(2), iElemNodes(nElemNodes),
     *     coordNodes(mcrd,nElemNodes), uNodes(mcrd,nElemNodes),
     *     volInt(nIntp), time(2), dt(2),
     *     temp(2,nElemNodes), predef(2,npredef,nElemNodes),
     *     svars(nsvars,2), sol(nElemNodes), dsol(nElemNodes),
     *     flux(nHeatEvents), dfluxdT(nHeatEvents),
     *     csiStart(3,nHeatEvents), csiEnd(3,nHeatEvents)

      user coding to define nHeatEvents, flux, dfluxdT, csiStart, csiEnd,
      and possibly update dt, svars

      return
      end
```

## Variables to Be Defined

| | |
|---|---|
| `nHeatEvents` | The total number of moving or stationary heat events (heat fluxes) in this element. |
| `flux(nHeatEvents)` | Magnitudes of the fluxes (units of $JT^{-1}$) flowing into this element. |

`dfluxdT(nHeatEvents)` The rate of change of the fluxes with respect to temperatures.

`csiStart(3,nHeatEvents)` The isoparametric coordinates of the moving fluxes' starting points.

`csiEnd(3,nHeatEvents)` The isoparametric coordinates of the moving fluxes' ending points. If the flux is stationary, the coordinates of the end points are the same as the coordinates of the start points (`csiStart=csiEnd`).

## Variables That Can Be Updated

`svars(nsvars,1)` An array for defining new values of element solution-dependent variables at the end of the increment. These are passed in as the values at the beginning of the increment unless they are updated by other user subroutines, in which case the updated values are passed in. The size of the array is `nsvars` (see below), which can be defined as described in *Allocating Space for Element Solution-Dependent Variables*.

## Variables Passed in for Information

`jFlags` An array containing the flags that define the current solution procedure and analysis type.

| | |
|---|---|
| `jFlags(1)` | Defines the procedure type. See *Results File* for the key used for each procedure. |
| `jFlags(2)=0` | Small-displacement analysis. |
| `jFlags(2)=1` | Large-displacement analysis (nonlinear geometric effects included in the step; see *General and Perturbation Procedures*). |

`amplitude` The current value of the amplitude referenced for this flux (set to unity if no amplitude is referenced). This is the amplitude defined on the **DFLUX* definition.

`noel` User-assigned element number.

`nElemNodes` Number of element nodes.

`iElemNodes(nElemNodes)` An array containing user-assigned node numbers of the element.

`mcrd` The maximum of the user-defined maximum number of coordinates needed at any node point (*Defining the Maximum Number of Coordinates Needed at Any Nodal Point*) and the value of the largest active degree of freedom of the user element that is less than or equal to 3. For example, if you specify that the maximum number of coordinates is 1 and the active degrees of freedom of the user element are 2, 3, and 6, `mcrd` will be 3. If you specify that the maximum number of coordinates is 2 and the active degrees of freedom of the user element are 11 and 12, `mcrd` will be 2.

`coordNodes(mcrd,nElemNodes)` An array containing the coordinates of the nodes of the element.

`uNodes(mcrd,nElemNodes)` An array containing displacements at the element nodes. In a pure heat transfer analysis, the displacements are set equal to the coordinates.

`kstep` Current step number.

`kinc` Current increment number.

`time(1)` Current value of step time.

`time(2)` Current value of total time.

`dt(1)` Time increment.

| | |
|---|---|
| `dt(2)` | Ratio of suggested new time increment set in user subroutine *UMDFLUXSETUP* to the time increment currently being used (`dt(1)`, above). |
| `jlTyp` | Identifies the moving flux type for which this call to *UMDFLUX* is being made; only the concentrated heat flux type is supported (jlTyp=1). |
| `temp(2,nElemNodes)` | An array containing pairs of values of predefined temperature at the element nodes. The first value in a pair, `temp(1,nElemNodes)`, corresponds to the value at the end of the increment; and the second value, `temp(2,nElemNodes)`, corresponds to the increment in temperature. |
| `npredef` | Number of field variables. |
| `predef(2,npredef,nElemNodes)` | An array containing pairs of values of predefined field variables at the element nodes. The first value in a pair, `predef(1,npredef,nElemNodes)`, corresponds to the value at the end of the increment; and the second value, `predef(2,npredef,nElemNodes)`, corresponds to the increment in the field variables. |
| `nsvars` | Number of element solution-dependent variables. |
| `svars(nsvars,2)` | An array containing values of element solution-dependent variables at the beginning of the increment. |
| `sol(nElemNodes)` | An array of the solution variables (temperature in a heat transfer analysis) at the beginning of the increment. |
| `dsol(nElemNodes)` | An array of estimated values of the increment of the solution variables (temperature in a heat transfer analysis). The values are always set equal to zero. |
| `nIntp` | Total number of integration points for this element type. |
| `volElm` | The total volume of the element. |
| `volInt(nIntp)` | The integration point volume. |

# UMDFLUXSETUP: User subroutine to communicate with the toolpath-mesh intersection module and to specify the list of elements that are subjected to the moving flux load.

**Products:** Abaqus/Standard

**References:**

- **DFLUX*

## Overview

User subroutine *UMDFLUXSETUP*:

- can be used to communicate with the toolpath-mesh intersection module at the start of a step;
- can be used to communicate with the toolpath-mesh intersection at the start of an increment; and
- can be used to modify the list of elements that are subjected to the moving flux load at the start of an increment.

## User Subroutine Interface

```
      subroutine umdfluxsetup(
     *   lop,
     *   cMBFNuElsetName,
     *   jFluxActiveElems,
     *   nFluxActiveElems,
     *   kstep,
     *   kinc,
     *   time,
     *   dt)
C
      include 'aba_param.inc'
C
      dimension
     *   jFluxActiveElems(*),
     *   time(2), dt(2)

      character*80 cMBFNuElsetName

      user coding to interact with the toolpath-mesh intersection module and optionally
      modify the list of elements that are affected by the moving flux load (MBFNU)

      return
      end
```

## Variables to Be Defined

None.

## Variables That Can Be Updated

`jFluxActiveElems` An array of elements on which the moving heat source can be applied in this increment. It is used only when `lop=2`.

`nFluxActiveElems` Number of elements on which the moving heat source can be applied in this increment. It is used only when `lop=2`.

`dt(2)` Ratio of suggested new time increment to the time increment currently being used [`dt(1)`, see below]. This variable allows you to provide input to the automatic time incrementation algorithms in Abaqus/Standard if automatic time incrementation is chosen.

`dt(2)` is set to a large value before each call to *UMDFLUX*.

If `dt(2)` is redefined to be different than 1.0, Abaqus/Standard must abandon the time increment and attempt it again with a new time increment. The suggested new time increment provided to the automatic time integration algorithms is `dt(2) × dt(1)`, where the `dt(2)` used is the minimum value for all calls to user subroutine *UMDFLUXSETUP* that allows redefinition of `dt(2)` for this iteration.

If automatic time incrementation is not selected in the analysis procedure, values of `dt(2)` greater than 1.0 are ignored and values of `dt(2)` less than 1.0 cause the job to terminate.

## Variables Passed in for Information

`lop` `lop`=1 indicates that the user subroutine is called at the start of a step.

`lop`=2 indicates that the user subroutine is called at the start of an increment.

`cMBFNuElsetName` The name of the element set that is specified in the data line of the **DFLUX* option with the MBFNU flux type.

`kstep` Current step number.

`kinc` Current increment number.

`time(1)` Current value of step time.

`time(2)` Current value of total time.

`dt(1)` Time increment.

# UMESHMOTION: User subroutine to specify mesh motion constraints during adaptive meshing.

**Products:** Abaqus/Standard

**References:**

- *Defining ALE Adaptive Mesh Domains in Abaqus/Standard*
- **ADAPTIVE MESH*
- **ADAPTIVE MESH CONSTRAINT*

## Overview

User subroutine *UMESHMOTION*:

- is called at the end of any increment where adaptive meshing is performed;
- can be used to define the motion of nodes in an adaptive mesh constraint node set; and
- can call utility routines GETVRN, GETNODETOELEMCONN, and GETVRMAVGATNODE to access results data at the node.

## Accessing Node Point Data

You are provided with access to the values of the node point quantities at the end of the increment through the utility routine GETVRN described in *Obtaining Node Point Information*. You can also access values of material point quantities extrapolated to, and averaged, at nodes at the end of the increment through the utility routine GETVRMAVGATNODE described in *Obtaining Material Point Information Averaged at a Node*. GETVRMAVGATNODE requires the list of elements attached to the node, which is obtained by calling the utility routine GETNODETOELEMCONN described in *Obtaining Node to Element Connectivity*.

## User Subroutine Interface

```
      SUBROUTINE UMESHMOTION(UREF,ULOCAL,NODE,NNDOF,
     *    LNODETYPE,ALOCAL,NDIM,TIME,DTIME,PNEWDT,
     *    KSTEP,KINC,KMESHSWEEP,JMATYP,JGVBLOCK,LSMOOTH)
C
      INCLUDE 'ABA_PARAM.INC'
C
      DIMENSION ULOCAL(NDIM),JELEMLIST(*)
      DIMENSION ALOCAL(NDIM,*),TIME(2)
      DIMENSION JMATYP(*),JGVBLOCK(*)
C


      user coding to define ULOCAL
      and, optionally PNEWDT


      RETURN
      END
```

## Variables to Be Defined

ULOCAL Components of the mesh displacement or velocity of the adaptive mesh constraint node, described in the coordinate system ALOCAL. ULOCAL will be passed into the routine as values determined by the mesh smoothing algorithm. All components of the mesh displacement or velocity will be applied; i.e., you do not have the ability to select the directions in which the mesh displacement should be applied.

## Variables That Can Be Updated

PNEWDT Ratio of suggested new time increment to the time increment currently being used (DTIME, see below). This variable allows you to provide input to the automatic time incrementation algorithms in Abaqus/Standard (if automatic time incrementation is chosen).

PNEWDT is set to a large value before each call to *UMESHMOTION*.

The suggested new time increment provided to the automatic time integration algorithms is PNEWDT × DTIME, where the PNEWDT used is the minimum value for all calls to user subroutines that allow redefinition of PNEWDT for this increment.

If automatic time incrementation is not selected in the analysis procedure, values of PNEWDT greater than 1.0 will be ignored and values of PNEWDT less than 1.0 will cause the job to terminate.

LSMOOTH Flag specifying that surface smoothing be applied after application of the mesh motion constraint. Set LSMOOTH to 1 to enable surface smoothing. When this flag is set, the constraint defined in ULOCAL will be modified by the smoothing algorithm. In cases where ULOCAL describes mesh motion normal to a surface, the smoothing will have a minor impact on this normal component of mesh motion.

## Variables Passed in for Information

UREF The value of the user-specified displacement or velocity provided as part of the adaptive mesh constraint definition. This value is updated based on any amplitude definitions used with the adaptive mesh constraint or default ramp amplitude variations associated with the current step.

NODE Node number.

NNDOF Number of degrees of freedom at the node.

LNODETYPE Node type flag.

LNODETYPE=1 indicates that the node is on the interior of the adaptive mesh region.

LNODETYPE=2 indicates that the node is involved in a tied constraint.

LNODETYPE=3 indicates that the node is at the corner of the boundary of an adaptive mesh region.

LNODETYPE=4 indicates that the node lies on the edge of a boundary of an adaptive mesh region.

LNODETYPE=5 indicates that the node lies on a flat surface on a boundary of the adaptive mesh region.

LNODETYPE=6 indicates that the node participates in a constraint (other than a tied constraint) as a main node.

LNODETYPE=7 indicates that the node participates in a constraint (other than a tied constraint) as a secondary node.

| | |
|---|---|
| | LNODETYPE=10 indicates that a concentrated load is applied to the node. |
| ALOCAL | Local coordinate system aligned with the tangent to the adaptive mesh domain at the node. If the node is on the interior of the adaptive mesh domain, ALOCAL will be set to the identity matrix. In other cases the 1-direction is along an edge or in the plane of a flat surface. When NDIM=2, the 2-direction is normal to the surface. When NDIM=3, the 2-direction also lies in the plane of a flat surface or is arbitrary if the node is on an edge. When NDIM=3 the 3-direction is normal to the surface or is arbitrary if the node is on an edge. |
| NDIM | Number of coordinate dimensions. |
| TIME(1) | Current value of step time. |
| TIME(2) | Current value of total time. |
| DTIME | Time increment. |
| KSTEP | Step number. |
| KINC | Increment number. |
| KMESHSWEEP | Mesh sweep number. |
| JMATYP | Variable that must be passed into the GETVRMAVGATNODE utility routine to access local results at the node. |
| JGVBLOCK | Variable that must be passed into the GETVRN, GETNODETOELEMCONN, and GETVRMAVGATNODE utility routines to access local results at the node. |

# UMIXMODEFATIGUE: User subroutine to specify a user-defined mixed-mode form of the Paris law in a fatigue crack growth analysis.

**Products:** Abaqus/Standard

**References:**

- **FRACTURE CRITERION*
- *Linear Elastic Fatigue Crack Growth Analysis*
- *Crack Propagation Analysis*
- *Modeling Discontinuities as an Enriched Feature Using the Extended Finite Element Method*

## Overview

User subroutine *UMIXMODEFATIGUE*:

- can be used to specify a user-defined fatigue crack growth criterion;
- will be called at the end of a completed loading cycle;
- can be used for debonding in brittle material interfaces with surface-based contact definition;
- can be used for crack growth in bulk brittle materials with enriched elements; and
- can be used in stress intensity factor–based form, where the corresponding stress intensity factor component rather than the strain energy release rate component will be passed in for information.

## User Subroutine Interface

```
      SUBROUTINE UMIXMODEFATIGUE( DADN, GI_MAX, GII_MAX, GIII_MAX,
     1 GI_MIN, GII_MIN, GIII_MIN, TEMP, DTEMP, PREDEF, DPRED,
     2 NFIELD, NPROPS, PROPS, NSTATV, STATEV,
     3 NIARRAY, I_ARRAY, NRARRAY, R_ARRAY, NCARRAY, C_ARRAY, DA )

or

      SUBROUTINE UMIXMODEFATIGUE( DADN, KI_MAX, KII_MAX, KIII_MAX,
     1 KI_MIN, KII_MIN, KIII_MIN, TEMP, DTEMP, PREDEF, DPRED,
     2 NFIELD, NPROPS, PROPS, NSTATV, STATEV,
     3 NIARRAY, I_ARRAY, NRARRAY, R_ARRAY, NCARRAY, C_ARRAY, DA )

      INCLUDE 'ABA_PARAM.INC'

C
      DIMENSION PROPS(NPROPS), STATEV(NSTATV), PREDEF(NFIELD), DPRED(NFIELD),

     1 I_ARRAY(NIARRAY), R_ARRAY(NRARRAY), C_ARRAY(NCARRAY)

      user coding to define DADN

      RETURN
      END
```

## Variables to Be Defined

| | |
|---|---|
| DADN | Crack growth rate |

## Variables That Can Be Updated

`STATEV` An array containing the user-defined solution-dependent state variables at this point. This array will be passed in containing the values of these variables at the start of the increment unless the values are updated in user subroutine *USDFLD*. They can be updated in this subroutine to their values at the end of the increment. You define the size of this array by allocating space for it (see *Allocating Space for Solution-Dependent State Variables* for more information).

## Variables Passed in for Information

`GI_MAX` Maximum value of mode I energy release rate GI over the entire cycle. Only available when the default strain energy release rate–based form specified.

`GII_MAX` Maximum value of mode II energy release rate GII over the entire cycle. Only available when the default strain energy release rate–based form specified.

`GIII_MAX` Maximum value of mode III energy release rate GIII over the entire cycle. Only available when the default strain energy release rate–based form specified.

`GI_MIN` Minimum value of mode I energy release rate GI over the entire cycle. Only available when the default strain energy release rate–based form specified.

`GII_MIN` Minimum value of mode II energy release rate GII over the entire cycle. Only available when the default strain energy release rate–based form specified.

`GIII_MIN` Minimum value of mode III energy release rate GIII over the entire cycle. Only available when the default strain energy release rate–based form specified.

`KI_MAX` Maximum value of mode I stress intensity factor KI over the entire cycle. Only available when stress intensity factor–based form specified.

`KII_MAX` Maximum value of mode II stress intensity factor KII over the entire cycle. Only available when stress intensity factor–based form specified.

`KIII_MAX` Maximum value of mode III stress intensity factor KIII over the entire cycle. Only available when stress intensity factor–based form specified.

`KI_MIN` Minimum value of mode I stress intensity factor KI over the entire cycle. Only available when stress intensity factor–based form specified.

`KII_MIN` Minimum value of mode II stress intensity factor KII over the entire cycle. Only available when stress intensity factor–based form specified.

`KIII_MIN` Minimum value of mode III stress intensity factor KIII over the entire cycle. Only available when stress intensity factor–based form specified.

`TEMP` Temperature at the start of the increment.

`DTEMP` Increment of temperature during the time increment.

`PREDEF` An array containing the values of all user-specified predefined variables at this point at the start of the increment.

`DPRED` An array containing the increments of all predefined variables during the time increment.

`NFIELD` Number of user-specified predefined variables.

`PROPS(NPROPS)` User-specified array of material constants associated with this user-defined crack growth criterion.

`NPROPS` User-defined number of material constants associated with this user-defined crack growth criterion.

`NSTATV` Number of solution-dependent state variables associated with this material (specified when space is allocated for the array; see *Allocating Space for Solution-Dependent State Variables*).

`NIARRAY` Size of array I_ARRAY.

`I_ARRAY` Not used.

`NRARRAY` Size of array R_ARRAY. Currently equal to 1.

`R_ARRAY(1)` Accumulated crack length along initially bonded crack surfaces.

| | |
|---|---|
| NCARRAY | Size of array C_ARRAY. |
| C_ARRAY | Not used. |
| DA | Length scale for calculating R-curve effects. |

## Example: User-Defined Fatigue Crack Growth Criterion

As a simple example of the coding of user subroutine *UMIXMODEFATIGUE*, you can calculate the fatigue crack growth rate based on the mode mix ratio, as illustrated below:

```
SUBROUTINE UMIXMODEFATIGUE( DADN, GI_MAX, GII_MAX, GIII_MAX,
     1 GI_MIN, GII_MIN, GIII_MIN, TEMP, DTEMP, PREDEF, DPRED,
     2 NFIELD, NPROPS, PROPS, NSTATV, STATEV,
     3 NIARRAY, I_ARRAY, NRARRAY, R_ARRAY, NCARRAY, C_ARRAY, DA )

      INCLUDE 'ABA_PARAM.INC'

C
     DIMENSION PROPS(NPROPS), STATEV(NSTATV), PREDEF(NFIELD), DPRED(NFIELD),

     1 I_ARRAY(NIARRAY), R_ARRAY(NRARRAY), C_ARRAY(NCARRAY)

      A = PROPS(1)
      B = PROPS(2)
      C = PROPS(3)
      D = PROPS(4)
      G_SHEAR = GII_MAX+GIII_MAX
      G_TOTAL = GI_MAX+GII_MAX+GIII_MAX
      RMODEMIX = G_SHEAR/G_TOTAL
      IF( RMODEMIX .LE. 0.5 ) THEN
                DADN=A*G_TOTAL**B
      ELSE
                DADN=C*G_TOTAl**D
      ENDIF
C
      RETURN
      END
```

# UMOTION: User subroutine to specify motions during cavity radiation heat transfer analysis or steady-state transport analysis.

**Products:** Abaqus/Standard

**References:**

- *Cavity Radiation in Abaqus/Standard*
- *Steady-State Transport Analysis*
- **MOTION*
- **TRANSPORT VELOCITY*

## Overview

User subroutine *UMOTION*:

- can be used either to define the magnitude of the translational motion for degrees of freedom specified as a predefined field in a cavity radiation heat transfer analysis or to define the magnitude of the rotational velocity in a steady-state transport step; and
- will overwrite any motion or transport velocity magnitudes if they are defined directly (and possibly modified by including an amplitude reference) outside the user subroutine.

## User Subroutine Interface

```
      SUBROUTINE UMOTION(U,KSTEP,KINC,TIME,NODE,JDOF)
C
      INCLUDE 'ABA_PARAM.INC'
C
      DIMENSION TIME(2)
C


      user coding to define U


      RETURN
      END
```

## Variables to Be Defined

`U` Total value of the component of the translation due to prescribed motion for the degree of freedom specified by `JDOF`. `U` will be passed into the routine as the value defined by any magnitude and/or amplitude specification in the motion definition for the degree of freedom `JDOF`. The total value of the translation must be given in user subroutine *UMOTION*, regardless of the type of motion defined (displacement or velocity).

When used in conjunction with a steady-state transport analysis, `U` defines the magnitude of the rotational velocity. In such a case `JDOF` is passed in as `0`.

## Variables Passed in for Information

| | |
|---|---|
| KSTEP | Step number. |
| KINC | Increment number. |
| TIME(1) | Current value of step time. |
| TIME(2) | Current value of total time. |
| NODE | Node number. |
| JDOF | Degree of freedom. When used in a steady-state transport analysis, JDOF is passed in as 0. |

# UMULLINS: User subroutine to define damage variable for the Mullins effect material model.

**Products:** Abaqus/Standard

**References:**

- *Stress Softening in Rubberlike Materials*
- **MULLINS EFFECT*
- *Mullins effect and permanent set*

## Overview

User subroutine *UMULLINS*:

- can be used to define the damage variable for the Mullins effect material model, including the use of the Mullins effect approach to model energy dissipation in elastomeric foams;
- will be called at all material calculation points of elements for which the material definition contains a user-defined Mullins effect; and
- should be used when you do not want to use the Ogden and Roxburgh form of the damage variable, $\eta$, that is used by Abaqus/Standard.

## User Subroutine Interface

```
      SUBROUTINE UMULLINS(NUMPROPS,PROPS,UMAXNEW,UMAXOLD,SEDDEV,
     1 ETA,DETADW,DMGDISSOLD,DMGDISSNEW,SENERNEW,NUMSTATEV,STATEV,
     2 TEMP,DTEMP,NUMFIELDV,FIELDV,FIELDVINC,CMNAME,LINPER)
C
      INCLUDE 'ABA_PARAM.INC'
C
      CHARACTER*80 CMNAME
      DIMENSION PROPS(*),STATEV(*),FIELDV(*),FIELDVINC(*)

      user coding to define ETA, DETADW,
      and, optionally, DMGDISSNEW, SENERNEW, STATEV


      RETURN
      END
```

## Variables to Be Defined

`ETA` The damage variable, $\eta$.

`DETADW` The derivative of the damage variable with respect to the elastic strain energy density of the undamaged material, $\frac{d\eta}{d\tilde{U}}$. This quantity is needed for the Jacobian of the overall system of equations and needs to be defined accurately to ensure good convergence characteristics.

## Variables That Can Be Updated

DMGDISSNEW — The energy dissipation density at the end of the increment. This quantity can be defined either in total form or in an incremental manner using the old value of the damage dissipation DMGDISSOLD and the increment in damage dissipation. This quantity is used for output purposes only.

SENERNEW — The recoverable strain energy density at the end of the increment. This quantity is used for output purposes only.

STATEV — Array containing the user-defined solution-dependent state variables at this point. These are supplied as values at the start of the increment or as values updated by other user subroutines (see *Allocating Space for Solution-Dependent State Variables*) and must be returned as values at the end of the increment.

## Variables Passed in for Information

UMAXNEW — The value, at the end of the increment, of the maximum primary strain energy density over its entire deformation history.

UMAXOLD — The value, at the beginning of the increment, of the maximum primary strain energy density over its entire deformation history.

SEDDEV — The value, at the end of the increment, of the deviatoric primary strain energy density when the primary material behavior is hyperelastic. The value, at the end of the increment, of the total primary strain energy density when the primary material behavior is hyperfoam.

DMGDISSOLD — The value of energy dissipated at the beginning of the increment.

CMNAME — User-specified material name, left justified.

NUMSTATEV — Number of solution-dependent state variables associated with this material (defined as described in *Allocating Space for Solution-Dependent State Variables*).

NUMPROPS — Number of material properties entered for this user-defined hyperelastic material.

PROPS — Array of material properties entered for this user-defined hyperelastic material.

TEMP — Temperature at the start of the increment.

DTEMP — Increment of temperature.

NUMFIELDV — Number of field variables.

FIELDV — Array of interpolated values of predefined field variables at this material point at the beginning of the increment based on the values read in at the nodes (initial values at the beginning of the analysis and current values during the analysis).

FIELDVINC — Array of increments of predefined field variables at this material point for this increment; this includes any values updated by user subroutine *USDFLD*.

LINPER — Linear perturbation flag. LINPER=1 if the step is a linear perturbation step. LINPER=0 if the step is a general step.

# UPOREP: User subroutine to define initial fluid pore pressure.

**Products:** Abaqus/Standard

**References:**

- *Initial Conditions*
- *Coupled Pore Fluid Diffusion and Stress Analysis*
- **INITIAL CONDITIONS*

## Overview

User subroutine *UPOREP*:

- allows for the specification of the initial pore pressure values of a porous medium;
- can be used to define initial pore pressure values as functions of nodal coordinates and/or node numbers; and
- will be called to define initial fluid pore pressure values at all nodes of a coupled pore fluid diffusion and stress analysis whenever user-defined initial pore pressure conditions are specified.

## User Subroutine Interface

```
      SUBROUTINE UPOREP(UW0,COORDS,NODE)
C
      INCLUDE 'ABA_PARAM.INC'
C
      DIMENSION COORDS(3)
C


      user coding to define UW0


      RETURN
      END
```

## Variables to Be Defined

| | |
|---|---|
| UW0 | Initial fluid pore pressure. |

## Variables Passed in for Information

| | |
|---|---|
| COORDS | An array containing the current coordinates of this node. |
| NODE | Node number. |

# UPRESS: User subroutine to specify prescribed equivalent pressure stress conditions.

**Products:** Abaqus/Standard

**References:**

- *Mass Diffusion Analysis*
- **PRESSURE STRESS*
- *UTEMP, VUTEMP, UFIELD, UMASFL, and UPRESS*

## Overview

User subroutine *UPRESS*:

- allows you to prescribe equivalent pressure stress values at the nodes of a model;
- will be called in a mass diffusion analysis whenever a current value of equivalent pressure stress is needed for a node that has user-defined pressure stress conditions;
- can be used to modify any pressure stresses read in from a results file; and
- ignores any equivalent pressure stresses provided for the associated pressure stress definition outside the user subroutine.

## User Subroutine Interface

```
      SUBROUTINE UPRESS(PRESS,KSTEP,KINC,TIME,NODE,COORDS)
C
      INCLUDE 'ABA_PARAM.INC'
C
      DIMENSION TIME(2), COORDS(3)
C


      user coding to define PRESS


      RETURN
      END
```

## Variables to Be Defined

`PRESS` Total value of the equivalent pressure stress at the node.

You may have also requested equivalent pressure stress to be set in one of two other ways: from a previously generated results file or via direct data input. When `PRESS` is passed into user subroutine *UPRESS*, it will contain equivalent pressure stresses obtained from the results file only. You can modify these values within this routine. Any values given as direct data input will be ignored.

## Variables Passed in for Information

`KSTEP` Step number.

| | |
|---|---|
| KINC | Increment number. |
| TIME(1) | Current value of step time. |
| TIME(2) | Current value of total time. |
| NODE | Node number. |
| COORDS | An array containing the coordinates of this node. |

# UPSD: User subroutine to define the frequency dependence for random response loading.

**Products:** Abaqus/Standard

**References:**

- *Random Response Analysis*
- **RANDOM RESPONSE*
- **PSD-DEFINITION*
- *Random response to jet noise excitation*

## Overview

User subroutine *UPSD*:

- will be called once for each frequency at which calculations will be made during a random response analysis if the frequency function is defined in a user subroutine;
- is used to define complicated frequency dependencies for the cross-spectral density matrix of the random loading; and
- ignores any data given for the associated frequency function outside the user subroutine.

## User Subroutine Interface

```
      SUBROUTINE UPSD(PSD,PSDR,PSDI,FREQ,KSTEP)
C
      INCLUDE 'ABA_PARAM.INC'
C
      CHARACTER*80 PSD


      user coding to define PSDR and PSDI


      RETURN
      END
```

## Variables to Be Defined

| | |
|---|---|
| PSDR | Real part of the frequency function at this frequency. |
| PSDI | Imaginary part of the frequency function at this frequency. |

## Variables Passed in for Information

| | |
|---|---|
| PSD | User-specified name for this frequency function definition, left justified. |
| FREQ | Frequency, in radians per time. |
| KSTEP | Step number. |

# URDFIL: User subroutine to read the results file.

**Products:** Abaqus/Standard

### References:

- *Results File*
- *Accessing the Results File Information*
- *Utility Routines for Accessing the Results File*

## Overview

User subroutine *URDFIL*:

- can be used to access the results file during an analysis;
- is called at the end of any increment in which new information is written to the results file;
- must call the utility routine DBFILE to read records from the results file (see *Utility Routines for Accessing the Results File*);
- can call the utility routine POSFIL to read from the results file starting at a specified step and increment as opposed to the beginning of the file, which would otherwise be done (see *Utility Routines for Accessing the Results File*);
- can force an analysis to terminate upon completion of a call by means of the variable LSTOP;
- allows the last increment written to the results file to be overwritten by means of the variable LOVRWRT; and
- allows access to the complete results file in a restarted job if the new results file is being appended to the old results file (see the description of the execution option **fil** in *Abaqus/Standard and Abaqus/Explicit Execution*).

## User Subroutine Interface

```
      SUBROUTINE URDFIL(LSTOP,LOVRWRT,KSTEP,KINC,DTIME,TIME)
C
      INCLUDE 'ABA_PARAM.INC'
C
      DIMENSION ARRAY(513),JRRAY(NPRECD,513),TIME(2)
      EQUIVALENCE (ARRAY(1),JRRAY(1,1))

      user coding to read the results file

      RETURN
      END
```

## Variables to Be Defined

**In all cases**

LSTOP — Flag to indicate whether an analysis should continue. The analysis will be terminated if LSTOP is set to 1. Otherwise, the analysis will continue.

LOVRWRT — Flag to indicate that the information written to the results file for the increment can be overwritten. If LOVRWRT is set to 1, information for the current increment will be overwritten by information written to the results file in a subsequent increment unless the current increment is the final increment written to the results file. The purpose of this flag is to reduce the size of

**In all cases**

the results file by allowing information for an increment to be overwritten by information for a subsequent increment.

`DTIME` Time increment. This variable allows you to provide input to the automatic time incrementation algorithms in Abaqus (if automatic time incrementation is chosen). It is passed in as the value of the next time increment to be taken and can be updated to increase or reduce the time increment. If automatic time incrementation is not selected in the analysis procedure, updated values of `DTIME` are ignored.

**Only if utility routine `POSFIL` is called**

`NSTEP` Desired step at which file reading will begin via utility routine `DBFILE`. If `NSTEP` is set to 0, the first available step will be read.

`NINC` Desired increment at which file reading will begin via utility routine `DBFILE`. If `NINC` is set to 0, the first available increment of the specified step will be read.

## Variables Passed in for Information

`KSTEP` Step number.

`KINC` Increment number.

`TIME(1)` Value of the step time at the end of the increment.

`TIME(2)` Value of the total time at the end of the increment.

## Example: Terminating an Analysis upon Exceeding a Mises Stress Limit

The example below reads the values of Mises stress for the current increment from record 12 in the results file and terminates the analysis if any of the values of Mises stress written to the results file exceed $2.09 \times 10^8$. Here, `POSFIL` is used to position you to read from the current increment.

```
      SUBROUTINE URDFIL(LSTOP,LOVRWRT,KSTEP,KINC,DTIME,TIME)
C
      INCLUDE 'ABA_PARAM.INC'
C
      DIMENSION ARRAY(513),JRRAY(NPRECD,513),TIME(2)
      EQUIVALENCE (ARRAY(1),JRRAY(1,1))
      PARAMETER(TOL=2.09D8)
C
C FIND CURRENT INCREMENT.
C
      CALL POSFIL(KSTEP,KINC,ARRAY,JRCD)
      DO K1=1,999999
         CALL DBFILE(0,ARRAY,JRCD)
         IF (JRCD .NE. 0) GO TO 110
         KEY=JRRAY(1,2)
C
C RECORD 12 CONTAINS VALUES FOR SINV
C
         IF (KEY.EQ.12) THEN
            IF (ARRAY(3).GT.TOL) THEN
               LSTOP=1
               GO TO 110
            END IF
         END IF
```

```
      END DO
 110  CONTINUE
C
      RETURN
      END
```

## Example: Terminating an Analysis When the Maximum Mises Stress Value Stops Increasing

This example demonstrates the use of *URDFIL* and POSFIL to stop an analysis when the maximum value of Mises stress in the model does not increase from one increment in the results file to the next. A data statement is used to save the maximum Mises stress value from the last increment. LOVRWRT is also used in this case to overwrite an increment in the results file once it has been read in *URDFIL*.

The subroutine shown below must be modified to define the maximum Mises stress in the data statement each time a job is restarted. This can be avoided by removing the LOVRWRT=1 statement and recoding the routine to read both the previous and the current increment to check that the Mises stress increases from one increment to the next (in this case you must correctly handle the first increment written to the results file as there will be no previous increment). The results file must also be properly appended on restart if you wish to compare the values of Mises stress between the first increment of a restart and the final increment of the job being restarted. This approach has the disadvantage that the results file may become quite large, as no information in the file will be overwritten.

```
      SUBROUTINE URDFIL(LSTOP,LOVRWRT,KSTEP,KINC,DTIME,TIME)
C
      INCLUDE 'ABA_PARAM.INC'
C
      DIMENSION ARRAY(513),JRRAY(NPRECD,513),TIME(2)
      EQUIVALENCE (ARRAY(1),JRRAY(1,1))
C
C INITIALIZE THE OLD MAXIMUM. FOR A JOB THAT IS BEING RESTARTED
C THIS VALUE SHOULD BE SET TO THE MAXIMUM MISES STRESS IN THE
C ORIGINAL ANALYSIS.
C
      DATA OLDMAX/-1.D0/
C
      CURRMAX = 0.D0
C
C FIND CURRENT INCREMENT.
C
      CALL POSFIL(KSTEP,KINC,ARRAY,JRCD)
C
C SEARCH FOR THE HIGHEST VALUE OF MISES STRESS
C AND STORE THIS IN CURRMAX
C
      DO K1=1,999999
         CALL DBFILE(0,ARRAY,JRCD)
         IF (JRCD.NE.0) GO TO 110
         KEY=JRRAY(1,2)
         IF (KEY.EQ.12) THEN
            IF (ARRAY(3).GT.CURRMAX) CURRMAX=ARRAY(3)
         END IF
      END DO
 110  CONTINUE
C
C COMPLETED READING OF CURRENT INCREMENT. NOW CHECK TO
C SEE IF VALUE OF MISES STRESS HAS INCREASED SINCE
C LAST INCREMENT
C
```

```
      IF (CURRMAX.LE.OLDMAX) LSTOP=1
      OLDMAX=CURRMAX
      LOVRWRT=1
C
      RETURN
      END
```

# USDFLD: User subroutine to redefine field variables at a material point.

**Products:** Abaqus/Standard

**References:**

- *Obtaining Material Point Information in an Abaqus/Standard Analysis*
- *Material Data Definition*
- **USER DEFINED FIELD*
- *Damage and failure of a laminated composite plate*
- *USDFLD*
- *Using Abaqus/Standard Output Variable Identifiers*

## Overview

User subroutine *USDFLD*:

- allows you to define field variables at a material point as functions of time or of any of the available material point quantities listed in the Output Variable Identifiers table (*Using Abaqus/Standard Output Variable Identifiers*) except the user-defined output variables UVARM and UVARM*n*;
- can be used to introduce solution-dependent material properties since such properties can easily be defined as functions of field variables;
- will be called at all material points of elements for which the material definition includes user-defined field variables;
- must call utility routine GETVRM to access material point data;
- can use and update state variables; and
- can be used in conjunction with user subroutine UFIELD to prescribe predefined field variables.

## Explicit Solution Dependence

Since this routine provides access to material point quantities only at the start of the increment, the solution dependence introduced in this way is explicit: the material properties for a given increment are not influenced by the results obtained during the increment. Hence, the accuracy of the results depends on the size of the time increment. Therefore, you can control the time increment in this routine by means of the variable PNEWDT.

## Defining Field Variables

Before user subroutine *USDFLD* is called, the values of the field variables at the material point are calculated by interpolation from the values defined at the nodes. Any changes to the field variables in the user subroutine are local to the material point: the nodal field variables retain the values defined as initial conditions, predefined field variables, or in user subroutine UFIELD. The values of the field variables defined in this routine are used to calculate values of material properties that are defined to depend on field variables and are passed into other user subroutines that are called at the material point, such as the following:

- *CREEP*
- *HETVAL*
- *UEXPAN*
- *UHARD*

- *UHYPEL*
- *UMAT*
- *UMATHT*
- *UTRS*

Output of the user-defined field variables at the material points can be obtained with the element integration point output variable FV (see *Element Integration Point Variables*).

## Accessing Material Point Data

You are provided with access to the values of the material point quantities at the start of the increment (or in the base state in a linear perturbation step) through the utility routine GETVRM described in *Obtaining Material Point Information in an Abaqus/Standard Analysis*. The values of the material point quantities are obtained by calling GETVRM with the appropriate output variable keys. The values of the material point data are recovered in the arrays ARRAY, JARRAY, and FLGRAY for floating point, integer, and character data, respectively. You may not get values of some material point quantities that have not been defined at the start of the increment; e.g., ER.

## State Variables

Since the redefinition of field variables in *USDFLD* is local to the current increment (field variables are restored to the values interpolated from the nodal values at the start of each increment), any history dependence required to update material properties by using this subroutine must be introduced with user-defined state variables.

The state variables can be updated in *USDFLD* and then passed into other user subroutines that can be called at this material point, such as those listed above. You specify the number of such state variables, as shown in the example at the end of this section (see also *Allocating Space for Solution-Dependent State Variables*).

## User Subroutine Interface

```
      SUBROUTINE USDFLD(FIELD,STATEV,PNEWDT,DIRECT,T,CELENT,
     1 TIME,DTIME,CMNAME,ORNAME,NFIELD,NSTATV,NOEL,NPT,LAYER,
     2 KSPT,KSTEP,KINC,NDI,NSHR,COORD,JMAC,JMATYP,MATLAYO,LACCFLA)
C
      INCLUDE 'ABA_PARAM.INC'
C
      CHARACTER*80 CMNAME,ORNAME
      CHARACTER*3  FLGRAY(15)
      DIMENSION FIELD(NFIELD),STATEV(NSTATV),DIRECT(3,3),
     1 T(3,3),TIME(2)
      DIMENSION ARRAY(15),JARRAY(15),JMAC(*),JMATYP(*),COORD(*)


      user coding to define FIELD and, if necessary, STATEV and PNEWDT


      RETURN
      END
```

## Variables to Be Defined

FIELD(NFIELD) An array containing the field variables at the current material point. These are passed in with the values interpolated from the nodes at the end of the current increment, as specified with initial condition definitions, predefined field variable definitions, or user subroutine UFIELD. The interpolation is performed using the same scheme used to interpolate temperatures: an

average value is used for linear elements; an approximate linear variation is used for quadratic elements (also see *Solid (Continuum) Elements*). The updated values are used to calculate the values of material properties that are defined to depend on field variables and are passed into other user subroutines (*CREEP*, *HETVAL*, *UEXPAN*, *UHARD*, *UHYPEL*, *UMAT*, *UMATHT*, and *UTRS*) that are called at this material point.

## Variables That Can Be Updated

`STATEV(NSTATV)` An array containing the solution-dependent state variables. These are passed in as the values at the beginning of the increment. In all cases `STATEV` can be updated in this subroutine, and the updated values are passed into other user subroutines (*CREEP*, *HETVAL*, *UEXPAN*, *UMAT*, *UMATHT*, and *UTRS*) that are called at this material point. The number of state variables associated with this material point is defined as described in *Allocating Space for Solution-Dependent State Variables*.

`PNEWDT` Ratio of suggested new time increment to the time increment being used (`DTIME`, see below). This variable allows you to provide input to the automatic time incrementation algorithms in Abaqus/Standard (if automatic time incrementation is chosen).

`PNEWDT` is set to a large value before each call to *USDFLD*.

If `PNEWDT` is redefined to be less than 1.0, Abaqus/Standard must abandon the time increment and attempt it again with a smaller time increment. The suggested new time increment provided to the automatic time integration algorithms is `PNEWDT` × `DTIME`, where the `PNEWDT` used is the minimum value for all calls to user subroutines that allow redefinition of `PNEWDT` for this iteration.

If `PNEWDT` is given a value that is greater than 1.0 for all calls to user subroutines for this iteration and the increment converges in this iteration, Abaqus/Standard may increase the time increment. The suggested new time increment provided to the automatic time integration algorithms is `PNEWDT` × `DTIME`, where the `PNEWDT` used is the minimum value for all calls to user subroutines for this iteration.

If automatic time incrementation is not selected in the analysis procedure, values of `PNEWDT` that are greater than 1.0 will be ignored and values of `PNEWDT` that are less than 1.0 will cause the job to terminate.

## Variables Passed in for Information

`DIRECT(3,3)` An array containing the direction cosines of the material directions in terms of the global basis directions. `DIRECT(1,1)`, `DIRECT(2,1)`, `DIRECT(3,1)` give the (1, 2, 3) components of the first material direction; `DIRECT(1,2)`, `DIRECT(2,2)`, `DIRECT(3,2)` give the second material direction, etc. For shell and membrane elements, the first two directions are in the plane of the element and the third direction is the normal. This information is not available for beam elements.

`T(3,3)` An array containing the direction cosines of the material orientation components relative to the element basis directions. This is the orientation that defines the material directions (`DIRECT`) in terms of the element basis directions. For continuum elements `T` and `DIRECT` are identical. For shell and membrane elements $\mathtt{T(1,1)} = \cos\theta$, $\mathtt{T(1,2)} = -\sin\theta$, $\mathtt{T(2,1)} = \sin\theta$, $\mathtt{T(2,2)} = \cos\theta$, $\mathtt{T(3,3)} = 1.0$, and all other components are zero, where $\theta$ is the counterclockwise rotation around the normal vector that defines the orientation. If no orientation is used, `T` is an identity matrix. Orientation is not available for beam elements.

| | |
|---|---|
| CELENT | Characteristic element length. This is a typical length of a line across an element for a first-order element; it is half of the same typical length for a second-order element. For beams and trusses it is a characteristic length along the element axis. For membranes and shells it is a characteristic length in the reference surface. For axisymmetric elements it is a characteristic length in the $(r, z)$ plane only. |
| TIME(1) | Value of step time at the beginning of the current increment. |
| TIME(2) | Value of total time at the beginning of the current increment. |
| DTIME | Time increment. |
| CMNAME | User-specified material name, left justified. |
| ORNAME | User-specified local orientation name, left justified. |
| NFIELD | Number of field variables defined at this material point. |
| NSTATV | User-defined number of solution-dependent state variables (see *Allocating Space for Solution-Dependent State Variables*). |
| NOEL | Element number. |
| NPT | Integration point number. |
| LAYER | Layer number (for composite shells and layered solids). |
| KSPT | Section point number within the current layer. |
| KSTEP | Step number. |
| KINC | Increment number. |
| NDI | Number of direct stress components at this point. |
| NSHR | Number of shear stress components at this point. |
| COORD | Coordinates at this material point. |
| JMAC | Variable that must be passed into the GETVRM utility routine to access an output variable. |
| JMATYP | Variable that must be passed into the GETVRM utility routine to access an output variable. |
| MATLAYO | Variable that must be passed into the GETVRM utility routine to access an output variable. |
| LACCFLA | Variable that must be passed into the GETVRM utility routine to access an output variable. |

## Example: Damaged Elasticity Model

Included below is an example of user subroutine *USDFLD*. In this example a truss element is loaded in tension. A damaged elasticity model is introduced: the modulus decreases as a function of the maximum tensile strain that occurred during the loading history. The maximum tensile strain is stored as a solution-dependent state variable—see "Defining solution-dependent field variables" in *Predefined Fields*.

### Input file

```
*HEADING
 DAMAGED ELASTICITY MODEL WITH USER SUBROUTINE USDFLD
*ELEMENT, TYPE=T2D2, ELSET=ONE
1, 1, 2
*NODE
1,  0., 0.
2, 10., 0.
*SOLID SECTION, ELSET=ONE, MATERIAL=ELASTIC
1.
*MATERIAL, NAME=ELASTIC
*ELASTIC, DEPENDENCIES=1
** Table of modulus values decreasing as a function
** of field variable 1.
```

```
2000., 0.3, 0., 0.00
1500., 0.3, 0., 0.01
1200., 0.3, 0., 0.02
1000., 0.3, 0., 0.04
*USER DEFINED FIELD
*DEPVAR
1
*BOUNDARY
1, 1, 2
2, 2
*STEP
*STATIC
0.1, 1.0, 0.0, 0.1
*CLOAD
2, 1, 20.
*END STEP
*STEP
*STATIC
0.1, 1.0, 0.0, 0.1
*CLOAD
2, 1, 0.
*END STEP
*STEP, INC=20
*STATIC
0.1, 2.0, 0.0, 0.1
*CLOAD
2, 1, 40.
*END STEP
```

## User subroutine

```
      SUBROUTINE USDFLD(FIELD,STATEV,PNEWDT,DIRECT,T,CELENT,
     1 TIME,DTIME,CMNAME,ORNAME,NFIELD,NSTATV,NOEL,NPT,LAYER,
     2 KSPT,KSTEP,KINC,NDI,NSHR,COORD,JMAC,JMATYP,MATLAYO,
     3 LACCFLA)
C
      INCLUDE 'ABA_PARAM.INC'
C
      CHARACTER*80 CMNAME,ORNAME
      CHARACTER*3  FLGRAY(15)
      DIMENSION FIELD(NFIELD),STATEV(NSTATV),DIRECT(3,3),
     1 T(3,3),TIME(2)
      DIMENSION ARRAY(15),JARRAY(15),JMAC(*),JMATYP(*),
     1 COORD(*)
C
C Absolute value of current strain:
      CALL
GETVRM('E',ARRAY,JARRAY,FLGRAY,JRCD,JMAC,JMATYP,MATLAYO,LACCFLA)
      EPS = ABS( ARRAY(1) )
C Maximum value of strain up to this point in time:
      CALL
GETVRM('SDV',ARRAY,JARRAY,FLGRAY,JRCD,JMAC,JMATYP,MATLAYO,LACCFLA)
      EPSMAX = ARRAY(1)
C Use the maximum strain as a field variable
      FIELD(1) = MAX( EPS , EPSMAX )
C Store the maximum strain as a solution dependent state
C variable
      STATEV(1) = FIELD(1)
C If error, write comment to .DAT file:
      IF(JRCD.NE.0)THEN
```

```
       WRITE(6,*) 'REQUEST ERROR IN USDFLD FOR ELEMENT NUMBER ',
     1     NOEL,'INTEGRATION POINT NUMBER ',NPT
      ENDIF
C
      RETURN

      END
```

# USETTLING: User subroutine to define the settling velocity field in slurry transport analysis.

**Products:** Abaqus/Standard

**References:**

- *Defining Slurry Transport and Placement within the Cohesive Element Gap*
- **GAP FLOW*

## Overview

User subroutine *USETTLING*:

- can be used in a hydraulic fracturing simulation using slurry to define the settling velocity as a function of position, slurry concentration, effective viscosity, etc.;
- will be called at each integration point for each element-based or surface-based slurry flow definition in the analysis; and
- assumes by default that the settling velocity vector is defined along the gravity direction but also allows the settling velocity to be defined in a more general way.

## User Subroutine Interface

```
      SUBROUTINE USETTLING( KSTEP, KINC, NOEL, NPT, COORDS, NDIR, RHOS,
     1 RHOW, DIAMP, RMINGAPOVERDP, DCRACK, CVN, JTYPEFLUID, CONSK, ALPHA,
TAU,
     2 EFFMU, CVTAKENASSOLID, GRAV, VM, VSET, TEMP, FIELD, STATEV, NFIELD,

     3 NSTATV, IVSETDIR, I_ARRAY, NIARRAY, R_ARRAY, NRARRAY, C_ARRAY,
     4 NCARRAY )
C
      INCLUDE 'ABA_PARAM.INC'
C
      DIMENSION VM(2), VSET(3), GRAV(2), COORDS(*)
      DIMENSION FIELD(NFIELD), STATEV(NSTATV)
      DIMENSION I_ARRAY(*), R_ARRAY(*)
      CHARACTER*80 C_ARRAY(*)
C
      user coding to define VSET


      RETURN
      END
```

## Variables to Be Defined

`VSET` Components of the settling velocity vector at this integration point.

When `IVSETDIR`=1, the components of the settling velocity vector are adjusted to ensure that the settling velocity vector is along the direction of the gravity vector. The magnitude of the vector is preserved. To specify only the settling velocity magnitude, it is recommended that you use `VSET(1)`.

When IVSETDIR=2, you can define all the components of the settling velocity vector in global coordinates. If the settling velocity vector is not parallel to the fracture plane, it will be projected to this plane.

## Variables That Can Be Updated

| | |
|---|---|
| STATEV(NSTATV) | Array containing the user-defined solution-dependent state variables at this point. |
| IVSETDIR | Set IVSETDIR=1 to indicate that the settling velocity vector components will be adjusted to ensure the vector is along the gravity direction.<br>Set IVSETDIR=2 to indicate that the settling velocity vector can be defined arbitrarily in global coordinates.<br>If this value is not defined, it is assumed to be 1. |

## Variables Passed in for Information

| | |
|---|---|
| KSTEP | Current step number. |
| KINC | Current increment number. |
| NOEL | Element number. |
| NPT | Integration point number. |
| COORDS | An array containing the coordinates of this point. These are the current coordinates if geometric nonlinearity is accounted for during the step (see *Defining an Analysis*); otherwise, the array contains the original coordinates of the point. |
| NDIR | Number of spatial dimensions at this point. |
| RHOS | Density of proppant particles. |
| RHOW | Density of carrier fluid. |
| DIAMP | Diameter of proppant particles. |
| RMINGAPOVERDP | Minimum value of the ratio of gap opening to particle diameter $\alpha$ (see *Section Controls*). Particles can only move into a fracture with a gap opening of at least $\alpha$ times the particle diameter. By default, the minimum ratio $\alpha$ is 1.0. |
| DCRACK | Crack opening. |
| CVN | Maximum slurry concentration of the current element. |
| JTYPEFLUID | JTYPEFLUID=1 for a *Newtonian Fluid*.<br>JTYPEFLUID=2 for a *Power Law Fluid*.<br>JTYPEFLUID=3 for a *Bingham Plastic Fluid*.<br>JTYPEFLUID=4 for a *Herschel-Bulkley Fluid*. |
| CONSK | Fluid consistency. This is a material property for defining a power law fluid, a Bingham plastic fluid, or a Herschel-Bulkley fluid. |
| ALPHA | Power law coefficient. This is a material property for defining a power law fluid or a Herschel-Bulkley fluid. |
| TAU | Yield stress. This is a material property for defining a Bingham plastic fluid or a Herschel-Bulkley fluid. |
| EFFMU | Apparent viscosity as a measure of fluid resistance to deformation at a given shear rate. For a Newtonian fluid, it is the fluid viscosity. For a non-Newtonian fluid, Abaqus will compute its value. |
| CVTAKENASSOLID | Maximum attainable slurry concentration. |

| | |
|---|---|
| GRAV(2) | Gravity vector in local coordinates. |
| VM(2) | Mean velocity vector of slurry flow in local coordinates. |
| TEMP | Temperature at the start of the increment. |
| FIELD(NFIELD) | Array of field variables at the start of the increment. |
| NIARRAY | Size of array I_ARRAY. |
| I_ARRAY | Not used. |
| NRARRAY | Size of array R_ARRAY. |
| R_ARRAY | Not used. |
| NCARRAY | Size of array C_ARRAY. |
| C_ARRAY | Not used. |

# USUPERELASHARDMOD: User subroutine to modify the material constants of the superelasticity model as a function of the plastic strain.

**Products:** Abaqus/Standard

**References:**

- *Plasticity Model for Superelastic Materials*
- **SUPERELASTIC HARDENING*
- **SUPERELASTIC HARDENING MODIFICATIONS*

## Overview

User subroutine *USUPERELASHARDMOD*:

- is called at all material calculation points of elements for which the material definition includes user-defined hardening modifications; and
- can use field variables or state variables.

## User Subroutine Interface

```
      SUBROUTINE USUPERELASHARDMOD(CMNAME,EQPLAS,
     1     NOEL,NPT,LAYER,KSPT,KINC,KSTEP,NSTATV,TEMP,DTEMP,REFT,
     2     TIME,DTIME,PROPS,NUMPROPS,NUMFIELDV,PREDEF,DPRED,
     3    EMODAREF,ENUAREF,EMODSREF,ENUSREF,SSASREF,SFASREF,SSSAREF,SFSAREF,

     4     CAS,CSA,EMODA,ENUA,EMODS,ENUS,SSAS,SFAS,SSSA,SFSA,STATEV)
C
      INCLUDE 'ABA_PARAM.INC'
C
      CHARACTER*80 CMNAME
      DIMENSION STATEV(NSTATV),TIME(*),
     $          PREDEF(NUMFIELDV),DPRED(NUMFIELDV),PROPS(NUMPROPS)

      user coding to define EMODA,ENUA,EMODS,ENUS,SSAS,SFAS,SSSA,SFSA

      RETURN
      END
```

## Variables to Be Defined

| | |
|---|---|
| EMODA | $E_A$, austenite's Young's modulus. |
| ENUA | $\nu_A$, austenite's Poisson's ratio. |
| EMODS | $E_M$, martensite's Young's modulus. |
| ENUS | $\nu_M$, martensite's Poisson's ratio. |
| SSAS | $\sigma_{tL}^{S}$, stress at which the transformation begins during loading in tension. |
| SFAS | $\sigma_{tL}^{E}$, stress at which the transformation ends during loading in tension. |

| | |
|---|---|
| SSSA | $\sigma_{tU}^{S}$, stress at which the reverse transformation begins during unloading in tension. |
| SFSA | $\sigma_{tU}^{E}$, stress at which the reverse transformation begins during unloading in tension. |
| STATEV(NSTATV) | Array containing the user-defined, solution-dependent state variables at this point. These are supplied as values at the beginning of the increment or as values updated by other user subroutines (see *Allocating Space for Solution-Dependent State Variables*) and must be returned as values at the end of the increment. |

## Variables Passed in for Information

| | |
|---|---|
| CMNAME | User-specified material name, left-justified. |
| EQPLAS | Equivalent plastic strain, $\bar{\varepsilon}^{pl}$. |
| NOEL | Element number. |
| NPT | Integration point number. |
| LAYER | Layer number (for composite shells and layered solids). |
| KSPT | Section point number within the current layer. |
| KINC | Increment number. |
| KSTEP | Step number. |
| NSTATV | User-specified number of solution-dependent state variables associated with this material (*Allocating Space for Solution-Dependent State Variables*). |
| TEMP | Temperature at the beginning of the increment. |
| DTEMP | Increment of temperature. |
| REFT | Reference temperature. |
| TIME(1) | Value of step time at the beginning of the current increment. |
| TIME(2) | Value of total time at the beginning of the current increment. |
| DTIME | Time increment. |
| NPROPS | Number of hardening properties entered for this user-defined hardening definition. |
| PROPS(NPROPS) | Array of hardening properties entered for this user-defined hardening definition. |
| NUMFIELDV | Number of field variables. |
| PREDEF(NUMFIELDV) | Array of interpolated values of predefined field variables at this material point at the start of the increment based on the values read in at the nodes (initial values at the beginning of the analysis and current values during the analysis). |
| DPRED(NUMFIELDV) | Array of increments of predefined field variables at this material point for this increment, which includes any values updated by user subroutine *USDFLD*. |
| CAS | $\left(\frac{\delta\sigma}{\delta T}\right)_L$, slope of the stress versus temperature curve for loading. |
| CSA | $\left(\frac{\delta\sigma}{\delta T}\right)_U$, slope of the stress versus temperature curve for unloading. |
| EMODAREF | $E_A$, reference value of austenite's Young's modulus. |
| ENUAREF | $\nu_A$, reference value of austenite's Poisson's ratio. |
| EMODSREF | $E_M$, reference value of martensite's Young's modulus. |
| ENUSREF | $\nu_M$, reference value of martensite's Poisson's ratio. |
| SSASREF | $\sigma_{tL}^{S}$, reference value of stress at which the transformation begins during loading in tension. |

SFASREF

$\sigma_{tL}^{E}$, reference value of stress at which the transformation ends during loading in tension.

SSSAREF

$\sigma_{tU}^{S}$, reference value of stress at which the reverse transformation begins during unloading in tension.

SFSAREF

$\sigma_{tU}^{E}$, reference value of stress at which the reverse transformation begins during unloading in tension.

# UTEMP: User subroutine to specify prescribed temperatures.

**Products:** Abaqus/Standard

**References:**

- *Predefined Fields*
- **TEMPERATURE*
- *LE11: Solid cylinder/taper/sphere—temperature loading*
- *UTEMP, VUTEMP, UFIELD, UMASFL, and UPRESS*
- *Beam Elements*
- *Shell Elements*

## Overview

User subroutine *UTEMP*:

- allows you to prescribe temperatures at the nodes of a model;
- will be called whenever a current value of temperature is needed for a node that is listed under a user-defined temperature field definition;
- ignores any temperatures provided for the associated temperature field definition outside the user subroutine; and
- can be used to modify any temperatures read in from a results file.

## User Subroutine Interface

```
      SUBROUTINE UTEMP(TEMP,NSECPT,KSTEP,KINC,TIME,NODE,COORDS)
C
      INCLUDE 'ABA_PARAM.INC'
C
      DIMENSION TEMP(NSECPT), TIME(2), COORDS(3)
C

      user coding to define TEMP


      RETURN
      END
```

## Variables to Be Defined

`TEMP(NSECPT)` Array of temperature values at node number `NODE`. If the node is not connected to a beam or shell element, only one value of temperature must be returned (`NSECPT`=1). Otherwise, the number of temperatures to be returned depends on the mode of temperature and field variable input selected for the beam or shell section. The following cases are possible:

1. Temperatures and field variables for a beam section are given as values at the points shown in the beam section descriptions. The number of values required, `NSECPT`, is determined by the particular section type specified, as described in *Beam Cross-Section Library*.
2. Temperatures and field variables are given as values at *n* equally spaced points through each layer of a shell section. The number of values required, `NSECPT`, is equal to *n*.

3. Temperatures and field variables for a beam section are given as values at the origin of the cross-section together with gradients with respect to the 2-direction and, for three-dimensional beams, the 1-direction of the section; or temperatures and field variables for a shell section are given as values at the reference surface together with gradients with respect to the thickness. The number of values required, NSECPT, is 3 for three-dimensional beams, 2 for two-dimensional beams, and 2 for shells. Give the midsurface value first, followed by the first and (if necessary) second gradients, as described in *Beam Elements* and *Shell Elements*.

You can also request temperatures to be set in one of two other ways: from a previously generated results file or via direct data input. When array TEMP is passed into user subroutine *UTEMP*, it will contain temperatures obtained from the results file only. You can modify these values within this routine. Any values given as direct data input will be ignored.

## Variables Passed in for Information

| | |
|---|---|
| NSECPT | Maximum number of section values required for any node in the model. |
| KSTEP | Step number. |
| KINC | Increment number. |
| TIME(1) | Current value of step time. |
| TIME(2) | Current value of total time. |
| NODE | Node number. |
| COORDS | An array containing the current coordinates of this point. These are the current coordinates if geometric nonlinearity is accounted for during the step (see *Defining an Analysis*); otherwise, the array contains the original coordinates of the node. |

# UTRACLOAD: User subroutine to specify nonuniform traction loads.

**Products:** Abaqus/Standard

**References:**

- *Distributed Loads*
- **DLOAD*
- **DSLOAD*
- *Distributed traction and edge loads*

## Overview

User subroutine *UTRACLOAD*:

- can be used to define the variation of the distributed traction load magnitude as a function of position, time, element number, load integration point number, etc.;
- if needed, can be used to define the *initial* loading direction for the distributed traction load as a function of position, element number, load integration point number, etc.;
- will be called at each load integration point for each element-based, edge-based, or surface-based nonuniform distributed traction load definition during stress analysis;
- cannot be used in mode-based procedures to describe the time variation of the load; and
- ignores any amplitude references that may appear with the associated step definition or nonuniform distributed traction load definition.

## User Subroutine Interface

```
      SUBROUTINE UTRACLOAD(ALPHA,T_USER,KSTEP,KINC,TIME,NOEL,NPT,
     1 COORDS,DIRCOS,JLTYP,SNAME)
C
      INCLUDE 'ABA_PARAM.INC'
C
      DIMENSION T_USER(3), TIME(2), COORDS(3), DIRCOS(3,3)
      CHARACTER*80 SNAME

      user coding to define ALPHA and T_USER


      RETURN
      END
```

## Variables to Be Defined

ALPHA Magnitude of the distributed traction load. Units are $FL^{-2}$ for surface loads, $FL^{-1}$ for edge loads, and F for edge moments. ALPHA is passed into the routine as the magnitude of the load specified as part of the element-based or surface-based distributed load definition. If the magnitude is not defined, ALPHA is passed in as zero. For a static analysis that uses the modified Riks method (*Unstable Collapse and Postbuckling Analysis*) ALPHA must be defined as a function of the load proportionality factor, $\lambda$. The distributed load magnitude is not available for output purposes.

`T_USER` Loading direction of the distributed traction load. `T_USER` is passed into the routine as the load direction specified as part of the element-based or surface-based distributed load definition. The vector `T_USER` passed out of the subroutine is used as the initial loading direction $\mathbf{t}_{user}$ discussed in *Distributed Loads*. The direction of `T_USER` as defined by the subroutine should not change during a step. If it does, convergence difficulties might arise. Load directions are needed only for a nonuniform general surface traction, shear surface traction, and general edge traction. If a direction is defined for the nonuniform normal edge traction, shear edge traction, transverse edge traction, or edge moment, it will be ignored. See *Distributed Loads* for details.

## Variables Passed in for Information

`KSTEP` Step number.

`KINC` Increment number.

`TIME(1)` Current value of step time or current value of the load proportionality factor, $\lambda$, in a Riks step.

`TIME(2)` Current value of total time.

`NOEL` User-defined element number.

`NPT` Load integration point number within the element or on the element's surface, depending on the load type.

`COORDS` An array containing the coordinates of the load integration point. These are the current coordinates if geometric nonlinearity is accounted for during the step (see *Defining an Analysis*); otherwise, the array contains the original coordinates of the point.

`DIRCOS` Orientation of the face or edge in the reference configuration. For three-dimensional facets the first and second columns are the normalized local directions in the plane of the surface, and the third column is the normal to the face. For solid elements the normal points inward, which is the negative of what is defined in *Conventions*; for shell elements the normal definition is consistent with the convention. For two-dimensional facets the first column is the normalized tangent, the second column is the facet normal, and the third column is not used. For three-dimensional shell edges the first column is the tangent to the shell edge (shear direction), the second column is the in-plane normal (normal direction), and the third column is the normal to the plane of the shell (transverse direction).

`JLTYP` Identifies the load type for which this call to *`UTRACLOAD`* is being made. The load type may be an element-based surface load, an edge-based load, or a surface-based load. This variable identifies the element face or edge for which this call to *`UTRACLOAD`* is being made. This information is useful when several different nonuniform distributed loads are being imposed on an element at the same time. See *About the Element Library* for element face and edge identification. The load labels are shown in *Table 1*. For surface- or edge-based loading (TRSHRNU, TRVECNU, EDLDNU, EDNORNU, EDSHRNU, EDTRANU, EDMOMNU), *j* in the load type identifies the face or edge of the element underlying the surface.

**Table 1: `JLTYP` values for surface traction and edge load labels.**

| Load Label | `JLTYP` | Load Label | `JLTYP` | Load Label | `JLTYP` |
|---|---|---|---|---|---|
| TRSHRNU | `510+j` | EDLDNU | `540+j` | EDTRANU | `570+j` |
| TRSHR1NU | `511` | EDLD1NU | `543` | EDTRANU | `573` |
| TRSHR2NU | `512` | EDLD2NU | `544` | EDTRANU | `574` |
| TRSHR3NU | `513` | EDLD3NU | `545` | EDTRANU | `575` |
| TRSHR4NU | `514` | EDLD4NU | `546` | EDTRANU | `576` |
| TRSHR5NU | `515` | EDNORNU | `550+j` | EDMOMNU | `580+j` |

| Load Label | JLTYP | Load Label | JLTYP | Load Label | JLTYP |
|---|---|---|---|---|---|
| TRSHR6NU | 516 | EDNOR1NU | 553 | EDMOM1NU | 583 |
| TRVECNU | 520+j | EDNOR2NU | 554 | EDMOM2NU | 584 |
| TRVEC1NU | 521 | EDNOR3NU | 555 | EDMOM3NU | 585 |
| TRVEC2NU | 522 | EDNOR4NU | 556 | EDMOM4NU | 586 |
| TRVEC3NU | 523 | EDSHRNU | 560+j | | |
| TRVEC4NU | 524 | EDSHRNU | 563 | | |
| TRVEC5NU | 525 | EDSHRNU | 564 | | |
| TRVEC6NU | 526 | EDSHRNU | 565 | | |
| | | EDSHRNU | 566 | | |

SNAME Surface name for a surface-based load definition. For an element-based or edge-based load the surface name is passed in as blank.

# UTRS: User subroutine to define a reduced time shift function for a viscoelastic material.

**Products:** Abaqus/Standard

**References:**

- *Frequency Domain Viscoelasticity*
- *Time Domain Viscoelasticity*
- **TRS*
- **VISCOELASTIC*
- *Transient thermal loading of a viscoelastic slab*

## Overview

User subroutine *UTRS*:

- can be used to define a temperature-time shift for a viscoelastic analysis;
- will be called for all material points of elements for which a user-defined shift function is specified to define the time-temperature correspondence as part of the viscoelastic material definition;
- will be called before user subroutine *UMATHT* and/or user subroutine *HETVAL* if either or both are to be used with *UTRS* in a fully coupled temperature-displacement or a coupled thermal-electrical-structural analysis;
- can use and update solution-dependent state variables; and
- can be used in conjunction with user subroutine *USDFLD* to redefine any field variables before they are passed in.

## User Subroutine Interface

```
      SUBROUTINE UTRS(SHIFT,TEMP,DTEMP,TIME,DTIME,PREDEF,DPRED,
     1 STATEV,CMNAME,COORDS)
C
      INCLUDE 'ABA_PARAM.INC'
C
      CHARACTER*80 CMNAME
      DIMENSION SHIFT(2),TIME(2),PREDEF(1),DPRED(1),STATEV(1),
     1 COORDS(1)
C

      user coding to define SHIFT(1) and SHIFT(2)

      RETURN
      END
```

## Variables to Be Defined

`SHIFT` An array of length two that defines the shift function, $A$ ($A > 0$), at this point.

In time domain viscoelasticity `SHIFT(1)` defines the shift function at the beginning of the increment and `SHIFT(2)` defines the shift function at the end of the increment. Abaqus/Standard applies an averaging scheme to these values that assumes that the natural logarithm of the shift function can be approximated by a linear function over the increment.

In frequency domain viscoelasticity `SHIFT(1)` defines the shift function and `SHIFT(2)` is ignored.

If either element of `SHIFT` is found to be less than or equal to zero, the analysis terminates with an error message.

## Variables That Can Be Updated

`STATEV` An array containing the solution-dependent state variables at this point. This array will be passed in containing the values of these variables at the start of the increment unless they are updated in user subroutines *USDFLD* or *UEXPAN*, in which case the updated values are passed in. If any of the solution-dependent state variables are being used in conjunction with the viscoelastic behavior, they must be updated in this subroutine to their values at the end of the increment.

## Variables Passed in for Information

`TEMP` Temperature at the end of the increment.

`DTEMP` Increment of temperature during the time increment.

`PREDEF` An array containing the values of all of the user-specified field variables at this point at the end of the increment (initial values at the beginning of the analysis and current values during the analysis).

`DPRED` An array containing the increments of all of the predefined field variables during the time increment.

`TIME(1)` Value of step time at the end of the current increment in time domain viscoelasticity or frequency in frequency domain viscoelasticity.

`TIME(2)` Value of total time at the end of the current increment. In frequency domain viscoelasticity this value is set to -1.0.

`DTIME` Time increment. If this subroutine is called during a procedure such as a static analysis in which the viscoelastic effects are not taken into account or if it is called during an analysis in the frequency domain, this variable is passed in as zero.

`CMNAME` User-specified material name, left justified.

`COORDS` An array containing the coordinates of the material point. These are the current coordinates if geometric nonlinearity is accounted for during the step (see *Defining an Analysis*); otherwise, the array contains the original coordinates of the point.

# UTRSNETWORK: User subroutine to define a reduced time shift function for models defined within the parallel rheological framework.

**Products:** Abaqus/Standard

**References:**

- *Parallel Rheological Framework*
- *Nonlinear large-strain viscoelasticity with hyperelasticity*
- *UTRSNETWORK*
- **VISCOELASTIC*

## Overview

User subroutine *UTRSNETWORK*:

- can be used to define a temperature-time shift for a nonlinear viscoelastic network for models defined using the parallel rheological framework;
- will be called for all material points of elements for which a user-defined shift function is specified to define the time-temperature correspondence as part of the viscoelastic material definition;
- can use and update solution-dependent state variables; and
- can be used in conjunction with user subroutine *USDFLD* to redefine any field variables before they are passed in.

## User Subroutine Interface

```
      subroutine utrsnetwork (
C Must be updated
     *   outputData,
C Can be updated
     *   statev,
C Information (Read only)
     *   nOutput,
     *   nstatv,
     *   networkid,
     *   coords,
     *   temp,
     *   dtemp,
     *   nfield,
     *   predef,
     *   dpred,
     *   nprops,
     *   props,
     *   i_array,
     *   niarray,
     *   r_array,
     *   nrarray,
     *   c_array,
     *   ncarray)
C
      include 'aba_param.inc'
C
      parameter( io_trs_shift_begin = 1,
```

```
     *           io_trs_shift_end   = 2 )
C
      parameter( i_trs_kstep   = 1,
     *           i_trs_kinc    = 2,
     *           i_trs_noel    = 3,
     *           i_trs_npt     = 4,
     *           i_trs_layer   = 5,
     *           i_trs_kspt    = 6 )
C
      parameter( ir_trs_step_time  = 1,
     *           ir_trs_total_time = 2,
     *           ir_trs_creep_time = 3,
     *           ir_trs_timeinc    = 4 )

C
      parameter( ic_trs_material_name = 1 )
C
      dimension
     *   statev(nstatv),
     *   predef(nfield),
     *   dpred(nfield),
     *   props(nprops),
     *   coords(*),
     *   outputData(nOutput),
     *   i_array(niarray),
     *   r_array(nrarray)

      character*80 c_array(ncarray)
C

      user coding to define outputData(io_trs_shift_begin)
         and outputData(io_trs_shift_end)

      return
      end
```

## Variables to Be Defined

| | |
|---|---|
| `outputData(io_trs_shift_begin)` | The shift function at the beginning of the increment. |
| `outputData(io_trs_shift_end)` | The shift function at the end of the increment. |

## Variables That Can Be Updated

| | |
|---|---|
| `statev` | An array containing the user-defined solution-dependent state variables at this point. |

## Variables Passed in for Information

| | |
|---|---|
| `nOutput` | Size of array outputData. Currently equal to 2. |
| `nstatv` | Number of solution-dependent state variables associated with this material. |
| `networkid` | Network identification number, which identifies the network for which creep is defined. |
| `coords` | An array containing the current coordinates at this point. |
| `temp` | Temperature at the end of the increment. |
| `dtemp` | Increment of temperature. |

| | |
|---|---|
| `nfield` | Number of field variables. |
| `predef` | An array of interpolated values of predefined field variables at this point at the end of the increment, based on the values read in at the nodes and, optionally, redefined in user subroutine *USDFLD*. |
| `dpred` | An array of increments of predefined field variables. |
| `nprops` | User-specified number of user-defined material properties. |
| `props` | An array of user-specified property values. |
| `i_array(i_trs_kstep)` | Step number. |
| `i_array(i_trs_kinc)` | Increment number. |
| `i_array(i_trs_noel)` | Element number. |
| `i_array(i_trs_npt)` | Integration point. |
| `i_array(i_trs_layer)` | Layer number (for layered solids). |
| `i_array(i_trs_kspt)` | Section point number within the current layer. |
| `niarray` | Size of array i_array. Currently equal to 6. |
| `r_array(ir_trs_step_time)` | Value of step time at the end of the increment. |
| `r_array(ir_trs_total_time)` | Value of total time at the end of the increment. |
| `r_array(ir_trs_creep_time)` | Value of creep time at the end of the increment. |
| `r_array(ir_trs_timeinc)` | Time increment. |
| `nrarray` | Size of array r_array. Currently equal to 4. |
| `c_array(ic_trs_material_name)` | User-specified material name, left justified. Some internal material models are given names starting with the "ABQ_" character string. To avoid conflict, you should not use "ABQ_" as the leading string for the material name. |
| `ncarray` | Size of array c_array. Currently equal to 1. |

## Example: Williams-Landel-Ferry Shift Function

As an example of the coding of user subroutine *UTRSNETWORK*, consider the William-Landel-Ferry model to define the shift function. In this case the shift function is expressed as (see *Thermorheologically Simple Temperature Effects*)

$$\log_{10}(A) = -\frac{C_1(\theta - \theta_0)}{C_2 + (\theta - \theta_0)},$$

where

$\theta$

is the temperature,

$\theta_0$

is the reference temperature, and

$C_1$ and $C_2$

are constants.

The user subroutine would be coded as follows:

```
      subroutine utrsnetwork (
C Must be updated
     *   outputData,
C Can be updated
     *   statev,
C Information (Read only)
     *   nOutput,
     *   nstatv,
     *   networkid,
     *   coords,
     *   temp,
     *   dtemp,
     *   nfield,
     *   predef,
     *   dpred,
     *   nprops,
     *   props,
     *   i_array,
     *   niarray,
     *   r_array,
     *   nrarray,
     *   c_array,
     *   ncarray)
C
      include 'aba_param.inc'
C
      parameter( io_trs_shift_begin = 1,
     *           io_trs_shift_end   = 2 )
C
      parameter( i_trs_kstep   = 1,
     *           i_trs_kinc    = 2,
     *           i_trs_noel    = 3,
     *           i_trs_npt     = 4,
     *           i_trs_layer   = 5,
     *           i_trs_kspt    = 6 )
C
      parameter( ir_trs_step_time  = 1,
     *           ir_trs_total_time = 2,
     *           ir_trs_creep_time = 3,
     *           ir_trs_timeinc    = 4 )

C
      parameter( ic_trs_material_name = 1 )
C
C
      parameter( zero=0.0d0, one=1.0d0, dln10=2.30258509299d0)
C
      dimension
     *   statev(nstatv),
     *   predef(nfield),
     *   dpred(nfield),
     *   props(nprops),
     *   coords(*),
     *   outputData(nOutput),
     *   i_array(niarray),
     *   r_array(nrarray)

      character*80 c_array(ncarray)
C
```

```
      outputData(io_trs_shift_begin) = zero
      outputData(io_trs_shift_end) = zero
      temp0 = temp-dtemp
C
C WLF
C
      theta0 = props(1)
      C1 = props(2)
      C2 = props(3)
      outputData(io_trs_shift_begin) =
     &   exp(-dln10*C1*(temp0-theta0)/(C2+(temp0-theta0)))
      outputData(io_trs_shift_end) =
     &   exp(-dln10*C1*(temp-theta0)/(C2+(temp-theta0)))
C
      return
      end
```

# UVAREL: User subroutine to generate element output.

**Products:** Abaqus/Standard

**References:**

- *Obtaining Material Point Information from User Subroutine UVAREL*
- *Using Abaqus/Standard Output Variable Identifiers*

## Overview

User subroutine UVAREL:

- will be called at all elements for which the element user output variables are defined;
- will be called for each converged increment in a step only once;
- allows you to define output quantities that are functions of any of the available integration point quantities listed in the Output Variable Identifiers table (*Using Abaqus/Standard Output Variable Identifiers*);
- allows you define whole element output variables (UVARE) as well as element section output variables (UVARPT),
- can be used with continuum solid, solid shell, conventional shell, continuum shell, and beam elements;
- can call utility routine `getElMPtVar` to access material point data for any material point within the element;
- can call utility routine `getSecVar` to access element section output variables;
- can call utility routines `getUVarElMaterialName` and `getUVarElOrientationName` to access the material name and the orientation name, respectively;
- cannot be used with linear perturbation procedures; and
- can be used only with procedures with mechanical degrees of freedom.

## Accessing Material Point Data

You can use utility routine `getElMPtVar` to access the values of the material point quantities for any material point within the element (see *Obtaining Material Point Information in an Abaqus/Standard Analysis*). The values of the material point data are recovered in the arrays `ARRAY`, `JARRAY`, and `FLGRAY` for floating point, integer, and character data, respectively. Floating point data are recovered as double-precision data.

## Using Element User-Defined Output Variables

You can use UVAREL to define both whole element as well as integration point output variables. The output identifiers for the user-defined output quantities are UVARE and UVARPT respectively. You access individual components with UVARE*n* (where $n = 1, 2, \ldots$, `NUVARE`) and UVARPT*n* (where $n = 1, 2, \ldots$, `NUVARPT`). You must specify the number of element user-defined output variables (`NUVARE`) and the number of user-defined output variables at the integration points (`NUVARPT`) for a given element to allocate space for each variable. You can use any number of user-defined output variables. Abaqus/Standard writes the user-defined output variables to the output database and restart files for contouring, printing, and *X*–*Y* plotting in Abaqus/CAE.

### Output Precision

Abaqus/Standard writes the data in single precision to the output database (`.odb`) file. Because you provide UVAREL output variables in double precision, numeric overflow errors related to output to the output database

file might occur in cases where the output results exceed the capacity for single-precision representation even when no overflow errors occur in *UVAREL*.

## User Subroutine Interface

```
      subroutine uvarel(uVarE, uVarPt, jOutput, time, dTime,
     * nuUarE, nuVarPt, noel, nPt, nLayer, nSpt, nRebar, kStep,
     * kInc, nDir, nShr, props, nProps, abqInternal)

      include 'aba_param.inc'

      dimension uVarE(nuVarE), uVarPt(nuVarPt,nPt), time(2),
     *     nSpt(nLayer), props(nProps)

      character*3 flgray(15)
      dimension array(15), jarray(15)

c     The dimensions of the variables flgray, array, and jarray
c     must be set equal to or greater than 15.


      user coding to define uVarE or uVarPt


      return
      end
```

## Variables to Be Defined

`uVarE(nuVarE)` Array containing the element user-defined output variables. These are passed in as the values at the beginning of the increment and must be returned as the values at the end of the increment.

`uVarPt(nuVarPt,nPt)` Array containing the user-defined output variables at the integration points. These are passed in as the values at the beginning of the increment and must be returned as the values at the end of the increment.

## Variables Passed in for Information

`jOutput` Output request flag

If `jOutput`=0, there is no output request at this increment.

If `jOutput`=1, there is an output request at this increment.

`time(1)` Value of step time at the end of the current increment.

`time(2)` Value of total time at the end of the current increment.

`dTime` Time increment.

`nuVarE` User-specified number of element user-defined output variables.

`nuVarPt` User-specified number of user-defined output variables at the integration points.

`noel` Element number.

`nPt` Number of integration points.

`nLayer` Number of layers (for composite shells and layered solids). 1 for all other elements.

`nSpt(nLayer)` Array containing the number of section points per layer.

| | |
|---|---|
| `nRebar` | Number of rebars. |
| `kStep` | Step number. |
| `kInc` | Increment number. |
| `nDir` | Number of direct stress components at this point. |
| `nShr` | Number of shear stress components at this point. |
| `props(nProps)` | User-specified array of constants associated with user-defined output variables. |
| `nProps` | Number of user-specified arrays of constants associated with user-defined output variables. |
| `abqInternal` | Variable that must be passed into the utility routines. |

# UVARM: User subroutine to generate element output.

**Products:** Abaqus/Standard

**References:**

- *Obtaining Material Point Information in an Abaqus/Standard Analysis*
- **USER OUTPUT VARIABLES*
- *UVARM*
- *Using Abaqus/Standard Output Variable Identifiers*

## Overview

User subroutine UVARM:

- will be called at all material calculation points of elements for which the material definition includes the specification of user-defined output variables;
- might be called multiple times for each material point in an increment, as Abaqus/Standard iterates to a converged solution;
- will be called for each increment in a step;
- allows you to define output quantities that are functions of any of the available integration point quantities listed in the Output Variable Identifiers table (*Using Abaqus/Standard Output Variable Identifiers*);
- allows you to define the material directions as output variables;
- can be used for gasket elements;
- can call utility routine GETVRM to access material point data;
- cannot be used with linear perturbation procedures, except for the static perturbation procedure;
- will be called once per load case in a static perturbation procedure with multiple load cases; and
- cannot be updated in the zero increment.

## Accessing Material Point Data

You are provided with access to the values of the material point quantities through the utility routine GETVRM described in *Obtaining Material Point Information in an Abaqus/Standard Analysis*. In a nonlinear analysis values returned will correspond to the current solution iteration, representing a converged solution only at the final iteration for each increment. In the static perturbation procedure, only those quantities for which perturbation values are computed can be accessed. The values of the material point data are recovered in the arrays ARRAY, JARRAY, and FLGRAY for floating point, integer, and character data, respectively. Floating point data are recovered as double-precision data.

## Using User-Defined Output Variables

The output identifier for the user-defined output quantities is UVARM. Individual components are accessed with UVARM*n*, where $n = 1, 2, \ldots$, NUVARM. You must specify the number of user-defined output variables, NUVARM, for a given material to allocate space at each material calculation point for each variable. The user-defined output variables are available for both printed and results file output and are written to the output database and restart

files for contouring, printing, and *X*–*Y* plotting in Abaqus/CAE. Any number of user-defined output variables can be used.

### Output Precision

The data are provided in double precision for output to the data (`.dat`) and results (`.fil`) files and are written to the output database (`.odb`) file in single precision. Because the user provides *UVARM* output variables in double precision, numeric overflow errors related to output to the output database file might occur in cases where the output results exceed the capacity for single-precision representation even when no overflow errors occur in *UVARM*.

## User Subroutine Interface

```
      SUBROUTINE UVARM(UVAR,DIRECT,T,TIME,DTIME,CMNAME,ORNAME,
     1 NUVARM,NOEL,NPT,LAYER,KSPT,KSTEP,KINC,NDI,NSHR,COORD,
     2 JMAC,JMATYP,MATLAYO,LACCFLA)
      INCLUDE 'ABA_PARAM.INC'
C
      CHARACTER*80 CMNAME,ORNAME
      CHARACTER*3 FLGRAY(15)
      DIMENSION UVAR(NUVARM),DIRECT(3,3),T(3,3),TIME(2)
      DIMENSION ARRAY(15),JARRAY(15),JMAC(*),JMATYP(*),COORD(*)

C     The dimensions of the variables FLGRAY, ARRAY and JARRAY
C     must be set equal to or greater than 15.


      user coding to define UVAR


      RETURN
      END
```

## Variables to Be Defined

`UVAR(NUVARM)` An array containing the user-defined output variables. These are passed in as the values at the beginning of the increment and must be returned as the values at the end of the increment.

## Variables Passed in for Information

`DIRECT(3,3)` An array containing the direction cosines of the material directions in terms of the global basis directions. `DIRECT(1,1)`, `DIRECT(2,1)`, `DIRECT(3,1)` give the (1, 2, 3) components of the first material direction; `DIRECT(1,2)`, `DIRECT(2,2)`, `DIRECT(3,2)` give the second material direction, etc. For shell and membrane elements, the first two directions are in the plane of the element and the third direction is the normal. This information is not available for beam and truss elements.

`T(3,3)` An array containing the direction cosines of the material orientation components relative to the element basis directions. This is the orientation that defines the material directions (`DIRECT`) in terms of the element basis directions. For continuum elements `T` and `DIRECT` are identical. For shell and membrane elements $\mathtt{T(1,1)} = \cos\theta$, $\mathtt{T(1,2)} = -\sin\theta$, $\mathtt{T(2,1)} = \sin\theta$, $\mathtt{T(2,2)} = \cos\theta$, $\mathtt{T(3,3)} = 1.0$, and all other components are zero, where $\theta$ is the counterclockwise rotation around the normal vector

| | |
|---|---|
| | that defines the orientation. If no orientation is used, `T` is an identity matrix. Orientation is not available for beam and truss elements. |
| `TIME(1)` | Value of step time at the end of the current increment. |
| `TIME(2)` | Value of total time at the end of the current increment. |
| `DTIME` | Time increment. |
| `CMNAME` | User-specified material name, left justified. |
| `ORNAME` | User-specified local orientation name, left justified. |
| `NUVARM` | User-specified number of user-defined output variables. |
| `NOEL` | Element number. |
| `NPT` | Integration point number. |
| `LAYER` | Layer number (for composite shells and layered solids). |
| `KSPT` | Section point number within the current layer. |
| `KSTEP` | Step number. |
| `KINC` | Increment number. |
| `NDI` | Number of direct stress components at this point. |
| `NSHR` | Number of shear stress components at this point. |
| `COORD` | Coordinates at this material point. |
| `JMAC` | Variable that must be passed into the `GETVRM` utility routine to access an output variable. |
| `JMATYP` | Variable that must be passed into the `GETVRM` utility routine to access an output variable. |
| `MATLAYO` | Variable that must be passed into the `GETVRM` utility routine to access an output variable. |
| `LACCFLA` | Variable that must be passed into the `GETVRM` utility routine to access an output variable. |

## Example: Calculation of Stress Relative to Shift Tensor

Below is an example of user subroutine *UVARM*. The subroutine calculates the position of the current state of stress relative to the center of the yield surface for the kinematic hardening plasticity model by subtracting the kinematic shift tensor, $\boldsymbol{\alpha}$, from the stress tensor, $\boldsymbol{\sigma}$. See *About metal plasticity models* for additional details.

```
      SUBROUTINE UVARM(UVAR,DIRECT,T,TIME,DTIME,CMNAME,ORNAME,
     1 NUVARM,NOEL,NPT,LAYER,KSPT,KSTEP,KINC,NDI,NSHR,COORD,
     2 JMAC,JMATYP,MATLAYO,LACCFLA)
C
      INCLUDE 'ABA_PARAM.INC'
C
      CHARACTER*80 CMNAME,ORNAME
      CHARACTER*3 FLGRAY(15)
      DIMENSION UVAR(NUVARM),DIRECT(3,3),T(3,3),TIME(2)
      DIMENSION ARRAY(15),JARRAY(15),JMAC(*),JMATYP(*),COORD(*)
C
C Error counter:
      JERROR = 0
C Stress tensor:
      CALL GETVRM('S',ARRAY,JARRAY,FLGRAY,JRCD,JMAC,JMATYP,
     1 MATLAYO,LACCFLA)
      JERROR = JERROR + JRCD
      UVAR(1) = ARRAY(1)
      UVAR(2) = ARRAY(2)
      UVAR(3) = ARRAY(3)
      UVAR(4) = ARRAY(4)
      UVAR(5) = ARRAY(5)
      UVAR(6) = ARRAY(6)
C Kinematic shift tensor:
```

```
      CALL GETVRM('ALPHA',ARRAY,JARRAY,FLGRAY,JRCD,JMAC,JMATYP,
     1 MATLAYO,LACCFLA)
      JERROR = JERROR + JRCD
C Calculate the position relative to the center of the
C yield surface:
      UVAR(1) = UVAR(1) - ARRAY(1)
      UVAR(2) = UVAR(2) - ARRAY(2)
      UVAR(3) = UVAR(3) - ARRAY(3)
      UVAR(4) = UVAR(4) - ARRAY(4)
      UVAR(5) = UVAR(5) - ARRAY(5)
      UVAR(6) = UVAR(6) - ARRAY(6)
C If error, write comment to .DAT file:
      IF(JERROR.NE.0)THEN
        WRITE(6,*) 'REQUEST ERROR IN UVARM FOR ELEMENT NUMBER ',
     1      NOEL,'INTEGRATION POINT NUMBER ',NPT
      ENDIF
      RETURN
      END
```

# UWAVE: User subroutine to define wave kinematics for an Abaqus/Aqua analysis.

**Products:** Abaqus/Standard Abaqus/Aqua

**References:**

- *Abaqus/Aqua Analysis*
- **WAVE*

## Overview

User subroutine *UWAVE*:

- will be called at each load integration point for which an Abaqus/Aqua load is specified and a user-defined gravity wave is specified;
- can be used to define the wave kinematics (fluid velocity and acceleration, dynamic pressure, vertical gradient of the dynamic pressure, and the instantaneous fluid surface elevation) as a function of time and space; and
- for stochastic analysis, can be used to determine when during the analysis the current configuration should be retained as the intermediate configuration upon which the wave kinematics are based.

## User Subroutine Interface

```
      SUBROUTINE UWAVE(V,A,PDYN,DPDYNDZ,SURF,LPDYN
     1 LRECOMPUTE,LUPLOCAL,LUPGLOBAL,
     2 LSURF,NDIM,XCUR,XINTERMED,
     3 GRAV,DENSITY,ELEVB,ELEVS,
     4 SEED,NSPECTRUM,FREQWAMP,
     5 TIME,DTIME,NOEL,NPT,KSTEP,KINC)
C
      INCLUDE 'ABA_PARAM.INC'
C
      DIMENSION V(NDIM),A(NDIM),XCUR(NDIM),XINTERMED(NDIM),
     1 FREQWAMP(2,NSPECTRUM),TIME(2)


      user coding to define V, A, PDYN, DPDYNDZ, SURF
      and, if necessary, LUPGLOBAL and LUPLOCAL


      RETURN
      END
```

## Variables to Be Defined

**When LSURF=0**

`V(NDIM)` The total fluid velocity at the current load integration location. This array is passed into *UWAVE* as the steady current velocity. The array should be updated as the sum of the steady current velocity and the velocity contribution from the user-defined wave theory.

`A(NDIM)` The fluid acceleration at the current load integration location.

**When LSURF=0**

PDYN — The dynamic pressure contribution to the total pressure. This variable is needed only for buoyancy loads. The total pressure at a location below the instantaneous surface elevation is the sum of the atmospheric pressure, the hydrostatic pressure measured to the mean fluid elevation, and the dynamic pressure. See *Airy wave theory* and *Stokes wave theory* for definitions of the dynamic pressure for Airy and Stokes waves, respectively.

DPDYNDZ — The gradient of the dynamic pressure in the vertical direction. This variable is needed only for buoyancy loads.

**When LSURF=1**

SURF — The vertical coordinate of the instantaneous fluid surface corresponding to the horizontal position of the load integration point (given in XCUR). If the current location of the load integration point is above the instantaneous surface elevation, no fluid loads will be applied.

**Only in an analysis with stochastic wave kinematics based on an intermediate configuration**

LUPLOCAL — Flag to determine if the intermediate configuration will be updated for this element. This flag can be set only when LRECOMPUTE=1. Return LUPLOCAL as 0 (default) to indicate that the intermediate configuration should not be updated. Return LUPLOCAL as 1 if the intermediate configuration should be updated for this element. The intermediate configuration is stored on an element-by-element basis. Therefore, all integration points for a given element will have their intermediate configuration updated if an update is requested at any one integration point on the element.

LUPGLOBAL — Flag to determine if the intermediate configuration will be updated for all elements. This flag can be set only when LRECOMPUTE=1. Return LUPGLOBAL as 0 (default) to indicate that the intermediate configuration should not be updated. Return LUPGLOBAL as 1 if the intermediate configuration should be updated for all elements with Abaqus/Aqua loads.

## Variables Passed in for Information

LRECOMPUTE — For stochastic analysis LRECOMPUTE=1 indicates that an update to the intermediate configuration is permitted during this call to user subroutine *UWAVE*. The local and global update flags must be set accordingly. If the intermediate configuration is to be updated, the local update flag LUPLOCAL or the global update flag LUPGLOBAL must be set to 1. When LRECOMPUTE=1 and the intermediate configuration needs to be updated, the user subroutine should recompute all wave kinematics information based on the new intermediate configuration. For nonstochastic analysis this flag is always set to 0.

LPDYN — LPDYN=1 indicates that only the dynamic pressure and its gradient need to be calculated (i.e., buoyancy loads). LPDYN=0 indicates that only the fluid velocity and acceleration need to be calculated (i.e., drag or inertia loads).

LSURF — LSURF=1 indicates that subroutine *UWAVE* only needs to return the instantaneous fluid surface elevation. When LSURF=1, no velocity, acceleration, or dynamic pressure needs to be calculated. LSURF=0 indicates that the instantaneous fluid surface elevation SURF is not needed.

| | |
|---|---|
| NDIM | Two or three, indicating that the analysis is in two or three dimensions. The vertical direction is the global $y$-direction in two-dimensional analysis and the global $z$-direction in three-dimensional analysis. |
| XCUR(NDIM) | An array containing the current coordinates of the load integration point. |
| XINTERMED(NDIM) | An array containing the intermediate configuration coordinates of the load integration point. For nonstochastic analysis this array is not used. In a stochastic analysis the wave field is based upon this configuration. At the beginning of each load increment the LRECOMPUTE flag is set to 1 to prompt you for update action. If the intermediate configuration should be replaced by the current configuration, the flag LUPLOCAL should be set to 1 to update the intermediate configuration for this element only or the flag LUPGLOBAL should be set to 1 to update the intermediate configuration for all elements that have Abaqus/Aqua loading. At the beginning of the analysis the intermediate configuration is the reference configuration. |
| GRAV | The user-specified gravitational constant in the fluid variable definition. |
| DENSITY | The user-specified fluid mass density in the fluid variable definition. |
| ELEVB | The user-specified elevation of the seabed in the fluid variable definition. |
| ELEVS | The user-specified elevation of the still fluid level in the fluid variable definition. |
| SEED | For stochastic analysis the user-specified random number seed in the gravity wave definition. |
| NSPECTRUM | For stochastic analysis the number of user-specified frequency versus wave amplitude pairs in the gravity wave definition, used to define the wave spectrum. |
| FREQWAMP(1,NSPECTRUM) | For stochastic analysis the frequency values used to define the wave spectrum. |
| FREQWAMP(2,NSPECTRUM) | For stochastic analysis the wave amplitude values used to define the wave spectrum. |
| TIME(1) | Value of step time at the end of the current increment. |
| TIME(2) | Value of total time at the end of the current increment. |
| DTIME | Time increment. |
| NOEL | Element number. |
| NPT | Load integration point number. All line elements use full integration for the application of external loads. For distributed loads applied to the ends of the element, NPT corresponds to the end number of the element. |
| KSTEP | Step number. |
| KINC | Increment number. |

# UXFEMCRACK: User subroutine to define the interactions between the cracked element surfaces for enriched elements using a small sliding formulation.

**Products:** Abaqus/Standard

**References:**

- *Modeling Discontinuities as an Enriched Feature Using the Extended Finite Element Method*
- **SURFACE INTERACTION*

## Overview

User subroutine *UXFEMCRACK*:

- can be used to specify the crack opening offset for contact of cracked element surfaces;
- can also be used to define the mechanical (normal and shear) interactions between the cracked element surfaces;
- is called at all the points of intersection between the crack surface and the element edges;
- can include the effects of slurry concentration and the proppant particle size; and
- is available only for contact of cracked enriched element surfaces using a small-sliding formulation.

## User Subroutine Interface

```
 SUBROUTINE UXFEMCRACK( OFFSETOLD, OFFSETNEW, DELTA,
     1 OPENMAX, NOEL, NPT, COORDS, KSTEP, KINC, TIME1, DTIME1, TEMP, DTEMP,

     2 PREDEF, DPRED, NFIELD, STATEV, NSTATV, PROPS, NPROPS, JMAC, JMATYP,

     3 MATLAYO, LACCFLA, SDEG, GAPINI, DV, STRESSOLD, STRESSNEW,
     4 STIFFNESS, DIAMETERP, CV, JFLGOPT, NTENS, NDIR, NSHR, MCRD,
     5 NIARRAY, I_ARRAY, NRARRAY, R_ARRAY, NCARRAY, C_ARRAY)
C
      INCLUDE 'ABA_PARAM.INC'
C
      DIMENSION COORDS(*), TIME1(*), TEMP(*), JMAC(*), JMATYP(*),
STRESSOLD(MCRD),
     1 STRESSNEW(MCRD), STIFFNESS(MCRD,MCRD), DELTA(*), PROPS(NPROPS),
     2 STATEV(NSTATV), PREDEF(NFIELD), DPRED(NFIELD), I_ARRAY(NIARRAY),
     3 R_ARRAY(NRARRAY), C_ARRAY(NCARRAY)

C
      user coding to define JFLGOPT, OFFSETNEW, STIFFNESS, STRESSNEW,
      and, if necessary, STATEV

      RETURN
      END
```

## Variables to Be Defined

| | |
|---|---|
| OFFSETNEW | The offset at the end of the increment, $\delta_{off}$. If this value is not defined, it is assumed to be zero. |
| JFLGOPT | An integer flag that must be set to either 1 or 2. |

Set JFLGOPT=1 to update OFFSETNEW and the penalty stiffness matrix, STIFFNESS.

Set JFLGOPT=2 to update OFFSETNEW; the penalty stiffness matrix, STIFFNESS; and the stress at the end of the increment, STRESSNEW.

## Variables That Can Be Updated

STIFFNESS Penalty stiffness of the cracked surfaces.

STIFFNESS is initialized to a penalty stiffness, based on the bulk stiffness of the underlying elements and the element characteristic length, before *UXFEMCRACK* is called. Abaqus uses these values for all subsequent calculations if STIFFNESS is not updated in *UXFEMCRACK*.

STRESSNEW Traction vector on the cracked surfaces.

This variable is initialized to the nominal traction vector at the beginning of the increment and can be updated to the corresponding values at the end of the increment.

If you set JFLGOPT=1, Abaqus computes STRESSNEW based on the offset and the stiffness matrix, STIFFNESS. Any updates you make to STRESSNEW in this case are ignored.

If you set JFLGOPT=2, you must update STRESSNEW.

STATEV An array containing the user-defined solution-dependent state variables at this point. This array will be passed in containing the values of these variables at the start of the increment unless the values are updated in user subroutine *USDFLD*. They can be updated in this subroutine to their values at the end of the increment. The size of the array is defined as described in *Allocating Space for Element Solution-Dependent Variables*.

## Variables Passed in for Information

OFFSETOLD Offset at the end of the previous increment.

OPENMAX $\delta_{off}^{\max}$, the maximum value of $\delta_{off}$ at an interaction point between the crack surface and the element edge over the deformation history.

NDIR Number of direct components of the underlying enriched element at this contact point.

NSHR Number of shear components of the underlying enriched element at this contact point.

NTENS Number of total components (NDI + NSHR) of the underlying enriched element at this contact point.

MCRD Number of coordinate directions at the contact point.

DELTA Current crack opening

NOEL Element number.

NPT Integration point number.

COORDS An array containing the current coordinates of this point.

KSTEP Step number.

KINC Increment number.

TIME1(1) Value of step time at the beginning of the current increment.

TIME1(2) Value of total time at the beginning of the current increment.

DTIME1 Time increment.

TEMP Temperature at the start of the increment.

DTEMP Increment of temperature during the time increment.

PREDEF An array containing the values of all of the user-specified predefined variables at this point at the start of the increment .

| | |
|---|---|
| DPRED | An array containing the increments of all of the predefined variables during the time increment. |
| NFIELD | Number of user-specified predefined variables. |
| NSTATV | Number of solution-dependent state variables that are associated with this material type (defined as described in *Allocating Space for Element Solution-Dependent Variables*). |
| PROPS(NPROPS) | User-specified array of material constants associated with this material. |
| NPROPS | User-defined number of material constants associated with this material. |
| JMAC | Variable that must be passed into the GETVRM utility routine to access an output variable. |
| JMATYP | Variable that must be passed into the GETVRM utility routine to access an output variable. |
| MATLAYO | Variable that must be passed into the GETVRM utility routine to access an output variable. |
| LACCFLA | Variable that must be passed into the GETVRM utility routine to access an output variable. |
| SDEG | Damage variable at the end of the last increment. |
| STRESSOLD | Stresses at the end of previous increment. |
| DV | Integration point volume. |
| DIAMETERP | Diameter of slurry particles |
| CV | Slurry concentration at the end of previous increment. |
| GAPINI | Initial crack opening. |
| NIARRAY | Size of array I_ARRAY. |
| I_ARRAY | Not used. |
| NRARRAY | Size of array R_ARRAY. |
| R_ARRAY | Not used. |
| NCARRAY | Size of array C_ARRAY. |
| C_ARRAY | Not used. |

## Example: User-Defined Offset for Enriched Cracked Element Surfaces

The following is a simple example of how to specify the offset for cracked element surfaces by using *UXFEMCRACK*:

```
SUBROUTINE UXFEMCRACK( OFFSETOLD, OFFSETNEW, DELTA,
     1     OPENMAX, NOEL, NPT, COORDS, KSTEP, KINC, TIME1, DTIME1, TEMP,
DTEMP,
     2     PREDEF, DPRED, NFIELD, STATEV, NSTATV, PROPS, NPROPS, JMAC,
JMATYP,
     3     MATLAYO, LACCFLA, SDEG, GAPINI, DV, STRESSOLD, STRESSNEW,
     4     STIFFNESS, DIAMETERP, CV, JFLGOPT, NTENS, NDIR, NSHR, MCRD,
     5     NIARRAY, I_ARRAY, NRARRAY, R_ARRAY, NCARRAY, C_ARRAY )

      INCLUDE 'ABA_PARAM.INC'

C
      DIMENSION COORDS(*), TIME1(*), TEMP(*), JMAC(*), JMATYP(*),
STRESSOLD(*),
     1 STRESSNEW(MCRD), STIFFNESS(MCRD,MCRD), DELTA(*),
PROPS(NPROPS),STATEV(NSTATV),
     2 PREDEF(NFIELD),DPRED(NFIELD), I_ARRAY(NIARRAY), R_ARRAY(NRARRAY),
     3 C_ARRAY(NCARRAY)

CC   THIS EXAMPLE DEMONSTRATES HOW TO SPECIFY AN OFFSET IN STEP 3 TO KEEP
THE
```

```
CC   FRACTURE "PROPPED OPEN" WHEN EXTERNAL LOADING CONDITIONS WOULD
CC   OTHERWISE HAVE LED TO CLOSURE OF THE FRACTURE


     JFLGOPT = 1
     IF( KSTEP .LE. 2 ) THEN
         OFFSETNEW = 0.0
     ELSE
        OFFSETNEW = 0.15
 IF( OPENMAX .gt. OFFSETNEW .and. DELTA(1) .gt. OFFSETNEW ) THEN
    DO II = 1,MCRD
              STIFFNESS(II,II) = 0.0
           END DO
        END IF
     END IF
C
     RETURN
     END
```

# UXFEMNONLOCALWEIGHT: User subroutine to define the weight function used to compute the average stress/strain to determine the crack propagation direction.

**Products:** Abaqus/Standard

**References:**

- *Modeling Discontinuities as an Enriched Feature Using the Extended Finite Element Method*
- *About Progressive Damage and Failure*
- **DAMAGE INITIATION*

## Overview

User subroutine *UXFEMNONLOCALWEIGHT*:

- can be used to specify a user-defined weight function; and
- is currently available only for enriched elements.

## User Subroutine Interface

```
SUBROUTINE UXFEMNONLOCALWEIGHT(WEIGHT, JELNO, NPT, COORDS,
     & CRACKTIPCOORD, NNCRD, RADIUS, KSTEP, KINC, TIME)
C

      INCLUDE 'ABA_PARAM.INC'

C
      DIMENSION TIME(2), COORDS(NNCRD), CRACKTIPCOORD(NNCRD)
      user coding to define weight
      RETURN
      END
```

## Variables to Be Defined

| | |
|---|---|
| `weight` | A scalar weight function used to compute the average stress/strain at the crack tip. |

## Variables Passed in for Information

| | |
|---|---|
| `JELNO` | Element number. |
| `NPT` | Integration point number. |
| `COORDS` | An array containing the current coordinates of this integration point. |
| `CRACKTIPCOORDS` | An array containing the current coordinates of the crack tip. |
| `NNCRD` | Dimension of the model. |
| `RADIUS` | Influence radius in which the elements are included for averaging. |
| `KSTEP` | Step number. |
| `KINC` | Increment number. |
| `TIME(1)` | Value of step time at the beginning of the current increment. |

| | |
|---|---|
| `TIME(2)` | Value of total time at the beginning of the current increment. |

# VOIDRI: User subroutine to define initial void ratios.

**Products:** Abaqus/Standard

**References:**

- *Initial Conditions*
- *Coupled Pore Fluid Diffusion and Stress Analysis*
- **INITIAL CONDITIONS*

## Overview

User subroutine *VOIDRI*:

- will be called to define initial void ratio values at material calculation points of continuum elements (see *About the Element Library*) in a porous medium whenever a user-defined initial condition on void ratio is specified; and
- can be used to define initial void ratio values as functions of material point coordinates and/or element numbers.

## User Subroutine Interface

```
      SUBROUTINE VOIDRI(EZERO,COORDS,NOEL)
C
      INCLUDE 'ABA_PARAM.INC'
C
      DIMENSION COORDS(3)
C


      user coding to define EZERO


      RETURN
      END
```

## Variables to Be Defined

| | |
|---|---|
| EZERO | Initial void ratio. |

## Variables Passed in for Information

| | |
|---|---|
| COORDS | An array containing the current coordinates of this point. |
| NOEL | Element number. |

# Abaqus/Explicit User Subroutines

Abaqus/Explicit user subroutines increase the functionality of several Abaqus capabilities for which the usual data input methods alone may be too restrictive. They provide an extremely powerful and flexible tool for analysis.

**In this section:**


- *VDFLUX*
- *VDISP*
- *VDLOAD*
- *VEXTERNALDB*
- *VFABRIC*
- *VFILM*
- *VFRIC*
- *VFRIC_COEF*
- *VFRICTION*
- *VHETVAL*
- *VSDVINI*
- *VUAMP*
- *VUANISOHYPER_INV*
- *VUANISOHYPER_STRAIN*
- *VUCHARLENGTH*
- *VUCREEPNETWORK*
- *VUEL*
- *VUEOS*
- *VUEXPAN*
- *VUFIELD*
- *VUFLUIDEXCH*
- *VUFLUIDEXCHEFFAREA*
- *VUGENS*
- *VUHARD*
- *VUHYPER*
- *VUHYPER_STRETCH*
- *VUINTER*
- *VUINTERACTION*
- *VUMAT*
- *VUMATHT*
- *VUMULLINS*
- *VUSDFLD*
- *VUSUPERELASHARDMOD*
- *VUTEMP*
- *VUTRS*
- *VUTRSNETWORK*

- *VUVISCOSITY*
- *VWAVE*

# VDFLUX: User subroutine to specify nonuniform distributed fluxes in an explicit dynamic coupled temperature-displacement analysis.

**Products:** Abaqus/Explicit

**References:**

- *Thermal Loads*
- **DFLUX*
- **DSFLUX*

## Overview

User subroutine *VDFLUX*:

- can be used to define the variation of the distributed flux as a function of position, temperature, time, velocity, element number, etc. for a group of points in a dynamic coupled thermal-stress analysis using explicit integration (for more information, see *Fully Coupled Thermal-Stress Analysis*);
- will be called at each flux integration point associated with each element-based or surface-based nonuniform distributed flux definition in the analysis; and
- recognizes an amplitude reference (*Amplitude Curves*) if it appears with the associated nonuniform flux definition.

## User Subroutine Interface

```
      subroutine vdflux (
C Read only (unmodifiable)variables –
     1 nblock, ndim, kStep, kIncr, stepTime, totalTime, jUid,
     2 amplitude, temp, curCoords, velocity, dirCos, jltyp, sname,
C Write only (modifiable) variable –
     1 value )
C
      include 'vaba_param.inc'
C
      dimension curCoords(nblock,ndim), velocity(nblock,ndim),
     1  jUid(nblock), dirCos(nblock,ndim,ndim), temp(nblock),
     2  value(nblock)
      character*80 sname
C

      do 100 km = 1, nblock
        user coding to define value

  100 continue

      return
      end
```

## Variables to Be Defined

`value(nblock)` Magnitude of the distributed flux. Units are $JT^{-1}L^{-2}$ for surface fluxes and $JT^{-1}L^{-3}$ for body fluxes.

## Variables Passed in for Information

`nblock` Number of points to be processed in this call to `VDFLUX`.

`ndim` Number of coordinate directions: 2 for two-dimensional models, 3 for three-dimensional models. The model is considered three-dimensional if any three-dimensional elements are defined.

`kStep` Step number.

`kIncr` Increment number.

`stepTime` Value of time since the step began.

`totalTime` Value of total time. The time at the beginning of the step is given by `totalTime – stepTime`.

`jUid` User-defined element numbers.

`amplitude` Current value of the amplitude referenced for this flux (set to unity if no amplitude is referenced). You must multiply the flux by the current amplitude value within the user subroutine if the amplitude is required.

`TEMP` Current value of temperature at this integration point.

`curCoords(nblock, ndim)` Current coordinates of each point for which the flux is to be prescribed.

`velocity(nblock, ndim)` Current velocity of each point for which the flux is to be prescribed.

`dirCos(nblock, ndim, ndim)` Current orientation of the face or edge (not applicable for body flux type loads). The second dimension indicates the vector, and the third dimension indicates the components of that vector. For faces (surface fluxes on three-dimensional continuum and shell elements) the first and second vectors are the local directions in the plane of the surface and the third vector is the normal to the face, as defined in *Conventions*. For solid elements the normal points inward, which is the opposite of what is defined in the conventions; for shell elements the normal definition is consistent with the defined conventions. For edges (fluxes on two-dimensional continuum elements) the first vector is the normal to the edge; the second vector is the tangent to the edge; and, if `ndim`=3, the third vector is a unit normal in the out-of-plane direction.

`jltyp` Key that identifies the distributed flux type. The load type may be a body flux, a surface-based flux, or an element-based surface flux. For element-based surface fluxes this variable identifies the element face for which this call to *VDFLUX* is being made. See *About the Element Library* for element load type identification. This information is useful when several different nonuniform distributed loads are being imposed on an element at the same time. The key is as follows:

| `jltyp` | Load type |
|---|---|
| 0 | Surface-based load |
| 1 | BFNU |
| 11 | S1NU or SNEGNU |
| 12 | S2NU or SPOSNU |
| 13 | S3NU |
| 14 | S4NU |
| 15 | S5NU |
| 16 | S6NU |

| | |
|---|---|
| `sname` | Surface name for a surface-based flux load definition (`JLTYP`=0). For a body flux or an element-based face load the surface name is passed in as a blank. |

# VDISP: User subroutine to specify prescribed boundary conditions.

**Products:** Abaqus/Explicit

**References:**

- *Boundary Conditions*
- **BOUNDARY*
- *VDISP*

## Overview

User subroutine *VDISP*:

- can be used to prescribe translational and rotational boundary conditions;
- is called for all degrees of freedom listed in the associated boundary condition;
- allows you to specify values for either the degree of freedom or its time derivatives such as velocity and acceleration;
- releases the boundary condition by default if you do not specify a value for the boundary condition unless the degree of freedom is also participating in a constraint;
- can be used to apply a concentrated load, instead, by adjusting the default motion of the node;
- can be called for blocks of nodes for which the boundary conditions are defined in the subroutine.

## Initial Velocity

At the beginning of each step user subroutine *VDISP* is called once to establish the initial velocity; and then, it is called once on each configuration, including the initial configuration, to establish the nodal acceleration.

The first call to user subroutine *VDISP* is made to establish the initial velocity, which is indicated by the passing of a step time value of $-dt$ into the subroutine, where $dt$ is the current time increment. If displacement is prescribed, the returned variable, `rval`, corresponds to $u_o - v_o dt$, where $u_o$ and $v_o$ are the initial displacement and velocity respectively. If velocity is prescribed, the returned variable corresponds to the initial velocity $v_o$.

If acceleration is prescribed, the returned variable corresponds to $\frac{v_o}{dt}$, where $v_o$ is the initial velocity.

The default value of `rval` is consistent with the velocity at the end of previous step or that specified as an initial condition in case of the first step. You only need to reset the `rval` if a different initial velocity is desired. The arrays `u` and `v` stand for the default initial displacement and velocity, respectively. The array `a` contains a zero value.

## Acceleration

During time incrementation user subroutine *VDISP* is called once for each configuration, including the initial configuration, to establish the nodal acceleration.

If displacement is prescribed, the returned variable should be set equal to the displacement at `stepTime+dtNext`, where `stepTime` is the step time and `dtNext` is the next time increment. If velocity is prescribed, the returned variable should be set equal to the mean velocity at `stepTime+dtNext/2`. If acceleration is prescribed, the returned variable should be set equal to the acceleration at `stepTime`. Note that `stepTime` is zero for the initial configuration.

The variable rval has a default value that is computed with the constraints not yet applied. You can leave rval at this value, if desired. The variable u contains values at stepTime. Whereas, the variable v contains initial velocity when stepTime is zero and, otherwise, velocity at stepTime−dt/2. The variable a contains values at stepTime computed with the constraints not yet applied.



**Tip:** If the node does not participate in any other conflicting constraints or connector elements, leaving rval at the incoming value would amount to leaving the node unconstrained. Alternatively, if you want to apply a concentrated load at such a node (instead of the boundary condition), you can compute the change in acceleration due to this load and modify the rval value to account for that change. The nodal mass and the rotary inertia are available in *VDISP* for computing the change in acceleration. In addition, the default value of rval at the node already reflects the forces from any contact, user-defined load, and elements (except for connectors) at that node.

## User Subroutine Interface

```
      subroutine vdisp(
c Read only variables –
     1   nblock, nDof, nCoord, kstep, kinc,
     2   stepTime, totalTime, dtNext, dt,
     3   cbname, jBCType, jDof, jNodeUid, amp,
     4   coordNp, u, v, a, rf, rmass, rotaryI,
c Write only variable –
     5   rval )
c
      include 'vaba_param.inc'
c
      character*80 cbname
      dimension jDof(nDof), jNodeUid(nblock),
     1          amp(nblock), coordNp(nCoord,nblock),
     2          u(nDof,nblock), v(nDof,nblock), a(nDof,nblock),
     3          rf(nDof,nblock), rmass(nblock), rotaryI(3,3,nblock),
     4          rval(nDof,nblock)
c
      do 100 k = 1, nblock
      do 100 j = 1, nDof
         if( jDof(j) .gt. 0 ) then
            user coding to define rval(j, k)
         end if
  100 continue
c
      return
      end
```

## Variables to Be Defined

| | |
|---|---|
| rval(nDof, nblock) | Values of the prescribed variable for degrees of freedom 1–6 (translation and rotation) at the nodes. The variable can be displacement, velocity, or acceleration, depending on the type specified in the associated boundary condition. The variable type is indicated by jBCType. The variable rval has a default value that is computed without taking the constraints into account. You may reset the rval. |

## Variables Passed in for Information

| | |
|---|---|
| nblock | Number of nodal points to be processed in this call to *VDISP*. |
| nDof | Number of degrees of freedom (equals 6). |

| | |
|---|---|
| `nCoord` | Number of coordinate components (equals 3). |
| `kstep` | Step number. |
| `kinc` | Increment number. |
| `stepTime` | Value of time since the step began. |
| `totalTime` | Value of total time. The time at the beginning of the step is given by `totalTime-stepTime`. |
| `dtNext` | Next time increment size. |
| `dt` | Current time increment size. |
| `cbname` | User-specified name corresponding to the associated boundary condition. |
| `jBCType` | Indicator for type of prescribed variable: 0 for displacement, 1 for velocity, and 2 for acceleration. |
| `jDof(nDof)` | Indicator for prescribed degrees of freedom. The values given by `rval(j,k)` are prescribed only if `jDof(j)` equals 1. |
| `jNodeUid(nblock)` | Node numbers. |
| `amp(nblock)` | Amplitude values corresponding to the associated amplitude functions. These values are passed in for information only and will not contribute to the values of the prescribed variable automatically. |
| `coordNp(nCoord, nblock)` | Nodal point coordinates. |
| `u(nDof, nblock)` | Initial displacements when `stepTime` is negative, and, otherwise, displacement at `stepTime`. All translations are included if one or more translational degrees of freedom are prescribed. All rotations are included if one or more rotational degrees of freedom are prescribed. |
| `v(nDof, nblock)` | Initial nodal velocities when `stepTime` is non-positive and, otherwise, mean velocities at `stepTime-dt/2` during time incrementation. All translational velocities are included if one or more translational degrees of freedom are prescribed. All angular velocities are included if one or more rotational degrees of freedom are prescribed. |
| `a(nDof, nblock)` | Contains a zero value when `stepTime` is negative and, otherwise, the accelerations, computed without accounting for the boundary condition, at `stepTime`. All translational accelerations are included if one or more translational degrees of freedom are prescribed. All angular accelerations are included if one or more rotational degrees of freedom are prescribed. |
| `rf(nDof, nblock)` | Nodal point reaction at `stepTime-dt`. All reaction forces are included if one or more translational degrees of freedom are prescribed. All reaction moments are included if one or more rotational degrees of freedom are prescribed. |
| `rmass(nblock)` | Nodal point masses. |
| `rotaryI(3, 3, nblock)` | Nodal point rotary inertia. |

## Example: Imposition of Acceleration on a Rigid Body with Nonzero Initial Velocity

In this example a sinusoidal acceleration is imposed on the reference node of a rigid body. Nonzero initial velocity is also specified for the rigid body. User subroutine *VDISP* given below illustrates how the return value array is to be computed for different phases of the solution. The analysis results show that both the initial velocity and acceleration are correctly specified.

## Input file

```
*HEADING
 Test VDISP with S4R element
*NODE, NSET=NALL
 1,
 2, 2., 0.
 3, 0., 2.
 4, 2., 2.
 9, 1., 1., 0.
*ELEMENT, TYPE=S4R, ELSET=SHELL
 10, 1,2,4,3
*SHELL SECTION, ELSET=SHELL, MATERIAL=ELSHELL
  2.0000000e-02,     3
*MATERIAL, NAME=ELSHELL
*DENSITY
7850.0,
*ELASTIC
  2.5000000e+11,   3.0000000e-01
*RIGID BODY, REF NODE=9, ELSET=SHELL
*INITIAL CONDITIONS, Type=VELOCITY
 9, 1, 0.4
*STEP
*DYNAMIC, EXPLICIT, DIRECT USER CONTROL
 0.01, 0.8
*BOUNDARY, USER, TYPE=ACCELERATION
 9, 1
*OUTPUT, HISTORY, TIME INTERVAL=0.01, OP=NEW
*NODE OUTPUT, NSET=NALL
 U, V, A
*END STEP
```

## User subroutine

```
      subroutine vdisp(
c Read only variables -
     *   nblock, nDof, nCoord, kstep, kinc,
     *   stepTime, totalTime, dtNext, dt,
     *   cbname, jBCType, jDof, jNodeUid, amp,
     *   coordNp, u, v, a, rf, rmass, rotaryI,
c Write only variable -
     *   rval )
c
      include 'vaba_param.inc'
      parameter( zero = 0.d0, half = 0.5d0, one = 1.d0 )
c
      character*80 cbname
      dimension jDof(nDof), jNodeUid(nblock),
     *   amp(nblock), coordNp(nCoord,nblock),
     *   u(nDof,nblock), v(nDof,nblock), a(nDof,nblock),
     *   rf(nDof,nblock), rmass(nblock),
     *   rotaryI(3,3,nblock), rval(nDof,nblock)
c
c     Impose acceleration
c
      if( jBCType .eq. 2 ) then
c
         if( stepTime .lt. zero ) then
c
```

```
c           Initialization 1
c
            do 310 k=1, nblock
            do 310 j=1, nDof
               if ( jDof(j) .gt. 0 ) then
                  v0 = v(j,k)
                  rval(j,k)  = v0/dt
               end if
310         continue
c
         else
c
c           Time incrementation
c
            amplitude = 2.0
            period = 0.8
            twopi = 6.2831853d0
c
            do 350 k=1, nblock
            do 350 j=1, nDof
               if ( jDof(j) .gt. 0 ) then
                  rval(j,k) = amplitude*
*                             sin( twopi*stepTime / period )
               end if
350         continue
         end if
      end if
c
      return

end
```

# VDLOAD: User subroutine to specify nonuniform distributed loads.

**Products:** Abaqus/Explicit

**References:**

- *About Loads*
- *Distributed Loads*
- **DLOAD*
- **DSLOAD*
- *Deformation of a sandwich plate under CONWEP blast loading*

## Overview

User subroutine *VDLOAD*:

- can be used to define the variation of the distributed load magnitude as a function of position, time, velocity, displacement, acceleration, etc. for a group of points, each of which appears in an element-based or surface-based nonuniform load definition;
- will be called for load integration points associated with each nonuniform load definition including PENU and PINU loads applicable for pipe elements;
- does not make available the current value of the nonuniform distributed loads for file output purposes; and
- recognizes an amplitude reference (*Amplitude Curves*) if it appears with the associated nonuniform load definition.

## User Subroutine Interface

```
      subroutine vdload (
C Read only (unmodifiable)variables -
     1 nBlock, ndim, stepTime, totalTime,
     2 amplitude, curCoords, vua, dirCos, jltyp, sname,
C Write only (modifiable) variable -
     1 value )
C
      include 'vaba_param.inc'
C
      dimension curCoords(nBlock,ndim), vua(nBlock,ndim,3),
     1  dirCos(nBlock,ndim,ndim), value(nBlock)
      character*80 sname
C

      do 100 km = 1, nBlock
        user coding to define value

  100 continue

      return
      end
```

## Variables to Be Defined

`value (nBlock)` Magnitude of the distributed load. Units are $FL^{-2}$ for surface loads, $FL^{-3}$ for body forces.

## Variables Passed in for Information

`nBlock` Number of points to be processed in this call to `VDLOAD`.

`ndim` Number of coordinate directions: 2 for two-dimensional models, 3 for three-dimensional models. The model will be considered three-dimensional if any three-dimensional elements are defined (including SPRINGA elements).

`stepTime` Value of time since the step began.

`totalTime` Value of total time. The time at the beginning of the step is given by `totalTime – stepTime`.

`amplitude` Current value of the amplitude referenced for this load (set to unity if no amplitude is referenced). You must multiply the load by the current amplitude value within the user subroutine if the amplitude is required.

`curCoords (nBlock, ndim)` Current coordinates of each point for which the load is to be calculated.

`vua(nBlock, ndim,3)` Current velocity, displacement, and previous acceleration at each point for which the load is to be calculated. The array is defined as `vua(1,1,1)` for the first component of the velocity array, `vua(1,1,2)` for the first component of the displacement array, and `vua(1,1,3)` for the first component of the acceleration array. The acceleration values correspond to the end of the previous increment.

`dirCos (nBlock, ndim, ndim)` Current orientation of the face, edge, pipe, or beam for pressure type loads (not applicable for body force type loads). The second dimension indicates the vector, and the third dimension indicates the components of that vector. For faces (pressures on three-dimensional continuum, shell, and membrane elements), the first and second vectors are the local directions in the plane of the surface and the third vector is the normal to the face, as defined in *Conventions*. For solid elements the normal points inward, which is the opposite of what is defined in the conventions; for shell elements the normal definition is consistent with the defined conventions. For edges (pressures on two-dimensional continuum elements and two-dimensional beams and pipes), the first vector is the normal to the edge, the second vector is the tangent to the edge, and, if `ndim`=3, the third vector will be a unit normal in the out-of-plane direction. For three-dimensional beam and pipe elements, the first and second vectors are the local axes ($\mathbf{n_1}$, $\mathbf{n_2}$) and the third vector is the tangent vector ($\mathbf{t}$), as defined in *Beam Element Cross-Section Orientation*.

For a discrete element analysis using PD3D elements, the first column of the array is the radius of the element.

`jltyp` Key that identifies the distributed load type. The load type might be a body force, a surface-based load, or an element-based surface load. For element-based surface loads, this variable identifies the element face for which this call to *VDLOAD* is being made. See *About the Element Library* for element load type identification. This information is useful when several different nonuniform distributed loads are being imposed on an element at the same time. The key is as follows:

| Jltype | Load type |
|---|---|
| 0 | Surface-based load |
| 1 | BXNU |

| Jltype | Load type |
|---|---|
| 2 | BYNU |
| 3 | BZNU |
| 20 | PNU |
| 21 | P1NU |
| 22 | P2NU |
| 23 | P3NU |
| 24 | P4NU |
| 25 | P5NU |
| 26 | P6NU |
| 27 | PINU |
| 28 | PENU |
| 41 | PXNU |
| 42 | PYNU |
| 43 | PZNU |

`sname` Surface name for a surface-based load definition (`JLTYP`=0). For a body force or an element-based load the surface name is passed in as a blank.

# VEXTERNALDB: User subroutine that gives control to the user at key moments of the analysis so that data can be exchanged dynamically among Abaqus user subroutines and with external programs or files.

**Products:** Abaqus/Explicit

**References:**

- *About User Subroutines and Utilities*

## Overview

User subroutine *VEXTERNALDB*:

- is called once at the beginning of the analysis, at the beginning of each step, before each increment, at the start of each increment, at the end of each increment, at the end of each step, and finally at the end of the analysis;
- is called on each active thread or process in the case of parallel execution in the sequence outlined above;
- can be used to communicate data between external programs and user subroutines within Abaqus/Explicit;
- can be used to control the time incrementation of the Abaqus/Explicit analysis;
- can be used to control the output of the restart data for the analysis;
- can be used either to skip the remainder of an Abaqus step or to terminate the analysis;
- can be used to open and close external files as needed for exchange of data with the Abaqus analysis;
- can be used to exchange data with other user subroutines, called either on the given thread or the other threads in the same MPI process, via user-allocated global and thread-local arrays (see *Allocatable Arrays*);
- can be used to exchange data with other Abaqus processes via an MPI mechanism (see *Obtaining Parallel Processes Information*) in parallel analyses; and
- can be used to request additional field output frames during the analysis.

## Dynamic Exchange of Data with Other Abaqus User Subroutines and External Programs

Typically, Abaqus user subroutines are called with the context data limited to a specific material point, a specific element, etc. Rarely, you need to know some nonlocal information such as the state of the neighboring material points or elements. In other situations, you want to specify the behavior in the user subroutines to depend dynamically on the external programs. Both these complex scenarios can be addressed using user subroutine *VEXTERNALDB*.

*VEXTERNALDB* is called once at the beginning of the analysis, at the beginning of each step, before each increment, at the start of each increment, at the end of each increment, at the end of each step, and finally at the end of the analysis. Other Abaqus subroutines are called after the call to user subroutine *VEXTERNALDB* at the start of the increment but before the next call at the end of that increment.

## User Subroutine Interface

```
      subroutine vexternaldb(lOp, i_Array, niArray, r_Array, nrArray)
C
      include 'vaba_param.inc'
c
C     Contents of i_Array
```

```
      parameter( i_int_nTotalNodes     = 1,
     *           i_int_nTotalElements  = 2,
     *           i_int_kStep           = 3,
     *           i_int_kInc            = 4,
     *           i_int_iStatus         = 5,
     *           i_int_lWriteRestart   = 6,
     *           i_int_ExtraOutputFrame= 7 )

C     Possible values for the lOp argument
      parameter( j_int_StartAnalysis    = 0,
     *           j_int_StartStep        = 1,
     *           j_int_SetupIncrement   = 2,
     *           j_int_StartIncrement   = 3,
     *           j_int_EndIncrement     = 4,
     *           j_int_EndStep          = 5,
     *           j_int_EndAnalysis      = 6 )


C     Possible values for i_Array(i_int_iStatus)
      parameter( j_int_Continue          = 0,
*               j_int_TerminateStep     = 1,
*               j_int_TerminateAnalysis = 2)

C     Contents of r_Array
      parameter( i_flt_TotalTime   = 1,
     *           i_flt_StepTime    = 2,
     *           i_flt_dTime       = 3 )
C
      dimension i_Array(niArray),
     *   r_Array(nrArray)

      kStep = i_Array(i_int_kStep)
      kInc  = i_Array(i_int_kInc)
```

*Note that you can use the MPI communication between parallel Abaqus processes to gather and scatter the data.*

```
C     Start of the analysis
      if (lOp .eq. j_int_StartAnalysis) then
```

*User coding to set up the environment, open files, launch/connect to the external programs, etc.*

```
C        continuation from a previous analysis (restart)
         if (kStep .ne. 0) then
         end if

C     Start of the step
      else if (lOp .eq. j_int_StartStep) then
```

*Set up or exchange (import and export) initial values with external programs.*

```
C         The initial values may need to match those at the
          point of restart.
          if ( kInc .ne. 0) then
          end if


C     Setup the increment
      else if (lOp .eq. j_int_SetupIncrement) then
```

```
      Change i_Array(i_int_lWriteRestart) and
             i_Array(i_int_iStatus) if desired.
      Change r_Array(i_flt_dTime) if desired.

C     Start of the increment
      else if (lOp .eq. j_int_StartIncrement) then

      The time increment is finalized.  Use r_Array(i_flt_dTime) if desired.
      If needed, gather and export data  from the configuration at the end of the previous
 increment to external programs.
      Import and scatter data from external program to influence the current Abaqus increment.

C     End of the increment
      else if (lOp .eq. j_int_EndIncrement) then

      Change i_Array(i_int_iStatus) if desired.
      Gather and export data  from the configuration at the end of the current increment
      to external programs.

C     End of the step
      else if (lOp .eq. j_int_EndStep) then

      In the case of multiple steps, prepare the transition to the next step.
      For example, these data can serve as initial values for the next step.

C     End of the analysis
      else if (lOp .eq. j_int_EndAnalysis) then

      User coding to close  files and disconnect any external programs, etc.

      end if

      return
      end
```

## Variables to Be Defined

None.

## Variables That Can Be Updated

`i_Array(i_int_lWriteRestart)` `i_Array(i_int_lWriteRestart)` indicates whether restart data are currently scheduled to be written. When `lOp=j_int_SetupIncrement`, you can optionally modify it either to write restart data or to skip it. A value of 1 would capture the data for a possible future restart of the analysis from the current time point; whereas 0 would forego such restart from the current time point.

`i_Array(i_int_ExtraFrameOutput)` `i_Array(i_int_ExtraFrameOutput)` indicates whether an on-demand output request containing all the field-type requests is written. The value is always passed in as 0. Resetting to a value of 1 triggers the on-demand output. The on-demand output is triggered in the subsequent increment. If an on-demand output was already triggered for this increment, that output will satisfy

this request and on-demand output will not be triggered in the subsequent increment.

`i_Array(i_int_iStatus)` `i_Array(i_int_iStatus)` indicates the status of the analysis and has a default value of `j_int_Continue`. When `lOp=j_int_SetupIncrement` or `j_int_EndIncrement`, you can optionally modify it either to a value of `j_int_TerminateStep` to skip the remainder of the current step or to a value of `j_int_TerminateAnalysis` to terminate the analysis. If you request to terminate the analysis, the analysis will go through one additional increment with a zero time increment size to generate the field output that reflects the state at termination, as described in *Abaqus/Explicit Output as a Result of Analysis Termination*. When the passed in value is not equal to `j_int_Continue`, you can coordinate the necessary events with any external program.

`r_Array(i_flt_dTime)` Time increment. When `lOp=j_int_SetupIncrement`, it is the time increment proposed for the current increment and it can be modified to control the incrementation. When `lOp=j_int_StartIncrement`, it is the finalized time increment for the increment to be taken; whereas when `lOp=j_int_EndIncrement`, it is the time increment just taken.

## Variables Passed in for Information

`i_Array(i_int_nTotalNodes)` Total number of nodes in the model.

`i_Array(i_int_nTotalElements)` Total number of elements in the model.

`i_Array(i_int_kStep)` Current step number. When `lOp=j_int_StartAnalysis`, `i_Array(i_int_kStep)` gives the restart step number.

`i_Array(i_int_kInc)` Current increment number. When `lOp=j_int_StartStep`, `i_Array(i_int_kInc)` gives the restart increment number.

`lOp` `lOp=j_int_StartAnalysis` indicates that the user subroutine is being called at the start of the analysis. A nonzero `i_Array(i_int_kStep)` indicates that the analysis is starting from a prior analysis (restart).

`lOp=j_int_StartStep` indicates that the user subroutine is being called at the start of a step. A nonzero `i_Array(i_int_kInc)` indicates a continuation of the step from a prior analysis (restart).

`lOp=j_int_SetupIncrement` indicates that the user subroutine is being called to set up an increment and `r_Array(i_flt_dTime)` can be modified. In addition, `i_Array(i_int_lWriteRestart)` can be modified to control output of restart data at the end of the current increment. You can also control the continuation of the analysis via `i_Array(i_int_iStatus)`.

`lOp=j_int_StartIncrement` indicates that the user subroutine is being called at the start of the agreed increment. You need to import or compute the data necessary for starting the increment.

| | |
|---|---|
| | `lOp=j_int_EndIncrement` indicates that the user subroutine is being called at the end of the increment. If you have results to export, this is a good time to do so. You can also control the continuation of the analysis via `i_Array(i_int_iStatus)`.<br><br>`lOp=j_int_EndStep` indicates that the user subroutine is being called at the end of the step.<br><br>`lOp=j_int_EndAnalysis` indicates that the user subroutine is being called at the end of the analysis. |
| `r_Array(i_flt_StepTime)` | Value of current step time. When `lOp=j_int_SetupIncrement` or `j_int_StartIncrement`, the step time is at the start of the increment. When `lOp =j_int_EndIncrement`, the step time is at the end of the increment. |
| `r_Array(i_flt_TotalTime)` | Value of current total time. When `lOp =j_int_SetupIncrement` or `j_int_StartIncrement`, the total time is at the start of the increment. When `lOp =j_int_EndIncrement`, the total time is at the end of the increment. |

# VFABRIC: User subroutine to define fabric material behavior.

**Products:** Abaqus/Explicit

**References:**

- *Fabric Material Behavior*
- **FABRIC*

## Overview



**Warning:** The use of this user subroutine generally requires considerable expertise. You are cautioned that the implementation of any realistic constitutive model requires extensive development and testing. Initial testing on a single-element model with prescribed traction loading is strongly recommended.

User subroutine *VFABRIC*:

- is used to define the mechanical constitutive behavior of a fabric material in the plane of the fabric;
- is valid for materials that exhibit two "structural" directions that may not be orthogonal to each other with deformation;
- is used to update the nominal fabric stresses for a given nominal fabric strain where the direct strains are defined as the nominal strain measured along the two yarn directions of the fabric and the engineering shear strain is defined as the drop in the angle between the two yarn directions going from the reference configuration to the current configuration;
- can be used with elements under plane stress conditions;
- will be called for blocks of material calculation points for which the material is defined in a user subroutine (*Material Data Definition*);
- can use and update solution-dependent state variables;
- can use any field variables that are passed in; and
- cannot be used in an adiabatic analysis.

## Component Ordering in Tensors

The component ordering depends upon whether the tensor is a "strain" variable or a "stress" variable.

### Symmetric Tensors

Tensors such as the strain and strain increment have four components, and tensors such as stress have three components, with the difference between the two sets of variables arising from the assumed plane stress condition. The component order with the arrays for these variables is listed in the table below:

| Component | Strain | Stress |
|---|---|---|
| 1 | $\varepsilon_{11}$ | $\sigma_{11}$ |
| 2 | $\varepsilon_{22}$ | $\sigma_{22}$ |
| 3 | $\varepsilon_{33}$ | $\sigma_{12}$ |

| Component | Strain | Stress |
|---|---|---|
| 4 | $\varepsilon_{12}$ | |

The shear strain components in user subroutine *VFABRIC* are stored as tensor components and not as engineering components.

## Initial Calculations and Checks

In the **datacheck** phase of the analysis Abaqus/Explicit calls user subroutine *VFABRIC* with a set of fictitious strains and a `totalTime` and `stepTime` that are both equal to 0.0. This step serves as a check on your constitutive relation and calculates the equivalent initial material properties, upon which the initial elastic wave speeds are computed.

## Orientation of the Fabric Yarn

In general, the yarn directions may not be orthogonal to each other in the reference configuration. You can specify these local directions with respect to the in-plane axes of an orthogonal orientation system at a material point. Both the local directions and the orthogonal system are defined together as a single orientation definition. If the local directions are not specified, these directions are assumed to match the in-plane axes of the orthogonal system. The local direction may not remain orthogonal with deformation. Abaqus updates the local directions with deformation and computes the nominal strains along these directions and the drop in angle between them (the fabric engineering shear strain). The constitutive behavior for the fabric defines the fabric nominal stresses as a function of the fabric strains. Abaqus converts these fabric stresses into the Cauchy stress and the resulting internal forces.

## Material Point Deletion

Material points that satisfy a user-defined failure criterion can be deleted from the model (see *User-Defined Mechanical Material Behavior*). You must specify the state variable number controlling the element deletion flag when you allocate space for the solution-dependent state variables, as explained in *Fabric Material Behavior*. The deletion state variable should be set to a value of one or zero in user subroutine *VFABRIC*. A value of one indicates that the material point is active, while a value of zero indicates that Abaqus/Explicit should delete the material point from the model by setting the stresses to zero. The structure of the block of material points passed to user subroutine *VFABRIC* remains unchanged during the analysis; deleted material points are not removed from the block. Abaqus/Explicit will pass zero stresses and strain increments for all deleted material points. Once a material point has been flagged as deleted, it cannot be reactivated.

## User Subroutine Interface

```
      subroutine vfabric(
C Read only (unmodifiable)variables –
     1     nblock, ndim, npt, layer, kspt, kstep, kinc,
     2     nstatev, nfieldv, nprops,
     3     lOp, jElem, stepTime, totalTime, dt, cmname, coordMp,
     4     charLength, props, density, braidAngle, fabricStrain,
     5     fabricStrainInc,
     6     tempOld, fieldOld, fabricStressOld, stateOld,
     7     tempNew, fieldNew, enerIntern,
C Write only (modifiable) variables –
     8     fabricStressNew, stateNew, enerInelas )
C
C NOTE: In addition to the above "Write only" variables,
C the thickness direction component of fabricStrainInc
C i.e, fabricStrainInc(*,ndirStrain) may also be set by
```

```
C the user for changing thickness as a function
C of material in-plane state.
C
      include 'vaba_param.inc'
C
      parameter ( ndirStrain = 3, nshr = 1, ndirStress = 2)
C
C NOTE: The constants defined above are used for array
C dimensions below.
C
      dimension
     *     jElem(nblock),
     *     coordMp(nblock,ndim),
     *     charLength(nblock),
     *     props(nprops),
     *     density(nblock),
     *     braidAngle(nblock),
     *     fabricStrain(nblock,ndirStrain+nshr),
     *     strainFabricInc(nblock,ndirStrain+nshr),
     *     tempOld(nblock),
     *     fieldOld(nblock,nfieldv),
     *     fabricStressOld(nblock,ndirStress+nshr),
     *     stateOld(nblock,nstatev),
     *     tempNew(nblock),
     *     fieldNew(nblock,nfieldv),
     *     fabricStressNew(nblock,ndirStress+nshr),
     *     stateNew(nblock,nstatev),
     *     enerIntern(nblock),
     *     enerInelas(nblock)
*
      character*80 cmname
C
      do 100 km = 1,nblock
        user coding
  100 continue

      return
      end
```

## Variables to Be Defined

`fabricStressNew(nblock,ndirStress+nshr)` Nominal fabric stress at each material point at the end of the increment. This nominal fabric stress can be requested as output variable SFABRIC.

`stateNew(nblock,nstatev)` State variables at each material point at the end of the increment. You define the size of this array by allocating space for it (see *About User Subroutines and Utilities* for more information). This variable can be requested as output variable SDV.

`enerInelas(nblock)` Total inelastic energy density at material points at the end of the increment. This variable can be requested as output variable ENER.

## Variables That Can Be Updated

`fabricStrainInc(*,ndirStrain)` Thickness direction strain increment. The thickness can be requested as output variable STH.

## Variables Passed in for Information

| | |
|---|---|
| `nblock` | Number of material points to be processed in this call to `VFABRIC`. |
| `ndim` | Two for a two-dimensional model and three for a three-dimensional model. |
| `npt` | Current integration point number. |
| `layer` | Current layer number in the case of a composite section. |
| `kspt` | Current material point number within the section. |
| `kStep` | Current Abaqus step number. |
| `kInc` | Increment number of the current Abaqus step. |
| `nstatev` | Number of user-defined state variables that are associated with this material type (you define this as described in *Allocating Space for Solution-Dependent State Variables*). |
| `nfieldv` | Number of user-defined external field variables. |
| `nprops` | User-specified number of user-defined material properties. |
| `lOp` | Integer flag indicating the computation that is expected. `lOp` = –2 indicates that the routine is being called to initialize the stresses corresponding to the initial strains, which can be large. `lOp` = –1 indicates that the routine is being called to update the stresses based on the instantaneous elastic response for a small "artificial" strain increment given. `lOp` = 0 indicates that this is an annealing process and you should reinitialize the internal state variables, `stateNew`, if necessary. The stresses will be set to zero by Abaqus. `lOp` = 1 indicates that the routine is being called to update the stresses and the state for a given strain increment. |
| `jElem(nblock)` | Array of element numbers. |
| `stepTime` | Value of time since the step began. |
| `totalTime` | Value of total time. The time at the beginning of the step is given by `totalTime - stepTime`. |
| `dt` | Time increment size. |
| `cmname` | User-specified material name, left justified. It is passed in as an uppercase character string. Some internal material models are given names starting with the character string "ABQ_". To avoid conflict, you should not use "ABQ_" as the leading string for `cmname`. |
| `coordMp(nblock,*)` | Material point coordinates. It is the midplane material point for shell elements and the centroid for beam elements. |
| `charLength(nblock)` | Characteristic element length, which is either the default value based on the geometric mean or the |

| | |
|---|---|
| | user-defined characteristic element length defined in user subroutine VUCHARLENGTH. The default value is a typical length of a line across an element for a first-order element; it is half of the same typical length for a second-order element. For membranes and shells the default value is a characteristic length in the reference surface. |
| `props(nprops)` | User-supplied material properties. |
| `density(nblock)` | Current density at the material points in the midstep configuration. This value may be inaccurate in problems where the volumetric strain increment is very small. If an accurate value of the density is required in such cases, the analysis should be run in double precision. This value of the density is not affected by mass scaling. |
| `braidAngle(nblock)` | Angle in radians between the two yarn directions at the end of the increment. |
| `fabricStrain(nblock,ndirStrain+nshr)` | Total nominal strain in the fabric at the end of increment. This variable can be requested as output variable EFABRIC. |
| `fabricStrainInc(nblock,ndirStrain+nshr)` | Incremental nominal strain in the fabric. |
| `tempOld(nblock)` | Temperatures at each material point at the beginning of the increment. |
| `fieldOld(nblock,nfieldv)` | Values of the user-defined field variables at each material point at the beginning of the increment. |
| `fabricStressOld(nblock,ndirStress+nshr)` | Nominal fabric stress at each material point at the beginning of the increment. |
| `stateOld(nblock,nstatev)` | State variables at each material point at the beginning of the increment. |
| `tempNew(nblock)` | Temperatures at each material point at the end of the increment. |
| `fieldNew(nblock,nfieldv)` | Values of the user-defined field variables at each material point at the end of the increment. |
| `enerIntern(nblock)` | Internal energy per unit mass at each material point at the beginning of the increment. |

## Example: Using More than One User-Defined Material Model

To use more than one user-defined fabric material model, the variable `cmname` can be tested for different fabric material names inside user subroutine *VFABRIC*, as illustrated below:

```
if (cmname(1:4) .eq. 'MAT1') then
   call VFABRIC_MAT1(argument_list)
else if (cmname(1:4) .eq. 'MAT2') then
   call VFABRIC_MAT2(argument_list)
end if
```

`VFABRIC_MAT1` and `VFABRIC_MAT2` are the actual fabric material user subroutines containing the constitutive material models for each material `MAT1` and `MAT2`, respectively. User subroutine *VFABRIC* merely acts as a directory here. The argument list can be the same as that used in subroutine *VFABRIC*. The material names must be in uppercase characters since `cmname` is passed in as an uppercase character string.

## Example: Influence of Nonorthogonal Material Directions in Highly Anisotropic Elastic Material

As an example of the coding of user subroutine *VFABRIC*, consider a simple elastic lamina material with highly anisotropic properties. For a fabric the material definitions need not remain orthogonal with deformation, whereas the directions do remain orthogonal for a built-in elastic material. The simple *VFABRIC* routine given below defines an elastic fabric and can be used to compare the fabric and the built-in elastic materials under different loading conditions.

The user subroutine would be coded as follows:

```
      subroutine vfabric(
C Read only (unmodifiable)variables –
     1     nblock, ndim, npt, layer, kspt, kstep, kinc,
     2     nstatev, nfieldv, nprops,
     3     lOp, jElem, stepTime, totalTime, dt, cmname, coordMp,
     4     charLength, props, density, braidAngle, fabricStrain,
     5     fabricStrainInc,
     6     tempOld, fieldOld, fabricStressOld, stateOld,
     7     tempNew, fieldNew, enerIntern,
C Write only (modifiable) variables –
     8     fabricStressNew, stateNew, enerInelas )
C
C NOTE: In addition to the above "Write only" variables,
C the thickness direction component of fabricStrainInc
C i.e, fabricStrainInc(*,ndirStrain) may also be set by the user
C for changing thickness as a function of material in-plane
C state.
C
       include 'vaba_param.inc'
C
       parameter( ndirStrain = 3, nshr = 1, ndirStress = 2,
          one = 1.d0, two = 2.d0 )
C
C NOTE: The constants defined above are used for array
C dimensions and computation below.

C
      dimension
     *     jElem(nblock),
     *     coordMp(nblock,ndim),
     *     charLength(nblock),
     *     props(nprops),
     *     density(nblock),
     *     braidAngle(nblock),
     *     fabricStrain(nblock,ndirStrain+nshr),
     *     fabricStrainInc(nblock,ndirStrain+nshr),
     *     tempOld(nblock),
     *     fieldOld(nblock,nfieldv),
     *     fabricStressOld(nblock,ndirStress+nshr),
     *     stateOld(nblock,nstatev),
     *     tempNew(nblock),
     *     fieldNew(nblock,nfieldv),
     *     fabricStressNew(nblock,ndirStress+nshr),
     *     stateNew(nblock,nstatev),
     *     enerIntern(nblock),
     *     enerInelas(nblock)
C
      character*80 cmname
```

```
C
C
C  Read properties
      E1      = props(1)
      E2      = props(2)
      xnu12   = props(3)
      twiceG12     = two * props(4)
C
      xnu21 = E2 * xnu12 / E1
C
C Let us assume:
      xnu13 = xnu12
      xnu23 = xnu21
C
      xnu13OverE1 = xnu13/E1
      xnu23OverE2 = xnu23/E2
C
      fr = one / ( one - xnu12 * xnu21 )
      D11 = E1 * fr
      D22 = E2 * fr
      D12 = E2 * xnu12 * fr
C
      do k = 1 , nblock
C
C   Update the stress
         stressInc11 = D11 * fabricStrainInc(k,1)
     *               + D12 * fabricStrainInc(k,2)
         stressInc22 = D22 * fabricStrainInc(k,2)
     *               + D12 * fabricStrainInc(k,1)
         stressInc12 = twiceG12 *
     *                 fabricStrainInc(k,ndirStrain + 1)
C
         fabricStressNew(k,1) = fabricStressOld(k,1)
     *                        + stressInc11
         fabricStressNew(k,2) = fabricStressOld(k,2)
     *                        + stressInc22
C
C              shear stress
         fabricStressNew(k,ndirStress+1) =
     *        fabricStressOld(k,ndirStress+1) + stressInc12
C
C              Thickness direction strain
C        fabricStrainInc(k,ndirStrain) =
C    *        - ( xnu13OverE1 * stressInc11
C    *        + xnu23OverE2 * stressInc22
C    *        )
C
      end do

      return
      end
```

# VFILM: User subroutine to define nonuniform film coefficients and sink temperatures for heat transfer analysis.

**Products:** Abaqus/Explicit

**References:**

- *Thermal Loads*
- **CFILM*
- **FILM*
- **SFILM*
- *Temperature-dependent film condition*

## Overview

User subroutine *VFILM*:

- can be used to define node-based, element-based, or surface-based nonuniform film coefficients;
- can be used to define sink temperatures as functions of position, time, temperature, field variables, node number, element number, etc.;
- will be called during procedures that allow heat transfer analysis at each node or surface integration point of those surfaces and elements for which node-based, element-based, or surface-based nonuniform film conditions are defined; and
- uses the amplitude references for the sink temperature or film coefficient that may appear with the associated nonuniform film definition only to modify the sink temperature or film coefficient passed into the user subroutine.

## User Subroutine Interface

```
      SUBROUTINE vfilm(
C Read only (unmodifiable) variables -
     *     nblock, ndim, kStep, kIncr, stepTime, totalTime, jElemUid,
jNodeUid,
     *     numExfv, temp, field, curCoords, areaNode, jltyp, sname,
C Write only (modifiable) variables -
     *     h, sink )
C
      include 'vaba_param.inc'
C
      dimension curCoords(nblock,ndim), jElemUid(nblock), stepTime(2),
     *     jNodeUid(nblock), field(nblock,numExfv), areaNode(nblock),
     *     temp(nblock), h(nblock), sink(nblock)
      character*80 sname
C
      do 100 km = 1, nblock
      user coding to define valueh and valuet
  100 continue
      return
      end
```

## Variables to Be Defined

`h(nblock)` Film coefficient. Units are $JT^{-1}L^{-2}\theta^{-1}$. `h` is passed into the routine as the magnitude of the film coefficient specified as part of the node-based, element-based, or surface-based film condition definition. If the magnitude is not defined, `h` will be initialized to zero.

`sink(nblock)` Sink temperature. `sink` is passed into the routine as the sink temperature specified as part of the node-based, element-based, or surface-based film condition definition. If the sink temperature is not defined, `sink` will be initialized to zero.

## Variables Passed in for Information

`nblock` Number of points to be processed in this call to *VFILM*.

`ndim` Number of coordinate directions: two for two-dimensional models and three for three-dimensional models. The model is considered to be three-dimensional if any three-dimensional elements are defined.

`kStep` Step number.

`kIncr` Increment number.

`stepTime` Value of time since the step began.

`totalTime` Value of total time. The time at the beginning of the step is given by `totalTime-stepTime`

`jElemUid` User-defined element numbers.

`jNodeUid` User-defined node numbers.

`numExfv` Number of field variables.

`temp(nblock)` Current value of temperature at this integration point.

`field(nblock,numExfv)` Current value of field variable at this integration point.

`curCoords(nblock,ndim)` An array containing the coordinates of this point. These are the current coordinates if geometric nonlinearity is accounted for during the step (see *Defining an Analysis*); otherwise, the array contains the original coordinates of the point.

`areaNode(nblock)` Array of nodal area for node-based films. `areaNode` is passed into the routine as the nodal area specified as part of the node-based film coefficient specification. This nodal area is not available for output purposes. This variable is passed in as blank for both element-based and surface-based films.

`jltyp` Element face for which this call to *VFILM* is being made for an element-based film coefficient specification. This information is useful when several different nonuniform film conditions are being imposed on an element at the same time. See *About the Element Library* for element face identification. The key is as follows:

| JLTYP | Film type |
|---|---|
| 0 | Node-based or surface-based loading |
| 11 | F1NU (FNEGNU for heat transfer shells) |
| 12 | F2NU (FPOSNU for heat transfer shells) |
| 13 | F3NU |
| 14 | F4NU |
| 15 | F5NU |
| 16 | F6NU |

| | |
|---|---|
| `sname` | Surface name for which this call to *VFILM* is being made for a surface-based film coefficient specification (`JLTYP`=0). This variable is passed in as blank for both node-based and element-based films. |

# VFRIC: User subroutine to define frictional behavior for contact surfaces.

**Products:** Abaqus/Explicit

**References:**

- *Frictional Behavior*
- **FRICTION*
- *VFRIC, VFRIC_COEF, and VFRICTION*

## Overview

User subroutine *VFRIC*:

- can be used to define the frictional behavior between contact pair surfaces;
- can be used when the classical Coulomb friction model is too restrictive and a more complex definition of shear transmission between contacting surfaces is required;
- must provide the entire definition of shear interaction between the contacting surfaces;
- can use and update solution-dependent state variables;
- cannot be used in conjunction with softened tangential surface behavior; and
- cannot be used with the general contact algorithm.

## Terminology

The use of user subroutine *VFRIC* requires familiarity with the following terminology.

### Surface Node Numbers

The "surface node number" refers to the position of a particular node in the list of nodes on the surface. For example, there are `nSecNod` nodes on the secondary surface. Number $n, n = 1, 2...$`nSecNod`, is the surface node number of the *n*th node in this list; `jSecUid`$(n)$ is the user-defined global number of this node. An Abaqus/Explicit model can be defined in terms of an assembly of part instances (see *Assembly Definition*). In such models a node number in `jSecUid` is an internally generated node number. If the original node number and part instance name are required, call the utility routine `VGETPARTINFO` (see *Obtaining Part Information*).

### Contact Points

The nodes on the secondary surface that are in contact in the current time increment are defined as "contact points." The number of contact points is passed into this subroutine as `nContact`. The array `jConSecid(nContact)` gives the surface node numbers for the contact points.

## Local Coordinate System

A local coordinate system is defined for each contact point to facilitate specification of frictional forces and incremental slips. The local 1-direction for both two-dimensional and three-dimensional contact is tangential to the main surface, and it is defined by $\mathbf{t}_1 = \boldsymbol{ds}/|\boldsymbol{ds}|$, where $\boldsymbol{ds}$ is the incremental slip vector. The incremental

slip vector used to define $\mathbf{t_1}$ corresponds to the incremental slip in the current time increment for penalty contact and the predicted incremental slip for kinematic contact. The main surface normal direction, $\mathbf{n}$, is the local 2-direction for two-dimensional contact and the local 3-direction for three-dimensional contact. The local 2-direction for three-dimensional contact is given by $\mathbf{t_2} = \mathbf{n} \times \mathbf{t_1}$, which is also tangent to the main surface. The vectors are shown in *Figure 1* and *Figure 2*. The direction cosines for $\mathbf{t_1}$ and $\mathbf{n}$ with respect to the global coordinate system are available in `dirCosT1` and `dirCosN`, respectively. In the case of zero incremental slip ($|\boldsymbol{ds}| = \mathbf{0}$) we choose an arbitrary direction for $\mathbf{t_1}$ that is orthogonal to the normal direction, $\mathbf{n}$.

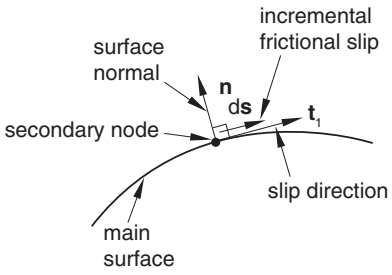



**Figure 1: Local coordinate system for two-dimensional contact with `VFRIC`.**

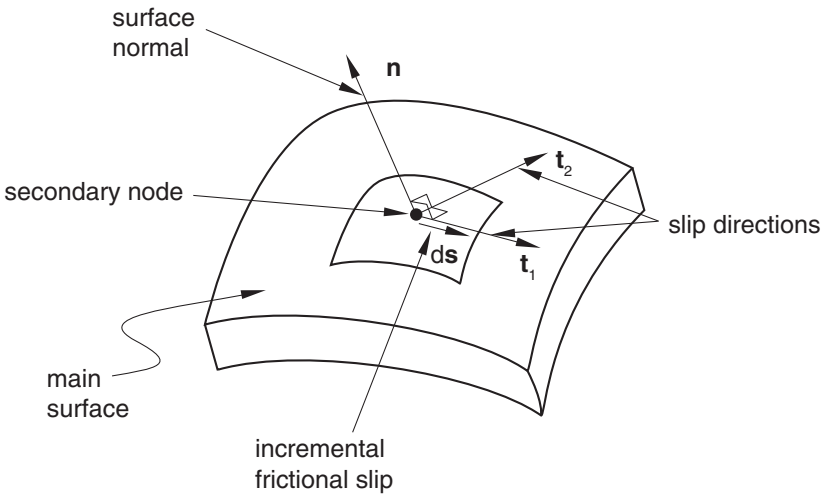



**Figure 2: Local coordinate system for three-dimensional contact with `VFRIC`.**

## Frictional Forces

You specify the frictional force, `fTangential`, at each contact point in local coordinates in this subroutine. The array `fTangential` is dimensioned such that only the tangential components can be specified. Any components of the frictional force that are not specified will remain equal to zero. For three-dimensional contact with isotropic friction, only the first component of the frictional force need be specified since the second component should be zero. A "stick force" at each contact point is provided in the array `fStickForce` to assist you in setting the appropriate frictional force values. The stick force is the force required to prevent additional "plastic" slipping. The stick force at each contact point is provided as a scalar value as it would act in the direction opposite to $\mathbf{t_1}$. The stick force is computed prior to calling user subroutine *VFRIC* by either the kinematic or the penalty contact algorithm. See *Contact Constraint Enforcement Methods in Abaqus/Explicit* for descriptions of the kinematic and penalty contact algorithms and the user interface for choosing between them. The first component of the frictional force should be in the range between zero and minus the stick force value. Typically, the stick force will be positive and the first component of the applied frictional force will be negative, opposing the incremental slip. Penalty contact includes an elastic slip regime due to finite penalty stiffness, so occasionally,

during recovery of elastic slip, the stick force will be negative, indicating that it is appropriate for the first component of the frictional force to be positive (i.e., acting in the same direction as the incremental slip). A noisy or unstable solution is likely to result if the first component of `fTangential` is set outside of the range between zero and negative the value of the stick force.

After user subroutine *VFRIC* is called, frictional forces that oppose the forces specified at the contact points are distributed to the main nodes. For balanced main-secondary contact we then compute weighted averages of the frictional forces for both main-secondary orientations. These forces are directly applied if the penalty contact algorithm is being used. If the kinematic contact algorithm is being used, the frictional forces are converted to acceleration corrections by dividing by the nodal masses.

## User Subroutine Interface

```
      subroutine vfric(
C Write only -
     1 fTangential,
C Read/Write -
     2 statev,
C Read  only -
     3 kStep, kInc, nContact, nFacNod, nSecNod, nMainNod,
     4 nFricDir, nDir, nStateVar, nProps, nTemp, nPred, numDefTfv,
     5 jSecUid,jMainUid, jConSecid, jConMainid, timStep, timGlb,
     6 dTimCur, surfInt, surfSec, surfMain, lContType,
     7 dSlipFric, fStickForce, fTangPrev, fNormal, frictionWork,
     8 shape, coordSec, coordMain, dirCosSl, dircosN, props,
     9 areaSec, tempSec, preDefSec, tempMain, preDefMain)
C
      include `vaba_param.inc'
C
      character*80 surfInt, surfSec, surfMain
C
      dimension props(nProps), statev(nStateVar,nSecNod),
     1 dSlipFric(nDir,nContact),
     2 fTangential(nFricDir,nContact),
     3 fTangPrev(nDir,nContact),
     4 fStickForce(nContact), areaSec(nSecNod),
     5 fNormal(nContact), shape(nFacNod,nContact),
     6 coordSec(nDir,nSecNod), coordMain(nDir,nMainNod),
     7 dirCosSl(nDir,nContact), dircosN(nDir,nContact),
     8 jSecUid(nSecNod), jMainUid(nMainNod),
     9 jConSecid(nContact), jConMainid(nFacNod,nContact)
     1 tempSec(nContact), preDefSec(nContact,nPred),
     2 tempMain(numDefTfv), preDefMain(numDefTfv,nPred)

      user coding to define fTangential
      and, optionally, statev

      return
      end
```

## Variables to Be Defined

`fTangential(nFricDir, nContact)` This array must be updated to the current values of the frictional force components for all contact points in the local tangent directions. See *Figure 1* and *Figure 2* for definition of the local coordinate system. This array will be zero (no friction force) until you reset it.

## Variables That Can Be Updated

`statev(nstateVar, nSecNod)` This array contains the user-defined solution-dependent state variables for all the nodes on the secondary surface. You define the size of this array (see *Frictional Behavior* for more information). This array will be passed in containing the values of these variables prior to the call to user subroutine *VFRIC*. If any of the solution-dependent state variables is being used in conjunction with the friction behavior, it must be updated in this subroutine. The state variables are available even for secondary nodes that are not in contact. This may be useful when, for example, the state variables need to be reset for secondary nodes that are not in contact.

## Variables Passed in for Information

`kStep` Step number.

`kInc` Increment number.

`nContact` Number of contacting secondary nodes.

`nFacNod` Number of nodes on each main surface facet (`nFacNod` is 2 for two-dimensional surfaces, `nFacNod` is 4 for three-dimensional surfaces). If the main surface is an analytical rigid surface, this variable is passed in as 0.

`nSecNod` Number of secondary nodes.

`nMainNod` Number of main surface nodes, if the main surface is made up of facets. If the main surface is an analytical rigid surface, this variable is passed in as 0.

`nFricDir` Number of tangent directions at the contact points (`nFricDir = nDir - 1`).

`nDir` Number of coordinate directions at the contact points. (In a three-dimensional model `nDir` will be two if the surfaces in the contact pair are two-dimensional analytical rigid surfaces or are formed by two-dimensional elements.)

`nStateVar` Number of user-defined state variables.

`nProps` User-specified number of property values associated with this friction model.

`nTemp` 1 if the temperature is defined and 0 if the temperature is not defined.

`nPred` Number of predefined field variables.

`numDefTfv` Equal to `nContact` if the main surface is made up of facets. If the main surface is an analytical rigid surface, this variable is passed in as 1.

`jSecUid(nSecNod)` This array lists the user-defined global node numbers (or internal node numbers for models defined in terms of an assembly of part instances) of the nodes on the secondary surface.

`jMainUid(nMainNod)` This array lists the user-defined global node numbers (or internal node numbers for models defined in terms of an assembly of part instances) of the nodes on the main surface. If the main surface is an analytical rigid surface, this array is passed in as a dummy array.

`jConSecid(nContact)` This array lists the surface node numbers of the secondary surface nodes that are in contact.

| | |
|---|---|
| `jConMainid(nFacNod, nContact)` | This array lists the surface node numbers of the main surface nodes that make up the facet with which each contact point is in contact. If the main surface is an analytical rigid surface, this array is passed in as a dummy array. |
| `timStep` | Value of step time. |
| `timGlb` | Value of total time. |
| `dtimCur` | Current increment in time from $t = t_{curr} - \Delta t$ to $t = t_{curr}$. |
| `surfInt` | User-specified surface interaction name, left justified. |
| `surfSec` | Secondary surface name. |
| `surfMain` | Main surface name. |
| `lContType` | Contact type flag. This flag is set based on the type of constraint enforcement method (see *Contact Constraint Enforcement Methods in Abaqus/Explicit*) being used: 1 for kinematic contact and 2 for penalty contact. Stick conditions are satisfied exactly with the kinematic contact algorithm; they are satisfied only approximately (subject to an automatically chosen penalty stiffness value) with the penalty contact algorithm. |
| `dSlipFric(nDir, nContact)` | This array contains the incremental frictional slip during the current time increment for each contact point in the current local coordinate system. These incremental slips correspond to tangential motion in the time increment from $t = t_{curr} - \Delta t$ to $t = t_{curr}$. For penalty contact this incremental slip is used to define the local coordinate system at each contact point (see *Figure 1* and *Figure 2*) so that only the first component of `dSlipFric` can be nonzero in the local system. The contact points for kinematic contact are determined based on penetrations detected in the predicted configuration (at $t = t_{curr} + \Delta t$), and the predicted incremental slip direction is used to define the local coordinate system at each contact point. If the slip direction changes between increments, `dSlipFric` may have a nonzero component in the local 2-direction and, if the surface is faceted and the contact point moves from one facet to another, in the local 3-direction. |
| `fStickForce(nContact)` | This array contains the magnitude of frictional force required to enforce stick conditions at each contact point. For kinematic contact this force corresponds to no slip; for penalty contact this force depends on the previous frictional force, the value of the penalty stiffness, and the previous incremental slip. The penalty stiffness is assigned automatically. Occasionally, during recovery of elastic slip associated with the penalty method, the stick force will be assigned a negative value. |
| `fTangPrev(nDir, nContact)` | This array contains the values of the frictional force components calculated in the previous increment but provided in the current local coordinate system (zero for nodes that were not in contact). |
| `fNormal(nContact)` | This array contains the magnitude of the normal force for the contact points applied at the end of current time increment; i.e., at time $t = t_{curr}$. |
| `frictionWork` | This variable contains the value of the total frictional dissipation in the entire model from the beginning of the analysis. The units are energy per unit area. |
| `shape(nFacNod, nContact)` | For each contact point this array contains the shape functions of the nodes of its main surface facet, evaluated at the location of the contact |

| | |
|---|---|
| | point. If the main surface is an analytical rigid surface, this array is passed in as a dummy array. |
| `coordSec(nDir, nSecNod)` | Array containing the `nDir` components of the current coordinates of the secondary nodes. |
| `coordMain(nDir, nMainNod)` | Array containing the `nDir` components of the current coordinates of the main nodes. If the main surface is an analytical rigid surface, this array is passed in as a dummy array. |
| `dirCosSl(nDir, nContact)` | Direction cosines of the incremental slip at the contact points. |
| `dircosN(nDir, nContact)` | Direction cosines of the normals to the main surface at the contact points. |
| `props(nProps)` | User-specified vector of property values to define the frictional behavior between the contacting surfaces. |
| `areaSec(nSecNod)` | Area associated with the secondary nodes (equal to 1 for node-based surface nodes). |
| `tempSec(nContact)` | Current temperature at the secondary nodes. |
| `preDefSec(nContact,nPred)` | Current user-specified predefined field variables at the secondary nodes (initial values at the beginning of the analysis and current values during the analysis). |
| `tempMain(numDefTfv)` | Current temperature at the nearest points on the main surface. |
| `preDefMain(numDefTfv,nPred)` | Current user-specified predefined field variables at the nearest points on the main surface (initial values at the beginning of the analysis and current values during the analysis). |

# VFRIC_COEF: User subroutine to define the frictional coefficient for contact surfaces.

**Products:** Abaqus/Explicit

**References:**

- *Frictional Behavior*
- **FRICTION*
- *VFRIC, VFRIC_COEF, and VFRICTION*

## Overview

User subroutine *VFRIC_COEF*:

- can be used to define the isotropic frictional coefficient between contacting surfaces;
- corresponds to the classical Coulomb friction model; and
- can be used only with the general contact algorithm.

## User Subroutine Interface

```
      subroutine vfric_coef (
C Write only -
     *   fCoef,
C Not applicable -
     *   fCoefDeriv,
C Read only, except state variables within rData array -
     *   nBlock, nProps, nTemp, nFields,
     *   iData, rData,
     *   surfInt, surfSec, surfMain,
     *   props, slipData, pressure,
     *   tempAvg, fieldAvg )
C
      include 'vaba_param.inc'
C
      dimension fCoef(nBlock,2),
     *   fCoefDeriv(nBlock,20,2),
     *   props(nProps),
     *   slipData(nBlock,6),
     *   pressure(nBlock),
     *   tempAvg(nBlock),
     *   fieldAvg(nBlock,nFields)
c
c    Offset parameters to various integer quantities in iData(nDataI)
c
      parameter( i_ifc_KStep   = 1,
     *           i_ifc_KInc    = 2,
     *           i_ifc_Kit     = 3,
     *           i_ifc_Node    = 4,
     *           i_ifc_nStateV = 5,
     *           i_ifc_nFDir   = 6,
     *           nDataI        = 6 )
c
```

```
c    Offset parameters to various real quantities in rData(*)
c
      parameter( i_rfc_TimStep = 1,
     *           i_rfc_TimGlb  = 2,
     *           i_rfc_DTimCur = 3,
     *           i_rfc_Drot    = 4,
     *           i_rfc_StateV  = 8 )
c
c    Offset parameters to slip variables in slipData
c
      parameter( i_SD_SlipRateMag  = 1,
     *           i_SD_SlipRate1    = 2,
     *           i_SD_SlipRate2    = 3,
     *           i_SD_SlipEq       = 4,
     *           i_SD_Slip1        = 5,
     *           i_SD_Slip2        = 6,
     *           n_SD_Length       = 6 )
C
      dimension iData(nDataI), rData(*)
C
      character*80 surfInt, surfSec, surfMain
c
C
      user coding to define fCoef
c
c     Pointers to access blocks of data from rData
c     For example, Solution-dependent state variables indexed from
c           1 to number of state variables nStateV where
c           nStateV = iData(i_ifc_nStateV)
c
      pointer (ptrStateVariables,StateVariables)
      real*8 StateVariables(*) !indexed 1 to iData(i_ifc_nStateV)
      ......
c
c     Access state variables
c
      ptrStateVariables = loc(rData(i_rfc_StateV))
      ......

c
c     Compute fCoef
c
      ......
C
      return
      end
```

## Variables to Be Defined

`fCoef(nBlock,2)` `fCoef(1,1)` and other entries of the first column of the `fCoef` array should be updated to current values of the friction coefficient for all contacting points. `fCoef(1,2)` and other entries of the second column of the `fCoef` array should not be updated for Abaqus/Explicit analyses. You can model anisotropic friction with directional preference specified as a surface property (see *Anisotropic Friction with Directional Preference as a Surface Property* and a nominal friction coefficient specified in `fCoef(1,1) and other entries of the first column of the fCoef` array.

`fCoefDeriv(nBlock,20,2)` This array is not applicable to Abaqus/Explicit analyses.

## Variables That Can Be Updated

`rData(i_rfc_StateV)` The array of real values starting from this offset corresponds to the solution-dependent state variables. You specify the number of available state variables (*Allocating Space for Solution-Dependent State Variables*). These state variables are passed in with values at the start of the increment. Update them to their corresponding values at the end of the increment. See the user subroutine interface provided earlier for an example of how to access the array of state variables from the buffer `rData`.

## Variables Passed in for Information

| | |
|---|---|
| `nBlock` | Number of contacting points to be processed in this call to *VFRIC_COEF*. |
| `nProps` | User-specified number of property values associated with this friction model. |
| `nTemp` | 1 if the temperature is defined, and 0 if the temperature is not defined. |
| `nFields` | Number of user-specified field variables. |
| `iData(i_ifc_KStep)` | Step number. |
| `iData(i_ifc_KInc)` | Increment number. |
| `iData(i_ifc_Kit)` | Not applicable to Abaqus/Explicit analyses. |
| `iData(i_ifc_Node)` | Equal to 0 in Abaqus/Explicit analyses. The secondary node number varies among the contact constraints processed in a single call to this user subroutine. |
| `iData(i_ifc_nStateV)` | The number of state variables you have specified. The default value is 0. |
| `iData(i_ifc_nFDir)` | Equal to 1 in Abaqus/Explicit analyses. You are expected to provide a single friction coefficient for each contact constraint in the user subroutine. |
| `rData(i_rfc_TimStep)` | Value of step time. |
| `rData(i_rfc_TimGlb)` | Value of total time. |
| `rData(i_rfc_DTimCur)` | Current time increment size. |
| `rData(i_rfc_Drot)` | The list of four value entries starting from this offset corresponds to the column major storage of the $2 \times 2$ rotation increment matrix. For Abaqus/Explicit, this rotation increment matrix is always provided as the identity matrix. |
| `surfInt` | User-specified surface interaction name, left justified. |
| `surfSec` | Not applicable to Abaqus/Explicit analyses. |
| `surfMain` | Not applicable to Abaqus/Explicit analyses. |
| `props(nProps)` | User-specified vector of property values to define the frictional coefficient at contacting points. |
| `slipData(nBlock,6)` | The entries `slipData(kBlock,i_SD_SlipRateMag)` and `slipData(kBlock,i_SD_SlipEq)` contain the slip rate magnitude and total accumulated slip distance for a contacting point indexed by `kBlock` for the current time increment. Other entries of `slipData` are not applicable to Abaqus/Explicit analyses. |
| `pressure(nBlock)` | Array containing the pressure at the contact point projected at the end of the current time increment. |
| `tempAvg(nBlock)` | Average current temperature between the main and secondary surfaces at the contacting points. |

`fieldAvg(nBlock,nFields)` Average current value of all the user-specified field variables between the main and secondary surfaces at the contacting points.

# VFRICTION: User subroutine to define frictional behavior for contact surfaces.

**Products:** Abaqus/Explicit

**References:**

- *Frictional Behavior*
- **FRICTION*
- *VFRIC, VFRIC_COEF, and VFRICTION*

## Overview

User subroutine *VFRICTION*:

- can be used to define the frictional behavior between contacting surfaces;
- can be used when the classical Coulomb friction model is too restrictive and a more complex definition of shear transmission between contacting surfaces is required;
- must provide the entire definition of shear interaction between the contacting surfaces;
- can use and update solution-dependent state variables for node-to-face and node-to-analytical rigid surface contact;
- cannot be used in conjunction with softened tangential surface behavior; and
- can be used only with the general contact algorithm.

## Contact Points

The points considered in user subroutine *VFRICTION* are called contact points. Each contact point is primarily associated with a secondary node or a point along a secondary edge; the contact point also references the corresponding main surface that it contacts. A contact point exists for each pairing of secondary node and main surface. Therefore, more than one contact point may reference the same secondary node but different main surfaces, such as with contact at a corner.

The number of contact points currently being passed into user subroutine *VFRICTION* is `nBlock`. The array `jConSecUid(nNodSec,nBlock)` gives the secondary surface node numbers associated with the contact points. The variable `nNodSec` indicates whether a single secondary node (for node-to-face contact) or two secondary nodes of an edge (for edge-to-edge contact) are associated with each contact point. Similarly, the array `jConMainUid(nNodMain,nBlockAnal)` gives the main surface node numbers associated with each contact point; the nodes can belong to a facet, an edge, or an analytical surface. The variable `nNodMain` indicates the number of main nodes associated with each contact point.

An Abaqus/Explicit model can be defined in terms of an assembly of part instances (see *Assembly Definition*). In such models a node number is an internally generated node number. If the original node number and part instance name are required, call the utility routine `VGETPARTINFO` (see *Obtaining Part Information*).

## Local Coordinate System

A local coordinate system is defined for each contact point to facilitate specification of frictional forces and incremental slip. The local 1-direction for both two-dimensional and three-dimensional contact is tangential to the main surface; it is defined by $\mathbf{t}_1 = \boldsymbol{ds}/|\boldsymbol{ds}|$, where $\boldsymbol{ds}$ is the incremental slip vector. The incremental slip

vector used to define $\mathbf{t_1}$ corresponds to the incremental slip in the current time increment. The main surface normal direction, $\mathbf{n}$, is the local 2-direction for two-dimensional contact and the local 3-direction for three-dimensional contact. The local 2-direction for three-dimensional contact is given by $\mathbf{t_2} = \mathbf{n} \times \mathbf{t_1}$, which is also tangent to the main surface. The vectors are shown in *Figure 1* and *Figure 2*. The direction cosines for $\mathbf{t_1}$ and $\mathbf{n}$ with respect to the global coordinate system are available in `dirCosS1` and `dirCosN`, respectively.

In the case of zero incremental slip ($|\boldsymbol{ds}| = \mathbf{0}$) we choose an arbitrary direction for $\mathbf{t_1}$ that is orthogonal to the normal direction, $\mathbf{n}$.

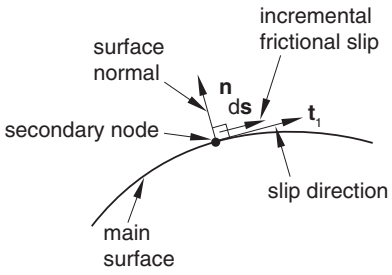



**Figure 1: Local coordinate system for two-dimensional contact with `VFRICTION`.**

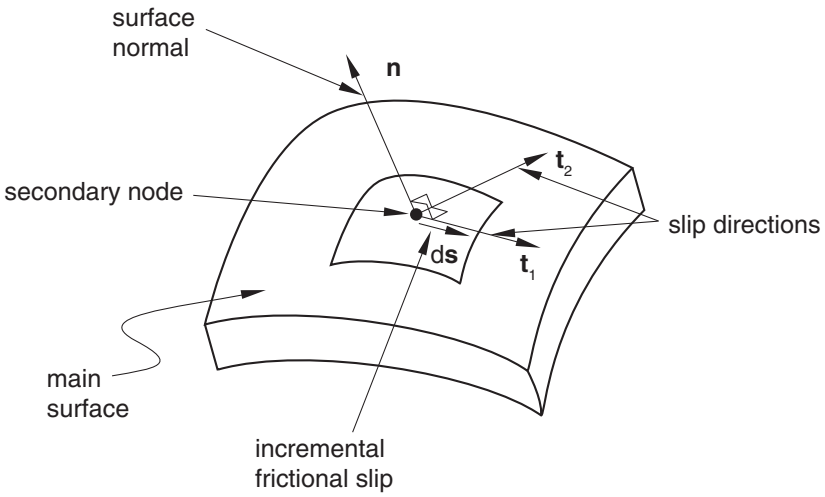



**Figure 2: Local coordinate system for three-dimensional contact with `VFRICTION`.**

## Frictional Forces

You specify the frictional force, `fTangential`, at each contact point in local coordinates in this subroutine. The array `fTangential` is dimensioned such that only the tangential components can be specified. Any components of the frictional force that are not specified will remain equal to zero. For isotropic friction, only the first component of the frictional force need be specified since the second component should be zero. A "stick force" at each contact point is provided in the array `fStickForce` to assist you in setting appropriate frictional force values. The stick force is the force required to prevent additional "plastic" slipping. The stick force at each contact point is provided as a scalar value as it would act in the direction opposite to $\mathbf{t_1}$. The stick force is computed prior to calling user subroutine *VFRICTION*. The first component of the frictional force should be in the range between zero and the negative of the stick force value. Typically, the stick force will be positive and the first component of the applied frictional force will be negative, opposing the incremental slip. Penalty contact includes an elastic slip regime due to finite penalty stiffness; so occasionally the stick force will be negative during recovery of elastic slip, indicating that it is appropriate for the first component of the frictional force to be positive (that is, acting in the same direction as the incremental slip). A noisy or unstable solution is

likely to result if the first component of `fTangential` is set outside the range between zero and the negative of the stick force value.

After user subroutine *VFRICTION* is called, frictional forces that oppose the forces specified at the contact points are distributed to the main nodes.

## User Subroutine Interface

```
       subroutine vfriction (
C Write only -
     *   fTangential,
C Read/Write -
     *   state,
C Read only -
     *   nBlock, nBlockAnal, nBlockEdge,
     *   nNodState, nNodSec, nNodMain,
     *   nFricDir, nDir,
     *   nStates, nProps, nTemp, nFields,
     *   jFlags, rData,
     *   surfInt, surfSec, surfMain,
     *   jConSecUid, jConMainUid, props,
     *   dSlipFric, fStickForce, fTangPrev, fNormal,
     *   areaCont, dircosN, dircosS1,
     *   shapeSec, shapeMain,
     *   coordSec, coordMain,
     *   velSec, velMain,
     *   tempSec, tempMain,
     *   fieldSec, fieldMain )
C
      include `vaba_param.inc'
C
      dimension fTangential(nFricDir,nBlock),
     *   state(nStates,nNodState,nBlock),
     *   jConSecUid(nNodSec,nBlock),
     *   jConMainUid(nNodMain,nBlockAnal),
     *   props(nProps),
     *   dSlipFric(nDir,nBlock),
     *   fStickForce(nBlock),
     *   fTangPrev(nDir,nBlock),
     *   fNormal(nBlock),
     *   areaCont(nBlock),
     *   dircosN(nDir,nBlock),
     *   dircosS1(nDir,nBlock),
     *   shapeSec(nNodSec,nBlockEdge),
     *   shapeMain(nNodMain,nBlockAnal),
     *   coordSec(nDir,nNodSec,nBlock),
     *   coordMain(nDir,nNodMain,nBlockAnal),
     *   velSec(nDir,nNodSec,nBlock),
     *   velMain(nDir,nNodMain,nBlockAnal),
     *   tempSec(nBlock),
     *   tempMain(nBlockAnal),
     *   fieldSec(nFields,nBlock),
     *   fieldMain(nFields,nBlockAnal)
C
      parameter( iKStep    = 1,
     *           iKInc     = 2,
     *           iLConType = 3,
     *           nFlags    = 3 )
C
      parameter( iTimStep      = 1,
```

```
     *            iTimGlb       = 2,
     *            iDTimCur      = 3,
     *            iFrictionWork = 4,
     *            nData         = 4 )
C
      dimension jFlags(nFlags), rData(nData)
C
      character*80 surfInt, surfSec, surfMain
C
      user coding to define fTangential
      and, optionally, state
C
      return
      end
```

## Variables to Be Defined

`fTangential(nFricDir,nBlock)` This array must be updated to the current values of the frictional force components for all contact points in the local tangent directions. See *Figure 2* for a definition of the local coordinate system. This array will be zero (no friction force) until it is set.

## Variables That Can Be Updated

`state(nStates,nNodState,nBlock)` This array contains the user-defined, solution-dependent state variables for all the nodes on the secondary surface. The use of state variables is applicable for node-to-face and node-to-analytical rigid surface contact. See *Frictional Behavior* for more information on the size of this array. This array will be passed in containing the values of these variables prior to the call to user subroutine *VFRICTION*.

If any of the solution-dependent state variables are being used in conjunction with the friction behavior, they must be updated in this subroutine. These state variables need to be updated with care: outside the user subroutine these state variables are single-valued per secondary node, but multiple contact points may refer to the same secondary node (if it contacts a main surface at more than one point). Each contact point may be passed into the user subroutine independently in a given increment, possibly on separate calls to the user subroutine; therefore, you may end up advancing the state variables for the associated node multiple times for a single increment. To keep track of whether or not a node state is advanced, you may want to use one of the state variables exclusively for this purpose. You could set that selected state variable to the current increment number and update the state only if it is not already set to the current increment number.

## Variables Passed in for Information

`nBlock` Number of contact points to be processed in this call to *VFRICTION*.

`nBlockAnal` 1 for analytical rigid main surface; `nBlock` otherwise.

`nBlockEdge` `nBlock` for edge-type secondary surface; 1 otherwise.

| | |
|---|---|
| `nNodState` | 1 for node-to-face contact and node-to-analytical rigid surface contact. |
| `nNodSec` | 1 for node-to-face and node-to-analytical rigid surface contact; 2 for edge-to-edge contact. |
| `nNodMain` | 1 for analytical rigid main surface; 2 for edge-type main surface; 2 for two-dimensional facet-type main surface; 4 for three-dimensional facet-type main surface. |
| `nFricDir` | Number of tangent directions at the contact points (`nFricDir = nDir - 1`). |
| `nDir` | Number of coordinate directions at the contact points (2 for two-dimensional models; 3 for three-dimensional models). |
| `nStates` | Number of user-defined state variables. |
| `nProps` | User-specified number of property values associated with this friction model. |
| `nTemp` | 1 if the temperature is defined and 0 if the temperature is not defined. |
| `nFields` | Number of predefined field variables. |
| `jFlag(1)` | Step number. |
| `jFlag(2)` | Increment number. |
| `jFlag(3)` | 1 for node-to-face contact, 2 for edge-to-edge contact, and 3 for node-to-analytical rigid surface contact. |
| `rData(1)` | Value of step time. |
| `rData(2)` | Value of total time. |
| `rData(3)` | Current increment in time from $t = t_{curr} - \Delta t$ to $t = t_{curr}$. |
| `rData(4)` | This variable contains the value of the total frictional dissipation in the entire model from the beginning of the analysis. The units are energy per unit area. |
| `surfInt` | User-specified surface interaction name, left justified. |
| `surfSec` | Secondary surface name, currently set to a blank. |
| `surfMain` | Main surface name, currently set to a blank. |
| `jConSecUid(nNodSec,nBlock)` | This array lists the surface node numbers of the secondary surface nodes associated with each contact point. |
| `jConMainUid(nNodMain,nBlockAnal)` | This array lists the surface node numbers of the main surface nodes that make up the facet, edge, or analytical rigid surface associated with each contact point. |
| `props(nProps)` | User-specified vector of property values to define the frictional behavior between the contacting surfaces. |
| `dSlipFric(nDir,nBlock)` | This array contains the incremental frictional slip during the current time increment for each contact point in the current local coordinate system. These incremental slips correspond to tangential motion in the time increment from $t = t_{curr} - \Delta t$ to $t = t_{curr}$. This incremental slip is used to define the local coordinate system at each contact point (see *Figure 2*) so that only the first component of `dSlipFric` can be nonzero in the local system. |
| `fStickForce(nBlock)` | This array contains the magnitude of frictional force required to enforce stick conditions at each contact point. |

| | |
|---|---|
| | This force depends on the previous frictional force, the value of the penalty stiffness, and the previous incremental slip. The penalty stiffness is assigned automatically. Occasionally, during recovery of elastic slip associated with the penalty method, the stick force will be assigned a negative value. |
| `fTangPrev(nDir,nBlock)` | This array contains the values of the frictional force components calculated in the previous increment but provided in the current local coordinate system (zero for nodes that were not in contact). |
| `fNormal(nBlock)` | This array contains the magnitude of the normal force for the contact points applied at the end of current time increment; that is,, at time $t = t_{curr}$. |
| `areaCont(nBlock)` | Area associated with the contact points. The sum of the contact areas among all contact points associated with a single secondary node equals the surface area associated with that secondary node (equal to 1 for node-based surface nodes). Therefore, the contact area at a contact point depends on the number of contact points currently associated with the same secondary node. A contact point contributes a frictional stress to the associated secondary node that is equal to `fTangential(1,k)` divided by `areaCont(k)`. |
| `dircosN(nDir,nBlock)` | Direction cosines of the normals to the main surface at the contact points. |
| `dirCosS1(nDir,nBlock)` | Direction cosines of the incremental slip at the contact points. The direction cosines are undefined (all components zero) if the incremental frictional slip is zero. |
| `shapeSec(nNodSec,nBlockEdge)` | For edge-to-edge contact this array contains the shape functions of the nodes of its secondary edge, evaluated at the location of the contact point. If the contact is not edge-to-edge, this array is passed in as a dummy array. |
| `shapeMain(nNodMain,nBlockAnal)` | For node-to-face and edge-to-edge contact this array contains the shape functions of the nodes of its main surface, evaluated at the location of the contact point. If the main surface is an analytical rigid surface, this array is passed in as a dummy array. |
| `coordSec(nDir,nNodSec,nBlock)` | Array containing the `nDir` components of the current coordinates of the contact points. |
| `coordMain(nDir,nNodMain,nBlockAnal)` | Array containing the `nDir` components of the current coordinates of the main nodes associated with the contact points. If the main surface is an analytical rigid surface, this array is passed in as a dummy array. |
| `velSec(nDir,nNodSec,nBlock)` | Array containing the `nDir` components of the current velocity of the contact points. |
| `velMain(nDir,nNodMain,nBlockAnal)` | Array containing the `nDir` components of the current velocity of the main nodes associated with the contact points. If the main surface is an analytical rigid surface, this array is passed in as a dummy array. |

| | |
|---|---|
| `tempSec(nBlock)` | Current temperature of the secondary surface at the contact points. |
| `tempMain(nBlockAnal)` | Current temperature at the points on the main surface associated with the contact points. |
| `fieldSec(nFields,nBlock)` | Current user-specified predefined field variables on the secondary surface at the contact points (initial values at the beginning of the analysis and current values during the analysis). |
| `fieldMain(nFields,nBlockAnal)` | Current user-specified predefined field variables at the points on the main surface associated with the contact points (initial values at the beginning of the analysis and current values during the analysis). |

# VHETVAL: User subroutine to provide internal heat generation in a heat transfer analysis.

**Products:** Abaqus/Explicit

**References:**

- *Fully Coupled Thermal-Stress Analysis*
- **HEAT GENERATION*
- *VHETVAL*

## Overview

User subroutine *VHETVAL*:

- can be used to define a heat flux due to internal heat generation in a material (for example, as might be associated with phase changes occurring during the solution);
- will be called for blocks of material calculation points for which the material definition contains internal heat generation;
- can be useful if it is necessary to include a kinetic theory for a phase change associated with latent heat release (for example, in the prediction of crystallization in a polymer casting process);
- can use and update solution-dependent state variables as an alternative to user subroutine *VUMATHT* without a requirement to completely define material thermal behavior;
- can use any field variables that are passed in; and
- can be used with thermally coupled continuum elements and Eulerian elements.

## User Subroutine Interface

```
      subroutine vhetval (
C Read only (unmodifiable) variables -
     *     nblock, nElem, nIntPt, nLayer, nSectPt,
     *     ntgrad, nstatev, nfieldv, nprops,
     *     cmname, stepTime, totalTime, dt,
     *     coordMp, density, props,
     *     tempOld, fieldOld, stateOld,
     *     tempNew, tempgradNew, fieldNew,
C Write only (modifiable) variables -
     *     stateNew, flux )
C
      include 'vaba_param.inc'
C
      dimension nElem(nblock)
C
      dimension coordMp(nblock,*), density(nblock), props(nprops),
     *    tempOld(nblock), fieldOld(nblock, nfieldv),
     *    stateOld(nblock, nstatev), tempNew(nblock),
     *    tempgradNew(nblock, ntgrad), fieldNew(nblock, nfieldv),
     *    stateNew(nblock, nstatev), flux(nblock)
C
      character*80 cmname
C
      do km = 1, nblock
C         user coding
```

```
      end do
C
      return
      end
```

## Variables to Be Defined

| | |
|---|---|
| `flux(nblock)` | Volumetric heat flux at each material point. |
| `stateNew(nblock, nstatev)` | State variables at each material point at the end of the increment. |

## Variables Passed in for Information

| | |
|---|---|
| `nblock` | Number of material points to be processed in this call to user subroutine *VHETVAL*. |
| `nElem(nblock)` | User-defined element number. |
| `nIntPt` | Integration point number. |
| `nLayer` | Layer number. |
| `nSectPt` | Section point number in the current layer. |
| `ntgrad` | Number of spatial gradients of temperature. |
| `nstatev` | Number of user-defined state variables that are associated with this material type (see *Allocating Space for Solution-Dependent State Variables*). |
| `nfieldv` | Number of user-defined external field variables. |
| `nprops` | User-specified number of user-defined material properties. |
| `cmname` | User-specified material name, left justified. It is passed in as an uppercase character string. Some internal material models are given names starting with the "ABQ_" character string. To avoid conflict, you should not use "ABQ_" as the leading string for `cmname`. |
| `stepTime` | Value of time since the step began. |
| `totalTime` | Value of total time. The time at the beginning of the step is given by `totalTime - stepTime`. |
| `dt` | Time increment size. |
| `coordMp(nblock,*)` | Material point coordinates. |
| `density(nblock)` | Current density at the material points in the midstep configuration. |
| `props(nprops)` | User-supplied material properties. |
| `tempOld(nblock)` | Temperatures at each material point at the beginning of the increment. |
| `fieldOld(nblock, nfieldv)` | Values of the user-defined field variables at each material point at the beginning of the increment. |
| `stateOld(nblock, nstatev)` | State variables at each material point at the beginning of the increment. |
| `tempNew(nblock)` | Temperatures at each material point at the end of the increment. |
| `tempgradNew(nblock, ntgrad)` | Current values of the spatial gradients of temperature. |
| `fieldNew(nblock, nfieldv)` | Values of the user-defined field variables at each material point at the end of the increment. |

# VSDVINI: User subroutine to define initial solution-dependent state variable fields.

**Products:** Abaqus/Explicit

**References:**

- *About User Subroutines and Utilities*
- **INITIAL CONDITIONS*
- *VSDVINI*

## Overview

User subroutine *VSDVINI*:

- will be called for user-defined initial solution-dependent state variable fields at particular material points or shell section points (see *Initial Conditions*);
- can be used to initialize solution-dependent state variables allocated at particular material points or shell section points, as described in *About User Subroutines and Utilities*; and
- returns a value of zero for any solution-dependent state variables that have no defined initial condition.

## Use of Solution-Dependent State Variables in Other User Subroutines

Solution-dependent state variables initialized in *VSDVINI* can be used and updated in other user subroutines called at particular material points (such as *VUMAT* or *VUHARD*) or shell section points (for example, *VUGENS*).

## User Subroutine Interface

```
      subroutine vsdvini(
C Read only -
     *     nblock, jElem, nIntPt, nLayer, nSectPt,
     *     ndim, nStateV, cMName, coordMp, charLength,
C Write only (modifiable) -
     *     stateIni )
C
      include 'vaba_param.inc'
C
      dimension charLength(nblock), coordMp(nblock,ndim),
     *   stateIni(nblock,nStateV), jElem(nblock)
C
      character*80 cMName
C
      do 100 km = 1,nblock
      user coding to define stateIni(nblock,nStateV)
  100 continue
C
      return
      end
```

## Variables to Be Defined

`stateIni(nblock,nStateV)` An array containing initial solution-dependent state variables at the material points or shell section points. The allocation of this array is described in *About User Subroutines and Utilities*.

## Variables Passed in for Information

| | |
|---|---|
| `nblock` | Number of material points to be processed in this call to *VSDVINI*. |
| `jElem(nblock)` | Array of element numbers. |
| `nIntPt` | Integration point number in the element. |
| `nLayer` | Layer number (for composite shells and layered solids). |
| `nSecPt` | Section point number within the current layer or section. Section point 1 is used for all pure heat transfer, coupled temperature-displacement, and coupled thermal-electrical-structural analyses. |
| `nDim` | Space dimension of coordinates. |
| `nStateV` | User-defined number of solution-dependent state variables (see *About User Subroutines and Utilities*). |
| `cMName` | Material name, left justified. It is passed in as an uppercase character string. Some internal material models are given names starting with the "ABQ_" character string. To avoid conflict, "ABQ_" should not be used as the leading string for `cMName`. |
| `coordMp(nblock,ndim)` | Material point coordinates. It is the midplane material point for shell elements and the centroid for beam elements. |
| `charLength(nblock)` | Characteristic element length, which is either the default value based on the geometric mean or the user-defined characteristic element length defined in user subroutine *VUCHARLENGTH*. The default value is a typical length of a line across an element for a first-order element; it is half of the same typical length for a second-order element. For beams and trusses the default value is a characteristic length along the element axis. For membranes and shells, it is a characteristic length in the reference surface. For axisymmetric elements, it is a characteristic length in the r–z plane only. For cohesive elements, it is equal to the constitutive thickness. |

# VUAMP: User subroutine to specify amplitudes.

**Products:** Abaqus/Explicit

**References:**

- *Amplitude Curves*
- **AMPLITUDE*
- **OUTPUT*

## Overview

User subroutine *VUAMP*:

- allows you to define the current value of an amplitude definition as a function of time;
- can be used to model control engineering aspects of your system when sensors are used (sensor values are from the beginning of the increment);
- can use a predefined number of state variables in its definition; and
- can optionally compute the derivatives and integrals of the amplitude function.

## Explicit Solution Dependence

The solution dependence introduced in this user subroutine is explicit: all data passed in the subroutine for information or to be updated are values at the beginning of that increment.

## User Subroutine Interface

```
      SUBROUTINE VUAMP(
     *     ampName, time, ampValueOld, dt, nprops, props, nSvars,
     *     svars, lFlagsInfo, nSensor, sensorValues, sensorNames,
     *     jSensorLookUpTable,
     *     AmpValueNew,
     *     lFlagsDefine,
     *     AmpDerivative, AmpSecDerivative, AmpIncIntegral)

      INCLUDE 'VABA_PARAM.INC'

C     time indices
      parameter (iStepTime        = 1,
     *           iTotalTime       = 2,
     *           nTime            = 2)
C     flags passed in for information
      parameter (iInitialization   = 1,
     *           iRegularInc       = 2,
     *           ikStep            = 3,
     *           nFlagsInfo        = 3)
C     optional flags to be defined
      parameter (iComputeDeriv     = 1,
     *           iComputeSecDeriv  = 2,
     *           iComputeInteg     = 3,
     *           iStopAnalysis     = 4,
     *           iConcludeStep     = 5,
```

```
     *           nFlagsDefine      = 5)
      dimension time(nTime), lFlagsInfo(nFlagsInfo),
     *          lFlagsDefine(nFlagsDefine),
     *          sensorValues(nSensor),
     *          props(nprops),
     *          sVars(nSvars)

      character*80 sensorNames(nSensor)
      character*80 ampName
      dimension jSensorLookUpTable(*)

      user coding to define AmpValueNew, and
      optionally lFlagsDefine, AmpDerivative, AmpSecDerivative,
      AmpIncIntegral

      RETURN
      END
```

## Variables to Be Defined

`AmpValueNew` Current value of the amplitude.

## Variables That Can Be Updated

`lFlagsDefine` Integer flag array to determine whether the computation of additional quantities is necessary or to set step continuation requirements.

| | |
|---|---|
| `lFlagsDefine(iComputeDeriv)` | If set to 1, you must provide the computation of the amplitude derivative. The default is 0, which means that Abaqus computes the derivative automatically. |
| `lFlagsDefine(iComputeSecDeriv)` | If set to 1, you must provide the computation of the amplitude second derivative. The default is 0, which means that Abaqus computes the second derivative automatically. |
| `lFlagsDefine(iComputeInteg)` | If set to 1, you must provide the computation of the amplitude incremental integral. The default is 0, which means that Abaqus computes the incremental integral automatically. |
| `lFlagsDefine(iStopAnalysis)` | If set to 1, the analysis will be stopped and an error message will be issued. The default is 0, which means that Abaqus will not stop the analysis. |
| `lFlagsDefine(iConcludeStep)` | If set to 1, Abaqus will conclude the step execution and advance to the next step (if a next step is available). The default is 0. |

`svars` An array containing the values of the solution-dependent state variables associated with this amplitude definition. The number of such variables is `nsvars` (see above). You define the meaning of these variables.

This array is passed into *VUAMP* containing the values of these variables at the start of the current increment. In most cases they should be updated to be the values at the end of the increment.

`AmpDerivative` Current value of the amplitude derivative.

`AmpSecDerivative` Current value of the amplitude second derivative.

`AmpIncIntegral` Current value of the amplitude incremental integral.

## Variables Passed in for Information

`ampName` User-specified amplitude name, left justified.

`time(iStepTime)` Current value of step time.

`time(iTotalTime)` Current value of total time.

`ampValueOld` Old value of the amplitude from the previous increment.

`dt` Current stable time increment.

`nprops` User-defined number of properties associated with this amplitude definition.

`props(nprops)` User-supplied amplitude properties.

`nSvars` User-defined number of solution-dependent state variables associated with this amplitude definition.

`lFlagsInfo` Integer flag array with information regrading the current call to *VUAMP*:

| | |
|---|---|
| `lFlagsInfo(iInitialization)` | This flag is equal to 1 if *VUAMP* is called from the initialization phase of each step and is set to 0 otherwise. |
| `lFlagsInfo(iRegularInc)` | This flag is equal to 1 if *VUAMP* is called from a regular increment and is set to 0 otherwise. |
| `lFlagsInfo(ikStep)` | Step number. |

`nSensor` Total number of sensors in the model.

`sensorValues` Array with sensor values at the end of the previous increment. Each sensor value corresponds to a history output variable associated with the output database request defining the sensor.

`sensorNames` Array with user-defined sensor names in the entire model, left justified. Each sensor name corresponds to a sensor value provided with the output database request. All names will be converted to uppercase characters if lowercase or mixed-case characters were used in their definition.

`jSensorLookUpTable` Variable that must be passed into the utility functions `IVGETSENSORID` and `VGETSENSORVALUE`.

## Example: Amplitude Definition Using Sensor and State Variables

```
c     user amplitude subroutine
      Subroutine VUAMP(
C           passed in for information and state variables
     *     ampName, time, ampValueOld, dt, nprops, props, nSvars,
     *     svars, lFlagsInfo, nSensor, sensorValues, sensorNames,
     *     jSensorLookUpTable,
C           to be defined
     *     ampValueNew,
     *     lFlagsDefine,
     *     AmpDerivative, AmpSecDerivative, AmpIncIntegral)

      include 'vaba_param.inc'

C     svars - additional state variables, similar to (V)UEL
      dimension sensorValues(nSensor), props(nprops),
     *     svars(nSvars)
      character*80 sensorNames(nSensor)
      character*80 ampName

C     time indices
```

```
      parameter( iStepTime        = 1,
     *           iTotalTime       = 2,
     *           nTime            = 2)
C     flags passed in for information
      parameter( iInitialization   = 1,
     *           iRegularInc       = 2,
     *           ikStep            = 3,
     *           nFlagsInfo        = 3)
C     optional flags to be defined
      parameter( iComputeDeriv     = 1,
     *           iComputeSecDeriv  = 2,
     *           iComputeInteg     = 3,
     *           iStopAnalysis     = 4,
     *           iConcludeStep     = 5,
     *           nFlagsDefine      = 5)

      parameter( tStep=0.18, tAccelerateMotor = .00375,
     *           omegaFinal=23.26)

c    Alternatively, assign the user-defined amplitude
c    properties on the data lines rather than using a parameter
c    definition above.
c     tStep = props(1)
c     tAccelerateMotor = props(2)
c     omegaFinal = props(3)

      dimension time(nTime), lFlagsInfo(nFlagsInfo),
     *          lFlagsDefine(nFlagsDefine)
      dimension jSensorLookUpTable(*)

      lFlagsDefine(iComputeDeriv)    = 1
      lFlagsDefine(iComputeSecDeriv) = 1

c     get sensor value
      vTrans_CU1  = vGetSensorValue('HORIZ_TRANSL_MOTION',
     *                              jSensorLookUpTable,
     *                              sensorValues)

      if (ampName(1:22) .eq. 'MOTOR_WITH_STOP_SENSOR' ) then
         if (lFlagsInfo(iInitialization).eq.1) then
            ampValueNew      = ampValueOld

            svars(1) = 0.0
            svars(2) = 0.0
         else
            tim = time(iStepTime)

c           ramp up the angular rot velocity  of the electric
c           motor after which hold constant
            if (tim .le. tAccelerateMotor) then
               ampValueNew = omegaFinal*tim/tAccelerateMotor

            else
               ampValueNew = omegaFinal
            end if

c           retrieve old sensor value
            vTrans_CU1_old = svars(1)

c           detect a zero crossing and count the number of
c           crossings
```

```
            if (vTrans_CU1_old*vTrans_CU1 .le. 0.0 .and.
     *          tim .gt. tAccelerateMotor ) then
               svars(2) = svars(2) + 1.0
            end if
            nrCrossings = int(svars(2))

c           stop the motor if sensor crosses zero the second
c           time
            if (nrCrossings.eq.2) then
               ampValueNew =  0.0
               lFlagsDefine(iConcludeStep)=1
            end if

c           store sensor value
            svars(1) = vTrans_CU1

         end if
      end if

      return
      end
```

# VUANISOHYPER_INV: User subroutine to define anisotropic hyperelastic material behavior using the invariant formulation.

**Products:** Abaqus/Explicit

**References:**

- *Anisotropic Hyperelastic Behavior*
- **ANISOTROPIC HYPERELASTIC*
- *UANISOHYPER_INV and VUANISOHYPER_INV*

## Overview

User subroutine *VUANISOHYPER_INV*:

- can be used to define the strain energy potential of anisotropic hyperelastic materials as a function of an irreducible set of scalar invariants;
- will be called for blocks of material calculation points for which the material definition contains user-defined anisotropic hyperelastic behavior with invariant-based formulation (*Anisotropic Hyperelastic Behavior*);
- can use and update solution-dependent state variables;
- can use any field variables that are passed in; and
- requires that the values of the derivatives of the strain energy density function be defined with respect to the scalar invariants.

## Enumeration of Invariants

To facilitate coding and provide easy access to the array of invariants passed to user subroutine *VUANISOHYPER_INV*, an enumerated representation of each invariant is introduced. Any scalar invariant can, therefore, be represented uniquely by an *enumerated* invariant, $I_n^*$, where the subscript $n$ denotes the order of the invariant according to the enumeration scheme in the following table:

| Invariant | Enumeration, $n$ |
|---|---|
| $\bar{I}_1$ | $1$ |
| $\bar{I}_2$ | $2$ |
| $J$ | $3$ |
| $\bar{I}_{4(\alpha\beta)}$ | $4+2(\alpha-1)+\beta(\beta-1)\,;\qquad \alpha\le\beta$ |
| $\bar{I}_{5(\alpha\beta)}$ | $5+2(\alpha-1)+\beta(\beta-1)\,;\qquad \alpha\le\beta$ |

For example, in the case of three families of fibers there are a total of 15 invariants: $\bar{I}_1$, $\bar{I}_2$, $J$, six invariants of type $\bar{I}_{4(\alpha\beta)}$, and six invariants of type $\bar{I}_{5(\alpha\beta)}$, with $\alpha,\beta=1,2,3 \quad (\alpha\le\beta)$. The following correspondence exists between each of these invariants and their enumerated counterpart:

| Enumerated invariant | Invariant |
|---|---|
| $I_1^*$ | $\overline{I}_1$ |
| $I_2^*$ | $\overline{I}_2$ |
| $I_3^*$ | $J$ |
| $I_4^*$ | $\overline{I}_{4(11)}$ |
| $I_5^*$ | $\overline{I}_{5(11)}$ |
| $I_6^*$ | $\overline{I}_{4(12)}$ |
| $I_7^*$ | $\overline{I}_{5(12)}$ |
| $I_8^*$ | $\overline{I}_{4(22)}$ |
| $I_9^*$ | $\overline{I}_{5(22)}$ |
| $I_{10}^*$ | $\overline{I}_{4(13)}$ |
| $I_{11}^*$ | $\overline{I}_{5(13)}$ |
| $I_{12}^*$ | $\overline{I}_{4(23)}$ |
| $I_{13}^*$ | $\overline{I}_{5(23)}$ |
| $I_{14}^*$ | $\overline{I}_{4(33)}$ |
| $I_{15}^*$ | $\overline{I}_{5(33)}$ |

A similar scheme is used for the array `zeta` of terms $\zeta_{\alpha\beta} = \mathbf{A}_\alpha \cdot \mathbf{A}_\beta$. Each term can be represented uniquely by an enumerated counterpart $\zeta_m^*$, as shown below:

| Dot product | Enumeration, $m$ |
|---|---|
| $\zeta_{\alpha\beta}$ | $\alpha + \frac{1}{2}(\beta - 2)(\beta - 1); \quad \alpha < \beta$ |

As an example, for the case of three families of fibers there are three $\zeta_{\alpha\beta}$ terms: $\zeta_{12}$, $\zeta_{13}$, and $\zeta_{23}$. These are stored in the `zeta` array as $(\zeta_1^*, \zeta_2^*, \zeta_3^*)$.

## Storage of Arrays of Derivatives of Energy Function

The components of the array `duDi` of first derivatives of the strain energy potential with respect to the scalar invariants, $\partial U / \partial I_i^*$, are stored using the enumeration scheme discussed above for the scalar invariants.

The elements of the array `d2uDiDi` of second derivatives of the strain energy function, $\partial^2 U / \partial I_i^* \partial I_j^*$, are laid out in memory using triangular storage: if $k$ denotes the component in this array corresponding to the term

$\partial^2 U/\partial I_i^* \partial I_j^*$, then $k = i + j \times (j-1)/2; \quad (i \le j)$. For example, the term $\partial^2 U/\partial I_2^* \partial I_5^*$ is stored in component $k = 2 + (5 \times 4)/2 = 12$ in the `d2uDiDi` array.

## Special Considerations for Shell Elements

When *VUANISOHYPER_INV* is used to define the material response of shell elements, Abaqus/Explicit cannot calculate a default value for the transverse shear stiffness of the element. Hence, you must define the material transverse shear modulus (see *Defining the Elastic Transverse Shear Modulus*) or the element's transverse shear stiffness (see *Shell Section Behavior*).

## Material Point Deletion

Material points that satisfy a user-defined failure criterion can be deleted from the model (see *User-Defined Mechanical Material Behavior*). You must specify the state variable number controlling the element deletion flag when you allocate space for the solution-dependent state variables, as explained in *User-Defined Mechanical Material Behavior*. The deletion state variable should be set to a value of one or zero in *VUANISOHYPER_INV*. A value of one indicates that the material point is active, and a value of zero indicates that Abaqus/Explicit should delete the material point from the model by setting the stresses to zero. The structure of the block of material points passed to user subroutine *VUANISOHYPER_INV* remains unchanged during the analysis; deleted material points are not removed from the block. Abaqus/Explicit will "freeze" the values of the invariants passed to *VUANISOHYPER_INV* for all deleted material points; that is, the values remain constant after deletion is triggered. Once a material point has been flagged as deleted, it cannot be reactivated.

## User Subroutine Interface

```
      subroutine vuanisohyper_inv (
C Read only (unmodifiable) variables -
     1     nblock, nFiber, nInv,
     2     jElem, kIntPt, kLayer, kSecPt,
     3     cmname,
     4     nstatev, nfieldv, nprops,
     5     props, tempOld, tempNew, fieldOld, fieldNew,
     6     stateOld, sInvariant, zeta,
C Write only (modifiable) variables -
     7     uDev, duDi, d2uDiDi,
     8     stateNew )
C
      include 'vaba_param.inc'
C
      dimension props(nprops),
     1  tempOld(nblock),
     2  fieldOld(nblock,nfieldv),
     3  stateOld(nblock,nstatev),
     4  tempNew(nblock),
     5  fieldNew(nblock,nfieldv),
     6  sInvariant(nblock,nInv),
     7  zeta(nblock,nFiber*(nFiber-1)/2),
     8  uDev(nblock), duDi(nblock,nInv),
     9  d2uDiDi(nblock,nInv*(nInv+1)/2),
     *  stateNew(nblock,nstatev)
C
      character*80 cmname
C

      do 100 km = 1,nblock
        user coding
```

```
  100 continue

      return
      end
```

## Variables to Be Defined

`udev(nblock)`
$\tilde{U}_{dev}$, the deviatoric part of the strain energy density of the primary material response. This quantity is needed only if the current material definition also includes Mullins effect (see *Stress Softening in Rubberlike Materials*).

`duDi(nblock,nInv)`
Array of derivatives of strain energy potential with respect to the scalar invariants, $\partial U / \partial I_i^*$, ordered using the enumeration scheme discussed above.

`d2uDiDi(nblock,nInv*(nInv+1)/2)`
Arrays of second derivatives of strain energy potential with respect to the scalar invariants (using triangular storage), $\partial^2 U / \partial I_i^* \partial I_j^*$.

`stateNew(nblock,nstatev)`
State variables at each material point at the end of the increment. You define the size of this array by allocating space for it (see *About User Subroutines and Utilities* for more information).

## Variables Passed in for Information

`nblock`
Number of material points to be processed in this call to *VUANISOHYPER_STRAIN*.

`nFiber`
Number of families of fibers defined for this material.

`nInv`
Number of scalar invariants.

`jElem(nblock)`
Array of element numbers.

`kIntPt`
Integration point number.

`kLayer`
Layer number (for composite shells).

`kSecPt`
Section point number within the current layer.

`cmname`
User-specified material name, left justified. It is passed in as an uppercase character string. Some internal material models are given names starting with the "ABQ_" character string. To avoid conflict, you should not use "ABQ_" as the leading string for `cmname`.

`nstatev`
Number of user-defined state variables that are associated with this material type (you define this as described in *Allocating Space for Solution-Dependent State Variables*).

`nfieldv`
Number of user-defined external field variables.

`nprops`
User-specified number of user-defined material properties.

`props(nprops)`
User-supplied material properties.

`tempOld(nblock)`
Temperatures at each material point at the beginning of the increment.

`tempNew(nblock)`
Temperatures at each material point at the end of the increment.

`fieldOld(nblock,nfieldv)`
Values of the user-defined field variables at each material point at the beginning of the increment.

| | |
|---|---|
| `fieldNew(nblock,nfieldv)` | Values of the user-defined field variables at each material point at the end of the increment. |
| `stateOld(nblock,nstatev)` | State variables at each material point at the beginning of the increment. |
| `sInvariant(nblock,nInv)` | Array of scalar invariants, $I_i^*$ , at each material point at the end of the increment. The invariants are ordered using the enumeration scheme discussed above. |
| `zeta(nblock,nFiber*(nFiber-1)/2)` | Array of dot product between the directions of different families of fiber in the reference configuration, $\zeta_{\alpha\beta} = \mathbf{A}_\alpha \cdot \mathbf{A}_\beta$ . The array contains the enumerated values $\zeta_m^*$ using the scheme discussed above. |

## Example: Using More than One User-Defined Anisotropic Hyperelastic Material Model

To use more than one user-defined anisotropic hyperelastic material model, the variable `cmname` can be tested for different material names inside user subroutine *VUANISOHYPER_INV*, as illustrated below:

```
if (cmname(1:4) .eq. 'MAT1') then
   call VUANISOHYPER_INV1(argument_list)
else if (cmname(1:4) .eq. 'MAT2') then
   call VUANISOHYPER_INV2(argument_list)
end if
```

`VUANISOHYPER_INV1` and `VUANISOHYPER_INV2` are the actual subroutines containing the anisotropic hyperelastic models for each material `MAT1` and `MAT2`, respectively. Subroutine `VUANISOHYPER_INV` merely acts as a directory here. The argument list can be the same as that used in subroutine `VUANISOHYPER_INV`. The material names must be in uppercase characters since `cmname` is passed in as an uppercase character string.

## Example: Anisotropic Hyperelastic Model of Kaliske and Schmidt

As an example of the coding of subroutine `VUANISOHYPER_INV`, consider the model proposed by Kaliske and Schmidt (2005) for nonlinear anisotropic elasticity with two families of fibers. The strain energy function is given by a polynomial series expansion in the form

$$U = \frac{1}{D}(J-1)^2 + \sum_{i=1}^{3} a_i \left(\bar{I}_1 - 3\right)^i + \sum_{j=1}^{3} b_j \left(\bar{I}_2 - 3\right)^j + \sum_{k=2}^{6} c_k \left(\bar{I}_{4(11)} - 1\right)^k + \sum_{l=2}^{6} d_l \left(\bar{I}_{5(11)} - 1\right)^l$$
$$+ \sum_{m=2}^{6} e_m \left(\bar{I}_{4(22)} - 1\right)^m + \sum_{n=2}^{6} f_n \left(\bar{I}_{5(22)} - 1\right)^n + \sum_{p=2}^{6} g_p \left(\zeta_{12}\bar{I}_{4(12)} - \zeta_{12}^2\right)^p.$$

The code in subroutine `VUANISOHYPER_INV` must return the derivatives of the strain energy function with respect to the scalar invariants, which are readily computed from the above expression. In this example auxiliary functions are used to facilitate enumeration of pseudo-invarinats of type $\bar{I}_{4(\alpha\beta)}$ and $\bar{I}_{5(\alpha\beta)}$, as well as for indexing into the array of second derivatives using symmetric storage. The subroutine would be coded as follows:

```
      subroutine vuanisohyper_inv (
C Read only -
     *     nblock, nFiber, nInv,
     *     jElem, kIntPt, kLayer, kSecPt,
     *     cmname,
     *     nstatev, nfieldv, nprops,
     *     props, tempOld, tempNew, fieldOld, fieldNew,
     *     stateOld, sInvariant, zeta,
```

```
C     Write only -
     *     uDev, duDi, d2uDiDi,
     *     stateNew )
C
      include 'vaba_param.inc'
C
      dimension props(nprops),
     *  tempOld(nblock),
     *  fieldOld(nblock,nfieldv),
     *  stateOld(nblock,nstatev),
     *  tempNew(nblock),
     *  fieldNew(nblock,nfieldv),
     *  sInvariant(nblock,nInv),
     *  zeta(nblock,nFiber*(nFiber-1)/2),
     *  uDev(nblock), duDi(nblock,*),
     *  d2uDiDi(nblock,*),
     *  stateNew(nblock,nstatev)
C
      character*80 cmname
C
      parameter ( zero = 0.d0, one = 1.d0, two = 2.d0,
     *     three = 3.d0, four = 4.d0, five = 5.d0, six = 6.d0 )
C
C     Kaliske energy function (3D)
C
C     Read material properties
      d=props(1)
      dinv = one / d
      a1=props(2)
      a2=props(3)
      a3=props(4)
      b1=props(5)
      b2=props(6)
      b3=props(7)
      c2=props(8)
      c3=props(9)
      c4=props(10)
      c5=props(11)
      c6=props(12)
      d2=props(13)
      d3=props(14)
      d4=props(15)
      d5=props(16)
      d6=props(17)
      e2=props(18)
      e3=props(19)
      e4=props(20)
      e5=props(21)
      e6=props(22)
      f2=props(23)
      f3=props(24)
      f4=props(25)
      f5=props(26)
      f6=props(27)
      g2=props(28)
      g3=props(29)
      g4=props(30)
      g5=props(31)
      g6=props(32)
C
      do k = 1, nblock
```

```
         Udev(k) = zero
C     Compute Udev and 1st and 2nd derivatives w.r.t invariants
C     - I1
         bi1 = sInvariant(k,1)
         term = bi1-three
         Udev(k) = Udev(k)
     *        + a1*term + a2*term**2 + a3*term**3
         duDi(k,1) = a1 + two*a2*term + three*a3*term**2
         d2uDiDi(k,indx(1,1)) = two*a2 + three*two*a3*term
C     - I2
         bi2 = sInvariant(k,2)
         term = bi2-three
         Udev(k) = Udev(k)
     *        + b1*term + b2*term**2 + b3*term**3
         duDi(k,2) = b1 + two*b2*term + three*b3*term**2
         d2uDiDi(k,indx(2,2)) = two*b2 + three*two*b3*term
C     - I3 (=J)
         bi3 = sInvariant(k,3)
         term = bi3-one
         duDi(k,3) = two*dinv*term
         d2uDiDi(k,indx(3,3)) = two*dinv
C     - I4(11)
         nI411 = indxInv4(1,1)
         bi411 = sInvariant(k,nI411)
         term = bi411-one
         Udev(k) = Udev(k)
     *        + c2*term**2 + c3*term**3 + c4*term**4
     *        + c5*term**5 + c6*term**6
         duDi(k,nI411) =
     *          two*c2*term
     *        + three*c3*term**2
     *        + four*c4*term**3
     *        + five*c5*term**4
     *        + six*c6*term**5
         d2uDiDi(k,indx(nI411,nI411)) =
     *          two*c2
     *        + three*two*c3*term
     *        + four*three*c4*term**2
     *        + five*four*c5*term**3
     *        + six*five*c6*term**4
C     - I5(11)
         nI511 = indxInv5(1,1)
         bi511 = sInvariant(k,nI511)
         term = bi511-one
         Udev(k) = Udev(k)
     *        + d2*term**2 + d3*term**3 + d4*term**4
     *        + d5*term**5 + d6*term**6
         duDi(k,nI511) =
     *          two*d2*term
     *        + three*d3*term**2
     *        + four*d4*term**3
     *        + five*d5*term**4
     *        + six*d6*term**5
         d2uDiDi(k,indx(nI511,nI511)) =
     *          two*d2
     *        + three*two*d3*term
     *        + four*three*d4*term**2
     *        + five*four*d5*term**3
     *        + six*five*d6*term**4
C     - I4(22)
         nI422 = indxInv4(2,2)
```

```
          bi422 = sInvariant(k,nI422)
          term = bi422-one
          Udev(k) = Udev(k)
     *         + e2*term**2 + e3*term**3 + e4*term**4
     *         + e5*term**5 + e6*term**6
          duDi(k,nI422) =
     *           two*e2*term
     *         + three*e3*term**2
     *         + four*e4*term**3
     *         + five*e5*term**4
     *         + six*e6*term**5
          d2uDiDi(k,indx(nI422,nI422)) =
     *           two*e2
     *         + three*two*e3*term
     *         + four*three*e4*term**2
     *         + five*four*e5*term**3
     *         + six*five*e6*term**4
C      - I5(22)
          nI522 = indxInv5(2,2)
          bi522 = sInvariant(k,nI522)
          term = bi522-one
          Udev(k) = Udev(k)
     *         + f2*term**2 + f3*term**3 + f4*term**4
     *         + f5*term**5 + f6*term**6
          duDi(k,nI522) =
     *           two*f2*term
     *         + three*f3*term**2
     *         + four*f4*term**3
     *         + five*f5*term**4
     *         + six*f6*term**5
          d2uDiDi(k,indx(nI522,nI522)) =
     *           two*f2
     *         + three*two*f3*term
     *         + four*three*f4*term**2
     *         + five*four*f5*term**3
     *         + six*five*f6*term**4
C      - I4(12)
          nI412 = indxInv4(1,2)
          bi412 = sInvariant(k,nI412)
          term = zeta(k,1)*(bi412-zeta(k,1))
          Udev(k) = Udev(k)
     *         + g2*term**2 + g3*term**3
     *         + g4*term**4 + g5*term**5
     *         + g6*term**6
          duDi(k,nI412) = zeta(k,1) * (
     *           two*g2*term
     *         + three*g3*term**2
     *         + four*g4*term**3
     *         + five*g5*term**4
     *         + six*g6*term**5 )
          d2uDiDi(k,indx(nI412,nI412)) = zeta(k,1)**2 * (
     *           two*g2
     *         + three*two*g3*term
     *         + four*three*g4*term**2
     *         + five*four*g5*term**3
     *         + six*five*g6*term**4 )
C
       end do
C
       return
       end
```

```
C
C     Function to map index from Square to Triangular storage
C     of symmetric matrix
C
      integer function indx( i, j )
      include 'vaba_param.inc'
      ii = min(i,j)
      jj = max(i,j)
      indx = ii + jj*(jj-1)/2
      return
      end
C
C     Function to generate enumeration of scalar
C     Pseudo-Invariants of type 4

C     integer function indxInv4( i, j )
      include 'vaba_param.inc'
      ii = min(i,j)
      jj = max(i,j)
      indxInv4 = 4 + jj*(jj-1) + 2*(ii-1)
      return
      end
C
C     Function to generate enumeration of scalar
C     Pseudo-Invariants of type 5
C
      integer function indxInv5( i, j )
      include 'vaba_param.inc'
      ii = min(i,j)
      jj = max(i,j)
      indxInv5 = 5 + jj*(jj-1) + 2*(ii-1)
      return
      end
```

## Additional Reference


- Kaliske, M., and J. Schmidt, “Formulation of Finite Nonlinear Anisotropic Elasticity,” *CADFEM GmbH Infoplaner 2/2005*, vol. 2, pp. 22–23, 2005.

# VUANISOHYPER_STRAIN: User subroutine to define anisotropic hyperelastic material behavior based on Green strain.

**Products:** Abaqus/Explicit

**References:**

- *Anisotropic Hyperelastic Behavior*
- **ANISOTROPIC HYPERELASTIC*
- *UANISOHYPER_INV and VUANISOHYPER_INV*

## Overview

User subroutine *VUANISOHYPER_STRAIN*:

- can be used to define the strain energy potential of anisotropic hyperelastic materials as a function of the components of the Green strain tensor;
- will be called for blocks of material calculation points for which the material definition contains user-defined anisotropic hyperelastic behavior with Green strain-based formulation (*Anisotropic Hyperelastic Behavior*);
- can use and update solution-dependent state variables;
- can use any field variables that are passed in; and
- requires that the values of the derivatives of the strain energy density function be defined with respect to the components of the modified Green strain tensor and the volume ratio.

## Component Ordering in Tensors

The component ordering depends upon whether the tensor is second or fourth order.

### Symmetric Second-Order Tensors

For symmetric second-order tensors, such as the modified Green strain tensor, there are `ndir+nshr` components; the component order is given as a natural permutation of the indices of the tensor. The direct components are first and then the indirect components, beginning with the 12-component. For example, a stress tensor contains `ndir` direct stress components and `nshr` shear stress components, which are passed in as

| Component | 2D Case | 3D Case |
|---|---|---|
| 1 | $\bar{\varepsilon}^G_{11}$ | $\bar{\varepsilon}^G_{11}$ |
| 2 | $\bar{\varepsilon}^G_{22}$ | $\bar{\varepsilon}^G_{22}$ |
| 3 | $\bar{\varepsilon}^G_{33}$ | $\bar{\varepsilon}^G_{33}$ |
| 4 | $\bar{\varepsilon}^G_{12}$ | $\bar{\varepsilon}^G_{12}$ |
| 5 | | $\bar{\varepsilon}^G_{23}$ |
| 6 | | $\bar{\varepsilon}^G_{31}$ |

The shear strain components are stored as tensor components and not as engineering components.

## Symmetric Fourth-Order Tensors

For symmetric fourth-order tensors, such as the deviatoric elasticity tensor $\partial^2 U/\partial \bar{\varepsilon}^G_{ij}\,\partial \bar{\varepsilon}^G_{kl}$, there are `(ndir+nshr)*(ndir+nshr+1)/2` independent components. These components are ordered using the following triangular storage scheme:

| Component | 2D Case | 3D Case |
|---|---|---|
| 1 | $\partial^2 U/\partial \bar{\varepsilon}^G_{11}\,\partial \bar{\varepsilon}^G_{11}$ | $\partial^2 U/\partial \bar{\varepsilon}^G_{11}\,\partial \bar{\varepsilon}^G_{11}$ |
| 2 | $\partial^2 U/\partial \bar{\varepsilon}^G_{11}\,\partial \bar{\varepsilon}^G_{22}$ | $\partial^2 U/\partial \bar{\varepsilon}^G_{11}\,\partial \bar{\varepsilon}^G_{22}$ |
| 3 | $\partial^2 U/\partial \bar{\varepsilon}^G_{22}\,\partial \bar{\varepsilon}^G_{22}$ | $\partial^2 U/\partial \bar{\varepsilon}^G_{22}\,\partial \bar{\varepsilon}^G_{22}$ |
| 4 | $\partial^2 U/\partial \bar{\varepsilon}^G_{11}\,\partial \bar{\varepsilon}^G_{33}$ | $\partial^2 U/\partial \bar{\varepsilon}^G_{11}\,\partial \bar{\varepsilon}^G_{33}$ |
| 5 | $\partial^2 U/\partial \bar{\varepsilon}^G_{22}\,\partial \bar{\varepsilon}^G_{33}$ | $\partial^2 U/\partial \bar{\varepsilon}^G_{22}\,\partial \bar{\varepsilon}^G_{33}$ |
| 6 | $\partial^2 U/\partial \bar{\varepsilon}^G_{33}\,\partial \bar{\varepsilon}^G_{33}$ | $\partial^2 U/\partial \bar{\varepsilon}^G_{33}\,\partial \bar{\varepsilon}^G_{33}$ |
| 7 | $\partial^2 U/\partial \bar{\varepsilon}^G_{11}\,\partial \bar{\varepsilon}^G_{12}$ | $\partial^2 U/\partial \bar{\varepsilon}^G_{11}\,\partial \bar{\varepsilon}^G_{12}$ |
| 8 | $\partial^2 U/\partial \bar{\varepsilon}^G_{22}\,\partial \bar{\varepsilon}^G_{12}$ | $\partial^2 U/\partial \bar{\varepsilon}^G_{22}\,\partial \bar{\varepsilon}^G_{12}$ |
| 9 | $\partial^2 U/\partial \bar{\varepsilon}^G_{33}\,\partial \bar{\varepsilon}^G_{12}$ | $\partial^2 U/\partial \bar{\varepsilon}^G_{33}\,\partial \bar{\varepsilon}^G_{12}$ |
| 10 | $\partial^2 U/\partial \bar{\varepsilon}^G_{12}\,\partial \bar{\varepsilon}^G_{12}$ | $\partial^2 U/\partial \bar{\varepsilon}^G_{12}\,\partial \bar{\varepsilon}^G_{12}$ |
| 11 | | $\partial^2 U/\partial \bar{\varepsilon}^G_{11}\,\partial \bar{\varepsilon}^G_{23}$ |
| 12 | | $\partial^2 U/\partial \bar{\varepsilon}^G_{22}\,\partial \bar{\varepsilon}^G_{23}$ |
| 13 | | $\partial^2 U/\partial \bar{\varepsilon}^G_{33}\,\partial \bar{\varepsilon}^G_{23}$ |
| 14 | | $\partial^2 U/\partial \bar{\varepsilon}^G_{12}\,\partial \bar{\varepsilon}^G_{23}$ |
| 15 | | $\partial^2 U/\partial \bar{\varepsilon}^G_{23}\,\partial \bar{\varepsilon}^G_{23}$ |
| 16 | | $\partial^2 U/\partial \bar{\varepsilon}^G_{11}\,\partial \bar{\varepsilon}^G_{31}$ |
| 17 | | $\partial^2 U/\partial \bar{\varepsilon}^G_{22}\,\partial \bar{\varepsilon}^G_{31}$ |
| 18 | | $\partial^2 U/\partial \bar{\varepsilon}^G_{33}\,\partial \bar{\varepsilon}^G_{31}$ |
| 19 | | $\partial^2 U/\partial \bar{\varepsilon}^G_{12}\,\partial \bar{\varepsilon}^G_{31}$ |
| 20 | | $\partial^2 U/\partial \bar{\varepsilon}^G_{23}\,\partial \bar{\varepsilon}^G_{31}$ |
| 21 | | $\partial^2 U/\partial \bar{\varepsilon}^G_{31}\,\partial \bar{\varepsilon}^G_{31}$ |

If $Q$ denotes the component number of term $\partial^2 U/\partial \bar{\varepsilon}^G_{ij}\,\partial \bar{\varepsilon}^G_{kl}$ in the above table and $M$ and $N$ (with $M \le N$) denote the component numbers of $\bar{\varepsilon}^G_{ij}$ and $\bar{\varepsilon}^G_{kl}$, respectively, in the table for second-order tensors, $Q$ is given by the relationship $Q = M + N \times (N-1)/2$. For example, consider the term $\partial^2 U/\partial \bar{\varepsilon}^G_{11}\,\partial \bar{\varepsilon}^G_{23}$. The component numbers for $\bar{\varepsilon}^G_{11}$ and $\bar{\varepsilon}^G_{23}$ are $M = 1$ and $N = 5$, respectively, giving $Q = 1 + (5 \times 4)/2 = 11$.

## Special Consideration for Shell Elements

When *VUANISOHYPER_STRAIN* is used to define the material response of shell elements, Abaqus/Explicit cannot calculate a default value for the transverse shear stiffness of the element. Hence, you must define the material transverse shear modulus (see *Defining the Elastic Transverse Shear Modulus*) or the element's transverse shear stiffness (see *Shell Section Behavior*).

## Material Point Deletion

Material points that satisfy a user-defined failure criterion can be deleted from the model (see *User-Defined Mechanical Material Behavior*). You must specify the state variable number controlling the element deletion flag when you allocate space for the solution-dependent state variables, as explained in *User-Defined Mechanical Material Behavior*. The deletion state variable should be set to a value of one or zero in *VUANISOHYPER_STRAIN*. A value of one indicates that the material point is active, and a value of zero indicates that Abaqus/Explicit should delete the material point from the model by setting the stresses to zero. The structure of the block of material points passed to user subroutine *VUANISOHYPER_STRAIN* remains unchanged during the analysis; deleted material points are not removed from the block. Abaqus/Explicit will "freeze" the values of the strains passed to *VUANISOHYPER_STRAIN* for all deleted material points; that is, the strain values remain constant after deletion is triggered. Once a material point has been flagged as deleted, it cannot be reactivated.

## User Subroutine Interface

```
      subroutine vuanisohyper_strain(
C Read only (unmodifiable) variables -
     1 nblock,jElem,kIntPt,kLayer,kSecPt,cmname,
     2 ndir,nshr,nstatev,nfieldv,nprops,
     3 props,tempOld,tempNew,fieldOld,fieldNew,
     4 stateOld,ebar,detu,
C Write only (modifiable) variables -
     4 udev,duDe,duDj,
     5 d2uDeDe,d2uDjDj,d2uDeDj,
     6 stateNew)
C
      include 'vaba_param.inc'
C
      dimension jElem(nblock),
     1  props(nprops),
     2  tempOld(nblock),
     3  fieldOld(nblock,nfieldv),
     4  stateOld(nblock,nstatev),
     5  tempNew(nblock),
     6  fieldNew(nblock,nfieldv),
     7  ebar(nblock,ndir+nshr), detu(nblock),
     8  uDev(nblock),
     9  duDe(nblock,ndir+nshr), duDj(nblock),
     *  d2uDeDe(nblock,(ndir+nshr)*(ndir+nshr+1)/2),
     1  d2uDjDj(nblock),
     2  d2uDeDj(nblock,ndir+nshr),
```

```
     3  stateNew(nblock,nstatev)
C
      character*80 cmname
C

      do 100 km = 1,nblock
        user coding
  100 continue

      return
      end
```

## Variables to Be Defined

`udev(nblock)`
$\tilde{U}_{dev}$, the deviatoric part of the strain energy density of the primary material response. This quantity is needed only if the current material definition also includes Mullins effect (see *Stress Softening in Rubberlike Materials*).

`duDe(nblock,ndir+nshr)`
Derivatives of strain energy potential with respect to the components of the modified Green strain tensor, $\partial U/\partial \bar{\varepsilon}^G_{ij}$.

`duDj(nblock,ndir+nshr)`
Derivatives of strain energy potential with respect to volume ratio, $\partial U/\partial J$.

`d2uDeDe(nblock,(ndir+nshr)*(ndir+nshr+1)/2)`
Second derivatives of strain energy potential with respect to the components of the modified Green strain tensor (using triangular storage), $\partial^2 U/\partial \bar{\varepsilon}^G_{ij}\,\partial \bar{\varepsilon}^G_{kl}$.

`d2uDjDj(nblock)`
Second derivatives of strain energy potential with respect to volume ratio, $\partial^2 U/\partial J^2$.

`d2uDeDj(nblock,ndir+nshr)`
Cross derivatives of strain energy potential with respect to components of the modified Green strain tensor and volume ratio, $\partial^2 U/\partial \bar{\varepsilon}^G_{ij}\,\partial J^2$.

`stateNew(nblock,nstatev)`
State variables at each material point at the end of the increment. You define the size of this array by allocating space for it (see *About User Subroutines and Utilities* for more information).

## Variables Passed in for Information

`nblock`
Number of material points to be processed in this call to *VUANISOHYPER_STRAIN*.

`jElem(nblock)`
Array of element numbers.

`kIntPt`
Integration point number.

`kLayer`
Layer number (for composite shells).

`kSecPt`
Section point number within the current layer.

`cmname`
User-specified material name, left justified. It is passed in as an uppercase character string. Some internal material models are given names starting

| | |
|---|---|
| | with the "ABQ_" character string. To avoid conflict, you should not use "ABQ_" as the leading string for `cmname`. |
| `ndir` | Number of direct components in a symmetric tensor. |
| `nshr` | Number of indirect components in a symmetric tensor. |
| `nstatev` | Number of user-defined state variables that are associated with this material type (you define this as described in *Allocating Space for Solution-Dependent State Variables*). |
| `nfieldv` | Number of user-defined external field variables. |
| `nprops` | User-specified number of user-defined material properties. |
| `props(nprops)` | User-supplied material properties. |
| `tempOld(nblock)` | Temperatures at each material point at the beginning of the increment. |
| `tempNew(nblock)` | Temperatures at each material point at the end of the increment. |
| `fieldOld(nblock,nfieldv)` | Values of the user-defined field variables at each material point at the beginning of the increment. |
| `fieldNew(nblock,nfieldv)` | Values of the user-defined field variables at each material point at the end of the increment. |
| `stateOld(nblock,nstatev)` | State variables at each material point at the beginning of the increment. |
| `ebar(nblock,ndir+nshr)` | Modified Green strain tensor, $\bar{\boldsymbol{\varepsilon}}^G$, at each material point at the end of the increment. |
| `detu(nblock)` | $J$, determinant of deformation gradient (volume ratio) at the end of the increment. |

## Example: Using More than One User-Defined Anisotropic Hyperelastic Material Model

To use more than one user-defined anisotropic hyperelastic material model, the variable `cmname` can be tested for different material names inside user subroutine *VUANISOHYPER_STRAIN*, as illustrated below:

```
if (cmname(1:4) .eq. 'MAT1') then
   call VUANISOHYPER_STRAIN1(argument_list)
else if (cmname(1:4) .eq. 'MAT2') then
   call VUANISOHYPER_STRAIN2(argument_list)
end if
```

`VUANISOHYPER_STRAIN1` and `VUANISOHYPER_STRAIN2` are the actual subroutines containing the anisotropic hyperelastic models for each material `MAT1` and `MAT2`, respectively. Subroutine `VUANISOHYPER_STRAIN` merely acts as a directory here. The argument list can be the same as that used in subroutine `VUANISOHYPER_STRAIN`. The material names must be in uppercase characters since `cmname` is passed in as an uppercase character string.

## Example: Orthotropic Saint-Venant Kirchhoff Model

As a simple example of the coding of subroutine `VUANISOHYPER_STRAIN`, consider the generalization to anisotropic hyperelasticity of the Saint-Venant Kirchhoff model. The strain energy function of the Saint-Venant Kirchhoff model can be expressed as a quadratic function of the Green strain tensor, $\boldsymbol{\varepsilon}^G$, as

$$U\left(\boldsymbol{\varepsilon}^G\right) = \tfrac{1}{2}\boldsymbol{\varepsilon}^G : \mathbf{D} : \boldsymbol{\varepsilon}^G,$$

where $\mathbf{D}$ is the fourth-order elasticity tensor. The derivatives of the strain energy function with respect to the Green strain are given as

$$\frac{\partial U}{\partial \boldsymbol{\varepsilon}^G} = \mathbf{D} : \boldsymbol{\varepsilon}^G,$$

$$\frac{\partial^2 U}{\partial \boldsymbol{\varepsilon}^G \partial \boldsymbol{\varepsilon}^G} = \mathbf{D}.$$

However, subroutine VUANISOHYPER_STRAIN must return the derivatives of the strain energy function with respect to the modified Green strain tensor, $\overline{\boldsymbol{\varepsilon}}^G$, and the volume ratio, $J$, which can be accomplished easily using the following relationship between $\boldsymbol{\varepsilon}^G$, $\overline{\boldsymbol{\varepsilon}}^G$, and $J$:

$$\boldsymbol{\varepsilon}^G = J^{\frac{2}{3}}\overline{\boldsymbol{\varepsilon}}^G + \frac{1}{2}\left(J^{\frac{2}{3}} - 1\right)\mathbf{I},$$

where $\mathbf{I}$ is the second-order identity tensor. Thus, using the chain rule we find

$$\frac{\partial U}{\partial \overline{\boldsymbol{\varepsilon}}^G} = J^{\frac{2}{3}}\frac{\partial U}{\partial \boldsymbol{\varepsilon}^G},$$

$$\frac{\partial U}{\partial J} = \frac{\partial \boldsymbol{\varepsilon}^G}{\partial J} : \frac{\partial U}{\partial \boldsymbol{\varepsilon}^G},$$

$$\frac{\partial^2 U}{\partial \overline{\boldsymbol{\varepsilon}}^G \partial \overline{\boldsymbol{\varepsilon}}^G} = J^{\frac{4}{3}}\frac{\partial^2 U}{\partial \boldsymbol{\varepsilon}^G \partial \boldsymbol{\varepsilon}^G},$$

$$\frac{\partial^2 U}{\partial J^2} = \frac{\partial^2 \boldsymbol{\varepsilon}^G}{\partial J^2} : \frac{\partial U}{\partial \boldsymbol{\varepsilon}^G} + \frac{\partial \boldsymbol{\varepsilon}^G}{\partial J} : \frac{\partial^2 U}{\partial \boldsymbol{\varepsilon}^G \partial \boldsymbol{\varepsilon}^G} : \frac{\partial \boldsymbol{\varepsilon}^G}{\partial J},$$

$$\frac{\partial^2 U}{\partial \overline{\boldsymbol{\varepsilon}}^G \partial J} = \frac{2}{3J}J^{\frac{2}{3}}\frac{\partial U}{\partial \boldsymbol{\varepsilon}^G} + J^{\frac{2}{3}}\frac{\partial^2 U}{\partial \boldsymbol{\varepsilon}^G \partial \boldsymbol{\varepsilon}^G} : \frac{\partial \boldsymbol{\varepsilon}^G}{\partial J},$$

where

$$\frac{\partial \boldsymbol{\varepsilon}^G}{\partial J} = \frac{2}{3J}J^{\frac{2}{3}}\left(\overline{\boldsymbol{\varepsilon}}^G + \frac{1}{2}\mathbf{I}\right) = \frac{2}{3J}\left(\boldsymbol{\varepsilon}^G + \frac{1}{2}\mathbf{I}\right)$$

and

$$\frac{\partial^2 \boldsymbol{\varepsilon}^G}{\partial J^2} = -\frac{1}{3J}\frac{\partial \boldsymbol{\varepsilon}^G}{\partial J}.$$

In this example an auxiliary function is used to facilitate indexing into a fourth-order symmetric tensor. The subroutine would be coded as follows:

```
      subroutine vuanisohyper_strain (
C Read only -
     *     nblock,
     *     jElem, kIntPt, kLayer, kSecPt,
     *     cmname,
     *     ndir, nshr, nstatev, nfieldv, nprops,
     *     props, tempOld, tempNew, fieldOld, fieldNew,
     *     stateOld, ebar, detu,
C Write only -
     *     uDev, duDe, duDj,
     *     d2uDeDe, d2uDjDj, d2uDeDj,
     *     stateNew )
C
      include 'vaba_param.inc'
C
      dimension props(nprops),
     *  tempOld(nblock),
     *  fieldOld(nblock,nfieldv),
     *  stateOld(nblock,nstatev),
     *  tempNew(nblock),
     *  fieldNew(nblock,nfieldv),
     *  ebar(nblock,ndir+nshr), detu(nblock),
     *  uDev(nblock), duDe(nblock,ndir+nshr), duDj(nblock),
     *  d2uDeDe(nblock,*), d2uDjDj(nblock),
```

```
     *  d2uDeDj(nblock,ndir+nshr),
     *  stateNew(nblock,nstatev)
C
      character*80 cmname
C
      parameter( half = 0.5d0, one = 1.d0, two = 2.d0,
     *     third = 1.d0/3.d0, twothds = 2.d0/3.d0, four = 4.d0,
     *     dinv = 0.d0 )
C
C     Orthotropic Saint-Venant Kirchhoff strain energy function
C     (3D)
C
      D1111 = props(1)
      D1122 = props(2)
      D2222 = props(3)
      D1133 = props(4)
      D2233 = props(5)
      D3333 = props(6)
      D1212 = props(7)
      D1313 = props(8)
      D2323 = props(9)
C
      do k = 1, nblock
C
         d2UdE11dE11 = D1111
         d2UdE11dE22 = D1122
         d2UdE11dE33 = D1133
         d2UdE22dE11 = d2UdE11dE22
         d2UdE22dE22 = D2222
         d2UdE22dE33 = D2233
         d2UdE33dE11 = d2UdE11dE33
         d2UdE33dE22 = d2UdE22dE33
         d2UdE33dE33 = D3333
         d2UdE12dE12 = D1212
         d2UdE13dE13 = D1313
         d2UdE23dE23 = D2323
C
         xpow = exp ( log(detu(k)) * twothds )
         detuInv = one / detu(k)
C
         E11 = xpow * ebar(k,1) + half * ( xpow - one )
         E22 = xpow * ebar(k,2) + half * ( xpow - one )
         E33 = xpow * ebar(k,3) + half * ( xpow - one )
         E12 = xpow * ebar(k,4)
         E23 = xpow * ebar(k,5)
         E13 = xpow * ebar(k,6)
C
         term1 = twothds * detuInv
         dE11Dj = term1 * ( E11 + half )
         dE22Dj = term1 * ( E22 + half )
         dE33Dj = term1 * ( E33 + half )
         dE12Dj = term1 * E12
         dE23Dj = term1 * E23
         dE13Dj = term1 * E13
         term2 = - third * detuInv
         d2E11DjDj = term2 * dE11Dj
         d2E22DjDj = term2 * dE22Dj
         d2E33DjDj = term2 * dE33Dj
         d2E12DjDj = term2 * dE12Dj
         d2E23DjDj = term2 * dE23Dj
         d2E13DjDj = term2 * dE13Dj
```

```
C
          dUdE11 = d2UdE11dE11 * E11
     *           + d2UdE11dE22 * E22
     *           + d2UdE11dE33 * E33
          dUdE22 = d2UdE22dE11 * E11
     *           + d2UdE22dE22 * E22
     *           + d2UdE22dE33 * E33
          dUdE33 = d2UdE33dE11 * E11
     *           + d2UdE33dE22 * E22
     *           + d2UdE33dE33 * E33
          dUdE12 = two * d2UdE12dE12 * E12
          dUdE23 = two * d2UdE23dE23 * E23
          dUdE13 = two * d2UdE13dE13 * E13
C
          U = half * ( E11*dUdE11 + E22*dUdE22 + E33*dUdE33 )
     *         + E12*dUdE12 + E13*dUdE13 + E23*dUdE23
          uDev(k) = U
C
          duDe(k,1) = xpow * dUdE11
          duDe(k,2) = xpow * dUdE22
          duDe(k,3) = xpow * dUdE33
          duDe(k,4) = xpow * dUdE12
          duDe(k,5) = xpow * dUdE23
          duDe(k,6) = xpow * dUdE13
C
          xpow2 = xpow * xpow
C Only update nonzero components
          d2uDeDe(k,indx(1,1)) = xpow2 * d2UdE11dE11
          d2uDeDe(k,indx(1,2)) = xpow2 * d2UdE11dE22
          d2uDeDe(k,indx(2,2)) = xpow2 * d2UdE22dE22
          d2uDeDe(k,indx(1,3)) = xpow2 * d2UdE11dE33
          d2uDeDe(k,indx(2,3)) = xpow2 * d2UdE22dE33
          d2uDeDe(k,indx(3,3)) = xpow2 * d2UdE33dE33
          d2uDeDe(k,indx(4,4)) = xpow2 * d2UdE12dE12
          d2uDeDe(k,indx(5,5)) = xpow2 * d2UdE23dE23
          d2uDeDe(k,indx(6,6)) = xpow2 * d2UdE13dE13
C
          duDj(k) = dUdE11*dE11Dj + dUdE22*dE22Dj + dUdE33*dE33Dj
     *         + two * ( dUdE12*dE12Dj + dUdE13*dE13Dj
     *                 + dUdE23*dE23Dj )
          d2uDjDj(k)= dUdE11*d2E11DjDj+dUdE22*d2E22DjDj
     *               +dUdE33*d2E33DjDj
     *         + two*(dUdE12*d2E12DjDj+dUdE13*d2E13DjDj
     *                +dUdE23*d2E23DjDj)
     *         + d2UdE11dE11 * dE11Dj * dE11Dj
     *         + d2UdE22dE22 * dE22Dj * dE22Dj
     *         + d2UdE33dE33 * dE33Dj * dE33Dj
     *         + two  * ( d2UdE11dE22 * dE11Dj * dE22Dj
     *                  + d2UdE11dE33 * dE11Dj * dE33Dj
     *                  + d2UdE22dE33 * dE22Dj * dE33Dj )
     *         + four * ( d2UdE12dE12 * dE12Dj * dE12Dj
     *                    d2UdE13dE13 * dE13Dj * dE13Dj
     *                    d2UdE23dE23 * dE23Dj * dE23Dj )
C
          d2uDeDj(k,1) = xpow * ( term1 * dUdE11
     *                 + d2UdE11dE11 * dE11Dj
     *                 + d2UdE11dE22 * dE22Dj
     *                 + d2UdE11dE33 * dE33Dj )
          d2uDeDj(k,2) = xpow * ( term1 * dUdE22
     *                 + d2UdE22dE11 * dE11Dj
     *                 + d2UdE22dE22 * dE22Dj
```

```
     *                + d2UdE22dE33 * dE33Dj )
          d2uDeDj(k,3) = xpow * ( term1 * dUdE33
     *                + d2UdE33dE11 * dE11Dj
     *                + d2UdE33dE22 * dE22Dj
     *                + d2UdE33dE33 * dE33Dj )
          d2uDeDj(k,4) = xpow * ( term1 * dUdE12
     *                + two * d2UdE12dE12 * dE12Dj )
          d2uDeDj(k,5) = xpow * ( term1 * dUdE23
     *                + two * d2UdE23dE23 * dE23Dj )
          d2uDeDj(k,6) = xpow * ( term1 * dUdE13
     *                + two * d2UdE13dE13 * dE13Dj )
      end do
C
      return
      end
C
      integer function indx( i, j )
C
      include 'vaba_param.inc'
C
C     Function to map index from Square to Triangular storage
C     of symmetric matrix
C
      ii = min(i,j)
      jj = max(i,j)
C
      indx = ii + jj*(jj-1)/2
C
      return
      end
```

# VUCHARLENGTH: User subroutine to define characteristic element length at a material point.

**Products:** Abaqus/Explicit

**References:**

- **CHARACTERISTIC LENGTH*
- *VUCHARLENGTH*

## Overview

User subroutine *VUCHARLENGTH*:

- is called at all material points of elements for which the material definition includes a user-defined characteristic element length and the constitutive model requires a characteristic length;
- allows the definition of characteristic element length at a material point as a function of element topology, nodal and material point coordinates, and material orientation;
- can use field variables that are passed in; and
- can use solution-dependent state variables that are passed in.

## Defining Characteristic Element Length

The characteristic element length defined in user subroutine *VUCHARLENGTH* is used by Abaqus in regularization schemes needed to mitigate mesh dependency in constitutive models that include strain-softening, such as damage models (*Damage Evolution and Element Removal for Ductile Metals*), and concrete (*Concrete Smeared Cracking*). It could be used with built-in Abaqus material models as well as user subroutine–based material models.

The characteristic element length coming in user subroutine *VUCHARLENGTH* has a default value based on the geometric mean. This default value is a typical length of a line across an element for a first-order element and is half of the same typical length for a second-order element. For trusses the default value is a characteristic length along the element axis. For membranes and shells the default value is a characteristic length in the reference surface. For axisymmetric elements the default value is a characteristic length in the $r$–$z$ plane only.

Inside user subroutine *VUCHARLENGTH* you can redefine the value of the characteristic element length based on the element topology and geometry. The characteristic element length defined in user subroutine *VUCHARLENGTH* at a particular material point is passed to other user subroutines that are called at the same material point, such as user subroutines *VFABRIC*, *VUMAT*, *VUSDFLD*, and *VUEOS*. The characteristic element length calculated in user subroutine *VUCHARLENGTH* at a particular material point is also used in built-in Abaqus material models that require characteristic length and are called at the same material point.

## Array of Element Type and Geometric Properties

`jElType` contains information about the element type and the geometry. `jElType(1)` provides information about the shape of the element.

| jElType(1) | Shape |
|---|---|
| 1 | line |
| 2 | triangle |

| jElType(1) | Shape |
|---|---|
| 3 | quadrilateral |
| 4 | tetrahedron |
| 5 | wedge |
| 6 | hexahedron |

`jElType(2)` provides information about the element's dimensionality.

| jElType(2) | Space |
|---|---|
| 1 | 2D and plane strain |
| 2 | 3D |
| 3 | axisymmetric |
| 4 | plane stress |

`jElType(3)` provides information about the section of the element.

| jElType(3) | Section |
|---|---|
| 1 | solid |
| 2 | shell |
| 3 | truss |
| 4 | membrane |

## Elements

User subroutine *VUCHARLENGTH* can be used with membrane elements; shell elements; truss elements; and plane stress, plane strain, axisymmetric, and three-dimensional solid elements.

## Special Consideration for 8-Node Continuum Shell Elements

For 8-node hexahedron continuum shell elements (SC8R and SC8RT), the order of nodes in the array of nodal coordinates (`coordNode`) passed to user subroutine *VUCHARLENGTH* depends on the stacking direction. For a single element nodes 1–4 correspond to the bottom face and nodes 5–6 correspond to the top face.

## User Subroutine Interface

```
      subroutine vucharlength(
c Read only variables-
     1     nblock, nfieldv, nprops, ncomp, ndim, nnode, nstatev,
     2     kSecPt, kLayer, kIntPt, jElType, jElem,
     3     totalTime, stepTime, dt,
     4     cmname, coordMp, coordNode, direct, T, props,
     5     field, stateOld,
c Write only variables-
     6     charLength )
c
      include 'vaba_param.inc'
c
      dimension jElType(3), jElem(nblock), coordMp(nblock,ndim),
     1          coordNode(nblock,nnode,ndim),
     2          direct(nblock,3,3), T(nblock,3,3), props(nprops),
     3          stateOld(nblock,nstatev), charLength(nblock,ncomp),
```

```
     4          field(nblock, nfieldv)
c
     character*80 cmname
c
      do 100 k = 1, nblock

         user coding to define charLength(nblock,ncomp)

  100 continue
c
      return
      end
```

## Variables to Be Defined

| | |
|---|---|
| `charLength(nblock,ncomp)` | Characteristic element length. |

## Variables Passed in for Information

| | |
|---|---|
| `nblock` | Number of material points to be processed in this call to user subroutine *VUCHARLENGTH*. |
| `nfieldv` | Number of user-defined external field variables. |
| `nprops` | User-specified number of user-defined material properties. |
| `ncomp` | User-specified number of components of characteristic element length. If `ncomp` is greater than 1, only the first component of characteristic element length would be used in the built-in Abaqus material models. However, all the components could be used in the above-mentioned user subroutines. |
| `ndim` | Number of coordinate directions: 2 for two-dimensional models and 3 for three-dimensional models. |
| `nnode` | Number of nodes of the element. |
| `nstatev` | Number of user-defined state variables that are associated with this material type (define this as described in *Allocating Space for Solution-Dependent State Variables*). |
| `kSecPt` | Section point number within the current layer. |
| `kLayer` | Layer number (for composite shells). |
| `kIntPt` | Integration point number. |
| `jElType(3)` | Array containing information about the element type and geometry. |
| `jElem(nblock)` | Array of element numbers. |
| `totalTime` | Value of total time. The time at the beginning of the step is given by `totalTime-stepTime`. |
| `stepTime` | Value of time since the step began. |
| `dt` | Time increment size. |
| `cmname` | User-specified material name, left justified. It is passed in as an uppercase character string. |
| `coordMp(nblock,ndim)` | Material point coordinates. It is the midplane material point for shell elements. |
| `coordNode(nblock,nnode,ndim)` | Nodal coordinates. |

| | |
|---|---|
| `direct(nblock,3,3)` | An array containing the direction cosines of the material directions in terms of the global basis directions. For material point *k* `direct(k,1,1)`, `direct(k,2,1)`, and `direct(k,3,1)` give the (1, 2, 3) components of the first material direction; `direct(k,1,2)`, `direct(k,2,2)`, and `direct(k,3,2)` give the second material direction and so on. For shell and membrane elements the first two directions are in the plane of the element and the third direction is the normal. |
| `T(nblock,3,3)` | An array containing the direction cosines of the material orientation components relative to the element basis directions. For material point *k* this is the orientation that defines the material directions (`direct`) in terms of the element basis directions. For continuum elements `T` and `direct` are identical. For shell and membrane elements `T(k,1,1)` $= \cos\theta$, `T(k,1,2)` $= -\sin\theta$, `T(k,2,1)` $= \sin\theta$, `T(k,2,2)` $= \cos\theta$, `T(k,3,3)` $= 1.0$, and all other components are zero, where $\theta$ is the counterclockwise rotation around the normal vector that defines the orientation. If no orientation is used, `T` is an identity matrix. |
| `props(nprops)` | User-supplied material properties. |
| `field(nblock,nfieldv)` | Values of the user-defined field variables at each material point at the beginning of the increment. User subroutine *VUCHARLENGTH* is called before user subroutine *VUSDFLD*. Thus, any changes to the field variables made in user subroutine *VUSDFLD* are not available in user subroutine *VUCHARLENGTH*. |
| `stateOld (nblock, nstatev)` | State variables at each material point at the beginning of the increment. |

# VUCREEPNETWORK: User subroutine to define time-dependent behavior (creep) for models defined within the parallel rheological framework.

**Products:** Abaqus/Explicit

**References:**

- *Parallel Rheological Framework*
- *Nonlinear large-strain viscoelasticity with hyperelasticity*
- **VISCOELASTIC*

## Overview

User subroutine *VUCREEPNETWORK*:

- is intended to provide creep laws for nonlinear viscoelastic networks for models defined using the parallel rheological framework;
- can use and update solution-dependent state variables; and
- can be used in conjunction with user subroutine *VUSDFLD* to redefine any field variables before they are passed in.

## Model Description

The user subroutine allows a creep law of the following general form to be defined:

$$\dot{\bar{\varepsilon}}^{cr} = g^{cr}\left(\bar{\varepsilon}^{cr}, I_1^{cr}, \bar{I}_1, \bar{I}_2, J, p, q, t, \theta, FV\right),$$

where

$$I_1^{cr} = \mathbf{I} : \mathbf{C}^{cr},$$

and

$\mathbf{I}$

is the identity tensor,

$\mathbf{C}^{cr}$

is the right Cauchy-Green creep strain tensor,

$\dot{\bar{\varepsilon}}^{cr}$

is the equivalent creep strain rate,

$\bar{\varepsilon}^{cr}$

is the equivalent creep strain,

$\overline{I}_1$

is the first invariant of $\overline{\mathbf{B}}$,

$\overline{I}_2$

is the second invariant of $\overline{\mathbf{B}}$,

$J$

is the determinant of the deformation gradient, $\mathbf{F}$,

$p$

is the Kirchhoff pressure,

$q$

is the equivalent deviatoric Kirchhoff stress,

$t$

is the time,

$\theta$

is the temperature, and

$FV$

are field variables.

The left Cauchy-Green strain tensor, $\overline{\mathbf{B}}$, is defined as

$$\overline{\mathbf{B}} = \overline{\mathbf{F}}\overline{\mathbf{F}}^T,$$

where $\overline{\mathbf{F}}$ is the deformation gradient with volume change eliminated, which is computed using

$$\overline{\mathbf{F}} = J^{-\frac{1}{3}}\mathbf{F}.$$

The user subroutine must define the increment of creep equivalent strain, $\Delta\bar{\varepsilon}^{cr}$, as a function of the time increment, $\Delta t$, and the variables used in the definition of $g^{cr}$, as well as the derivatives of the equivalent creep strain increment with respect to those variables. If any solution-dependent state variables are included in the definition of $g^{cr}$, they must also be integrated forward in time in this user subroutine.

## User Subroutine Interface

```
      subroutine vucreepnetwork (
C Read only -
     *   nblock, jInfoArray, nstatev, nfieldv,
     *   nprops, nDg, stepTime, totalTime, dt,
     *   jElem, kIntPt, kLayer, kSecPt, cmname,
     *   props, coordMp, tempOld, fieldOld,
```

```
     *   stateOld, tempNew, fieldNew,
     *   nIarray, i_array, nRarray, r_array,
     *   q, p, eqcs, TrCc,
C Write only -
     *   dg, stateNew )
C
      include 'vaba_param.inc'
C
C     indices for information array
      parameter (i_inf_NetworkID = 1,
     *           i_inf_RateDepOff = 2)
C
C     indices for equivalent creep strain and its derivatives
      parameter( i_deqcs       = 1,
     *           i_DdeqcsDq    = 2,
     *           i_DdeqcsDeqcs = 3,
     *           i_DdeqcsDi1c  = 4 )
C
C     indices for strain invariants
      parameter( i_I1 = 1,
     *           i_I2 = 2,
     *           i_J  = 3 )
C
      dimension props(nprops),
     *  tempOld(nblock),
     *  fieldOld(nblock,nfieldv),
     *  stateOld(nblock,nstatev),
     *  tempNew(nblock),
     *  fieldNew(nblock,nfieldv),
     *  coordMp(nblock,*),
     *  jElem(nblock),
     *  i_array(nblock,nIarray),
     *  r_array(nblock,nRarray),
     *  q(nblock),
     *  p(nblock),
     *  eqcs(nblock),
     *  TrCc(nblock),
     *  stateNew(nblock,nstatev),
     *  dg(nblock,nDg),
     *  jInfoArray(*)

      character*80 cmname
C
      lNetworkID = jInfoArray(i_inf_NetworkID)
      lRateDepOff = jInfoArray(i_inf_RateDepOff)
C
      do 100 km = 1,nblock
        user coding
  100 continue

      return
      end
```

## Variables to Be Defined

`dg(nblock,i_deqcs)`

Equivalent creep strain increment, $\Delta\bar{\varepsilon}^{cr}$.

`dg(nblock,i_DdeqcsDq)`

The derivative: $\partial \Delta\bar{\varepsilon}^{cr} / \partial q$.

`dg(nblock,i_DdeqcsDeqcs)` The derivative: $\partial \Delta \bar{\varepsilon}^{cr} / \partial \bar{\varepsilon}^{cr}$.

`dg(nblock,i_DdeqcsDi1c)` The derivative: $\partial \Delta \bar{\varepsilon}^{cr} / \partial I_1^{cr}$.

## Variables That Can Be Updated

`stateNew(nblock,nstatev)` Array containing the user-defined solution-dependent state variables at this point.

## Variables Passed in for Information

`nblock` Number of material points to be processed in this call to user subroutine *VUCREEPNETWORK*.

`jInfoArray`

| | |
|---|---|
| `jInfoArray(i_inf_ NetworkID)` | Network identification number, which identifies the network for which creep is defined. |
| `jInfoArray(i_inf_RateDepOff)` | Flag indicating whether the routine is being called during a step where material rate dependence is ignored.<br>`jInfoArray(i_inf_RateDepOff)=0` indicates that the routine is being called during a step where material rate dependence is considered.<br>`jInfoArray(i_inf_RateDepOff)=1` indicates that the routine is being called during a step where material rate-dependence is ignored and that the step uses the fully relaxed long-term elastic response of the material. In this case, Abaqus/Explicit computes the equivalent creep strain increment from the kinematics and ignores all the variables defined in the routine except the internal state variables. If necessary, you should update the internal state variables in the array `stateNew` accordingly. |

`nstatev` Number of user-defined state variables that are associated with this material type (see *Allocating Space for Solution-Dependent State Variables*).

`nfieldv` Number of user-defined external field variables.

`nprops` User-specified number of user-defined material properties.

`nDg` Size of array `dg`.

`stepTime` Value of time since the step began.

`totalTime` Value of total time. The time at the beginning of the step is given by `totalTime-stepTime`.

`dt` Time increment size.

`jElem(nblock)` Array of element numbers.

`kIntPt` Integration point number.

`kLayer` Layer number (for composite shells).

`kSecPt` Section point number within the current layer.

`cmname` Material name, left justified. It is passed in as an uppercase character string. Some internal material models are given names starting with the “ABQ_”

| | |
|---|---|
| | character string. To avoid conflict, “ABQ_” should not be used as the leading string for `cmname`. |
| `props(nprops)` | User-supplied material properties. |
| `coordMp(nblock,*)` | Material point coordinates. It is the midplane material point for shell elements and the centroid for beam elements. |
| `tempOld(nblock)` | Temperatures at the material points at the beginning of the increment. |
| `fieldOld(nblock,nfieldv)` | Values of the user-defined field variables at the material points at the beginning of the increment. |
| `stateOld(nblock,nstatev)` | State variables at the material points at the beginning of the increment. |
| `tempNew(nblock)` | Temperatures at the material points at the end of the increment. |
| `fieldNew(nblock)` | Values of the user-defined field variables at the material points at the end of the increment. |
| `nIarray` | Size of array `i_array`. |
| `i_array(nblock,nIarray)` | Array containing integer arguments. Currently it is not used. |
| `nRarray` | Size of array `r_array`. |
| `r_array(nblock,i_I1)` | The first invariant, $\overline{I}_1$, of the left Cauchy-Green strain tensor, $\overline{\mathbf{B}}$. |
| `r_array(nblock,i_I2)` | The second invariant, $\overline{I}_2$, of the left Cauchy-Green strain tensor, $\overline{\mathbf{B}}$. |
| `r_array(nblock,i_J)` | The determinant of the deformation gradient, $\mathbf{F}$. |
| `q(nblock)` | Array containing equivalent deviatoric Kirchhoff stresses. |
| `p(nblock)` | Array containing Kirchhoff pressures. |
| `eqcs(nblock)` | Array containing equivalent creep strains. |
| `TrCc(nblock)` | Array containing the first invariants, $I_1^{cr}$, of the right Cauchy-Green creep strain tensor, $\mathbf{C}^{\mathbf{cr}}$. |

## Example: Power Law Strain Hardening Model

As an example of the coding of user subroutine *VUCREEPNETWORK*, consider the power law strain hardening model. In this case the equivalent creep strain rate is expressed as

$$\dot{\bar{\varepsilon}}^{cr} = \left(Aq^n\left[(m+1)\,\bar{\varepsilon}^{cr}\right]^m\right)^{\frac{1}{m+1}},$$

where

$\bar{\varepsilon}^{cr}$

is the equivalent creep strain,

$q$

is the equivalent deviatoric Kirchhoff stress, and

### $A$, $m$, and $n$

are material parameters.

The user subroutine would be coded as follows:

```
      subroutine vucreepnetwork (
C Read only -
```

```
     *   nblock, jInfoArray, nstatev, nfieldv,
     *   nprops, nDg, stepTime, totalTime, dt,
     *   jElem, kIntPt, kLayer, kSecPt, cmname,
     *   props, coordMp, tempOld, fieldOld,
     *   stateOld, tempNew, fieldNew,
     *   nIarray, i_array, nRarray, r_array,
     *   q, p, eqcs, TrCc,
C Write only -
     *   dg, stateNew )
C
      include 'vaba_param.inc'
C
      parameter (i_inf_NetworkID = 1,
     *           i_inf_RateDepOff = 2) ! =1 if RATE DEPENDENCE = OFF
C
      parameter( i_deqcs       = 1,
     *           i_DdeqcsDq    = 2,
     *           i_DdeqcsDeqcs = 3,
     *           i_DdeqcsDi1c  = 4 )
C
      parameter ( one = 1.d0, half = 0.5d0 )
      parameter ( eqcsSmall = 1.d-8 )
      parameter ( rMinVal = 1.d-12 )
C
      dimension props(nprops),
     *  tempOld(nblock),
     *  fieldOld(nblock,nfieldv),
     *  stateOld(nblock,nstatev),
     *  tempNew(nblock),
     *  fieldNew(nblock,nfieldv),
     *  coordMp(nblock,*),
     *  jElem(nblock),
     *  i_array(nblock,nIarray),
     *  r_array(nblock,nRarray),
     *  q(nblock),
     *  p(nblock),
     *  eqcs(nblock),
     *  TrCc(nblock),
     *  stateNew(nblock,nstatev),
     *  dg(nblock,nDg),
     *  jInfoArray(*)
C
      character*80 cmname
C
      lNetworkID = jInfoArray(i_inf_NetworkID)
      lRateDepOff = jInfoArray(i_inf_RateDepOff)
C
C Read properties
C
      rA = props(1)
      rN = props(2)
      rM = props(3)
C
C Update equivalent creep strain and its derivatives
C
      do k = 1, nblock
        om1 = one / ( one + rM )
        test = half - sign( half, q(k) - rMinVal )
        qInv = ( one - test ) / ( q(k) + test )
        eqcs_t = eqcs(k)
        if ( eqcs_t .le. eqcsSmall .and. q(k).gt.rMinVal ) then
```

```
C Initial guess based on constant creep strain rate during increment
          eqcs_t = dt*(exp(log(rA)+rN*log(q(k)))*
     *             ((one+rM)*dt)**rM)
        end if
C
        test2 = half - sign( half, eqcs_t - rMinVal )
        eqcsInv = ( one - test2 ) / ( eqcs_t + test2 )
        g = dt*(exp(log(rA)+rN*log(q(k)))*
     *       ((one+rM)*(test2+eqcs_t))**rM)**om1
        dg(k,i_deqcs) = g
        dg(k,i_DdeqcsDq) = qInv * rN * om1 * g
        dg(k,i_DdeqcsDeqcs) = eqcsInv * rM * om1 * g
      end do
C
      return
      end
```

# VUEL: User subroutine to define an element.

**Products:** Abaqus/Explicit

**References:**

- *User-Defined Elements*
- *User-Defined Element Library*
- *UEL*
- **UEL PROPERTY*
- **USER ELEMENT*

## Overview



**Warning:** This feature is intended for advanced users only. Its use in all but the simplest test examples will require considerable coding by the user/developer. *User-Defined Elements* should be read before proceeding.

User subroutine *VUEL*:

- will be called for each element that is of a general user-defined element type each time element calculations are required; and
- (or subroutines called by user subroutine *VUEL*) must perform all the calculations for the element, appropriate to the current activity in the analysis.

## User Subroutine Interface

```
      SUBROUTINE VUEL(nblock,rhs,amass,dtimeStable,svars,nsvars,
     1                energy,
     2                nnode,ndofel,props,nprops,jprops,njprops,
     3                coords,mcrd,u,du,v,a,
     4                jtype,jElem,
     5                time,period,dtimeCur,dtimePrev,kstep,kinc,
     6                lflags,
     7                dMassScaleFactor,
     8                predef,npredef,
     9                jdltyp, adlmag)
C
      include 'vaba_param.inc'

C     operational code keys
      parameter ( jMassCalc            = 1,
     *            jIntForceAndDtStable = 2,
     *            jExternForce         = 3)

C     flag indices
      parameter (iProcedure = 1,
     *           iNlgeom    = 2,
     *           iOpCode    = 3,
     *           nFlags     = 3)

C     energy array indices
      parameter ( iElPd = 1,
     *            iElCd = 2,
```

```
     *             iElIe = 3,
     *             iElTs = 4,
     *             iElDd = 5,
     *             iElBv = 6,
     *             iElDe = 7,
     *             iElHe = 8,
     *             iUnused = 9,
     *             iElTh = 10,
     *             iElDmd = 11,
     *             iElDc = 12,
     *             nElEnergy = 12)

C     predefined variables indices
      parameter ( iPredValueNew = 1,
     *            iPredValueOld = 2,
     *            nPred         = 2)

C     time indices
      parameter (iStepTime  = 1,
     *           iTotalTime = 2,
     *           nTime      = 2)

     dimension rhs(nblock,ndofel),amass(nblock,ndofel,ndofel),
     1          dtimeStable(nblock),
     2          svars(nblock,nsvars),energy(nblock,nElEnergy),
     3          props(nprops),jprops(njprops),
     4          jElem(nblock),time(nTime),lflags(nFlags),
     5          coords(nblock,nnode,mcrd),
     6          u(nblock,ndofel), du(nblock,ndofel),
     7          v(nblock,ndofel), a(nblock, ndofel),
     8          dMassScaleFactor(nblock),
     9          predef(nblock,nnode,npredef,nPred),
     *          adlmag(nblock)


      do kblock = 1,nblock
         user coding to define  rhs, amass, dtimeStable, svars and energy
       end do

      RETURN
      END
```

## Variables to Be Defined

Some of the following arrays depend on the value of the `lflags` array.

`rhs` An array containing the contributions of each element to the right-hand-side vector of the overall system of equations. Depending on the settings of the `lflags` array, it contains either the internal force from the element or the external load calculated from the specified distributed loads.

`amass` An array containing the contribution of each element to the mass matrix of the overall system of equations.

All nonzero entries in `amass` should be defined. Moreover, the mass matrix must be symmetric. There are several other requirements that apply depending on the active degrees of freedom specified. These requirements are explained in detail below.

`dtimeStable` A scalar value defining, for each element, the upper limit of the time increment for stability considerations. This would be the maximum time increment to be used in the subsequent

increment for this element to be stable (to satisfy the Courant condition). This value depends strongly on the element formulation, and it is important that is computed appropriately.

`svars` An array containing the values of the solution-dependent state variables associated with each element. The number of such variables is `nsvars` (see below). You define the meaning of these variables.

This array is passed into *VUEL* containing the values of these variables at the start of the current increment. In most cases they should be updated to be the values at the end of the increment. In rare cases such an update is not required.

`energy` The array `energy` contains the values of the energy quantities associated with each element. The values in this array when *VUEL* is called are the element energy quantities at the start of the current increment. They should be updated to the correct values at the end of the current increment; otherwise, plots of the energy balance for the entire model will not be accurate. You cannot modify or update the kinetic energy. Depending on the element formulation, many of these energies could be zero at all times. The entries in the array are as follows:

| | |
|---|---|
| `energy(nblock,iElPd )` | Plastic dissipation. |
| `energy(nblock,iElCd)` | Creep dissipation. |
| `energy(nblock,iElIe)` | Internal energy. |
| `energy(nblock,iElTs)` | Transverse shear energy. |
| `energy(nblock,iElDd)` | Material damping dissipation. |
| `energy(nblock,iElBv)` | Bulk viscosity dissipation. |
| `energy(nblock,iElDe)` | Drill energy. |
| `energy(nblock,iElHe)` | Hourglass energy. |
| `energy(nblock,iElTh)` | Heat energy. |
| `energy(nblock,iElDmd)` | Damage dissipation. |
| `energy(nblock,iElDc )` | Distortion control energy. |

## Variables Passed in for Information

**Arrays:**

`props` A floating point array containing the `nprops` real property values defined for use with each element processed. `nprops` is the user-specified number of real property values. See *Defining the Element Properties*.

`jprops` An integer array containing the `njprops` integer property values defined for use with each element processed. `njprops` is the user-specified number of integer property values. See *Defining the Element Properties*.

`coords` An array containing the original coordinates of the nodes of the element. `coords(kblock,k1,k2)` is the `k2`th coordinate of the `k1`th node of the `kblock` element.

`u, du, v, a` Arrays containing the basic solution variables (displacements, rotations, temperatures, pressures, depending on the degree of freedom) at the nodes of the element. Values are provided as follows, illustrated below for the `k1`th degree of freedom of the `kblock` element:

| | |
|---|---|
| `u(kblock,k1)` | Total value of the variables (such as displacements or rotations) at the end of the current increment. |
| `du(kblock,k1)` | Incremental values of the variables in the current increment. |

**Arrays:**

| | |
|---|---|
| `v(kblock,k1)` | Time rate of change of the variables (velocities, rates of rotation) at the midpoint of the increment. |
| `a(kblock,k1)` | Accelerations of the variables at the beginning of the current increment. |

`jElem` `jElem(kblock)` contains the element number for the `kblock` element.

`adlmag` `adlmag(kblock)` is the total load magnitude of the load type `jdltyp` (integer identifying the load number for distributed load type U*n*) distributed load at the end of the current increment for distributed loads of type U*n*.

`predef` An array containing the values of predefined field variables, such as temperature in an uncoupled stress/displacement analysis, at the nodes of the element (*Predefined Fields*).

The second index, `k2`, indicates the local node number on the `kblock` element. The third index, `k3`, indicates the variable: the temperature is stored if the index is 1, and the predefined field variables are stored if the indices are greater than or equal to 2. The fourth index of the array, `k4`, is either 1 or 2, with 1 indicating the value of the field variable at the end of the increment and 2 indicating the value of the field variable at the beginning of the increment.

| | |
|---|---|
| `predef(kblock,k2,1,k4)` | Temperature. |
| `predef(kblock,k2,2,k4)` | First predefined field variable. |
| `predef(kblock,k2,3,k4)` | Second predefined field variable. |
| Etc. | Any other predefined field variable. |
| `predef(kblock,k2,k3,k4)` | Value of the `(k3-1)`th predefined field variable at the `k2`th node of the element at the beginning or the end of the increment. |
| `predef(kblock,k2,k3,1)` | Values of the variables at the end of the current increment. |
| `predef(kblock,k2,k3,2)` | Values of the variables at the beginning of the current increment. |

`lflags` An array containing the flags that define the current solution procedure and requirements for element calculations.

| | |
|---|---|
| `lflags(iProcedure)` | Defines the procedure type. See *Results File* for the key used for each procedure. |
| `lflags(iNlgeom)=0` | Small-displacement analysis. |
| `lflags(iNlgeom)=1` | Large-displacement analysis (nonlinear geometric effects included in the step; see *General and Perturbation Procedures*). |
| `lflags(iOpCode)=jMassCalc` | Define the mass matrix `amass` in the beginning of the analysis. |
| `lflags(iOpCode)=jIntForceAndDtStable` | Define the element internal force. Define the stable time increment as well. |
| `lflags(iOpCode)=jExternForce` | Define the distributed load effect on the external force associated with the element. |

`dMassScaleFactor` An array containing the mass scale factors for each element.

`time(iStepTime)` Current value of step time.

`time(iTotalTime)` Current value of total time.

**Scalar parameters:**

| | |
|---|---|
| `nblock` | Number of user elements to be processed in this call to `VUEL`. |
| `dtimeCur` | Current time increment. |
| `dtimePrev` | Previous time increment. |
| `period` | Time period of the current step. |
| `ndofel` | Number of degrees of freedom in the elements processed. |
| `nsvars` | User-defined number of solution-dependent state variables associated with the element (*Defining the Number of Solution-Dependent Variables That Must Be Stored within the Element*). |
| `nprops` | User-defined number of real property values associated with the elements processed (*Defining the Element Properties*). |
| `njprops` | User-defined number of integer property values associated with the elements processed (*Defining the Element Properties*). |
| `mcrd` | `mcrd` is defined as the maximum of the user-defined maximum number of coordinates needed at any node point (*Defining the Maximum Number of Coordinates Needed at Any Nodal Point*) and the value of the largest active degree of freedom of the user element that is less than or equal to 3. For example, if you specify that the maximum number of coordinates is 1 and the active degrees of freedom of the user element are 2, 3, and 6 `mcrd` will be 3. If you specify that the maximum number of coordinates is 2 and the active degree of freedom of the user element is 11, `mcrd` will be 2. |
| `nnode` | User-defined number of nodes on the elements (*Defining the Number of Nodes Associated with the Element*). |
| `jtype` | Integer defining the element type. This is the user-defined integer value *n* in element type VU*n* (*Assigning an Element Type Key to a User-Defined Element*). |
| `kstep` | Current step number. |
| `kinc` | Current increment number. |
| `npredef` | Number of predefined field variables, including temperature. For user elements Abaqus/Explicit uses one value for each field variable per node. |

## `VUEL` Conventions

The solution variables (displacement, velocity, etc.) are arranged on a node/degree of freedom basis. The degrees of freedom of the first node are first, followed by the degrees of freedom of the second node, etc. The degrees of freedom that will be updated automatically in Abaqus/Explicit are: 1–3 (displacements), 4–6 (rotations), 8 (pressure), and 11 (temperature). Depending on the procedure type (see below), only some of the degrees of freedom listed above will be updated. Other degrees of freedom will not be updated by the time integration procedure in Abaqus/Explicit and, hence, should not be used.

The mass matrix defined in user subroutine *VUEL* must be symmetric. In addition, the following requirements apply:

- The mass matrix entries associated with the translational degrees of freedom for a particular node must be diagonal. Moreover, these diagonal entries must be equal to each other.
- There must be no coupling (off-diagonal) entries specified between degrees of freedom of different kinds. For example, you cannot specify nonzero mass matrix entries to couple the translational degrees of freedom to the rotational degrees of freedom.
- There must be no coupling (off-diagonal) entries specified between degrees of freedom belonging to different nodes.

You must be using appropriate lumping techniques to provide a mass matrix that follows these requirements. For the rotational degrees of freedom at a particular node in three-dimensional analyses, you can specify a fully populated symmetric 3 × 3 inertia tensor.

## Usage with General Nonlinear Procedures

The following illustrates the use in explicit dynamic procedures:

### Direct-Integration Explicit Dynamic Analysis (`lflags(iProcedure)=17`)

- Automatic updates for degrees of freedom 1–6, 8, and 11.
- The governing equations are as described in *Explicit Dynamic Analysis*.
- Coding for the operational code `lflags(iOpCode)=jExternForce`is optional.

### Transient Fully Coupled Thermal-Stress Analysis (`lflags(iProcedure)=74`)

- Automatic updates for degrees of freedom 1–6 and 11.
- The governing equations are as described in *Fully Coupled Thermal-Stress Analysis in Abaqus/Explicit*.
- Coding for the operational code `lflags(iOpCode)=jExternForce` is optional.

## Example: Structural User Element

A structural user element has been created to demonstrate the usage of subroutine *VUEL*. These user-defined elements are applied in a number of analyses. The following excerpt is from the verification problem that invokes the structural user element in an explicit dynamic procedure:

```
*USER ELEMENT, NODES=2, TYPE=VU1, PROPERTIES=4, COORDINATES=3,
VARIABLES=12
1, 2, 3
*ELEMENT, TYPE=VU1
101, 101, 102
*ELGEN, ELSET=VUTRUSS
101, 5
*UEL PROPERTY, ELSET=VUTRUSS
0.002, 2.1E11, 0.3, 7200.
```

The user element consists of two nodes that are assumed to lie parallel to the *x*-axis. The element behaves similarly to a linear truss element. The supplied element properties are the cross-sectional area, Young's modulus, Poisson's ratio, and density, respectively.

The next excerpt shows the listing of the subroutine. The user subroutine has been coded for use in an explicit dynamic analysis. The names of the verification input files associated with the subroutine and these procedures can be found in *VUEL*.

```
      subroutine vuel(
     *     nblock,
     *     rhs,amass,dtimeStable,
     *     svars,nsvars,
     *     energy,
     *     nnode,ndofel,
     *     props,nprops,
     *     jprops,njprops,
     *     coords,ncrd,
     *     u,du,v,a,
```

```
     *     jtype,jElem,
     *     time,period,dtimeCur,dtimePrev,kstep,kinc,lflags,
     *     dMassScaleFactor,
     *     predef,npredef,
     *     ndload,adlmag)

      include 'vaba_param.inc'

c     operation code
      parameter ( jMassCalc            = 1,
     *            jIntForceAndDtStable = 4)

c     flags
      parameter (iProcedure = 1,
     *           iNlgeom    = 2,
     *           iOpCode    = 3,
     *           nFlags     = 3)

c     procedure flags
      parameter ( jDynExplicit = 17 )

c     time
      parameter (iStepTime  = 1,
     *           iTotalTime = 2,
     *           nTime      = 2)

c     energies
      parameter ( iElPd = 1,
     *            iElCd = 2,
     *            iElIe = 3,
     *            iElTs = 4,
     *            iElDd = 5,
     *            iElBv = 6,
     *            iElDe = 7,
     *            iElHe = 8,
     *            iElKe = 9,
     *            iElTh = 10,
     *            iElDmd = 11,
     *            iElDc = 12,
     *            nElEnergy = 12)

      parameter (factorStable = 0.99d0)
      parameter (zero = 0.d0, half = 0.5d0, one = 1.d0, two=2.d0)
C
      dimension rhs(nblock,ndofel), amass(nblock,ndofel,ndofel),
     *     dtimeStable(nblock),
     *     svars(nblock,nsvars), energy(nblock,nElEnergy),
     *     props(nprops), jprops(njprops),
     *     jElem(nblock), time(nTime), l(nFlags),
     *     coords(nblock,nnode,ncrd), u(nblock,ndofel),
     *     du(nblock,ndofel), v(nblock,ndofel), a(nblock, ndofel),
     *     predef(nblock, nnode, npredef, nPred), adlmag(nblock),
     *     dMassScaleFactor(nblock)


c     Notes:
c        Define only nonzero entries; the arrays to be defined
c        have been zeroed out just before this call

      if (jtype .eq. 1001 .and.
     *    lflags(iProcedure).eq.jDynExplicit) then
```

```
        area0 = props(1)
        eMod  = props(2)
        anu   = props(3)
        rho   = props(4)

        eDampTra    = zero
        amassFact0  = half*area0*rho

        if ( lflags(iOpCode).eq.jMassCalc ) then
           do kblock = 1, nblock
c             use original distance to compute mass
              alenX0 = (coords(kblock,2,1) - coords(kblock,1,1))
              alenY0 = (coords(kblock,2,2) - coords(kblock,1,2))
              alenZ0 = (coords(kblock,2,3) - coords(kblock,1,3))
              alen0 = sqrt(alenX0*alenX0 + alenY0*alenY0 +
     *                     alenZ0*alenZ0)
              am0   = amassFact0*alen0
              amass(kblock,1,1) = am0
              amass(kblock,2,2) = am0
              amass(kblock,3,3) = am0
              amass(kblock,4,4) = am0
              amass(kblock,5,5) = am0
              amass(kblock,6,6) = am0

           end do
        else if ( lflags(iOpCode) .eq.
     *            jIntForceAndDtStable) then
           do kblock = 1, nblock
              alenX0 = (coords(kblock,2,1) - coords(kblock,1,1))
              alenY0 = (coords(kblock,2,2) - coords(kblock,1,2))
              alenZ0 = (coords(kblock,2,3) - coords(kblock,1,3))
              alen0 = sqrt(alenX0*alenX0 + alenY0*alenY0 +
     *                     alenZ0*alenZ0)
              vol0 = area0*alen0
              amElem0 = two*amassFact0*alen0

              alenX = alenX0
     *              +  (u(kblock,4)        -  u(kblock,1))
              alenY = alenY0
     *              +  (u(kblock,5)        -  u(kblock,2))
              alenZ = alenZ0
     *              +  (u(kblock,6)        -  u(kblock,3))
              alen = sqrt(alenX*alenX + alenY*alenY + alenZ*alenZ)
              area   = vol0/alen
              ak     = area*eMod/alen

c             stable time increment for translations
              dtimeStable(kblock) = factorStable*sqrt(amElem0/ak)

c             force = E * logarithmic strain *current area
              strainLog = log(alen/alen0)
              fElasTra = eMod*strainLog*area

              forceTra = fElasTra

c             assemble internal load in RHS
              rhs(kblock,1) = -forceTra
              rhs(kblock,4) =  forceTra

c             internal energy calculation
```

```
               alenOld  = svars(kblock,1)
               fElasTraOld = svars(kblock,2)

               energy(kblock, iElIe) = energy(kblock, iElIe) +
     *              half*(fElasTra+fElasTraOld)*(alen - alenOld)

c              update state variables
               svars(kblock,1) = alen
               svars(kblock,2) = fElasTra

            end do
         end if

       end if
c
       return
       end
```

# VUEOS: User subroutine to define equation of state material model.

**Products:** Abaqus/Explicit

**References:**

- *Equation of State*
- **EOS*
- *Equation of state material*

## Overview

User subroutine *VUEOS*:

- can be used to define the hydrodynamic material model in which the material's volumetric response is determined by the user-defined equation of state;
- will be called for blocks of material calculation points for which the material definition contains a user-defined equation of state;
- can use and update solution-dependent state variables; and
- can use any field variables that are passed in.

## User Subroutine Interface

```
      subroutine vueos (
C Read only (unmodifiable) variables -
     1     nblock,
     2     jElem, kIntPt, kLayer, kSecPt,
     3     stepTime, totalTime, dt, cmname,
     4     nstatev, nfieldv, nprops,
     5     props, tempOld, tempNew, fieldOld, fieldNew,
     6     stateOld, charLength, coordMp,
     7     densityMean, refDensity, densityNew,
     8     dkk, Em,
C Write only (modifiable) variables -
     8     press, dPdRho, dPdEm,
     9     stateNew )
C
      include 'vaba_param.inc'
C
      dimension props(nprops),
     1  tempOld(nblock),
     2  fieldOld(nblock,nfieldv),
     3  stateOld(nblock,nstatev),
     4  tempNew(nblock),
     5  fieldNew(nblock,nfieldv),
     6  charLength(nblock), coordMp(nblock,*),
     7  densityMean(nblock), refDensity(nblock),
     8  densityNew(nblock),
     9  dkk(nblock), Em(nblock),
     1  press(nblock),dPdRho(nblock),dPdEm(nblock),
     2  stateNew(nblock)
C
      character*80 cmname
```

```
C
      do 100 km = 1,nblock
        user coding to define/update press, dPdRho, dPdEm
  100 continue

      return
      end
```

## Variables to Be Defined

`press(nblock)` The material point pressure stress, $p$.

`dPdRho(nblock)` The derivative of the pressure with respect to the density, $\partial p/\partial \rho$. This quantity is needed for the evaluation of the effective moduli of the material, which enters the stable time increment calculation.

`dPdEm(nblock)` The derivative of the pressure with respect to the internal energy, $\partial p/\partial E_m$. This quantity is needed for the iterative Newton loop used outside of the user subroutine to solve for pressure.

## Variables That Can Be Updated

`stateNew(nblock,nstatev)` State variables at each material point at the end of the increment. You define the size of this array by allocating space for it (see *About User Subroutines and Utilities* for more information).

## Variables Passed in for Information

`nblock` Number of material points to be processed in this call to *VUEOS*.

`jElem(nblock)` Array of element numbers.

`kIntPt` Integration point number.

`kLayer` Layer number.

`kSecPt` Section point number within the current layer.

`stepTime` Value of time since the step began.

`totalTime` Value of total time. The time at the beginning of the step is given by `totalTime - stepTime`.

`dt` Time increment size.

`cmname` User-specified material name, left justified. It is passed in as an uppercase character string. Some internal material models are given names starting with the "ABQ_" character string. To avoid conflict, you should not use "ABQ_" as the leading string for `cmname`.

`nstatev` Number of user-defined state variables that are associated with this material type (you define the number as described in *Allocating Space for Solution-Dependent State Variables*).

`nfieldv` Number of user-defined external field variables.

`nprops` User-specified number of user-defined material properties.

`props(nprops)` User-supplied material properties.

`tempOld(nblock)` Temperatures at each material point at the beginning of the increment.

`tempNew(nblock)` Temperatures at each material point at the end of the increment.

`fieldOld(nblock,nfieldv)` Values of the user-defined field variables at each material point at the beginning of the increment.

`fieldNew(nblock,nfieldv)` Values of the user-defined field variables at each material point at the end of the increment.

`stateOld(nblock,nstatev)` State variables at each material point at the beginning of the increment.

`charLength(nblock)` Characteristic element length, which is either the default value based on the geometric mean or the user-defined characteristic element length defined in user subroutine `VUCHARLENGTH`. The default value is a typical length of a line across an element for a first-order element; it is half the same typical length for a second-order element. For beams, pipes, and trusses, the default value is a characteristic length along the element axis. For membranes and shells it is a characteristic length in the reference surface. For axisymmetric elements it is a characteristic length in the $r$–$z$ plane only. For cohesive elements it is equal to the constitutive thickness.

`coordMp(nblock,*)` Material point coordinates.

`densityMean(nblock)` Value of density at the middle of the current increment.

`refDensity(nblock)` The reference density.

`densityNew(nblock)` The current density for this increment.

`dkk(nblock)` The volumetric strain increment.

`Em(nblock)` The element specific internal energy (per unit mass)

## Example: User Subroutine VUEOS to Reproduce Results Obtained with *EOS, TYPE=USUP

As a simple example of coding of user subroutine *VUEOS*, consider the following form of the Mie-Grüneisen equation of state with $s = 0.0$ and a linear dependency between pressure and internal energy:

$$p = \rho_0 c_0^2 \eta \left(1 - \frac{\Gamma_0 \eta}{2}\right) + \Gamma_0 \rho_0 E_m,$$

where $\eta = 1 - \rho_0/\rho$. Therefore, the results obtained with user subroutine *VUEOS* should be the same as the results obtained with the linear $U_s - U_p$ type of EOS already available in *Equation of state material*.

The code in user subroutine *VUEOS* must return the pressure, $p$, as in the above equation; the derivative of the pressure with respect to the density, $\partial p/\partial \rho$; and the derivative of the pressure with respect to the energy, $\partial p/\partial E_m$. For the case considered here, these values are

$$\frac{\partial p}{\partial \rho} = \frac{\rho_0^2 c_0^2}{\rho^2}\left(1 - \Gamma_0 \eta\right),$$

$$\frac{\partial p}{\partial E_m} = \Gamma_0 \rho_0.$$

```
      subroutine vueos (
C Read only (unmodifiable) variables -
     1     nblock,
     2     jElem, kIntPt, kLayer, kSecPt,
     3     stepTime, totalTime, dt, cmname,
     4     nstatev, nfieldv, nprops,
     5     props, tempOld, tempNew, fieldOld, fieldNew,
     6     stateOld, charLength, coordMp,
     7     densityMean, refDensity, densityNew,
     8     dkk, Em,
C Write only (modifiable) variables -
```

```
     8     press, dPdRho, dPdEm,
     9     stateNew )
C
      include 'vaba_param.inc'
C
      dimension props(nprops),
     1  tempOld(nblock),
     2  fieldOld(nblock,nfieldv),
     3  stateOld(nblock,nstatev),
     4  tempNew(nblock),
     5  fieldNew(nblock,nfieldv),
     6  charLength(nblock), coordMp(nblock,*),
     7  densityMean(nblock), refDensity(nblock),
     8  densityNew(nblock),
     9  dkk(nblock), Em(nblock),
     1  press(nblock),dPdRho(nblock),dPdEm(nblock),
     2  stateNew(nblock)
C
      character*80 cmname
C
      parameter ( zero = 0.d0, one = 1.d0, half = 0.5d0 )
C
      c0 = props(1)
      gamma0 = props(2)
      c02 = c0*c0
C
      do k=1, nblock

       rho0 = refDensity(k)
       eta = one - rho0/densityNew(k)
       f1 = rho0*c02*eta*(one-half*gamma0*eta)
       f2 = gamma0*rho0
       press(k) = f1 + f2*Em(k)
C dP/dEm
       dPdEm(k) = f2
C dP/dRho

       dPdRho(k)= c02*(rho0/densityNew(k))**2*(one-gamma0*eta)

      end do
C
      return
      end
```

# VUEXPAN: User subroutine to define thermal strain increments.

**Products:** Abaqus/Explicit

**References:**

## Overview

User subroutine *VUEXPAN*:

- can be used to define thermal strain increments as a function of temperature, time, element number, predefined field variables, etc.;
- will be called at all integration points of the elements for which the material contains user-defined thermal expansion; and
- is intended for models in which the thermal strains depend on temperature, time, state variables, and/or predefined fields in complex ways.

## User Subroutine Interface

```
      subroutine vuexpan (
C Read only (unmodifiable)variables –
     * nblock, nDir, nShr, nExpanType,
     * nElem, nIntPt, nLayer, nSectPt,
     * stepTime, totalTime, dt, cmname,
     * nstatev, nfieldv, nprops, props,
     * tempOld, tempNew, fieldOld, fieldNew,
     * stateOld,
C Write only (modifiable) variable –
     * stateNew, strainThInc, dStrainTherDT )
C
      include 'vaba_param.inc'
C
      dimension strainThInc(nblock,nDir+nShr),
     * dStrainTherDT(nblock,nDir+nShr),
     * nElem(nblock), props(nprops),
     * tempOld(nblock), tempNew(nblock),
     * fieldOld(nblock,nfieldv),
     * fieldNew(nblock,nfieldv),
     * stateOld(nblock,nstatev),
     * stateNew(nblock,nstatev)
C
      character*80 cmname
C
      do 100 km = 1, nblock
        user coding to define strainThInc, dStrainTherDT,
        and, optionally, stateNew
  100 continue

      return
      end
```

## Variables to Be Defined

`strainThInc(nblock,nDir+nShr)` Increments of thermal strain. The number of values to be defined and the order in which they are arranged depend on the type of thermal expansion being defined.

- For isotropic expansion give the isotropic thermal strain increment as the first and only component of the matrix.
- For orthotropic expansion give $\Delta\epsilon^{th}_{11}$, $\Delta\epsilon^{th}_{22}$, and $\Delta\epsilon^{th}_{33}$ as the first, second, and third components of the matrix, respectively.
- For anisotropic expansion give $\Delta\epsilon^{th}_{11}$, $\Delta\epsilon^{th}_{22}$, $\Delta\epsilon^{th}_{33}$, $\Delta\epsilon^{th}_{12}$, $\Delta\epsilon^{th}_{23}$, and $\Delta\epsilon^{th}_{13}$. Direct components are stored first, followed by shear components in the order presented here. For plane stress only three components of the matrix are needed; give $\Delta\epsilon^{th}_{11}$, $\Delta\epsilon^{th}_{22}$, and $\Delta\epsilon^{th}_{12}$ as the first, second, and third components, respectively.

`dStrainTherDT(nblock,nDir+nShr)` Variation of thermal strains with respect to temperature, $\partial\epsilon^{th}/\partial\theta$. The number of values and the order in which they are arranged depend on the type of thermal expansion being defined.

- For isotropic expansion give the variation of the isotropic thermal strain with respect to temperature as the first and only component of the matrix.
- For orthotropic expansion give $\partial\epsilon^{th}_{11}/\partial\theta$, $\partial\epsilon^{th}_{22}/\partial\theta$, and $\partial\epsilon^{th}_{33}/\partial\theta$ as the first, second, and third components of the matrix, respectively.
- For anisotropic expansion give $\partial\epsilon^{th}_{11}/\partial\theta$, $\partial\epsilon^{th}_{22}/\partial\theta$, $\partial\epsilon^{th}_{33}/\partial\theta$, $\partial\epsilon^{th}_{12}/\partial\theta$, $\partial\epsilon^{th}_{23}/\partial\theta$, and $\partial\epsilon^{th}_{13}/\partial\theta$. Direct components are stored first, followed by shear components in the order presented here. For plane stress only three components of the matrix are needed; give $\partial\epsilon^{th}_{11}/\partial\theta$, $\partial\epsilon^{th}_{22}/\partial\theta$, and $\partial\epsilon^{th}_{12}/\partial\theta$ as the first, second, and third components, respectively.

`stateNew(nblock,nstatev)` Values of state variables at each material point at the end of the increment.

## Variables Passed in for Information

| | |
|---|---|
| `nblock` | Number of points to be processed in this call to *VUEXPAN*. |
| `nDir` | Number of direct components in a symmetric tensor. |
| `nShr` | Number of indirect components in a symmetric tensor. |
| `nExpansionType` | The thermal expansion type: 1 for isotropic expansion, 2 for orthotropic expansion, and 3 for anisotropic expansion. |
| `nElem(nblock)` | User-defined element number. |

| | |
|---|---|
| `nInt` | Integration point number. |
| `nLayer` | Layer number. |
| `nSectPt` | Section point number in the current layer. |
| `stepTime` | Value of time since the step began. |
| `totalTime` | Value of total time. The time at the beginning of step is given by `totalTime - stepTime`. |
| `dt` | Time increment size. |
| `cmname` | User-specified material name, left justified. It is passed in as an uppercase character string. Some internal material models are given names starting with the "ABQ_" character string. To avoid a conflict, you should not use "ABQ_" as the leading string for `cmname`. |
| `nstatev` | Number of user-defined state variables that are associated with this material. |
| `nfieldv` | Number of user-defined external field variables. |
| `nprops` | User-specified number of user-defined thermal expansion properties. |
| `props(nprops)` | User-supplied thermal expansion properties. |
| `tempOld(nblock)` | Temperatures at each material point at the beginning of the increment. |
| `tempNew(nblock)` | Temperatures at each material point at the end of the increment. |
| `fieldOld(nblock,nfieldv)` | Values of user-defined field variables at each material point at the beginning of the increment. |
| `fieldNew(nblock,nfieldv)` | Values of user-defined field variables at each material point at the end of the increment. |
| `stateOld(nblock,nstatev)` | Values of state variables at each material point at the beginning of the increment. |

# VUFIELD: User subroutine to specify predefined field variables.

**Products:** Abaqus/Explicit

**References:**

- *Predefined Fields*
- **FIELD*

## Overview

User subroutine *VUFIELD*:

- allows you to prescribe predefined field variables at the nodes of a model—the predefined field variables at a node can be updated individually, or a number of field variables at the nodes can be updated simultaneously;
- can be called for blocks of nodes for which the field variable values are defined in the subroutine;
- ignores any field variable values specified directly;
- can be used to modify field variable values read from a results file; and
- can be used in conjunction with user subroutine *VUSDFLD* such that the field variables that are passed in from *VUFIELD* and interpolated to the material points can be modified (such changes are local to material point values, and nodal field variable values remain unaffected).

## Updating Field Variables

Two different methods are provided for updating field variables.

### Individual Variable Updates

By default, only one field variable is updated at a time for given nodes or a given node set in user subroutine *VUFIELD*. The user subroutine is called whenever a current value of a field variable is needed for the nodes that are listed in the field variable definition. This method is ideal for cases in which the field variables are independent of each other.

### Simultaneous Variable Updates

User subroutine *VUFIELD* can also be used to update multiple field variables simultaneously for given nodes or a given node set. This method is well-suited for cases in which there are dependencies between some of the field variables. In this case you must specify the number of field variables to be updated simultaneously, and the user subroutine will be called each time the field variable values are needed.

## User Subroutine Interface

```
      SUBROUTINE VUFIELD(FIELD, NBLOCK, NFIELD, KFIELD, NCOMP,
     1                   KSTEP, JFLAGS, JNODEUID, TIME,
     2                   COORDS, U, V, A)
C
      INCLUDE 'VABA_PARAM.INC'

C     indices for the time array TIME
```

```
      PARAMETER( i_ufld_Current   = 1,
     *           i_ufld_Increment = 2,
     *           i_ufld_Period    = 3,
     *           i_ufld_Total     = 4 )

C     indices for the coordinate array COORDS
      PARAMETER( i_ufld_CoordX = 1,
     *           i_ufld_CoordY = 2,
     *           i_ufld_CoordZ = 3 )

C     indices for the displacement array U
      PARAMETER( i_ufld_SpaDisplX = 1,
     *           i_ufld_SpaDisplY = 2,
     *           i_ufld_SpaDisplZ = 3,
     *           i_ufld_RotDisplX = 4,
     *           i_ufld_RotDisplY = 5,
     *           i_ufld_RotDisplZ = 6,
     *           i_ufld_AcoPress  = 7,
     *           i_ufld_Temp      = 8 )

C     indices for the velocity array V
      PARAMETER( i_ufld_SpaVelX   = 1,
     *           i_ufld_SpaVelY   = 2,
     *           i_ufld_SpaVelZ   = 3,
     *           i_ufld_RotVelX   = 4,
     *           i_ufld_RotVelY   = 5,
     *           i_ufld_RotVelZ   = 6,
     *           i_ufld_DAcoPress = 7,
     *           i_ufld_DTemp     = 8 )

C     indices for the acceleration array A
      PARAMETER( i_ufld_SpaAccelX  = 1,
     *           i_ufld_SpaAccelY  = 2,
     *           i_ufld_SpaAccelZ  = 3,
     *           i_ufld_RotAccelX  = 4,
     *           i_ufld_RotAccelY  = 5,
     *           i_ufld_RotAccelZ  = 6,
     *           i_ufld_DDAcoPress = 7,
     *           i_ufld_DDTemp     = 8 )

C     indices for JFLAGS
      PARAMETER( i_ufld_kInc   = 1,
     *           i_ufld_kPass  = 2 )
C
      DIMENSION FIELD(NBLOCK,NCOMP,NFIELD)
      DIMENSION JFLAGS(2), JNODEUID(NBLOCK), TIME(4),
     *          COORDS(3,NBLOCK)
      DIMENSION U(8,NBLOCK), V(8,NBLOCK), A(8,NBLOCK)
C

      user coding to define FIELD

      RETURN
      END
```

## Variables to Be Defined

`FIELD(NBLOCK,NCOMP,NFIELD)` Array of field variable values at a collective number of nodes `NBLOCK` (see `NBLOCK` below). When updating one field variable at a time, only

the value of the specified field variable KFIELD must be returned. In this case NFIELD is passed into user subroutine *VUFIELD* with a value of 1, and FIELD is thus dimensioned as FIELD(NBLOCK,NCOMP,1). When updating all field variables simultaneously, the values of the specified number of field variables must be returned. In this case FIELD is dimensioned as FIELD(NBLOCK,NCOMP,NFIELD), where NFIELD is the number of field variables specified and KFIELD, which is set to −1, has no meaning.

If fields are applied to nodes that are not part of pipe, beam, or shell elements, only one value of each field variable is required (NCOMP=1), and the user subroutine is invoked in a single pass. For nodes that are part of pipe, beam, or shell elements, *VUFIELD* is invoked in two passes per increment for these elements, and the number of values to be returned depends on the mode of temperature and field variable input selected for the beam or shell section. The following cases are possible:

1. Field variables are given as values at the points on the shell or beam section. For a beam section the number of values required is determined by the particular section type specified, as described in *Beam Cross-Section Library*. For a shell section temperatures and field variables are given as values at *n* equally spaced points through each layer of a shell section. In the first pass NCOMP is passed in with the value of 1 to define the field variable values at the first point. The second pass is used to define field variables at the remaining points.
2. Field variables for the shell or beam section are given as values at the origin of the cross-section together with gradients along the cross-section. The number of gradient values required is 2 for three-dimensional beams, 1 for two-dimensional beams, and 1 for shells. In the first pass NCOMP is passed in with the value of 1 to define the field variable values at the origin of the cross-section. The gradients are defined in the second pass.

Because field variables can also be defined directly, it is important to understand the hierarchy used in situations with conflicting information (see *Predefined Fields*).

When the array FIELD is passed into user subroutine *VUFIELD*, it will contain either the field variable values from the previous increment or those values obtained from the results file if this method was used. You can then modify these values within this subroutine.

## Variables Passed in for Information

NBLOCK — User-specified number of nodes to be processed as a block in this call to *VUFIELD*. The value is equal to the total number of nodes given in a node set when blocking is disabled. When blocking is enabled, NBLOCK is equal to a predefined number set in Abaqus/Explicit. You can also modify NBLOCK by specifying a blocking size in the Abaqus/Explicit analysis.

NFIELD — User-specified number of field variables to be updated. The default value is 1.

KFIELD — User-specified field variable number. This variable is meaningful only when updating individual field variables at a time; otherwise, the value is set to −1.

| | |
|---|---|
| NCOMP | Maximum number of section values to be defined for any node in the model in the current pass. The first pass to user subroutine *VUFIELD* has NCOMP passed in with the value of 1. |
| KSTEP | Current step number. |
| JFLAGS(i_ufld_kInc) | Increment number for step KSTEP. |
| JFLAGS(i_ufld_kPass) | This flag is equal to 1 for the first pass to user subroutine *VUFIELD* and is equal to 2 for the second pass. |
| JNODEUID(NBLOCK) | Array for user-defined node numbers. This array is dimensioned based on the size of NBLOCK, and the contained node numbers are identical to those defined in the input file. You can perform additional interdependent field variable operations by using nodal indices stored in this array. |
| TIME(4) | Array for information of analysis time. You can retrieve any time information from this array by using the parameters given above. TIME(i_ufld_Current) stores the current analysis time, TIME(i_ufld_Increment) gives the time increment at this instance, TIME(i_ufld_Period) is the time period of the current step, and TIME(i_ufld_Total) is the total analysis time up to this point. You can use this time information to perform possible time-dependent field variable operations. |
| COORDS(3,NBLOCK) | Coordinates for nodes in the array JNODEUID. This array stores current coordinates of nodes in which the order of coordinates stored corresponds to the order of nodes listed in the array JNODEUID. The coordinates can be retrieved by using the parameters given above. You can make use of COORDS to define possible position-dependent field variable operations. |
| U(8,NBLOCK), V(8,NBLOCK), and A(8,NBLOCK) | Arrays containing solution variables of displacements, rotations, temperatures, and pressures and their corresponding temporal derivatives. The order in which these solutions are stored follows the order defined in the array JNODEUID. For a specific node, its solution variables can be retrieved by using the parameter indices given above. Depending on the degrees of freedom, some solution variables are not valid for a given node. The displacement values correspond to the current increment. However, the acceleration is from a configuration that is one increment behind, and the velocity is such that it is consistent with the displacement increment and the time increment between the two successive configurations. |

# VUFLUIDEXCH: User subroutine to define the mass flow rate/heat energy flow rate for fluid exchange.

**Products:** Abaqus/Explicit

**References:**

- *Fluid Exchange Definition*
- **FLUID EXCHANGE*
- **FLUID EXCHANGE ACTIVATION*
- **FLUID EXCHANGE PROPERTY*

## Overview

User subroutine *VUFLUIDEXCH*:

- can be used to define mass flow rate and/or heat energy flow rate for fluid exchange;
- can be used when built-in fluid exchange property types cannot satisfactorily model the mass/heat energy flow;
- can use and update solution-dependent state variables;
- can use any field variables that are passed in; and
- requires that the derivatives of mass/heat energy flow rates be defined with respect to pressure and temperature in the primary and secondary fluid cavities.

## Conventions for Defining Mass Flow/Heat Energy Flow Rate

A positive mass/heat energy flow rate indicates flow from the primary fluid cavity to the secondary fluid cavity. A negative value for mass flow rate will be ignored if the fluid exchange is between a cavity and its environment.

## User Subroutine Interface

```
      subroutine vufluidexch(
C Read only (unmodifiable)variables –
     1     nstatev, nfieldv, nprops,
     2     stepTime, totalTime, dt,
     3     jCavType, fluExchName, effArea, amplitude,
     4     props, lExchEnv, pcavNew, pcavOld,
     5     ctempNew, ctempOld, cvol, cmass,
     6     rMix, CpMix, DCpDtemp,
     7     field, stateOld,
C Write only (modifiable) variables
     8     stateNew, rMassRate, rEneRate,
     9     DMassRateDPcav, DMassRateDTemp,
     *     DEneRateDPcav, DEneRateDTemp)
c
      include 'vaba_param.inc'
c
      dimension props(nprops),
     1     pcavNew(2), pcavOld(2),
     2     ctempNew(2), ctempOld(2), cvol(2), cmass(2),
     3     rMix(2), CpMix(2), dCpDtemp(2),
```

```
     4     field(nfieldv),
     5     stateOld(nstatev), stateNew(nstatev),
     6     DMassRateDPcav(2), DMassRateDTemp(2),
     7     DEneRateDPcav(2), DEneRateDTemp(2)

c     Fluid cavity type
      parameter( iHydraulic     = 1,
     *           iAdiabaticGas  = 2,
     *           iIsothermalGas = 3)

      character*80 fluExchName

c     User coding to calculate mass flow rate,
c     heat energy flow rate and its derivatives with respect
c     to fluid cavity pressure and temperature.

      return
      end
```

## Variables to Be Defined

| | |
|---|---|
| `rMassRate` | Mass flow rate. The mass flow rate is negative if the flow is into the primary cavity. |
| `DMassRateDPcav(2)` | Derivative of mass flow rate with respect to pressure in primary and secondary fluid cavities. |
| `DMassRateDTemp(2)` | Derivative of mass flow rate with respect to temperature in primary and secondary fluid cavities. |
| `rEneRate` | Heat energy flow rate. The energy flow rate is negative if the flow is into the primary cavity. |
| `DEneRateDPcav(2)` | Derivative of heat energy flow rate with respect to pressure in primary and secondary fluid cavities. |
| `DEneRateDTemp(2)` | Derivative of heat energy flow rate with respect to temperature in primary and secondary fluid cavities. |

## Variables That Can Be Updated

| | |
|---|---|
| `stateNew(nstatev)` | State variable for fluid exchange at the end of the increment. You define the size of this array by allocating space for it (see *Allocating Space for Solution-Dependent State Variables* for more information). |

## Variables Passed in for Information

| | |
|---|---|
| `nstatev` | Number of user-defined state variables that are associated with this fluid exchange (you define this as described in *Allocating Space for Solution-Dependent State Variables*). |
| `nfieldv` | Number of user-defined external field variables. |
| `nprops` | User-specified number of user-defined fluid exchange properties required to define mass/heat energy flow rate. |
| `stepTime` | Value of time since the step began. |
| `totalTime` | Value of total time. The time at the beginning of the step is given by `totalTime-stepTime`. |
| `dt` | Time increment size. |

| | |
|---|---|
| `jCavType` | Indicator of fluid cavity type: 1 for fluid cavity with hydraulic fluids, 2 for fluid cavity with adiabatic gases, and 3 for fluid cavity with isothermal gases. |
| `fluExchName` | User-specified fluid exchange name. |
| `effArea` | Effective area for fluid exchange. |
| `amplitude` | Current value of the amplitude referenced for this fluid exchange. You must multiply the flow rates by the current amplitude value within the user subroutine if the amplitude is required. |
| `props(nprop)` | User-defined fluid exchange properties. |
| `lExchEnv` | The fluid exchange is to the environment if `lExchEnv`=1 and to another fluid cavity if `lExchEnv`=0. |
| `pcavNew(2)` | Pressure in primary and secondary fluid cavities at the end of the increment. |
| `pcavOld(2)` | Pressure in primary and secondary fluid cavities at the beginning of the increment. |
| `ctempNew(2)` | Temperature in primary and secondary fluid cavities at the end of the increment. |
| `ctempOld(2)` | Temperature in primary and secondary fluid cavities at the beginning of the increment. |
| `cvol(2)` | Volume of primary and secondary fluid cavities. |
| `cmass(2)` | Mass of fluid in primary and secondary fluid cavities. |
| `rMix(2)` | Gas constant of mixture in primary and secondary fluid cavities. |
| `CpMix(2)` | Specific heat of mixture in primary and secondary fluid cavities. |
| `DCpDtemp(2)` | Derivative of specific heat with respect to temperature for primary and secondary fluid cavities. |
| `field(nfieldv)` | Field variables at orifice. |
| `stateOld(nstatev)` | State variables for fluid exchange at the beginning of the increment. |

# VUFLUIDEXCHEFFAREA: User subroutine to define the effective area for fluid exchange.

**Products:** Abaqus/Explicit

**References:**

- *Fluid Exchange Definition*
- **FLUID EXCHANGE*

## Overview

User subroutine *VUFLUIDEXCHEFFAREA*:

- can be used to define an effective area for fluid exchange that depends on the material state in the underlying elements on the fluid exchange surface;
- will be called for blocks of material calculation points on the fluid exchange surface;
- can be used only if the specified surface over which the fluid exchange occurs is a surface defined over membrane elements; and
- can be used with any fluid exchange property type.

## Defining Effective Area

The contribution of each material point can be defined as a function of:

- the original area associated with the material point;
- the current material state in the underlying elements; and
- the temperature and pressure in the primary fluid cavity and the secondary fluid cavity or environment.

The effective area for fabric materials can depend on the nominal strain in the yarn directions and the change in angle between the two yarn directions, as well as the current angle between the two yarn directions. For nonfabric materials the effective area can depend on the material point strain.

## User Subroutine Interface

```
      subroutine vufluidexcheffarea(
C Read only (unmodifiable) variables -
     1     nblock, nprop, props,
     2     stepTime, totalTime, fluExchName,
     3     cMatName, lFabric, braidAngle,
     4     strain, origArea,
     5     pcav, ctemp,
C Write only (modifiable) variables
     6     effArea)
c
      include 'vaba_param.inc'
c
      parameter (ndir = 3, nshr=1)
c
c     pointers for retrieving fabric constitutive strains
```

```
      parameter( iFiberStrain1    = 1,
     *           iFiberStrain2    = 2,
     *           iFiberChangeAng  = 4)
c
      dimension props(nprop),
     1     braidAngle(nblock),
     2     strain(nblock, ndir+nshr),
     3     origArea(nblock),
     4     pcav(2),ctemp(2),
     5     effArea(nblock)

      character*80 fluExchName, cMatName

c      do k = 1, nblock
c       User coding to update effArea(k) = area associated with
c       material point contributing to area for fluid exchange
c       (leakage).
c      end do

      return
      end
```

## Variables to Be Defined

`effArea(nblock)` Area associated with the material point contributing to the total effective area for fluid exchange. The subroutine is called with `effArea` set to the current area associated with the material point and should be updated to reflect the area that contributes to fluid exchange.

## Variables Passed in for Information

| | |
|---|---|
| `nBlock` | Number of material points to be processed in this call to *VUFLUIDEXCHEFFAREA*. |
| `nprop` | User-specified number of user-defined fluid exchange properties required to define the effective area. |
| `props(nprop)` | User-defined fluid exchange properties. |
| `stepTime` | Value of time since the step began. |
| `totalTime` | Value of total time. The time at the beginning of the step is given by `totalTime-stepTime`. |
| `fluExchName` | User-specified fluid exchange name. |
| `cMatName` | User-specified material name associated with material points processed in this call. |
| `lFabric` | Flag indicating whether the subroutine is called for material points on the fluid exchange surface with a fabric material (`lFabric`=1 if fabric material, `lFabric`=0 otherwise). |
| `braidAngle(nblock)` | Angle in radians between the two yarn directions for fabric materials. |
| `strain(nblock, ndir+nshr)` | Fabric constitutive strains (nominal strain in the yarn directions and change in angle between the two yarn directions) or strains for nonfabric materials at current location. |
| `origArea(nblock)` | Original area associated with current material point. |
| `pcav(2)` | Absolute pressure in primary and secondary (or ambient) fluid cavities at the start of the increment. |

| | |
|---|---|
| `ctemp(2)` | Temperature in primary and secondary (or ambient) fluid cavities at the start of the increment. |

# VUGENS: User subroutine to define the mechanical behavior of a shell section.

**Products:** Abaqus/Explicit

**References:**

- *Using a General Shell Section to Define the Section Behavior*
- **SHELL GENERAL SECTION*

## Overview

User subroutine *VUGENS*:

- is used to define the (nonlinear) mechanical behavior of a shell section directly in terms of generalized section quantities;
- requires you to define the section behavior of the shell directly in terms of membrane strains and curvature changes;
- can define a damage variable to be applied to the (elastic) transverse shear stiffness of the section;
- is called for blocks of integration points in shell elements with a general shell section definition with user-defined section behavior;
- can use and update solution-dependent state variables; and
- can use any field variables that are passed in.

## Storage of Membrane and Bending Components

The arrays of the section forces, moments, and curvatures contain `ndir` + `nshr` components, with `ndir` = 2 and `nshr` = 1 for three-dimensional shells and `ndir` = 2 and `nshr` = 0 for axisymmetric shells. The direct components (11 and 22) are stored first, followed by the indirect 12 component (for three-dimensional shells). The arrays of section membrane strains and strain increments contain (`ndir` +1) + `nshr` components, where the additional direct component corresponds to the thickness strain.

The shear strain components in user subroutine *VUGENS* are stored as tensor components, not as engineering components.

## Stability

You should ensure that the integration scheme coded in this routine is stable.

Abaqus/Explicit uses the tangent stiffness matrix `ddndde` to compute an effective modulus of the material, which in turn is used for the calculation of the stable time increment. To ensure stability, the matrix `ddndde` defined inside *VUGENS* should provide a conservative estimate of the stiffness. The matrix `ddndde` is assumed to be symmetric for efficiency reasons. If the constitutive response leads to nonsymmetric stiffness, compute the symmetric part of the matrix by taking one half of the sum of the matrix and its transpose.

## Use with Shells That Have Transverse Shear

If user subroutine *VUGENS* is used to describe the section behavior of shells with transverse shear, you must define the transverse shear stiffness (see *Defining the Transverse Shear Stiffness*).

This initial value of the transverse shear stiffness can be modified within *VUGENS* by defining a TVS DAMAGE factor.

## Use with Continuum Shell Elements

User subroutine *VUGENS* cannot be used to describe the section behavior of continuum shell elements.

## User Subroutine Interface

```
      subroutine vugens (
     C Read only -
      * nblock, ndir, nshr, nstatev, nfieldv,
      * jElem, kIntPt, kIncr,
      * stepTime, totalTime, dt,
      * cElsetName, coordMp,
      * strainInc, curvInc,
      * tempOld, fieldOld,
      * defgradOld, direct,
      * forceOld, xmomOld, stateOld,
      * enerInternOld, enerInelasOld,
      * props, nprops, jprops, njprops,
      * density, thick,
      * tempNew, fieldNew,
      * defgradNew, curvNew, gamma,
     C Write only -
      * ddndde,
      * forceNew, xmomNew, stateNew,
      * enerInternNew, enerInelasNew )
     *
      include 'vaba_param.inc'
     *
      character*80 cElsetName
     *
      dimension jElem(nblock),
      * forceNew(nblock,ndir+nshr),
      * forceOld(nblock,ndir+nshr),
      * xmomNew(nblock,ndir+nshr),
      * xmomOld(nblock,ndir+nshr),
      * stateOld(nblock,nstatev),
      * stateNew(nblock, nstatev),
      * coordMp(nblock,3),
      * strainInc(nblock,ndir+1+nshr),
      * curvInc(nblock,ndir+nshr),
      * ddndde(nblock,*),
      * defgradNew(nblock,ndir+1+nshr+nshr),
      * defgradOld(nblock,ndir+1+nshr+nshr),
      * direct(nblock,3,3),
      * curvNew(nblock,ndir+nshr),
      * gamma(nblock, 2),
      * tempNew(nblock),
      * tempOld(nblock),
      * fieldNew(nblock,nfieldv),
      * fieldOld(nblock,nfieldv),
      * props(nprops),
      * density(nblock),
      * thick(nblock),
      * enerInternNew(nblock),
      * enerInternOld(nblock),
      * enerInelasOld(nblock),
      * enerInelasNew(nblock)
```

## Variables to Be Defined

`ddndde(nblock, *)`
Symmetric part of the section stiffness matrix of the shell section, $\partial \mathbf{N} / \partial \mathbf{E}$, where $\mathbf{N}$ are the section forces and moments on the shell section and $\mathbf{E}$ are the generalized section strains in the shell. The components of `ddndde` define the change at the end of the time increment caused by an infinitesimal perturbation of the generalized section strain increment array. The size of this matrix is (`ndir` + `nshr` ) × ((`ndir` + `nshr`) × 2 +1) corresponding to the triangular storage of the symmetric part of $\partial \mathbf{N} / \partial \mathbf{E}$.

`forceNew (nblock, ndir+nshr)`
Array of section forces at each integration point at the end of the increment. Stored as $(N_{11}, N_{22}, N_{12})$ for three-dimensional shells and as $(N_{11}, N_{22})$ for axisymmetric shells.

`xmomNew (nblock, ndir+nshr)`
Array of section moments at each integration point at the end of the increment. Stored as $(M_{11}, M_{22}, M_{12})$ for three-dimensional shells and as $(M_{11}, M_{22})$ for axisymmetric shells.

`stateNew(nblock, nstatev)`
State variables at each integration point at the end of the increment. You define the size of this array by allocating space for it using the VARIABLES parameter in the **SHELL GENERAL SECTION* option.

## Variables That Can Be Updated

`enerInternNew(nblock)`
Internal energy per unit mass at each integration point at the end of the increment.

`enerInelasNew(nblock)`
Dissipated inelastic energy per unit mass at each integration point at the end of the increment.

## Variables Passed in for Information

`nblock`
Number of integration points to be processed in this call to *VUGENS*.

`ndir`
Number of direct components in a symmetric tensor.

`nshr`
Number of indirect components in a symmetric tensor.

`jElem(nblock)`
Array of element numbers.

`kIntPt`
Integration point number.

`kIncr`
Increment number.

`nstatev`
Number of user-defined state variables that are associated with this integration point. You define this using the VARIABLES parameter in the **SHELL GENERAL SECTION* option.

`nfieldv`
Number of user-defined external field variables.

`nprops`
User-defined number of real property values associated with this section (*Defining the Shell Section Behavior*).

`njprops`
User-defined number of integer property values associated with this section (*Defining the Shell Section Behavior*).

`stepTime`
Value of time since the step began.

`totalTime`
Value of total time. The time at the beginning of the step is given by `totalTime - stepTime`.

`dt`
Time increment size.

| | |
|---|---|
| `cElsetName` | User-specified element set name associated with this section, left justified. |
| `coordMp(nblock, *)` | An array containing the current coordinates for each integration point. |
| `direct(nblock, 3, 3)` | An array containing the direction cosines of the shell local surface coordinate system. For integration point k, `direct(k,1,1)`, `direct(k,2,1)`, and `direct(k,3,1)` give the (1, 2, 3) components of the first material direction; `direct(k,1,2)`, `direct(k,2,2)`, and `direct(k,3,2)` give the second material direction, etc. The first two directions are in the plane of the element, and the third direction is the normal. |
| `props(nprops)` | A floating point array containing the `nprops` real property values defined for use with this section. |
| `jprops(njprops)` | An integer array containing the `njprops` integer property values defined for use with this section. |
| `density(nblock)` | Current density at the integration points. |
| `strainInc(nblock, ndir+1+nshr)` | Array of membrane strain increments at each integration point. Stored as $(\Delta\varepsilon_{11}, \Delta\varepsilon_{22}, \Delta\varepsilon_{33}, \Delta\varepsilon_{12})$ for three-dimensional shells and as $(\Delta\varepsilon_{11}, \Delta\varepsilon_{22}, \Delta\varepsilon_{33})$ for axisymmetric shells. |
| `curvInc(nblock, ndir+nshr)` | Array of curvature increments at each integration point. Stored as $(\Delta\kappa_{11}, \Delta\kappa_{22}, \Delta\kappa_{12})$ for three-dimensional shells and as $(\Delta\kappa_{11}, \Delta\kappa_{22})$ for axisymmetric shells. |
| `tempOld(nblock)` | Temperatures at each integration point at the beginning of the increment. |
| `defgradOld(nblock, ndir+1+nshr+nshr)` | Midsurface deformation gradient tensor at each integration point at the beginning of the increment. Stored in three dimensions as $(F_{11}, F_{22}, F_{33}, F_{12}, F_{23}, F_{13}, F_{21}, F_{32}, F_{31})$ and in two dimensions as $(F_{11}, F_{22}, F_{33}, F_{12}, F_{21})$.<br>The deformation gradient curvature tensor is available for finite-strain shells, but it is not available for small-strain shells. The tensor is provided in the local shell coordinate system. |
| `fieldOld(nblock, nfieldv)` | Values of the user-defined field variables at each integration point at the beginning of the increment. |
| `forceOld (nblock, ndir+nshr)` | Array of section forces at each integration point at the beginning of the increment. Stored as $(N_{11}, N_{22}, N_{12})$ for three-dimensional shells and as $(N_{11}, N_{22})$ for axisymmetric shells. |
| `xmomOld (nblock, ndir+nshr)` | Array of section moments at each integration point at the beginning of the increment. Stored as $(M_{11}, M_{22}, M_{12})$ for three-dimensional shells and as $(M_{11}, M_{22})$ for axisymmetric shells. |
| `stateOld(nblock, nstatev)` | State variables at each integration point at the beginning of the increment. |
| `enerInternOld(nblock)` | Internal energy per unit mass at each integration point at the beginning of the increment. |
| `enerInelasOld(nblock)` | Dissipated inelastic energy per unit mass at each integration point at the beginning of the increment. |
| `tempNew(nblock)` | Temperatures at each integration point at the end of the increment. |
| `defgradNew(nblock, ndir+1+nshr+nshr)` | Midsurface deformation gradient tensor at each integration point at the end of the increment. Stored in three dimensions as |

| | |
|---|---|
| | ($F_{11}, F_{22}, F_{33}, F_{12}, F_{23}, F_{13}, F_{21}, F_{32}, F_{31}$) and in two dimensions as ($F_{11}, F_{22}, F_{33}, F_{12}, F_{21}$).<br>The deformation gradient curvature tensor is available for finite-strain shells, but it is not available for small-strain shells. The tensor is provided in the local shell coordinate system. |
| `fieldNew(nblock, nfieldv)` | Values of the user-defined field variables at each integration point at the end of the increment. |
| `curvNew(nblock, ndir+nshr)` | An array containing the curvature tensor at each integration point at the end of the increment. Stored as ($\kappa_{11}, \kappa_{22}, \kappa_{12}$) for three-dimensional shells and as ($\kappa_{11}, \kappa_{22}$) for axisymmetric shells.<br>The tensor is provided in the local shell coordinate system. |
| `gamma(nblock, 2)` | Transverse shear strains at each integration point. |
| `thick(nblock)` | Section thickness at each integration point. |

# VUHARD: User subroutine to define the yield surface size and hardening parameters for isotropic plasticity or combined hardening models.

**Products:** Abaqus/Explicit

**References:**

- *Classical Metal Plasticity*
- *Models for Metals Subjected to Cyclic Loading*
- **CYCLIC HARDENING*
- **PLASTIC*
- *Deformation of a sandwich plate under CONWEP blast loading*
- *VUHARD*

## Overview

User subroutine *VUHARD*:

- is called at all material points of elements for which the material definition includes user-defined isotropic hardening or cyclic hardening for metal plasticity;
- can be used to define a material's isotropic yield behavior;
- can be used to define the size of the yield surface in a combined hardening model;
- can include material behavior dependent on field variables or state variables; and
- requires that the derivatives of the yield stress (or yield surface size in combined hardening models) be defined with respect to the appropriate independent variables, such as strain, strain rate, and temperature.

## User Subroutine Interface

```
      subroutine vuhard(
C Read only -
     *     nblock,
     *     jElem, kIntPt, kLayer, kSecPt,
     *     jInfoArray, stepTime, totalTime, dt, cmname,
     *     nstatev, nfieldv, nprops,
     *     props, tempOld, tempNew, fieldOld, fieldNew,
     *     stateOld,
     *     eqps, eqpsRate,
C Write only -
     *     yield, dyieldDtemp, dyieldDeqps,
     *     stateNew )
C
      include 'vaba_param.inc'
      parameter (i_inf_AnnealFlag = 1,
     *           i_inf_RateDepOff = 2)

C
      dimension props(nprops), tempOld(nblock), tempNew(nblock),
     1   fieldOld(nblock,nfieldv), fieldNew(nblock,nfieldv),
     2   stateOld(nblock,nstatev), eqps(nblock), eqpsRate(nblock),
     3   yield(nblock), dyieldDtemp(nblock), dyieldDeqps(nblock,2),
     4   stateNew(nblock,nstatev), jElem(nblock), jInfoArray(*)
```

```
C
      character*80 cmname
C
      lAnneal = jInfoArray(i_inf_AnnealFlag)
      lRateDepOff = jInfoArray(i_inf_RateDepOff)
C
      do 100 km = 1,nblock
        user coding
  100 continue
C
      return
      end
```

## Variables to Be Defined

`yield(nblock)`
Array containing the yield stress (for isotropic plasticity) or yield surface size (for combined hardening) at the material points.

`dyieldDeqps(nblock,1)`
Array containing the derivative of the yield stress or yield surface size with respect to the equivalent plastic strain at the material points.

`dyieldDeqps(nblock,2)`
Array containing the derivative of the yield stress with respect to the equivalent plastic strain rate at the material points.

Abaqus/Explicit ignores the prescribed values during a step where material rate dependence is ignored (jInfoArray(i_inf_RateDepOff) = 1).

`dyieldDtemp(nblock)`
Array containing the derivative of the yield stress or yield surface size with respect to temperature at the material points.

This quantity is required only in adiabatic and fully coupled temperature-displacement analyses.

`stateNew(nblock,nstatev)`
Array containing the state variables at the material points at the end of the increment.

The allocation of this array is described in *Allocating Space for Solution-Dependent State Variables*.

## Variables Passed in for Information

`nblock`
Number of material points to be processed in this call to `VUHARD`.

`jElem(nblock)`
Array of element numbers.

`kIntPt`
Integration point number.

`kLayer`
Layer number (for composite shells).

`kSecPt`
Section point number within the current layer.

`jInfoArray`

| | |
|---|---|
| `jInfoArray(i_inf_AnnealFlag)` | Flag indicating whether the routine is being called during an annealing process.<br>`jInfoArray(i_inf_AnnealFlag)=0` indicates that the routine is being called during a normal mechanics increment.<br>`jInfoArray(i_inf_AnnealFlag)=1` indicates that this is an annealing process and you should reinitialize the internal state variables, `stateNew`, if necessary. Abaqus/Explicit automatically sets the stresses, stretches, and state to a value of zero during the annealing process. |

| | |
|---|---|
| `jInfoArray(i_inf_RateDepOff)` | Flag indicating whether the routine is being called during a step where material rate dependence is ignored.<br>`jInfoArray(i_inf_RateDepOff)=0` indicates that the routine is being called during a step where material rate dependence is considered.<br>`jInfoArray(i_inf_RateDepOff)=1` indicates that the routine is being called during a step where material rate dependence is ignored. |

`stepTime` Value of time since the step began.

`totalTime` Value of total time. The time at the beginning of the step is given by `totalTime-stepTime`.

`dt` Time increment size.

`cmname` Material name, left justified. It is passed in as an uppercase character string. Some internal material models are given names starting with the "ABQ_" character string. To avoid conflict, "ABQ_" should not be used as the leading string for `cmname`.

`nstatev` Number of user-defined state variables that are associated with this material type (see *Allocating Space for Solution-Dependent State Variables*).

`nfieldv` Number of user-defined external field variables.

`nprops` User-specified number of user-defined material properties.

`tempOld(nblock)` Temperatures at the material points at the beginning of the increment.

`tempNew(nblock)` Temperatures at the material points at the end of the increment.

`fieldOld(nblock,nfieldv)` Values of the user-defined field variables at the material points at the beginning of the increment.

`fieldNew(nblock,nfieldv)` Values of the user-defined field variables at the material points at the end of the increment.

`stateOld(nblock,nstatev)` State variables at the material points at the beginning of the increment.

`eqps(nblock)` Equivalent plastic strain at the material points.

`eqpsRate(nblock)` Equivalent plastic strain rate at the material points.

# VUHYPER: User subroutine to define isotropic hyperelastic material behavior using the invariant formulation.

**Products:** Abaqus/Explicit

**References:**

- *Hyperelastic Behavior of Rubberlike Materials*
- **HYPERELASTIC*
- *VUHYPER*

## Overview

User subroutine *VUHYPER*:

- can be used to define the strain energy potential for isotropic hyperelastic material behavior;
- is called at all material calculation points of elements for which the material definition contains user-defined hyperelastic behavior;
- can include material behavior dependent on field variables or state variables; and
- requires that the values of the derivatives of the strain energy density function of the hyperelastic material be defined with respect to the strain invariants.

## Special Considerations for Shell Elements

When user subroutine *VUHYPER* is used to define the material response of shell elements, Abaqus/Explicit cannot calculate a default value for the transverse shear stiffness of the element. Hence, you must define the material transverse shear modulus (see *Defining the Elastic Transverse Shear Modulus*) or the element's transverse shear stiffness (see *Shell Section Behavior*).

## Material Point Deletion

Material points that satisfy a user-defined failure criterion can be deleted from the model (see *User-Defined Mechanical Material Behavior*). You must specify the state variable number controlling the element deletion flag when you allocate space for the solution-dependent state variables, as explained in *User-Defined Mechanical Material Behavior*. The deletion state variable should be set to a value of one or zero in user subroutine *VUHYPER*. A value of one indicates that the material point is active, and a value of zero indicates that Abaqus/Explicit should delete the material point from the model by setting the stresses to zero.

The structure of the block of material points passed to user subroutine *VUHYPER* remains unchanged during the analysis; deleted material points are not removed from the block. Abaqus/Explicit will “freeze” the values of the invariants passed to user subroutine *VUHYPER* for all deleted material points; that is, the values remain constant after deletion is triggered. Once a material point has been flagged as deleted, it cannot be reactivated.

## User Subroutine Interface

```
      subroutine vuhyper(
C Read only (unmodifiable) variables
     1     nblock, networkId, jElem, nIntPt, nLayer, nSecPt,
     2     cmname,
     3     nstatev, nfieldv, nprops,
```

```
     4     props, tempOld, tempNew, fieldOld, fieldNew,
     5     stateOld, bi1, bi2, detu,
C Write only (modifiable) variables
     6     u, duDi, d2uDiDi,
     7     stateNew )
C
      include 'vaba_param.inc'
C
       dimension props(nprops),
     1     tempOld(nblock),
     2     fieldOld(nblock,nfieldv),
     3     stateOld(nblock,nstatev),
     4     tempNew(nblock),
     5     fieldNew(nblock,nfieldv),
     6     bi1(nblock), bi2(nblock), detu(nblock), u(nblock,2),
     7     duDi(nblock,3), d2uDiDi(nblock,6),
     8     stateNew(nblock,nstatev), d1inv(nblock)
C
      character*80 cmname
C

      do 100 km = 1,nblock
        user coding
  100 continue

      return
      end
```

## Variables to Be Defined

`u(nblock,1)`

$\tilde{U}_{dev}$, the deviatoric part of the strain energy density of the primary material response. Define this term if the material definition includes Mullins effect (see *Stress Softening in Rubberlike Materials*) or when *VUHYPER* is used with the parallel rheological framework and viscous softening (see *Parallel Rheological Framework*).

`u(nblock,2)`

The total strain energy density defined in terms of the strain invariants. The strain invariants—$\bar{I}_1$, $\bar{I}_2$, and $J$—are defined in *Hyperelastic Behavior of Rubberlike Materials*.Abaqus/Explicit uses this term to compute the initial strain energy when initial stress conditions are specified (see *Initial Conditions*).

`duDi(nblock,1)`

Array of derivatives of strain energy potential with respect to the first invariant $\bar{I}_1$, $\partial U/\partial \bar{I}_1$.

`duDi(nblock,2)`

Array of derivatives of strain energy potential with respect to the second invariant $\bar{I}_2$, $\partial U/\partial \bar{I}_2$.

`duDi(nblock,3)`

Array of derivatives of strain energy potential with respect to the third invariant $J$, $\partial U/\partial J$.

`d2uDiDi(nblock,1)`

$\partial^2 U/\partial \bar{I}_1^{\,2}$

`d2uDiDi(nblock,2)`

$\partial^2 U/\partial \bar{I}_1 \partial \bar{I}_2$

`d2uDiDi(nblock,3)`

$\partial^2 U/\partial \bar{I}_2^{\,2}$

`d2uDiDi(nblock,4)` $\partial^2 U/\partial \bar{I}_1 \partial J$

`d2uDiDi(nblock,5)` $\partial^2 U/\partial \bar{I}_2 \partial J$

`d2uDiDi(nblock,6)` $\partial^2 U/\partial J^2$

`stateNew(nblock,nstatev)` State variables at each material point at the end of the increment. You define the size of this array by allocating space for it (see *About User Subroutines and Utilities* for more information).

## Variables Passed in for Information

`nblock` Number of material points to be processed in this call to *VUHYPER*.

`networkId` Network identification number, which identifies the network for which the strain energy potential is defined when *VUHYPER* is used in combination with the Parallel Rheological Framework (see *Parallel Rheological Framework* and *VUHYPER*). You can define different strain energy functions for different networks.

`jElem(nblock)` Array of element numbers.

`nIntPt` Integration point number.

`nLayer` Layer number (for composite shells).

`nSecPt` Section point number within the current layer.

`cmname` User-specified material name, left justified. It is passed in as an uppercase character string. Some internal material models are given names starting with the "ABQ_" character string. To avoid conflict, you should not use "ABQ_" as the leading string for `cmname`.

`nstatev` Number of user-defined state variables that are associated with this material type (you define this as described in *Allocating Space for Solution-Dependent State Variables*).

`nfieldv` Number of user-defined external field variables.

`nprops` User-specified number of user-defined material properties.

`props(nprops)` User-supplied material properties.

`tempOld(nblock)` Temperatures at each material point at the beginning of the increment.

`tempNew(nblock)` Temperatures at each material point at the end of the increment.

`fieldOld(nblock,nfieldv)` Values of the user-defined field variables at each material point at the beginning of the increment.

`fieldNew(nblock,nfieldv)` Values of the user-defined field variables at each material point at the end of the increment.

`stateOld(nblock,nstatev)` State variables at each material point at the beginning of the increment.

`bi1(nblock)` Invariant $\bar{I}_1$, at each material point at the end of the increment.

`bi2(nblock)` Invariant $\bar{I}_2$, at each material point at the end of the increment.

`detu(nblock)` Invariant $J$, at each material point at the end of the increment.

## Example: Neo-Hookean Hyperelastic Model

Below is an example of the coding of user subroutine *VUHYPER* for a neo-Hookean hyperelastic material. The strain energy function is given in the form

$$U = C_{10}\left(\bar{I}_1 - 3\right) + \frac{1}{D_1}\left(J^{el} - 1\right)^2.$$

The code in user subroutine *VUHYPER* must return the derivatives of the strain energy function with respect to the invariants, which are readily computed from the above expression. The user subroutine would be coded as follows:

```
      subroutine vuhyper(
C Read only (unmodifiable) variables
     1     nblock, networkId, jElem, kIntPt, kLayer, kSecPt,
     2     cmname,
     3     nstatev, nfieldv, nprops,
     4     props, tempOld, tempNew, fieldOld, fieldNew,
     5     stateOld, bi1,bi2,detu,
C Write only (modifiable) variables
     6     u, duDi, d2uDiDi,
     7     stateNew )

C
      include 'vaba_param.inc'
C
      dimension props(nprops),
     1     tempOld(nblock),
     2     fieldOld(nblock,nfieldv),
     3     stateOld(nblock,nstatev),
     4     tempNew(nblock),
     5     fieldNew(nblock,nfieldv),
     6     bi1(nblock), bi2(nblock), detu(nblock), u(nblock,2),
     7     duDi(nblock,3), d2uDiDi(nblock,6),
     8     stateNew(nblock,nstatev)
C
      character*80 cmname
C
      parameter ( zero =0.d0, one = 1.d0, two = 2.d0, three = 3.d0 )

      c10 = props(1)
      d1 = props(2)
      d1inv = one/d1
      do km = 1,nblock
         a1 = bi1(km)-three
         a2 = bi2(km) - three
         a3 = detu(km) - one

         u(km,1) = c10*a1
         u(km,2) = c10*a1 + d1inv * a3**2
         duDi(km,1) = c10
         duDi(km,2) = zero
         duDi(km,3) = d1inv * a3 * two
C
         d2uDiDi(km,1) = zero
         d2uDiDi(km,2) = zero
         d2uDiDi(km,3) = zero
         d2uDiDi(km,6) = d1inv * two
      end do

      return
      end
```

# VUHYPER_STRETCH: User subroutine to define isotropic hyperelastic material behavior using the stretch formulation.

**Products:** Abaqus/Explicit

**References:**

- *Hyperelastic Behavior of Rubberlike Materials*
- **HYPERELASTIC*
- *VUHYPER_STRETCH*

## Overview

User subroutine *VUHYPER_STRETCH*:

- can be used to define the strain energy potential for isotropic hyperelastic material behavior;
- is called at all material calculation points of elements for which the material definition contains user-defined hyperelastic behavior;
- can include material behavior dependent on field variables or state variables; and
- requires that the values of the derivatives of the strain energy density function of the hyperelastic material be defined with respect to the principal stretches.

## Special Considerations for Shell Elements

When you use user subroutine *VUHYPER_STRETCH* to define the material response of shell elements, Abaqus/Explicit cannot calculate a default value for the transverse shear stiffness of the element. You must manually define the material transverse shear modulus (see *Defining the Elastic Transverse Shear Modulus*) or the element's transverse shear stiffness (see *Shell Section Behavior*).

## Material Point Deletion

Material points that satisfy a user-defined failure criterion can be deleted from the model (see *User-Defined Mechanical Material Behavior*). You must specify the state variable number controlling the element deletion flag when you allocate space for the solution-dependent state variables, as explained in *User-Defined Mechanical Material Behavior*. The deletion state variable should be set to a value of one or zero in user subroutine *VUHYPER_STRETCH*. A value of one indicates that the material point is active, and a value of zero indicates that Abaqus/Explicit should delete the material point from the model by setting the stresses to zero.

The structure of the block of material points passed to user subroutine *VUHYPER_STRETCH* remains unchanged during the analysis; deleted material points are not removed from the block. Abaqus/Explicit will “freeze” the values of the stretches passed to user subroutine *VUHYPER_STRETCH* for all deleted material points; that is, the values remain constant after deletion is triggered. Once a material point has been flagged as deleted, it cannot be reactivated.

## User Subroutine Interface

```
      subroutine vuhyper_stretch(
C Read only (unmodifiable) variables
     1     nblock, networkId, jElement, nIntPt, nLayer, nSecPt,
     2     cmname,
```

```
     3     nstatev, nfieldv, nprops,
     4     props, tempOld, tempNew, fieldOld, fieldNew,
     5     stateOld, stretchP1, stretchP2, stretchP3, detu,
C Write only (modifiable) variables
     6     u, duDx, d2uDxDx,
     7     stateNew )
C
      include 'vaba_param.inc'
C
      dimension props(nprops),
     1     tempOld(nblock),
     2     fieldOld(nblock,nfieldv),
     3     stateOld(nblock,nstatev),
     4     tempNew(nblock),
     5     fieldNew(nblock,nfieldv),
     6
stretchP1(nblock),stretchP2(nblock),stretchP3(nblock),detu(nblock),
     7     duDx(nblock,4), d2uDxDx(nblock,10),
     8     u(nblock,2),
     9     stateNew(nblock,nstatev)
C
      character*80 cmname
C
      do 100 km = 1,nblock
        user coding
  100 continue

      return
      end
```

## Variables to Be Defined

`u(nblock,1)`

$\widetilde{U}_{dev}$, the deviatoric part of the strain energy density of the primary material response. Define this term if the material definition includes Mullins effect (see *Stress Softening in Rubberlike Materials*) or when *VUHYPER_STRETCH* is used with the parallel rheological framework and viscous softening (see *Parallel Rheological Framework*).

`u(nblock,2)`

The total strain energy density defined in terms of the principal deviatoric stretches ($\overline{\lambda}_1$, $\overline{\lambda}_2$, $\overline{\lambda}_3$) and the total volume ratio,*J*. You can find the definitions of $\overline{\lambda}_1$, $\overline{\lambda}_2$, $\overline{\lambda}_3$, and *J* in *Hyperelastic Behavior of Rubberlike Materials*. Abaqus/Explicit uses this term to compute the initial strain energy when initial stress conditions are specified (see *Initial Conditions*).

`duDx(nblock,1)`

Array of derivatives of strain energy potential with respect to the first principal deviatoric stretch $\overline{\lambda}_1$, $\partial U/\partial\overline{\lambda}_1$.

`duDx(nblock,2)`

Array of derivatives of strain energy potential with respect to the second principal deviatoric stretch $\overline{\lambda}_2$, $\partial U/\partial\overline{\lambda}_2$.

`duDx(nblock,3)`

Array of derivatives of strain energy potential with respect to the third principal deviatoric stretch $\overline{\lambda}_3$, $\partial U/\partial\overline{\lambda}_3$.

`duDx(nblock,4)`

Array of derivatives of strain energy potential with respect to the third invariant *J*, $\partial U/\partial J$.

`d2uDxDx(nblock,1)`

$\partial^2 U/\partial\overline{\lambda}_1^{\,2}$.

`d2uDxDx(nblock,2)` $\partial^2 U/\partial \bar{\lambda}_2^{\,2}$.

`d2uDxDx(nblock,3)` $\partial^2 U/\partial \bar{\lambda}_3^{\,2}$.

`d2uDxDx(nblock,4)` $\partial^2 U/\partial \bar{\lambda}_1 \partial \bar{\lambda}_2$.

`d2uDxDx(nblock,5)` $\partial^2 U/\partial \bar{\lambda}_1 \partial \bar{\lambda}_3$.

`d2uDxDx(nblock,6)` $\partial^2 U/\partial \bar{\lambda}_2 \partial \bar{\lambda}_3$.

`d2uDxDx(nblock,7)` $\partial^2 U/\partial \bar{\lambda}_1 \partial J$.

`d2uDxDx(nblock,8)` $\partial^2 U/\partial \bar{\lambda}_2 \partial J$.

`d2uDxDx(nblock,9)` $\partial^2 U/\partial \bar{\lambda}_3 \partial J$.

`d2uDxDx(nblock,10)` $\partial^2 U/\partial J^2$.

`stateNew(nblock,nstatev)` State variables at each material point at the end of the increment. You define the size of this array by allocating space for it (see *About User Subroutines and Utilities* for more information).

## Variables Passed in for Information

`nblock` Number of material points to be processed in this call to user subroutine *VUHYPER_STRETCH*.

`networkId` Network identification number, which identifies the network for which the strain energy potential is defined when user subroutine *VUHYPER_STRETCH* is used in combination with the parallel rheological framework (see *Parallel Rheological Framework* and *VUHYPER_STRETCH*). You can define different strain energy functions for different networks.

`jElem(nblock)` Array of element numbers.

`nIntPt` Integration point number.

`nLayer` Layer number (for composite shells).

`nSecPt` Section point number within the current layer.

`cmname` User-specified material name, left justified. It is passed in as an uppercase character string. Some internal material models are given names starting with the "ABQ_" character string. To avoid conflict, you should not use "ABQ_" as the leading string for `cmname`.

`nstatev` Number of user-defined state variables that are associated with this material type (you define this as described in *Allocating Space for Solution-Dependent State Variables*).

`nfieldv` Number of user-defined external field variables.

`nprops` User-specified number of user-defined material properties.

`props(nprops)` User-supplied material properties.

`tempOld(nblock)` Temperatures at each material point at the beginning of the increment.

`tempNew(nblock)` Temperatures at each material point at the end of the increment.

`fieldOld(nblock,nfieldv)` Values of the user-defined field variables at each material point at the beginning of the increment.

`fieldNew(nblock,nfieldv)` Values of the user-defined field variables at each material point at the end of the increment.

`stateOld(nblock,nstatev)` State variables at each material point at the beginning of the increment.

`stretchP1(nblock)` First principal deviatoric stretch, $\overline{\lambda}_1$, at each material point at the end of the increment.

`stretchP2(nblock)` Second principal deviatoric stretch, $\overline{\lambda}_2$, at each material point at the end of the increment.

`stretchP3(nblock)` Third principal deviatoric stretch, $\overline{\lambda}_3$, at each material point at the end of the increment.

`detu(nblock)` Invariant $J$ at each material point at the end of the increment.

## Example: Hencky Hyperelastic Model

Below is an example of the coding of user subroutine *VUHYPER_STRETCH* for a Hencky hyperelastic material. The strain energy function is given in the form

$$U = \mu\left(\sum_{i=1}^{3} \left(\ln \overline{\lambda}_i\right)^2\right) + \frac{1}{2}K(\ln J)^2,$$

where

$U$
Strain energy per unit of reference volume.

$\mu = \frac{E}{2(1+\nu)}$
Shear modulus.

$K = \frac{E}{3(1-2\nu)}$
Bulk modulus, with $\nu$ being the Poisson's ratio.

The code in user subroutine *VUHYPER_STRETCH* must return the derivatives of the strain energy function with respect to the deviatoric stretches and the volume ratio, $J$, which are readily computed from the above expression. The user subroutine is coded as follows:

```
      C user subroutine vuhyper_stretch
       subroutine vuhyper_stretch(
C Read only (unmodifiable) variables
     1     nblock, networkId, jElement, nIntPt, nLayer, nSecPt,
     2     cmname,
     3     nstatev, nfieldv, nprops,
     4     props, tempOld, tempNew, fieldOld, fieldNew,
     5     stateOld, stretchP1, stretchP2, stretchP3, detu,
C Write only (modifiable) variables
     6     u, duDx, d2uDxDx,
     7     stateNew )
*
       include 'vaba_param.inc'
       dimension props(nprops),
     1     tempOld(nblock),
     2     fieldOld(nblock,nfieldv),
     3     stateOld(nblock,nstatev),
     4     tempNew(nblock),
     5     fieldNew(nblock,nfieldv),
     6
stretchP1(nblock),stretchP2(nblock),stretchP3(nblock),detu(nblock),
```

```
     7     duDx(nblock,4), d2uDxDx(nblock,10),
     8     u(nblock,2),
     9     stateNew(nblock,nstatev)
C
      character*80 cmname
      parameter (zero=0.d0, one = 1.d0, two = 2.d0, third = 1.d0/3.d0)

      xE = props(1)
      xNu = props(2)
      twoMu = xE/(one+xNu)
      BulkMod = third*xE/(one-two*xNu)
      do k = 1, nblock
         xl1 = stretchP1(k)
         xl2 = stretchP2(k)
         xl3 = stretchP3(k)
         e1 = log(xl1)
         e2 = log(xl2)
         e3 = log(xl3)
         eVol = log(detu(k))
         de1dxl1 = one/xl1
         de2dxl2 = one/xl2
         de3dxl3 = one/xl3
         detuInv = one/detu(k)
*     First derivatives
         duDx(k,1) = twoMu* e1* de1dxl1
         duDx(k,2) = twoMu* e2* de2dxl2
         duDx(k,3) = twoMu* e3* de3dxl3
         duDx(k,4) =  BulkMod* eVol*detuInv
*     Second derivatives
         d2uDxDx(k,1) = twoMu *(one-e1)* (de1dxl1)**2
         d2uDxDx(k,2) = twoMu *(one-e2)* (de2dxl2)**2
         d2uDxDx(k,3) = twoMu *(one-e3)* (de3dxl3)**2
         d2uDxDx(k,4) = zero
         d2uDxDx(k,5) = zero
         d2uDxDx(k,6) = zero
         d2uDxDx(k,10)=  detuInv*(BulkMod/detuInv-duDx(k,4))

      end do

      return
      end
```

# VUINTER: User subroutine to define the interaction between contact surfaces.

**Products:** Abaqus/Explicit

**References:**

- *User-Defined Interfacial Constitutive Behavior*
- **SURFACE INTERACTION*
- *VUINTER*

## Overview

User subroutine *VUINTER*:

- can be used to define the mechanical and thermal interaction between contacting surfaces;
- must provide the entire definition of the interaction between the contacting surfaces;
- can use and update solution-dependent state variables; and
- must be used with the penalty contact constraint algorithm.

## Terminology

The use of user subroutine *VUINTER* requires familiarity with the following terminology.

### Surface Node Numbers

The "surface node number" refers to the position of a particular node in the list of nodes on the surface. For example, there are `nSecNod` nodes on the secondary surface. Number $n, n = 1, 2...$`nSecNod`, is the surface node number of the *n*th node in this list; `jSecUid`$(n)$ is the user-defined global number of this node. An Abaqus/Explicit model can be defined in terms of an assembly of part instances (see *Assembly Definition*). In such models a node number in `jSecUid` is an internally generated node number. If the original node number and part instance name are required, call the utility routine `VGETPARTINFO` (see *Obtaining Part Information*).

## Local Coordinate System

The array `alocaldir` defines the direction cosines of a local coordinate system for each secondary node. The first local direction corresponds to the contact normal direction from the perspective of the secondary node. For a two-dimensional *VUINTER* model the second local direction is the tangent direction defined by the cross product of the vector into the plane of the model (0., 0., −1.0) and the secondary normal. For a three-dimensional *VUINTER* model the second and third local directions correspond to two orthogonal tangent directions $\mathbf{t}_1$ and $\mathbf{t}_2$, which are set as follows:

- If the main surface is a cylindrical analytical surface, the second local direction corresponds to the generator direction (see *Analytical Rigid Surface Definition*), and the third local direction is the cross product of the first and second local directions.
- If the main surface is an analytical surface of revolution, the third local direction corresponds to the hoop direction, and the second local direction is the cross product of the third and first local directions.

- If the main surface is a three-dimensional, element-based surface, the tangent directions are based on the secondary normal, using the standard convention for calculating surface tangents (see *Conventions*).

For the two cases listed above involving three-dimensional analytical surfaces, the local tangent directions will reflect a rotation of the main surface. For the last case (three-dimensional, element-based main surface) the tangent directions may not follow the rotation of either the main or secondary surfaces; for example, the local system would remain fixed with respect to the global system if a secondary node and its surrounding facets rotate about an axes parallel to the secondary normal.

The 2 × 2 array stored in `drot` for each secondary node represents the incremental rotation of the tangent directions within the tangent plane corresponding to the tracked point of a three-dimensional main surface. (This incremental rotation array is equal to a unit matrix if `nDir` is equal to 2.) This incremental rotation matrix is provided so that vector- or tensor-valued state variables defined within the tangent plane can be rotated in this subroutine. For example, the second and third components of the `rdisp` array (i.e., the relative slip components) are rotated by this amount before *VUINTER* is called. However, as already mentioned, the rotation of the tangent directions may not reflect a physical rotation of the main or secondary surface.

## Conventions for Heat Flux and Stress

A positive flux indicates heat flowing into a surface, and a negative flux denotes heat leaving the surface. Flux must be specified for both surfaces, and they need not be equal and opposite so that effects such as frictional dissipation and differential surface heating can be modeled.

A positive normal stress denotes a pressure directed into the surface (opposite the local normal direction). Positive shear stresses denote shear tractions in the direction of the local surface tangents.

## User Subroutine Interface

```
      subroutine vuinter(
C Write only
     1 sfd, scd, spd, svd,
C Read/Write -
     2 stress, fluxSec, fluxMain, sed, statev,
C Read only -
     3 kStep, kInc, nFacNod, nSecNod, nMainNod, nSurfDir,
     4 nDir, nStateVar, nProps, nTemp, nPred, numDefTfv,
     5 jSecUid, jMainUid, jConMainid, timStep, timGlb,
     6 dTimCur, surfInt, surfSec, surfMain,
     7 rdisp, drdisp, drot, stiffDflt, condDflt,
     8 shape, coordSec, coordMain, alocaldir, props,
     9 areaSec, tempSec, dtempSec, preDefSec, dpreDefSec,
     1 tempMain, dtempMain, preDefMain, dpreDefMain)
C
      include `vaba_param.inc'
C
      character*80 surfInt, surfSec, surfMain
C
      dimension props(nProps), statev(nStateVar,nSecNod),
     1 drot(2,2,nSecNod), sed(nSecNod), sfd(nSecNod),
     2 scd(nSecNod), spd(nSecNod), svd(nSecNod),
     3 rdisp(nDir,nSecNod), drdisp(nDir,nSecNod),
     4 stress(nDir,nSecNod), fluxSec(nSecNod),
     5 fluxMain(nSecNod), areaSec(nSecNod),
     6 stiffDflt(nSecNod), condDflt(nSecNod),
     7 alocaldir(nDir,nDir,nSecNod), shape(nFacNod,nSecNod),
     8 coordSec(nDir,nSecNod), coordMain(nDir,nMainNod),
     9 jSecUid(nSecNod), jMainUid(nMainNod),
     1 jConMainid(nFacNod,nSecNod), tempSec(nSecNod),
```

```
     2 dtempSec(nSecNod), preDefSec(nPred,nSecNod),
     3 dpreDefSec(nPred,nSecNod), tempMain(numDefTfv),
     4 dtempMain(numDefTfv), preDefMain(nPred,numDefTfv),
     5 dpreDefMain(nPred,numDefTfv)

      user coding to define stress,
      and, optionally, fluxSec, fluxMain, statev, sed, sfd, scd, spd,
      and svd

      return
      end
```

## Variables to Be Defined

`stress(nDir, nSecNod)` On entry this array contains the stress at the interface during the previous time increment. It must be updated to the stress at the interface in the current time increment.

## Variables That Can Be Updated

`fluxSec(nSecNod)` On entry this array contains the flux entering the secondary surface during the previous time increment. It must be updated to the flux entering the secondary surface during the current increment.

`fluxMain(nSecNod)` On entry this array contains the flux entering the main surface during the previous time increment. It must be updated to the flux entering the main surface during the current time increment.

`sfd(nSecNod)` This array can be updated to contain the increment in frictional dissipation at each node (units of energy per unit area). These values contribute to the output variables SFDR and ALLFD and have no effect on other solution variables.

`scd(nSecNod)` This array can be updated to contain the increment in creep dissipation at each node (units of energy per unit area). These values contribute to the output variables SFDR and ALLCD and have no effect on other solution variables.

`spd(nSecNod)` This array can be updated to contain the increment in plastic dissipation at each node (units of energy per unit area). These values contribute to the output variables SFDR and ALLPD and have no effect on other solution variables.

`svd(nSecNod)` This array can be updated to contain the increment in viscous dissipation at each node (units of energy per unit area). These values contribute to the output variables SFDR and ALLVD and have no effect on other solution variables.

`sed(nSecNod)` On entry this array contains the elastic energy density at the secondary nodes at the beginning of the increment. It can be updated to contain the elastic energy density at the end of the current time increment. These values contribute to the output variable ALLSE and have no effect on other solution variables.

`statev(nstateVar, nSecNod)` This array contains the user-defined solution-dependent state variables for all the nodes on the secondary surface. You define the size of this array (see *User-Defined Interfacial Constitutive Behavior* for more information). This array will be passed in containing the values of these variables prior to the call to user subroutine *VUINTER*. If any of the solution-dependent state variables is being used in conjunction with the surface interaction, it must be updated in this subroutine.

## Variables Passed in for Information

| | |
|---|---|
| `kStep` | Step number. |
| `kInc` | Increment number. |
| `nFacNod` | Number of nodes on each main surface facet. `nFacNod` is 2 for two-dimensional surfaces, and `nFacNod` is 4 for three-dimensional surfaces (the first and last nodes are the same for triangular facets). If the main surface is an analytical rigid surface, this variable is passed in as 0. |
| `nSecNod` | Number of secondary nodes. |
| `nMainNod` | Number of main surface nodes, if the main surface is made up of facets. If the main surface is an analytical rigid surface, this variable is passed in as 0. |
| `nSurfDir` | Number of tangent directions at the contact points (`nSurfDir = nDir - 1`). |
| `nDir` | Number of coordinate directions at the contact points. (In a three-dimensional model `nDir` will be 2 if the surfaces in the contact pair are two-dimensional analytical rigid surfaces or are formed by two-dimensional elements.) |
| `nStateVar` | Number of user-defined state variables. |
| `nProps` | User-specified number of property values associated with this surface interaction model. |
| `nTemp` | 1 if the temperature is defined and 0 if the temperature is not defined. |
| `nPred` | Number of predefined field variables. |
| `numDefTfv` | Equal to `nSecNod` if the main surface is made up of facets. If the main surface is an analytical rigid surface, this variable is passed in as 1. |
| `jSecUid(nSecNod)` | This array lists the user-defined global node numbers (or internal node numbers for models defined in terms of an assembly of part instances) of the nodes on the secondary surface. |
| `jMainUid(nMainNod)` | This array lists the user-defined global node numbers (or internal node numbers for models defined in terms of an assembly of part instances) of the nodes on the main surface. If the main surface is an analytical rigid surface, this array is passed in as a dummy array. |
| `jConMainid(nFacNod, nSecNod)` | This array lists the surface node numbers of the main surface nodes that make up the facet onto which each secondary node projects. If the main surface is an analytical rigid surface, this array is passed in as a dummy array. |
| `timStep` | Value of step time. |
| `timGlb` | Value of total time. |
| `dtimCur` | Current increment in time from $t = t_{curr} - \Delta t$ to $t = t_{curr}$. |
| `surfInt` | User-specified surface interaction name, left justified. |
| `surfSec` | Secondary surface name. |
| `surfMain` | Main surface name. |
| `rdisp(nDir, nSecNod)` | An array containing the relative positions between the two surfaces. The first component is the relative position of the secondary node, with respect to the main surface, in the normal direction (a positive value indicates a penetration, and a negative value indicates a gap). The second and third components, if applicable, are the accumulated incremental relative tangential displacements of the secondary node, measured from the beginning of the step in which the contact pair is defined. The local directions in which the relative displacements are defined are stored in `alocaldir`. If the main surface is an analytical surface, the elements in `rdisp` are set to `r_MaxVal` for the secondary nodes that are far from the main surface. |

`drdisp(nDir, nSecNod)` An array containing the increments in relative positions between the two surfaces during the current time increment. If the main surface is an analytical surface, the elements in `drdisp` are set to `r_MaxVal` for the secondary nodes that are far from the main surface.

`drot(2, 2, nSecNod)` Rotation increment matrix. This matrix represents the incremental rotation of the local surface tangent directions for a three-dimensional surface. This rotation matrix for each secondary node is defined as a unit matrix for two-dimensional surfaces. If the main surface is an analytical surface, the elements in `drot` are set to `r_MaxVal` for the secondary nodes that are far from the main surface.

`stiffDflt(nSecNod)` Values of the default penalty stiffnesses for each secondary node (units of $FL^{-3}$).

`condDflt(nSecNod)` Values of the default penalty conductances for each secondary node (units of $J^{-1}T^{-1}$).

`shape(nFacNod, nSecNod)` For each contact point this array contains the shape functions of the nodes of its main surface facet, evaluated at the location of the contact point. If the main surface is an analytical rigid surface, this array is passed in as a dummy array.

`coordSec(nDir, nSecNod)` Array containing the `nDir` components of the current coordinates of the secondary nodes.

`coordMain(nDir, nMainNod)` Array containing the `nDir` components of the current coordinates of the main nodes. If the main surface is an analytical rigid surface, this array is passed in as the coordinates of the contact points on the main surface.

`alocaldir(nDir, nDir, nSecNod)` Direction cosines of the local surface coordinate system. The first array index corresponds to the components of the local directions, and the second array index corresponds to the local direction number. The first direction (`alocaldir(1..nDir,1,...)`) is the normal to the surface. The second direction (`alocaldir(1..nDir,2,...)`) is the first surface tangent. For a three-dimensional surface, the third direction (`alocaldir(1..3,3,...)`) is the second surface tangent. If the main surface is an analytical rigid surface, the numbers in `alocaldir` are valid only if the corresponding parts in `rdisp` are valid (that is, not equal to `r_MaxVal`).

`props(nProps)` User-specified vector of property values to define the behavior between the contacting surfaces.

`areaSec(nSecNod)` Area associated with the secondary nodes (equal to 1 for node-based surface nodes).

`tempSec(nSecNod)` Current temperature at the secondary nodes.

`dtempSec(nSecNod)` Increment in temperature during the previous time increment at the secondary nodes.

`preDefSec(nPred, nSecNod)` Current user-specified predefined field variables at the secondary nodes (initial values at the beginning of the analysis and current values during the analysis).

`dpreDefSec(nPred, nSecNod)` Increment in the predefined field variables at the secondary nodes during the previous time increment.

`tempMain(numDefTfv)` Current temperature at the nearest points on the main surface.

`dtempMain(numDefTfv)` Increment in temperature during the previous time increment at the nearest points on the main surface.

`preDefMain(nPred, numDefTfv)` Current user-specified predefined field variables at the nearest points on the main surface (initial values at the beginning of the analysis and current values during the analysis).

`dpreDefMain(nPred, numDefTfv)` Increment in the predefined field variables during the previous time increment at the nearest points on the main surface.

# VUINTERACTION: User subroutine to define the contact interaction between surfaces with the general contact algorithm.

**Products:** Abaqus/Explicit

**References:**

- *User-Defined Interfacial Constitutive Behavior*
- **SURFACE INTERACTION*
- *VUINTERACTION*

## Overview

User subroutine *VUINTERACTION*:

- can be used to define the mechanical and thermal interaction between contact surfaces;
- must provide the entire definition of the interaction between the contact surfaces;
- can utilize a user-specified tracking thickness to determine potential points of interaction on a surface (and thus which nodes should be passed into the subroutine);
- can use and update solution-dependent state variables for node-to-face contact and node-to-analytical rigid surface contact; and
- must be used with the general contact algorithm.

## Terminology

The use of user subroutine *VUINTERACTION* requires familiarity with the following terminology.

### Tracking Thickness

For efficiency, user subroutine *VUINTERACTION* considers only regions of two surfaces that are likely to be in contact or come into contact in a given increment. This likelihood is defined by a tracking thickness: only portions of surfaces separated by less than the tracking thickness in a given increment are passed into the subroutine; portions of the surfaces with a separation larger than the tracking thickness are ignored for the current increment. Surface thicknesses are accounted for in the separation calculations.

Abaqus/Explicit provides an internal default value for the tracking thickness, but a nondefault value can be specified; see *Tracking Thickness When VUINTER or VUINTERACTION Is Used*. The tracking thickness is passed into *VUINTERACTION* using the variable `rData(4)`.

### Proximity Points

A proximity point is a potential point of interaction for you to consider in user subroutine *VUINTERACTION*. Each proximity point is primarily associated with a secondary node or a point along a secondary edge; the proximity point also references a corresponding, locally nearest point on the main surface within the tracking thickness. A proximity point exists for each pairing of secondary node and proximal main surface point. Therefore, more than one proximity point may reference the same node on the secondary surface but different points on the main surface if multiple local minimum distances to the secondary node exist on the main surface; this phenomenon commonly occurs near the corners of a main surface. No proximity points exist for a secondary

node that is separated from the main surface by more than the tracking thickness. A two-dimensional representation for contact between portions of shell surfaces is shown in *Figure 1*.

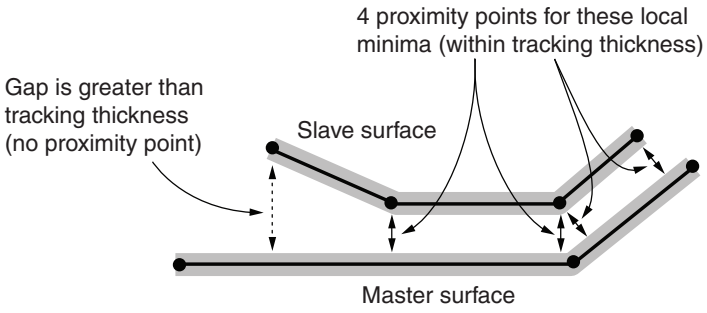



**Figure 1: Four proximity points are associated with three secondary nodes in this surface pairing.**

The number of proximity points currently being passed into user subroutine *VUINTERACTION* is `nBlock`. The array `jSecUid(nNodSec,nBlock)` gives the secondary surface node numbers associated with the proximity points. The variable `nNodSec` indicates whether a single secondary node (for node-to-face contact) or two secondary nodes of an edge (for edge-to-edge contact) are associated with each proximity point. Similarly, the array `jMainUid(nNodMain,nBlockAnal)` gives the main surface node numbers associated with the proximity points; the nodes can belong to a facet, an edge, or an analytical surface. The variable `nNodMain` indicates the number of main nodes associated with each proximity point.

An Abaqus/Explicit model can be defined in terms of an assembly of part instances (see *Assembly Definition*). In such models a node number is an internally generated node number. If the original node number and part instance name are required, call the utility routine `VGETPARTINFO` (see *Obtaining Part Information*).

## Local Coordinate System

The array `dircos` defines the direction cosines of a local coordinate system for each proximity point. The local 1-direction corresponds to the contact normal direction from the perspective of the secondary node. For a two-dimensional *VUINTERACTION* model the local 2-direction is the tangent direction defined by the cross product of the vector into the plane of the model (0., 0., −1.0) and the secondary normal. For a three-dimensional *VUINTERACTION* model the local 2-direction and 3-direction correspond to two orthogonal tangent directions, $\mathbf{t}_1$ and $\mathbf{t}_2$, which are set as follows:

- If the main surface is a cylindrical analytical surface, the local 2-direction corresponds to the generator direction (see *Analytical Rigid Surface Definition*), and the local 3-direction is the cross product of the local 1-direction and 2-direction.
- If the main surface is an analytical surface of revolution, the local 3-direction corresponds to the hoop direction, and the local 2-direction is the cross product of the local 1-direction and 3-direction.
- If the main surface is element-based, the tangential directions are based on the secondary normal and the line connecting the first and third nodes on the main facet.

For the two cases listed above involving analytical surfaces, the local tangential directions will reflect a rotation of the main surface. For the last case (element-based main surface) the tangential directions follow the rotation of the main surface only approximately. The second tangential direction is constructed such that it is perpendicular to the secondary normal and the line going from the first to the third node on the main facet. The secondary normal, the first tangent, and the second tangent form a right-handed system.

## Conventions for Stress and Heat Flux

A positive normal stress denotes a pressure directed into the surface (opposite the local normal direction). Positive shear stresses denote shear tractions in the direction of the local surface tangents.

A positive flux indicates heat flowing into a surface, and a negative flux denotes heat leaving the surface. Flux must be specified for both surfaces, and they need not be equal and opposite so that effects such as frictional dissipation and differential surface heating can be modeled.

## User Subroutine Interface

```
      subroutine vuinteraction (
C Read/Write -
     *   stress, fluxSec, fluxMain,
     *   state, sed,
C Write only -
     *   sfd, scd, spd, svd,
C Read only -
     *   nBlock, nBlockAnal, nBlockEdge,
     *   nNodState, nNodSec, nNodMain, nDir,
     *   nStates, nProps, nTemp, nFields,
     *   jFlags, rData,
     *   surfInt, surfSec, surfMain,
     *   jSecUid, jMainUid, props,
     *   penetration, drDisp, dRot, dircos, stiffDef, conductDef,
     *   coordSec, coordMain, areaProx, shapeSec, shapeMain,
     *   tempSec, tempMain, dTempSec, dTempMain,
     *   fieldSec, fieldMain, dFieldSec, dFieldMain )
C
      include `vaba_param.inc'
C
      dimension stress(nDir,nBlock),
     *   fluxSec(nBlock),
     *   fluxMain(nBlock),
     *   state(nStates,nNodState,nBlock),
     *   sed(nBlock),
     *   sfd(nBlock),
     *   scd(nBlock),
     *   spd(nBlock),
     *   svd(nBlock),
     *   jSecUid(nNodSec,nBlock),
     *   jMainUid(nNodMain,nBlockAnal),
     *   props(nProps),
     *   penetration(nBlock),
     *   drDisp(nDir,nBlock),
     *   dRot(2,2,nBlock),
     *   stiffDef(nBlock),
     *   conductDef(nBlock),
     *   dircos(nDir,nDir,nBlock),
     *   coordSec(nDir,nNodSec,nBlock),
     *   coordMain(nDir,nNodMain,nBlockAnal),
     *   areaProx(nBlock),
     *   shapeSec(nNodSec,nBlockEdge),
     *   shapeMain(nNodMain,nBlockAnal),
     *   tempSec(nBlock),
     *   tempMain(nBlockAnal),
     *   dTempSec(nBlock),
     *   dTempMain(nBlockAnal),
```

```
     *   fieldSec(nFields,nBlock),
     *   fieldMain(nFields,nBlockAnal),
     *   dFieldSec(nFields,nBlock),
     *   dFieldMain(nFields,nBlockAnal)
C
      parameter( iKStep    = 1,
     *           iKInc     = 2,
     *           iLConType = 3,
     *           nFlags    = 3 )
C
      parameter( iTimStep   = 1,
     *           iTimGlb    = 2,
     *           iDTimCur   = 3,
     *           iTrackThic = 4,
     *           nData      = 4 )
C
      dimension jFlags(nFlags), rData(nData)
C
      character*80 surfInt, surfSec, surfMain
C
      user coding to define stress,
      and, optionally, fluxSec, fluxMain, state, sed, sfd, scd, spd,
      and svd
C
      return
      end
```

## Variables to Be Defined

`stress(nDir, nBlock)` On entry this array contains the stress defined in the local system at the proximity points during the previous time increment. It must be updated to the stress at the interface in the current time increment.

## Variables That Can Be Updated

`fluxSec(nBlock)` On entry this array contains the flux entering the secondary surface during the previous time increment. It must be updated to the flux entering the secondary surface during the current increment.

`fluxMain(nBlock)` On entry this array contains the flux entering the main surface during the previous time increment. It must be updated to the flux entering the main surface during the current time increment.

`state(nStates,nNodState,nBlock)` This array contains the user-defined solution-dependent state variables for the proximity points. The use of the state variables is applicable only for node-to-face contact. See *User-Defined Interfacial Constitutive Behavior* for more information on the size of this array. This array will be passed in containing the values of these variables prior to the call to user subroutine *VUINTERACTION*.

If any of the solution-dependent state variables is being used in conjunction with the interaction, it must be updated in this subroutine. These state variables need to be updated with care: outside the user subroutine these state variables are single-valued per secondary node, but multiple proximity points may refer to the same secondary node. Each proximity point may be passed

into the user subroutine independently in a given increment, possibly on separate calls to the user subroutine; therefore, you may end up advancing the state variables for the associated node multiple times for a single time increment. To keep track of whether or not a node state is advanced, you may want to use one of the state variables exclusively for this purpose. You could set that selected state variable to the current increment number and update the state only if it is not already set to the current increment number.

`sed(nBlock)` On entry this array contains the elastic energy density at the proximity points at the beginning of the increment. It can be updated to contain the elastic energy density at the end of the current time increment. These values contribute to the output variable ALLSE and have no effect on other solution variables. The use of this variable is applicable only for node-to-face contact.

`sfd(nBlock)` This array can be updated to contain the increment in frictional dissipation at each proximity point (units of energy per unit area). These values contribute to the output variables SFDR and ALLFD and have no effect on other solution variables. The use of this variable is applicable only for node-to-face contact.

`scd(nBlock)` This array can be updated to contain the increment in creep dissipation at each proximity point (units of energy per unit area). These values contribute to the output variables SFDR and ALLCD and have no effect on other solution variables. The use of this variable is applicable only for node-to-face contact.

`spd(nBlock)` This array can be updated to contain the increment in plastic dissipation at each proximity point (units of energy per unit area). These values contribute to the output variables SFDR and ALLPD and have no effect on other solution variables. The use of this variable is applicable only for node-to-face contact.

`svd(nBlock)` This array can be updated to contain the increment in viscous dissipation at each proximity point (units of energy per unit area). These values contribute to the output variables SFDR and ALLVD and have no effect on other solution variables. The use of this variable is applicable only for node-to-face contact.

## Variables Passed in for Information

`nBlock` Number of proximity points to be processed in this call to *VUINTERACTION*.

`nBlockAnal` 1 for analytical rigid main surface; `nBlock` otherwise.

`nBlockEdge` `nBlock` for edge type secondary surface; 1 otherwise.

`nNodState` 1 for node-to-face and node-to-analytical rigid surface contact; not applicable for edge-to-edge contact.

`nNodSec` 1 for node-to-face and node-to-analytical rigid surface contact; 2 for edge-to-edge contact.

`nNodMain` 1 for analytical rigid main surface; 2 for edge-type main surface; 2 for two-dimensional facet-type main surface; 4 for three-dimensional facet-type main surface.

| | |
|---|---|
| `nDir` | Number of coordinate directions at the proximity points (2 for two-dimensional models; 3 for three-dimensional models). |
| `nStates` | Number of user-defined state variables. |
| `nProps` | User-specified number of property values associated with this interaction model. |
| `nTemp` | 1 if the temperature is defined and 0 if the temperature is not defined. |
| `nFields` | Number of predefined field variables. |
| `jFlag(1)` | Step number. |
| `jFlag(2)` | Increment number. |
| `jFlag(3)` | 1 for node-to-face contact, 2 for edge-to-edge contact, and 3 for node-to-analytical rigid surface contact. |
| `rData(1)` | Value of step time. |
| `rData(2)` | Value of total time. |
| `rData(3)` | Current increment in time from $t = t_{curr} - \Delta t$ to $t = t_{curr}$. |
| `rData(4)` | This variable contains the value of the tracking thickness specified for the surface interaction. |
| `surfInt` | User-specified surface interaction name, left justified. |
| `surfSec` | Secondary surface name, currently set to a blank. |
| `surfMain` | Main surface name, currently set to a blank. |
| `jSecUid(nNodSec,nBlock)` | This array lists the surface node numbers of the secondary surface nodes associated with each proximity point. |
| `jMainUid(nNodMain,nBlockAnal)` | This array lists the surface node numbers of the main surface nodes that make up the facet, edge, or analytical rigid surface associated with each proximity point. |
| `props(nProps)` | User-specified vector of property values to define the interaction between the tracking surfaces. |
| `penetration(nBlock)` | The relative position of the proximity points, with respect to the main surface, in the normal direction (a positive value indicates a penetration, and a negative value indicates a gap) during the current time increment. |
| `drDisp(nDir,nBlock)` | An array containing the increments in relative positions of the proximity points with respect to the associated main surfaces during the current time increment. |
| `dRot(2,2,nBlock)` | This argument is currently undefined. |
| `stiffDef(nBlock)` | Values of the default penalty stiffnesses (stress per unit penetration, units of $\mathbf{FL^{-3}}$). |
| `conductDef(nBlock)` | Values of the default penalty conductances (units of $\mathbf{J\theta^{-1}T^{-1}}$). |
| `dircos(nDir,nDir,nBlock)` | Direction cosines of the local surface coordinate system. The first array index corresponds to the components of the local directions, and the second array index corresponds to the local direction number. The first direction (`dircos(1..nDir,1,...)`) is the normal to the surface. The second direction (`dircos(1..nDir,2,...)`) is the first surface tangent. |

| | |
|---|---|
| | For a three-dimensional surface, the third direction (`dircos(1..3,3,...)`) is the second surface tangent. If the main surface is an analytical rigid surface, the numbers in `dircos` are valid only if the corresponding parts in `penetration` are valid (i.e., not equal to `r_MaxVal`). |
| `coordSec(nDir,nNodSec,nBlock)` | Array containing the `nDir` components of the current coordinates of the proximity points. |
| `coordMain(nDir,nNodMain,nBlockAnal)` | Array containing the `nDir` components of the current coordinates of the nodes on the main surface. If the main surface is an analytical rigid surface, this array is passed in as a dummy array. |
| `areaProx(nBlock)` | Contact area associated with a proximity point. The sum of the contact areas among all proximity points associated with a single secondary node equals the surface area associated with that secondary node (equal to 1 for node-based surface nodes). Therefore, the contact area at a proximity point depends on the number of other proximity points currently associated with the same secondary node. A proximity point contributes a contact normal force to the associated secondary node that is equal to `stress(1,k)` multiplied by `areaProx(k)`. |
| `shapeSec(nNodSec,nBlockEdge)` | For edge-to-edge contact this array contains the shape functions of the nodes of its secondary edge, evaluated at the location of the contact point. If the contact is not edge-to-edge, this array is passed in as a dummy array. |
| `shapeMain(nNodMain,nBlockAnal)` | For node-to-face and edge-to-edge contact this array contains the shape functions of the nodes of its main surface, evaluated at the location of the contact point. If the main surface is an analytical rigid surface, this array is passed in as a dummy array. |
| `tempSec(nBlock)` | Current temperature at the proximity points on the secondary surface. |
| `tempMain(nBlockAnal)` | Current temperature at the points on the main surface closest to the proximity points. |
| `dTempSec(nBlock)` | Increment in the temperature during the previous time increment at the proximity points on the secondary surface. |
| `dTempMain(nBlockAnal)` | Increment in the temperature during the previous time increment at the points on the main surface closest to the proximity points. |
| `fieldSec(nFields,nBlock)` | Current user-specified predefined field variables at the proximity points on the secondary surface (initial values at the beginning of the analysis and current values during the analysis). |
| `fieldMain(nFields,nBlockAnal)` | Current user-specified predefined field variables at the points on the main surface closest to the proximity points (initial values at the beginning of the analysis and current values during the analysis). |

| | |
|---|---|
| `dFieldSec(nFields,nBlock)` | Increment in the user-specified predefined field variables during the previous time increment at the proximity points on the secondary surface. |
| `dFieldMain(nFields,nBlockAnal)` | Increment in the user-specified predefined field variables during the previous time increment at the points on the main surface closest to the proximity points. |

# VUMAT: User subroutine to define a material's mechanical behavior.

**Products:** Abaqus/Explicit

**References:**

- *User-Defined Mechanical Material Behavior*
- **USER MATERIAL*

## Overview



**Warning:** The use of this user subroutine generally requires considerable expertise. You are cautioned that the implementation of any realistic constitutive model requires extensive development and testing. Initial testing on a single-element model with prescribed traction loading is strongly recommended. The component ordering of the symmetric and nonsymmetric tensors for three-dimensional elements is different from the ordering specified in *Three-Dimensional Solid Element Library* and the ordering used in Abaqus/Standard.

User subroutine *VUMAT*:

- is used to define the mechanical constitutive behavior of a material;
- will be called for blocks of material calculation points for which the material is defined in a user subroutine (*Material Data Definition*);
- can use and update solution-dependent state variables;
- can use any field variables that are passed in; and
- can be used in an adiabatic analysis, provided you define both the inelastic heat fraction and the specific heat for the appropriate material definitions and you store the temperatures and integrate them as user-defined state variables.

## Component Ordering in Tensors

The component ordering depends on whether the tensor is symmetric or nonsymmetric.

### Symmetric Tensors

For symmetric tensors such as the stress and strain tensors, there are `ndir+nshr` components, and the component order is given as a natural permutation of the indices of the tensor. The direct components are first and then the indirect components, beginning with the 12-component. For example, a stress tensor contains `ndir` direct stress components and `nshr` shear stress components, which are passed in as

| Component | 2D Case | 3D Case |
|---|---|---|
| 1 | $\sigma_{11}$ | $\sigma_{11}$ |
| 2 | $\sigma_{22}$ | $\sigma_{22}$ |
| 3 | $\sigma_{33}$ | $\sigma_{33}$ |
| 4 | $\sigma_{12}$ | $\sigma_{12}$ |
| 5 | | $\sigma_{23}$ |
| 6 | | $\sigma_{31}$ |

The shear strain components in user subroutine *VUMAT* are stored as tensor components and not as engineering components; this is different from user subroutine *UMAT* in Abaqus/Standard, which uses engineering components.

### Nonsymmetric Tensors

For nonsymmetric tensors there are `ndir+2*nshr` components, and the component order is given as a natural permutation of the indices of the tensor. The direct components are first and then the indirect components, beginning with the 12-component. For example, the deformation gradient is passed as

| Component | 2D Case | 3D Case |
|---|---|---|
| 1 | $F_{11}$ | $F_{11}$ |
| 2 | $F_{22}$ | $F_{22}$ |
| 3 | $F_{33}$ | $F_{33}$ |
| 4 | $F_{12}$ | $F_{12}$ |
| 5 | $F_{21}$ | $F_{23}$ |
| 6 | | $F_{31}$ |
| 7 | | $F_{21}$ |
| 8 | | $F_{32}$ |
| 9 | | $F_{13}$ |

## Initial Calculations and Checks

In the data check phase of the analysis Abaqus/Explicit calls user subroutine *VUMAT* with a set of fictitious strains and a `totalTime` and `stepTime` both equal to 0.0. This is done as a check on your constitutive relation and to calculate the equivalent initial material properties, based on which the transverse shear stiffness (if *VUMAT* is used to define the material response of shell elements and material transverse shear modulus is defined) and the initial elastic wave speeds are computed.

## Defining Local Orientations

All stresses, strains, stretches, and state variables are in the orientation of the local material axes. These local material axes form a basis system in which stress and strain components are stored. This represents a corotational coordinate system in which the basis system rotates with the material. If a user-specified coordinate system (*Orientations*) is used, it defines the local material axes in the undeformed configuration.

## Special Considerations for Various Element Types

In addition to defining the stresses and internal state variables, the use of user subroutine *VUMAT* requires special consideration for various element types.

### Shell and Plane Stress Elements

In the case of shell or plane stress elements, `NDIR`=3 and `NSHR`=1; you must define the thickness strain increment, `strainInc(*,3)`, inside *VUMAT* (typically by enforcing the plane stress constraint or the incompressibility constraint for fully incompressible materials). When the constitutive model employs the right stretch tensor $\mathbf{U}$

(such as in hyperelastic formulations), you should also define the thickness components of the stretch and deformation gradient tensors, `stretchNew (*, 3)` and `defgradNew (*,3)`. Optionally, you can define the internal energies. If you do not define them, Abaqus/Explicit cannot provide a meaningful energy balance.

### Shell Elements

When *VUMAT* is used to define the material response of shell elements, Abaqus/Explicit cannot calculate a default value for the transverse shear stiffness of the element. Hence, you must define the material transverse shear modulus (see *Defining the Elastic Transverse Shear Modulus*) or the element's transverse shear stiffness (see *Shell Section Behavior*).

### Beam Elements

For beam elements the stretch tensor and the deformation gradient tensor are not available. For beams in space you must define the thickness strains, `strainInc(*,2)` and `strainInc(*,3)`. `strainInc(*,4)` is the shear strain associated with twist. Thickness stresses, `stressNew(*,2)` and `stressNew(*,3)`, are assumed to be zero, and any values you assign are ignored.

### Pipe Elements

For pipe elements the stretch tensor and the deformation gradient tensor are not available. The axial strain, `strainInc(*,1)`, and the shear strain, `strainInc(*,4)`, associated with twist are provided along with the hoop stress, `stressNew(*,2)`. The hoop stress is predefined based on your pipe internal and external pressure load definitions (PE, PI, HPE, HPI, PENU, and PINU), and it should not be modified here. The thickness stress, `stressNew(*,3)`, is assumed to be zero and any value you assign is ignored. You must define the axial stress, `stressNew(*,1)`, and the shear stress, `stressNew(*,4)`. You must also define hoop strain, `strainInc(*,2)`, and the pipe thickness strain, `strainInc(*,3)`.

## Deformation Gradient

The polar decomposition of the deformation gradient is written as $\mathbf{F} = \mathbf{R} \cdot \mathbf{U} = \mathbf{V} \cdot \mathbf{R}$, where $\mathbf{U}$ and $\mathbf{V}$ are the right and left symmetric stretch tensors, respectively. The constitutive model is defined in a corotational coordinate system in which the basis system rotates with the material. All stress and strain tensor quantities are defined with respect to the corotational basis system. The right stretch tensor, $\mathbf{U}$, is used. The relative spin tensor $\mathbf{W} - \mathbf{\Omega}$ represents the spin (the antisymmetric part of the velocity gradient) defined with respect to the corotational basis system.

### Special Considerations for Hyperelasticity

Hyperelastic constitutive models in *VUMAT* should be defined in a corotational coordinate system in which the basis system rotates with the material. This is most effectively accomplished by formulating the hyperelastic constitutive model in terms of the stretch tensor, $\mathbf{U}$, instead of in terms of the deformation gradient, $\mathbf{F} = \mathbf{R} \cdot \mathbf{U}$. Using the deformation gradient can present some difficulties because the deformation gradient includes the rotation tensor and the resulting stresses would need to be rotated back to the corotational basis.

## Objective Stress Rates

The Green-Naghdi stress rate is used when the mechanical behavior of the material is defined using user subroutine *VUMAT*. The stress rate obtained with user subroutine *VUMAT* might differ from that obtained with a built-in Abaqus material model. For example, most material models used with solid (continuum) elements in Abaqus/Explicit employ the Jaumann stress rate. This difference in the formulation will cause significant

differences in the results only if finite rotation of a material point is accompanied by finite shear. For a discussion of the objective stress rates used in Abaqus, see *Stress rates*.

## Defining Effective Modulus to Control Time Incrementation in Abaqus/Explicit

The stable time increment in Abaqus/Explicit is a function of the dilatational wave speed of the material and the characteristic length of the element. By default, the dilatational wave speed is determined automatically by Abaqus/Explicit using a numerical calculation of the effective hypoelastic bulk and shear moduli from the user material's constitutive response. In situations when the user material is highly nonlinear, this calculation might not yield a conservative value of the time increment, leading to unstable behavior. To avoid these situations, you can define the values of the effective bulk and shear moduli directly inside user subroutine *VUMAT* when the EFFMOD parameter is specified in the **USER MATERIAL* option. These values are then used by Abaqus/Explicit to compute the dilatational wave speed and the stable time increment. For more information, see *Stability*.

## Material Point Deletion

Material points that satisfy a user-defined failure criterion can be deleted from the model (see *User-Defined Mechanical Material Behavior*). You must specify the state variable number controlling the element deletion flag when you allocate space for the solution-dependent state variables, as explained in *User-Defined Mechanical Material Behavior*. The deletion state variable should be set to a value of one or zero in *VUMAT*. A value of one indicates that the material point is active, while a value of zero indicates that Abaqus/Explicit should delete the material point from the model by setting the stresses to zero. The structure of the block of material points passed to user subroutine *VUMAT* remains unchanged during the analysis; deleted material points are not removed from the block. Abaqus/Explicit will pass zero stresses and strain increments for all deleted material points. Once a material point has been flagged as deleted, it cannot be reactivated.

## User Subroutine Interface

```
      subroutine vumat(
C Read only (unmodifiable)variables -
     1  nblock, ndir, nshr, nstatev, nfieldv, nprops, jInfoArray,
     2  stepTime, totalTime, dtArray, cmname, coordMp, charLength,
     3  props, density, strainInc, relSpinInc,
     4  tempOld, stretchOld, defgradOld, fieldOld,
     5  stressOld, stateOld, enerInternOld, enerInelasOld,
     6  tempNew, stretchNew, defgradNew, fieldNew,
C Write only (modifiable) variables -
     7  stressNew, stateNew, enerInternNew, enerInelasNew )
C
      include 'vaba_param.inc'
      parameter (i_inf_AnnealFlag = 1,
     *     i_inf_Intpt    = 2, ! Integration station number
     *     i_inf_layer  = 3, ! Layer number
     *     i_inf_kspt   = 4, ! Section point number in current layer
     *     i_inf_effModDefn = 5, !  if Bulk/ShearMod need to be defined
     *     i_inf_ElemNumStartLoc = 6, ! Start loc of user element number
     *    i_inf_RateDepOff = 7) ! if RATE DEPENDENCE is considered/ignored
C
      dimension props(nprops), density(nblock), coordMp(nblock,*),
     1  charLength(nblock), dtArray(2*(nblock)+1),
strainInc(nblock,ndir+nshr),
     2  relSpinInc(nblock,nshr), tempOld(nblock),
     3  stretchOld(nblock,ndir+nshr),
     4  defgradOld(nblock,ndir+nshr+nshr),
     5  fieldOld(nblock,nfieldv), stressOld(nblock,ndir+nshr),
     6  stateOld(nblock,nstatev), enerInternOld(nblock),
```

```
     7  enerInelasOld(nblock), tempNew(nblock),
     8  stretchNew(nblock,ndir+nshr),
     8  defgradNew(nblock,ndir+nshr+nshr),
     9  fieldNew(nblock,nfieldv),
     1  stressNew(nblock,ndir+nshr), stateNew(nblock,nstatev),
     2  enerInternNew(nblock), enerInelasNew(nblock), jInfoArray(*)
C
      character*80 cmname
C
      pointer (ptrjElemNum, jElemNum)
      dimension jElemNum(nblock)
C
      lAnneal = jInfoArray(i_inf_AnnealFlag)
      iLayer = jInfoArray(i_inf_layer)
      kspt   = jInfoArray(i_inf_kspt)
      intPt  = jInfoArray(i_inf_Intpt)
      iUpdateEffMod = jInfoArray(i_inf_effModDefn)
      iElemNumStartLoc = jInfoArray(i_inf_ElemNumStartLoc)
      ptrjElemNum = loc(jInfoArray(iElemNumStartLoc))
      lRateDepOff = jInfoArray(i_inf_RateDepOff)

      do 100 km = 1,nblock
        user coding
  100 continue

      return
      end
```

## Variables to Be Defined

`stressNew (nblock, ndir+nshr)` Stress tensor at each material point at the end of the increment.

`stateNew (nblock, nstatev)` State variables at each material point at the end of the increment. You define the size of this array by allocating space for it (see *About User Subroutines and Utilities* for more information).

## Variables That Can Be Updated

`enerInternNew (nblock)` Internal energy per unit mass at each material point at the end of the increment.

`enerInelasNew (nblock)` Dissipated inelastic energy per unit mass at each material point at the end of the increment.

`dtArray(2 : nblock+1)` Shear modulus at each material point for `nblock` elements in `dtArray(2 : nblock+1)`. The shear modulus returned by the user subroutine is used to compute the time increment only when the EFFMOD parameter is specified in the **USER MATERIAL* option.

`dtArray(nblock+2 : 2*(nblock)+1)` Bulk modulus at each material point for `nblock` elements in `dtArray(nblock+2 : 2*(nblock)+1)`. The bulk modulus returned by the subroutine is used to compute the time increment only when the EFFMOD parameter is specified in the **USER MATERIAL* option.

## Variables Passed in for Information

`nblock` Number of material points to be processed in this call to *VUMAT*.

`ndir` Number of direct components in a symmetric tensor.

`nshr` Number of indirect components in a symmetric tensor.

`nstatev` Number of user-defined state variables that are associated with this material type (you define this as described in *Allocating Space for Solution-Dependent State Variables*).

`nfieldv` Number of user-defined external field variables.

`nprops` User-specified number of user-defined material properties.

`jInfoArray`

| | |
|---|---|
| `jInfoArray(i_inf_AnnealFlag)` | Flag indicating whether the routine is being called during an annealing process.<br>`jInfoArray(i_inf_AnnealFlag)=0` indicates that the routine is being called during a normal mechanics increment.<br>`jInfoArray(i_inf_AnnealFlag)=1` indicates that this is an annealing process and you should reinitialize the internal state variables, `stateNew`, if necessary. Abaqus/Explicit automatically sets the stresses, stretches, and state to a value of zero during the annealing process. |
| `jInfoArray(i_inf_Intpt)` | Integration location number. |
| `jInfoArray(i_inf_layer)` | Layer number. |
| `jInfoArray(i_inf_kspt)` | Section point number in the current layer. |
| `jInfoArray(i_inf_effModDefn)` | Flag indicating whether the bulk modulus and shear modulus must be updated.<br>`jInfoArray(i_inf_effModDefn) = 1` indicates that you must update the values for the bulk and shear moduli. |
| `jInfoArray(i_inf_ElemNumStartLoc)` | Start location in `jInfoArray` for user-defined element numbers of all elements in `nblock`. |
| `jInfoArray(i_inf_RateDepOff)` | Flag indicating whether the routine is being called during a step where material rate dependence is ignored.<br>`jInfoArray(i_inf_RateDepOff)=0` indicates that the routine is being called during a step where material rate dependence is considered.<br>`jInfoArray(i_inf_RateDepOff)=1` indicates that the routine is being called during a step where material rate dependence is ignored. |

`stepTime` Value of time since the step began.

`totalTime` Value of total time. The time at the beginning of the step is given by `totalTime - stepTime`.

`dtArray(1)` Time increment size.

`cmname` User-specified material name, left justified. It is passed in as an uppercase character string. Some internal material models are given names starting with the “ABQ_” character string. To avoid conflict, you should not use “ABQ_” as the leading string for `cmname`.

`coordMp(nblock,*)` Material point coordinates. It is the midplane material point for shell elements and the centroid for beam and pipe elements.

`charLength(nblock)` Characteristic element length, which is either the default value based on the geometric mean or the user-defined characteristic element length defined in user

| | |
|---|---|
| | subroutine VUCHARLENGTH. The default value is a typical length of a line across an element for a first-order element; it is half of the same typical length for a second-order element. For beams, pipes, and trusses, the default value is a characteristic length along the element axis. For membranes and shells, it is a characteristic length in the reference surface. For axisymmetric elements, it is a characteristic length in the $r$–$z$ plane only. For cohesive elements it is equal to the constitutive thickness. |
| `props(nprops)` | User-supplied material properties. |
| `density(nblock)` | Current density at the material points in the midstep configuration. This value might be inaccurate in problems where the volumetric strain increment is very small. If an accurate value of the density is required in such cases, the analysis should be run in double precision. This value of the density is not affected by mass scaling. |
| `strainInc (nblock, ndir+nshr)` | Strain increment tensor at each material point. |
| `relSpinInc (nblock, nshr)` | Incremental relative rotation vector at each material point defined in the corotational system. Defined as $\Delta t\ (\mathbf{W} - \boldsymbol{\Omega})$, where $\mathbf{W}$ is the antisymmetric part of the velocity gradient, $\mathbf{L}$, and $\boldsymbol{\Omega} = \dot{\mathbf{R}} \cdot \mathbf{R}^{\mathbf{T}}$. Stored in 3D as $(32, 13, 21)$ and in 2D as $(21)$. |
| `tempOld(nblock)` | Temperatures at each material point at the beginning of the increment. |
| `stretchOld (nblock, ndir+nshr)` | Stretch tensor, $\mathbf{U}$, at each material point at the beginning of the increment defined from the polar decomposition of the deformation gradient by $\mathbf{F} = \mathbf{R} \cdot \mathbf{U}$. |
| `defgradOld (nblock,ndir+2*nshr)` | Deformation gradient tensor at each material point at the beginning of the increment. Stored in 3D as ($F_{11}$, $F_{22}$, $F_{33}$, $F_{12}$, $F_{23}$, $F_{31}$, $F_{21}$, $F_{32}$, $F_{13}$) and in 2D as ($F_{11}$, $F_{22}$, $F_{33}$, $F_{12}$, $F_{21}$). |
| `fieldOld (nblock, nfieldv)` | Values of the user-defined field variables at each material point at the beginning of the increment. |
| `stressOld (nblock, ndir+nshr)` | Stress tensor at each material point at the beginning of the increment. |
| `stateOld (nblock, nstatev)` | State variables at each material point at the beginning of the increment. |
| `enerInternOld (nblock)` | Internal energy per unit mass at each material point at the beginning of the increment. |
| `enerInelasOld (nblock)` | Dissipated inelastic energy per unit mass at each material point at the beginning of the increment. |
| `tempNew(nblock)` | Temperatures at each material point at the end of the increment. |
| `stretchNew (nblock, ndir+nshr)` | Stretch tensor, $\mathbf{U}$, at each material point at the end of the increment defined from the polar decomposition of the deformation gradient by $\mathbf{F} = \mathbf{R} \cdot \mathbf{U}$. |
| `defgradNew (nblock,ndir+2*nshr)` | Deformation gradient tensor at each material point at the end of the increment. Stored in 3D as ($F_{11}$, $F_{22}$, $F_{33}$, $F_{12}$, $F_{23}$, $F_{31}$, $F_{21}$, $F_{32}$, $F_{13}$) and in 2D as ($F_{11}$, $F_{22}$, $F_{33}$, $F_{12}$, $F_{21}$). |
| `fieldNew (nblock, nfieldv)` | Values of the user-defined field variables at each material point at the end of the increment. |

## Example: Using More than One User-Defined Material Model

To use more than one user-defined material model, the variable cmname can be tested for different material names inside user subroutine *VUMAT*, as illustrated below:

```
if (cmname(1:4) .eq. 'MAT1') then
   call VUMAT_MAT1(argument_list)
else if (cmname(1:4) .eq. 'MAT2') then
   call VUMAT_MAT2(argument_list)
end if
```

VUMAT_MAT1 and VUMAT_MAT2 are the actual user material subroutines containing the constitutive material models for each material MAT1 and MAT2, respectively. Subroutine *VUMAT* merely acts as a directory here. The argument list can be the same as that used in subroutine *VUMAT*. The material names must be in uppercase characters since cmname is passed in as an uppercase character string.

## Example: Elastic/Plastic Material with Kinematic Hardening

As a simple example of the coding of subroutine *VUMAT*, consider the generalized plane strain case for an elastic/plastic material with kinematic hardening. The basic assumptions and definitions of the model are as follows.

Let $\boldsymbol{\sigma}$ be the current value of the stress, and define $\mathbf{S}$ to be the deviatoric part of the stress. The center of the yield surface in deviatoric stress space is given by the tensor $\boldsymbol{\alpha}$, which has initial values of zero. The stress difference, $\boldsymbol{\xi}$, is the stress measured from the center of the yield surface and is given by

$$\boldsymbol{\xi} = \mathbf{S} - \boldsymbol{\alpha}.$$

The von Mises yield surface is defined as

$$f(\boldsymbol{\sigma}) = \tfrac{1}{2}\boldsymbol{\xi} : \boldsymbol{\xi} - \tfrac{1}{3}\sigma_0^2,$$

where $\sigma_0$ is the uniaxial equivalent yield stress. The von Mises yield surface is a cylinder in deviatoric stress space with a radius of

$$R = \sqrt{\tfrac{2}{3}}\sigma_0.$$

For the kinematic hardening model, $R$ is a constant. The normal to the Mises yield surface can be written as

$$\mathbf{Q} = \sqrt{\tfrac{3}{2}}\frac{\boldsymbol{\xi}}{\sigma_0} \quad .$$

We decompose the strain rate into an elastic and plastic part using an additive decomposition:

$$\dot{\boldsymbol{\epsilon}} = \dot{\boldsymbol{\epsilon}}^{el} + \dot{\boldsymbol{\epsilon}}^{pl}.$$

The plastic part of the strain rate is given by a normality condition

$$\dot{\boldsymbol{\epsilon}}^{pl} = \dot{\gamma}\mathbf{Q},$$

where the scalar multiplier $\dot{\gamma}$ must be determined. A scalar measure of equivalent plastic strain rate is defined by

$$\dot{\bar{\epsilon}}^{pl} = \sqrt{\tfrac{2}{3}\dot{\boldsymbol{\epsilon}}^{pl} : \dot{\boldsymbol{\epsilon}}^{pl}} \quad .$$

The stress rate is assumed to be purely due to the elastic part of the strain rate and is expressed in terms of Hooke's law by

$$\dot{\boldsymbol{\sigma}} = \lambda \operatorname{trace}\left(\dot{\boldsymbol{\epsilon}}^{el}\right)\mathbf{I} + 2\mu\dot{\boldsymbol{\epsilon}}^{el},$$

where $\lambda$ and $2\mu$ are the Lamés constants for the material.

The evolution law for $\boldsymbol{\alpha}$ is given as

$$\dot{\boldsymbol{\alpha}} = \tfrac{2}{3}\dot{\gamma}H\mathbf{Q},$$

where $H$ is the slope of the uniaxial yield stress versus plastic strain curve.

During active plastic loading the stress must remain on the yield surface, so that

$$\sqrt{\mathbf{Q} : \mathbf{Q}} = 1.$$

The equivalent plastic strain rate is related to $\dot{\gamma}$ by

$$\dot{\bar{\epsilon}}^{pl} = \sqrt{\tfrac{2}{3}}\dot{\gamma}.$$

The kinematic hardening constitutive model is integrated in a rate form as follows. A trial elastic stress is computed as

$$\boldsymbol{\sigma}_{new}^{trial} = \boldsymbol{\sigma}_{old} + \lambda \operatorname{trace}\left(\boldsymbol{\Delta\epsilon}\right)\mathbf{I} + 2\mu\boldsymbol{\Delta\epsilon},$$

where the subscripts *old* and *new* refer to the beginning and end of the increment, respectively. If the trial stress does not exceed the yield stress, the new stress is set equal to the trial stress. If the yield stress is exceeded, plasticity occurs in the increment. We then write the incremental analogs of the rate equations as

$$\boldsymbol{\sigma}_{new} = \boldsymbol{\sigma}_{new}^{trial} - 2\mu\boldsymbol{\Delta\epsilon}^{pl} = \boldsymbol{\sigma}_{new}^{trial} - 2\mu\Delta\gamma\mathbf{Q},$$

$$\boldsymbol{\alpha}_{new} = \boldsymbol{\alpha}_{old} + \tfrac{2}{3}H\Delta\gamma\mathbf{Q},$$

$$\bar{\epsilon}_{new}^{pl} = \bar{\epsilon}_{old}^{pl} + \sqrt{\tfrac{2}{3}}\Delta\gamma,$$

where

$$\Delta\gamma = \dot{\gamma}\Delta t.$$

From the definition of the normal to the yield surface at the end of the increment, $\mathbf{Q}$,

$$\boldsymbol{\alpha}_{new} + \sqrt{\tfrac{2}{3}}\sigma_0\mathbf{Q} = \mathbf{S}_{new}.$$

This can be expanded using the incremental equations as

$$\boldsymbol{\alpha}_{old} + \tfrac{2}{3}H\Delta\gamma\mathbf{Q} + \sqrt{\tfrac{2}{3}}\sigma_0\mathbf{Q} = \mathbf{S}_{new}^{trial} - \Delta\gamma 2\mu\mathbf{Q}.$$

Taking the tensor product of this equation with $\mathbf{Q}$, using the yield condition at the end of the increment, and solving for $\boldsymbol{\Delta\gamma}$:

$$\Delta\gamma = \tfrac{1}{2\mu(1+H/3\mu)}\left(\left(\boldsymbol{\xi}_{new}^{trial} : \boldsymbol{\xi}_{new}^{trial}\right)^{1/2} - \sqrt{\tfrac{2}{3}}\sigma_0\right).$$

The value for $\boldsymbol{\Delta\gamma}$ is used in the incremental equations to determine $\boldsymbol{\sigma}_{new}$, $\boldsymbol{\alpha}_{new}$, and $\bar{\boldsymbol{\epsilon}}_{new}^{pl}$.

This algorithm is often referred to as an elastic predictor, radial return algorithm because the correction to the trial stress under the active plastic loading condition returns the stress state to the yield surface along the direction

defined by the vector from the center of the yield surface to the elastic trial stress. The subroutine would be coded as follows:

```
       subroutine vumat(
C Read only -
     1  nblock, ndir, nshr, nstatev, nfieldv, nprops, jInfoArray,
     2  stepTime, totalTime, dtArray, cmname, coordMp, charLength,
     3  props, density, strainInc, relSpinInc,
     4  tempOld, stretchOld, defgradOld, fieldOld,
     3  stressOld, stateOld, enerInternOld, enerInelasOld,
     6  tempNew, stretchNew, defgradNew, fieldNew,
C Write only -
     5  stressNew, stateNew, enerInternNew, enerInelasNew )
C
      include 'vaba_param.inc'
C
C J2 Mises Plasticity with kinematic hardening for plane
C strain case.
C Elastic predictor, radial corrector algorithm.
C
C The state variables are stored as:
C      STATE(*,1) = back stress component 11
C      STATE(*,2) = back stress component 22
C      STATE(*,3) = back stress component 33
C      STATE(*,4) = back stress component 12
C      STATE(*,5) = equivalent plastic strain
C
C
C All arrays dimensioned by (*) are not used in this algorithm
      dimension props(nprops), density(nblock),
     1  coordMp(nblock,*),
     2  charLength(*), dtArray(*), strainInc(nblock,ndir+nshr),
     3  relSpinInc(*), tempOld(*),
     4  stretchOld(*), defgradOld(*),
     5  fieldOld(*),  stressOld(nblock,ndir+nshr),
     6  stateOld(nblock,nstatev), enerInternOld(nblock),
     7  enerInelasOld(nblock), tempNew(*),
     8  stretchNew(*), defgradNew(*), fieldNew(*),
     9  stressNew(nblock,ndir+nshr), stateNew(nblock,nstatev),
     2  enerInternNew(nblock), enerInelasNew(nblock), jInfoArray(*)
C
      parameter (i_inf_AnnealFlag = 1,
     *     i_inf_Intpt    = 2, ! Integration station number
     *     i_inf_layer  = 3, ! Layer number
     *     i_inf_kspt   = 4, ! Section point number in current layer
     *     i_inf_effModDefn = 5, ! =1 if Bulk/ShearMod need to be defined
     *     i_inf_ElemNumStartLoc = 6, ! Start loc of user element number
     *     i_inf_RateDepOff = 7) ! = 1 if RATE DEPENDENCE = OFF
C
      parameter( zero = 0., one = 1., two = 2., three = 3.,
     1  third = one/three, half = .5, twoThirds = two/three,
     2  threeHalfs = 1.5 )

C
      character*80 cmname
C
      pointer (ptrjElemNum, jElemNum)
      dimension jElemNum(nblock)
C
      lAnneal = jInfoArray(i_inf_AnnealFlag)
      iLayer = jInfoArray(i_inf_layer)
```

```
      kspt   = jInfoArray(i_inf_kspt)
      intPt  = jInfoArray(i_inf_Intpt)
      iUpdateEffMod = jInfoArray(i_inf_effModDefn)
      iElemNumStartLoc = jInfoArray(i_inf_ElemNumStartLoc)
      ptrjElemNum = loc(jInfoArray(iElemNumStartLoc))
      lRateDepOff = jInfoArray(i_inf_RateDepOff)
c
      e     = props(1)
      xnu   = props(2)
      yield = props(3)
      hard  = props(4)
C
      twomu  = e / ( one + xnu )
      thremu = threeHalfs * twomu
      sixmu  = three * twomu
      alamda = twomu * ( e - twomu ) / ( sixmu - two * e )
      term   = one / ( twomu * ( one + hard/thremu ) )
      con1   = sqrt( twoThirds )
C
      do 100 i = 1,nblock
C
C Trial stress
        trace  = strainInc(i,1) + strainInc(i,2) + strainInc(i,3)
        sig1 = stressOld(i,1) + alamda*trace + twomu*strainInc(i,1)
        sig2 = stressOld(i,2) + alamda*trace + twomu*strainInc(i,2)
        sig3 = stressOld(i,3) + alamda*trace + twomu*strainInc(i,3)
        sig4 = stressOld(i,4)                + twomu*strainInc(i,4)
C
C Trial stress measured from the back stress
        s1 = sig1 - stateOld(i,1)
        s2 = sig2 - stateOld(i,2)
        s3 = sig3 - stateOld(i,3)
        s4 = sig4 - stateOld(i,4)
C
C Deviatoric part of trial stress measured from the back stress
        smean = third * ( s1 + s2 + s3 )
        ds1 = s1 - smean
        ds2 = s2 - smean
        ds3 = s3 - smean
C
C Magnitude of the deviatoric trial stress difference
        dsmag = sqrt( ds1**2 + ds2**2 + ds3**2 + 2.*s4**2 )
C
C Check for yield by determining the factor for plasticity,
C zero for elastic, one for yield
        radius = con1 * yield
        facyld = zero
        if(  dsmag - radius .ge. zero ) facyld = one
C
C Add a protective addition factor to prevent a divide by zero
C when dsmag is zero. If dsmag is zero, we will not have exceeded
C the yield stress and facyld will be zero.
        dsmag  = dsmag + ( one - facyld )
C
C Calculated increment in gamma (this explicitly includes the
C time step)
        diff   = dsmag - radius
        dgamma = facyld * term * diff
C
C Update equivalent plastic strain
        deqps  = con1 * dgamma
```

```
        stateNew(i,5) = stateOld(i,5) + deqps
C
C Divide dgamma by dsmag so that the deviatoric stresses are
C explicitly converted to tensors of unit magnitude in the
C following calculations
        dgamma = dgamma / dsmag
C
C Update back stress
        factor  = hard * dgamma * twoThirds
        stateNew(i,1) = stateOld(i,1) + factor * ds1
        stateNew(i,2) = stateOld(i,2) + factor * ds2
        stateNew(i,3) = stateOld(i,3) + factor * ds3
        stateNew(i,4) = stateOld(i,4) + factor *  s4
C
C Update the stress
        factor   = twomu * dgamma
        stressNew(i,1) = sig1 - factor * ds1
        stressNew(i,2) = sig2 - factor * ds2
        stressNew(i,3) = sig3 - factor * ds3
        stressNew(i,4) = sig4 - factor *  s4
C
C Update the specific internal energy -
        stressPower = half * (
     1    ( stressOld(i,1)+stressNew(i,1) )*strainInc(i,1)
     1    +     ( stressOld(i,2)+stressNew(i,2) )*strainInc(i,2)
     1    +     ( stressOld(i,3)+stressNew(i,3) )*strainInc(i,3)
     1    + two*( stressOld(i,4)+stressNew(i,4) )*strainInc(i,4) )
C
        enerInternNew(i) = enerInternOld(i)
     1    + stressPower / density(i)
C
C Update the dissipated inelastic specific energy -
        plasticWorkInc = dgamma * half * (
     1    ( stressOld(i,1)+stressNew(i,1) )*ds1
     1    +     ( stressOld(i,2)+stressNew(i,2) )*ds2
     1    +     ( stressOld(i,3)+stressNew(i,3) )*ds3
     1    + two*( stressOld(i,4)+stressNew(i,4) )*s4 )
        enerInelasNew(i) = enerInelasOld(i)
      1   + plasticWorkInc / density(i)
  100 continue
C
      return
      end
```

# VUMATHT: User subroutine to define a material's thermal behavior.

**Products:** Abaqus/Explicit

**References:**

- *User-Defined Thermal Material Behavior*
- **USER MATERIAL*
- *Freezing of a square solid: the two-dimensional Stefan problem*
- *VUMATHT*

## Overview



**Warning:** The use of this user subroutine generally requires considerable expertise. You are cautioned that the implementation of any realistic constitutive model requires extensive development and testing. Initial testing on a single-element model with prescribed traction loading is strongly recommended.

User subroutine *VUMATHT*:

- can be used to define the thermal constitutive behavior of a material as well as internal heat generation during heat transfer processes;
- will be called for blocks of material calculation points for which the thermal material behavior is defined in a user subroutine;
- can use and update solution-dependent state variables;
- can use any field variables that are passed in;
- can be used with thermally coupled continuum elements and Eulerian elements; and
- is described further in *User-Defined Thermal Material Behavior*.

## User Subroutine Interface

```
      subroutine vumatht (
C Read only (unmodifiable) variables -
     *     nblock, nElem, nIntPt, nLayer, nSectPt,
     *     ntgrad, nstatev, nfieldv, nprops,
     *     cmname, stepTime, totalTime, dt,
     *     coordMp, density, props,
     *     tempOld, fieldOld, stateOld, enerThermOld,
     *     tempNew, tempgradNew, fieldNew,
C Write only (modifiable) variables -
     *     stateNew, fluxNew, enerThermNew, dEnerThDTemp, condEff )
C
      include 'vaba_param.inc'
C
      dimension nElem(nblock)
C
      dimension coordMp(nblock,*), density(nblock), props(nprops),
     *    tempOld(nblock), fieldOld(nblock, nfieldv),
     *    stateOld(nblock, nstatev), enerThermOld(nblock),
     *    tempNew(nblock), tempgradNew(nblock, ntgrad),
     *    fieldNew(nblock, nfieldv), stateNew(nblock, nstatev),
     *    fluxNew(nblock, ntgrad), enerThermNew(nblock),
     *    dEnerThDTemp(nblock,2), condEff(nblock)
```

```
C
      character*80 cmname
C
      do km = 1, nblock
C         user coding
      end do
C
      return
      end
```

## Variables to Be Defined

| | |
|---|---|
| `stateNew(nblock, nstatev)` | State variables at each material point at the end of the increment. You define the size of this array by allocating space for it (see *About User Subroutines and Utilities* for more information). |
| `fluxNew(nblock, ntgrad)` | Heat flux vector at each material point at the end of the increment. |
| `enerThermNew(nblock)` | Internal thermal energy per unit mass at each material point at the end of the increment. |
| `dEnerThDTemp(nblock,1)` | Variation of internal energy per unit mass with respect to temperature at each material point at the end of the increment. |
| `condEff(nblock)` | Effective conductivity at each material point at the end of the increment. This variable is required to estimate the element stable time increment size in Abaqus/Explicit. You can provide the largest value in conductivity tensor at each material point for a conservative estimate of the element stable time increment size. |

## Variables That Can Be Updated

| | |
|---|---|
| `dEnerThDTemp(nblock,2)` | Variation of internal heat energy per unit mass resulting from internal heat generation (for example, latent heat release and absorption) with respect to the temperature at each material point at the end of the increment. This variable can be defined when a more conservative estimate of the element stable time increment size is needed during internal heat generation. |

## Variables Passed in for Information

| | |
|---|---|
| `nblock` | Number of material points to be processed in this call to *VUMATHT*. |
| `nElem(nblock)` | User-defined element number. |
| `nIntPt` | Integration point number. |
| `nLayer` | Layer number. |
| `nSectPt` | Section point number in the current layer. |
| `ntgrad` | Number of spatial gradients of temperature. |
| `nstatev` | Number of user-defined state variables that are associated with this material type (see *About User Subroutines and Utilities*). |
| `nfieldv` | Number of user-defined external field variables. |
| `nprops` | User-specified number of user-defined material properties. |
| `cmname` | User-specified material name, left justified. It is passed in as an uppercase character string. Some internal material models are given names starting with the “ABQ_” character string. To avoid conflict, you should not use “ABQ_” as the leading string for `cmname`. |
| `stepTime` | Value of time since the step began. |
| `totalTime` | Value of total time. The time at the beginning of the step is given by `totalTime - stepTime`. |
| `dt` | Time increment size. |

| | |
|---|---|
| `coordMp(nblock,*)` | Material point coordinates. |
| `density(nblock)` | Current density at the material points in the midstep configuration. |
| `props(nprops)` | User-supplied material properties. |
| `tempOld(nblock)` | Temperatures at each material point at the beginning of the increment. |
| `fieldOld(nblock, nfieldv)` | Values of the user-defined field variables at each material point at the beginning of the increment. |
| `stateOld(nblock, nstatev)` | State variables at each material point at the beginning of the increment. |
| `enerThermOld(nblock)` | Internal thermal energy per unit mass at each material point at the beginning of the increment. |
| `tempNew(nblock)` | Temperatures at each material point at the end of the increment. |
| `tempgradNew(nblock, ntgrad)` | Current values of the spatial gradients of temperature. |
| `fieldNew(nblock, nfieldv)` | Values of the user-defined field variables at each material point at the end of the increment. |

## Example: Using More than One User-Defined Material Model

To use more than one user-defined material model, the variable `cmname` can be tested for different material names inside user subroutine *VUMATHT*, as illustrated below:

```
if (cmname(1:4) .eq. 'MAT1') then
   call VUMATHT_MAT1(argument_list)
else if (cmname(1:4) .eq. 'MAT2') then
   call VUMATHT_MAT2(argument_list)
end if
```

`VUMATHT_MAT1` and `VUMATHT_MAT2` are the actual user material subroutines containing the thermal constitutive material models for each material `MAT1` and `MAT2`, respectively. User subroutine *VUMATHT* merely acts as a directory here. The argument list can be the same as that used in user subroutine *VUMATHT*. The material names must be in uppercase characters since `cmname` is passed in as an uppercase character string.

## Example: A Simple Thermal Material

As a simple example of the coding of user subroutine *VUMATHT*, consider a material with isotropic conductivity and a constant specific heat. The heat transfer equations are developed here, and the example *VUMATHT* is given. The problem can also be solved by directly specifying thermal conductivity, specific heat, density, and internal heat generation.

First, the heat transfer equations are outlined.

The basic energy balance is

$$\int_V \rho \dot{U} dV = \int_S q\, dS + \int_V r\, dV,$$

where $V$ is the volume of solid material with surface area $S$, $\rho$ is the density of the material, $\dot{U}$ is the material time rate of the internal thermal energy, $q$ is the heat flux per unit area of the body flowing into the body, and $r$ is the heat supplied externally into the body per unit volume. $q$ and $r$ can depend on the strains or displacements of the body.

In Abaqus/Explicit the heat transfer and mechanical solutions are obtained simultaneously by an explicit coupling. Thus, the material time rate of the internal thermal energy can be written as

$$\dot{U} = \frac{\partial U}{\partial \theta}\dot{\theta}.$$

A heat flux vector $\mathbf{f}$ is defined such that

$$q = -\mathbf{f} \cdot \mathbf{n},$$

where $\mathbf{n}$ is the unit outward normal to the surface *S*. Introducing the above relation into the energy balance equation and using the divergence theorem, the following relation is obtained:

$$\int_V \rho \frac{\partial U}{\partial \theta} \dot{\theta} dV = -\int_V \frac{\partial}{\partial \mathbf{x}} \cdot \mathbf{f} dV + \int_V r \, dV.$$

The corresponding weak form is given by

$$\int_V \delta\theta \, \rho \frac{\partial U}{\partial \theta} \dot{\theta} dV = \int_V \delta \mathbf{g} \cdot \mathbf{f} dV + \int_V \delta\theta \, r \, dV + \int_{S_q} \delta\theta \, q \, dS,$$

where

$$\mathbf{g} = \frac{\partial \theta}{\partial \mathbf{x}}$$

is the temperature gradient and $\delta\theta$ is an arbitrary variational field satisfying the essential boundary conditions.

In Abaqus/Explicit this nonlinear system is solved using the explicit central-difference time integration rule with a lumped capacitance matrix as described in *Fully Coupled Thermal-Stress Analysis*.

The thermal constitutive behavior for this example is now defined. Isotropic conductivity and a constant specific heat for the material are assumed. The heat conduction in the material is assumed to be governed by Fourier's law.

The internal thermal energy per unit mass is defined as

$$U = \int c \, d\theta = c\left(\theta - \theta^Z\right),$$

and its variation with respect to temperature is

$$\frac{\partial U}{\partial \theta} = c,$$

where *c* is the specific heat of the material and $\theta^Z$ is the value of absolute zero on the temperature scale being used.

Fourier's law for heat conduction is given as

$$\mathbf{f} = -\mathbf{k} \cdot \mathbf{g},$$

where $\mathbf{k}$ is the thermal conductivity matrix.

An approximation to the stability limit for the forward-difference operator in the thermal solution response is given by

$$\Delta t \approx L_{min}^2 \frac{\rho c_e}{2k_e},$$

where $L_{min}$ is the smallest element dimension in the mesh and the parameters $c_e$ and $k_e$ represent the material's effective specific heat and effective thermal conductivity, respectively. For a conservative estimate of the time increment size of the thermal solution response, we can specify

$$c_e = c,$$
$$k_e = \max(k_{ij}),$$

where $k_{ij}$ are components of the thermal conductivity matrix $\mathbf{k}$.

No state variables are needed for this material, so the allocation of space for them is not necessary.

A thermal user material definition can be used to read in the two constants for our simple case, namely the specific heat, *c*, and the coefficient of isotropic thermal conductivity, *k*, so that

$$\texttt{PROPS(1)} = k,$$

**PROPS(2)** = $c$.

```
      subroutine vumatht (
C Read only (unmodifiable) variables -
     *     nblock, nElem, nIntPt, nLayer, nSectPt,
     *     ntgrad, nstatev, nfieldv, nprops,
     *     cmname, stepTime, totalTime, dt,
     *     coordMp, density, props,
     *     tempOld, fieldOld, stateOld, enerThermOld,
     *     tempNew, tempgradNew, fieldNew,
C Write only (modifiable) variables -
     *     stateNew, fluxNew, enerThermNew, dEnerThDTemp, condEff )
C
      include 'vaba_param.inc'
C
      dimension nElem(nblock)
C
      dimension coordMp(nblock,*), density(nblock), props(nprops),
     *    tempOld(nblock), fieldOld(nblock, nfieldv),
     *    stateOld(nblock, nstatev), enerThermOld(nblock),
     *    tempNew(nblock), tempgradNew(nblock, ntgrad),
     *    fieldNew(nblock, nfieldv), stateNew(nblock, nstatev),
     *    fluxNew(nblock, ntgrad), enerThermNew(nblock),
     *    dEnerThDTemp(nblock,2), condEff(nblock)
C
      character*80 cmname
*
      parameter ( zero = 0.d0 )
      parameter ( tempZero = 0.d0)
* properties
* 1. conductivity
* 2. specific heat
*
      cond = props(1)
      specHeat = props(2)
*
      do km = 1, nblock
*
*        update heat flux vector
         do i = 1, ntgrad
            fluxNew(km, i) = -cond*tempgradNew(km,i)
         end do
         condEff(km) = cond
*
*        update internal thermal energy and its derivatives
         if (totalTime .eq. zero) then
            dEnerTh = specHeat*(tempNew(km) - tempZero)
         else
            dEnerTh = specHeat*(tempNew(km) - tempOld(km))
         end if
         enerThermNew(km) = enerThermOld(km) + dEnerTh
         dEnerThDTemp(km,1) = specHeat
         dEnerThDTemp(km,2) = zero
      end do
*
      return
      end
```

# VUMULLINS: User subroutine to define damage variable for the Mullins effect material model.

**Products:** Abaqus/Explicit

**References:**

- *Stress Softening in Rubberlike Materials*
- *Stress Softening in Elastomeric Foams*
- *MULLINS EFFECT
- *Mullins effect and permanent set*

## Overview

User subroutine VUMULLINS:

- can be used to define the damage variable for the Mullins effect material model (*Stress Softening in Rubberlike Materials*), including the use of the Mullins effect approach to model energy dissipation in elastomeric foams (*Stress Softening in Elastomeric Foams*);
- will be called for blocks of material calculation points for which the material definition contains a user-defined Mullins effect;
- can be used to define a failure criterion based on the strain energy density of the material;
- can use and update solution-dependent state variables;
- can use any field variables that are passed in; and
- should be used when you do not want to use the Ogden and Roxburgh form of the damage variable, $\eta$, that is used by Abaqus/Explicit.

## Material Point Deletion

Material points that satisfy a user-defined failure criterion can be deleted from the model (see *User-Defined Mechanical Material Behavior*). You must specify the state variable number controlling the element deletion flag when you allocate space for the solution-dependent state variables, as explained in *User-Defined Mechanical Material Behavior*. The deletion state variable can be set to a value of one or zero inside user subroutine VUMULLINS. A value of one indicates that the material point is active, and a value of zero indicates that Abaqus/Explicit should delete the material point from the model by setting the stresses to zero. The structure of the block of material points passed to user subroutine VUMULLINS remains unchanged during the analysis; deleted material points are not removed from the block. Abaqus/Explicit will "freeze" the values of the strain energy density passed to user subroutine VUMULLINS for all deleted material points; that is, the values remain constant after deletion is triggered. Once a material point has been flagged as deleted, it cannot be reactivated.

## User Subroutine Interface

```
      subroutine vumullins (
C Read only (unmodifiable) variables -
     1     nblock,
     2     jElem, kIntPt, kLayer, kSecPt,
     3     cmname,
     4     nstatev, nfieldv, nprops,
     5     props, tempOld, tempNew, fieldOld, fieldNew,
```

```
     6     stateOld, enerDamageOld,
     7     uMaxOld, uMaxNew, uDev,
C Write only (modifiable) variables -
     8     eta, detaDuDev,
     9     stateNew, enerDamageNew )
C
      include 'vaba_param.inc'
C
      dimension props(nprops),
     1  tempOld(nblock),
     2  fieldOld(nblock,nfieldv),
     3  stateOld(nblock,nstatev),
     4  tempNew(nblock),
     5  fieldNew(nblock,nfieldv),
     6  enerDamageOld(nblock),
     7  uMaxOld(nblock), uMaxNew(nblock),
     8  uDev(nblock),
     9  eta(nblock), detaDuDev(nblock),
     1  stateNew(nblock,nstatev),
     2  enerDamageNew(nblock)
C
      character*80 cmname
C

      do 100 km = 1,nblock
        user coding
  100 continue

      return
      end
```

## Variables to Be Defined

`eta(nblock)` The damage variable, $\eta$.

`detaDuDev(nblock)` The derivative of the damage variable with respect to the deviatoric elastic strain energy density of the undamaged material, $d\eta/d\tilde{U}_{dev}$, when the primary material behavior is hyperelastic. The derivative of the damage variable with respect to the total elastic strain energy density of the undamaged material, $d\eta/d\tilde{U}$, when the primary material behavior is hyperfoam. This quantity is needed for the evaluation of the effective moduli of the material, which enters the stable time increment calculation.

## Variables That Can Be Updated

`stateNew(nblock,nstatev)` State variables at each material point at the end of the increment. You define the size of this array by allocating space for it (see *About User Subroutines and Utilities* for more information).

`enerDamageNew(nblock)` The energy dissipation density at the end of the increment. This quantity can be defined either in total form or in an incremental manner using the old value of the damage dissipation `enerDamageOld` and the increment in damage dissipation. This quantity is used for output purposes only.

## Variables Passed in for Information

`nblock` Number of material points to be processed in this call to *VUMULLINS*.

| | |
|---|---|
| `jElem(nblock)` | Array of element numbers. |
| `kIntPt` | Integration point number. |
| `kLayer` | Layer number (for composite shells). |
| `kSecPt` | Section point number within the current layer. |
| `cmname` | User-specified material name, left justified. It is passed in as an uppercase character string. Some internal material models are given names starting with the "ABQ_" character string. To avoid conflict, you should not use "ABQ_" as the leading string for `cmname`. |
| `nstatev` | Number of user-defined state variables that are associated with this material type (you define the number as described in *Allocating Space for Solution-Dependent State Variables*). |
| `nfieldv` | Number of user-defined external field variables. |
| `nprops` | User-specified number of user-defined material properties. |
| `props(nprops)` | User-supplied material properties. |
| `tempOld(nblock)` | Temperatures at each material point at the beginning of the increment. |
| `tempNew(nblock)` | Temperatures at each material point at the end of the increment. |
| `fieldOld(nblock,nfieldv)` | Values of the user-defined field variables at each material point at the beginning of the increment. |
| `fieldNew(nblock,nfieldv)` | Values of the user-defined field variables at each material point at the end of the increment. |
| `stateOld(nblock,nstatev)` | State variables at each material point at the beginning of the increment. |
| `enerDamageOld(nblock)` | The value of energy dissipated at the beginning of the increment. |
| `uMaxOld(nblock)` | The value, at the beginning of the increment, of the maximum primary strain energy density over its entire deformation history. |
| `uMaxNew(nblock)` | The value, at the end of the increment, of the maximum primary strain energy density over its entire deformation history. |
| `uDev(nblock)` | The value, at the end of the increment, of the deviatoric primary strain energy density, $\widetilde{U}_{dev}$, when the primary material behavior is hyperelastic. The value, at the end of the increment, of the total primary strain energy density, $\widetilde{U}$, when the primary material behavior is hyperfoam. |

## Example: Hyperelasticity with Softening

As a simple example of the coding of user subroutine *VUMULLINS*, consider the following damage model based on the softening hyperelasticity approach proposed by Volokh (2007). The damage variable is assumed to vary with the deformation according to

$$\eta = \exp\left(\frac{-U_{dev}^{m}}{U_0}\right),$$

where $U_{dev}^{m}$ is the maximum value of $\widetilde{U}_{dev}$ at a material point during its deformation history, $\widetilde{U}_{dev}$ is the deviatoric part of the strain energy density of the undamaged hyperelastic behavior, and $U_0$ is a material parameter with units of strain energy density. The energy dissipation function for this model takes the form

$$\phi(\eta) = U_0\left(1 - \eta + \eta\ln(\eta)\right).$$

It can be shown that the functions $\eta$ and $\phi(\eta)$ satisfy the following condition:

$$\left(\tilde{U}_{dev} + \phi'(\eta)\right)\dot{\eta} = 0.$$

The code in user subroutine *VUMULLINS* must return the damage variable, $\eta$; the derivative of the damage variable with respect to the elastic strain energy density of the undamaged material, $d\eta/d\tilde{U}_{dev}$; and the energy dissipation $\phi(\eta)$. The user subroutine would be coded as follows:

```
      subroutine vumullins (
C Read only (unmodifiable) variables -
     1     nblock,
     2     jElem, kIntPt, kLayer, kSecPt,
     3     cmname,
     4     nstatev, nfieldv, nprops,
     5     props, tempOld, tempNew, fieldOld, fieldNew,
     6     stateOld, enerDamageOld,
     7     uMaxOld, uMaxNew, uDev,
C Write only (modifiable) variables -
     8     eta, detaDuDev,
     9     stateNew, enerDamageNew )
C
      include 'vaba_param.inc'
C
      dimension props(nprops),
     1  tempOld(nblock),
     2  fieldOld(nblock,nfieldv),
     3  stateOld(nblock,nstatev),
     4  tempNew(nblock),
     5  fieldNew(nblock,nfieldv),
     6  enerDamageOld(nblock),
     7  uMaxOld(nblock), uMaxNew(nblock),
     8  uDev(nblock),
     9  eta(nblock), detaDuDev(nblock),
     1  stateNew(nblock,nstatev),
     2  enerDamageNew(nblock)
C
      character*80 cmname
C
      parameter ( zero = 0.d0, one = 1.d0 )
C
      u0 = props(1)
      u0Inv = zero
      if ( u0 .gt. zero ) u0Inv = one / u0
C
      do k=1, nblock
         eta(k) = exp( -uMaxNew(k) * u0Inv )
         detaDUdev(k) = zero
         if ( uMaxNew(k) .gt. uMaxOld(k) )
     1        detaDUdev(k) = -u0Inv * eta(k)
         enerDamageNew(k) = u0*(one-eta(k)+eta(k)*log(eta(k)))
       end do
C
      return
      end
```

## Additional Reference

- Volokh, K. Y., "Hyperelasticity with Softening for Modeling Materials Failure," *Journal of the Mechanics and Physics of Solids*, vol. 55, pp. 2237–2264, 2007.

# VUSDFLD: User subroutine to redefine field variables at a material point.

**Products:** Abaqus/Explicit

**References:**

- *Obtaining Material Point Information in an Abaqus/Explicit Analysis*
- *Material Data Definition*
- **USER DEFINED FIELD*
- *Damage and failure of a laminated composite plate*
- *VUSDFLD*

## Overview

User subroutine *VUSDFLD*:

- allows the redefinition of field variables at a material point as functions of time or of any of the available material point quantities listed in *Available Output Variable Keys*;
- can be used to introduce solution-dependent material properties since such properties can be easily defined as functions of field variables;
- is called at all material points of elements for which the material definition includes user-defined field variables;
- can call utility routine VGETVRM to access material point data; and
- can use and update solution-dependent state variables.

## Explicit Solution Dependence

Since this routine provides access to material point quantities only at the start of the increment, the material properties for a given increment are not influenced by the results obtained during the increment. Hence, the accuracy of the results depends on the size of the time increment. However, in most situations this is not a concern for explicit dynamic analysis because the stable time increment is usually sufficiently small to ensure good accuracy.

## Defining Field Variables

Before user subroutine *VUSDFLD* is called, the values of the field variables at the material point are calculated by interpolation from the values defined at the nodes. Any changes to the field variables in the user subroutine are local to the material point: the nodal field variables retain the values defined as initial conditions or predefined field variables or the values defined in user subroutine *VUFIELD*. The values of the field variables defined in this routine are used to calculate values of material properties that are defined to depend on field variables and are passed into other user subroutines that are called at the material point, such as the following:

- *VHETVAL*
- *VUANISOHYPER_INV*
- *VUANISOHYPER_STRAIN*
- *VUHARD*
- *VUMAT*
- *VUMATHT*

- *VUTRS*
- *VUVISCOSITY*

Output of the user-defined field variables at the material points can be obtained with the element integration point output variable FV (see *Element Integration Point Variables*.)

## State Variables

Since the redefinition of field variables in *VUSDFLD* is local to the current increment (field variables are restored to the values interpolated from the nodal values at the start of each increment,) any history dependence required to update material properties by using this subroutine must be introduced with user-defined state variables.

The state variables can be updated in *VUSDFLD* and then passed into other user subroutines that can be called at this material point, such as those listed above. The number of such state variables can be specified as shown in the example at the end of this section (see *About User Subroutines and Utilities*).

## Accessing Material Point Data

The values of the material point quantities at the start of the increment can be accessed through the utility routine VGETVRM described in *Obtaining Material Point Information in an Abaqus/Explicit Analysis*. The values of the material point quantities are obtained by calling VGETVRM with the appropriate output variable keys.

## Component Ordering in Symmetric Tensors

For symmetric tensors such as the stress and strain tensors there are ndir+nshr components, and the component order is given as a natural permutation of the indices of the tensor. The direct components are first and then the indirect components, beginning with the 12-component. For example, a stress tensor contains ndir direct stress components and nshr shear stress components, which are returned as:

| Component | 2D Case | 3D Case |
|---|---|---|
| 1 | $\sigma_{11}$ | $\sigma_{11}$ |
| 2 | $\sigma_{22}$ | $\sigma_{22}$ |
| 3 | $\sigma_{33}$ | $\sigma_{33}$ |
| 4 | $\sigma_{12}$ | $\sigma_{12}$ |
| 5 | | $\sigma_{23}$ |
| 6 | | $\sigma_{31}$ |

The shear strain components in user subroutine *VUSDFLD* are stored as tensor components and not as engineering components; unlike user subroutine *USDFLD* in Abaqus/Standard, which uses engineering components.

## User Subroutine Interface

```
      subroutine vusdfld(
c Read only variables -
     1   nblock, nstatev, nfieldv, nprops, ndir, nshr,
     2   jElem, kIntPt, kLayer, kSecPt,
     3   stepTime, totalTime, dt, cmname,
     4   coordMp, direct, T, charLength, props,
     5   stateOld,
c Write only variables -
     6   stateNew, field )
c
```

```
      include 'vaba_param.inc'
c
      dimension jElem(nblock), coordMp(nblock,*),
     1          direct(nblock,3,3), T(nblock,3,3),
     2          charLength(nblock), props(nprops),
     3          stateOld(nblock,nstatev),
     4          stateNew(nblock,nstatev),
     5          field(nblock,nfieldv)
      character*80 cmname
c
c     Local arrays from vgetvrm are dimensioned to
c     maximum block size (maxblk)
c
      parameter( nrData=6 )
      character*3 cData(maxblk*nrData)
      dimension rData(maxblk*nrData), jData(maxblk*nrData)
c
      do 100 k = 1, nblock

         user coding to define field(nblock,nfieldv)
         and, if necessary, stateNew(nblock,nstatev)

  100 continue
c
      return
      end
```

## Variables to Be Defined

`field(nblock,nfieldv)` An array containing the field variables at the material points. These are passed in with the values interpolated from the nodes at the end of the current increment, as specified with initial condition definitions, predefined field variable definitions, or user subroutine *VUFIELD*. The updated values are used to calculate the values of material properties that are defined to depend on field variables and are passed into other user subroutines that are called at the material points.

## Variables That Can Be Updated

`stateNew(nblock,nstatev)` An array containing the solution-dependent state variables at the material points. In all cases `stateNew` can be updated in this subroutine, and the updated values are passed into other user subroutines that are called at the material points. The number of state variables associated with this material point is defined as described in *Allocating Space for Solution-Dependent State Variables*.

## Variables Passed in for Information

| | |
|---|---|
| `nblock` | Number of material points to be processed in this call to *VUSDFLD*. |
| `nstatev` | Number of user-defined state variables that are associated with this material type (you define this as described in *Allocating Space for Solution-Dependent State Variables*). |
| `nfieldv` | Number of user-defined external field variables. |
| `nprops` | User-specified number of user-defined material properties. |
| `ndir` | Number of direct components in a symmetric tensor. |

| | |
|---|---|
| `nshr` | Number of indirect components in a symmetric tensor. |
| `jElem` | Array of element numbers. |
| `kIntPt` | Integration point number. |
| `kLayer` | Layer number (for composite shells). |
| `kSecPt` | Section point number within the current layer. |
| `stepTime` | Value of time since the step began. |
| `totalTime` | Value of total time. The time at the beginning of the step is given by `totalTime-stepTime`. |
| `dt` | Time increment size. |
| `cmname` | User-specified material name, left justified. It is passed in as an uppercase character string. Some internal material models are given names starting with the "ABQ_" character string. To avoid conflict, you should not use "ABQ_" as the leading string for `cmname`. |
| `coordMp(nblock,*)` | Material point coordinates. It is the midplane material point for shell elements and the centroid for beam elements. |
| `direct(nblock,3,3)` | An array containing the direction cosines of the material directions in terms of the global basis directions. For material point k, `direct(k,1,1)`, `direct(k,2,1)`, `direct(k,3,1)` give the (1, 2, 3) components of the first material direction; `direct(k,1,2)`, `direct(k,2,2)`, `direct(k,3,2)` give the second material direction, etc. For shell and membrane elements, the first two directions are in the plane of the element and the third direction is the normal. This information is not available for beam elements. |
| `T(nblock,3,3)` | An array containing the direction cosines of the material orientation components relative to the element basis directions. For material point k, this is the orientation that defines the material directions (`direct`) in terms of the element basis directions. For continuum elements `T` and `direct` are identical. For shell and membrane elements $\texttt{T(k,1,1)} = \cos\theta$, $\texttt{T(k,1,2)} = -\sin\theta$, $\texttt{T(k,2,1)} = \sin\theta$, $\texttt{T(k,2,2)} = \cos\theta$, $\texttt{T(k,3,3)} = 1.0$ and all other components are zero, where $\theta$ is the counterclockwise rotation around the normal vector that defines the orientation. If no orientation is used, `T` is an identity matrix. Orientation is not available for beam elements. |
| `charLength(nblock)` | Characteristic element length, which is either the default value based on the geometric mean or the user-defined characteristic element length defined in user subroutine `VUCHARLENGTH`. The default value is a typical length of a line across an element for a first-order element; it is half of the same typical length for a second-order element. For beams and trusses the default value is a characteristic length along the element axis. For membranes and shells it is a characteristic length in the reference surface. For axisymmetric elements it is a characteristic length in the $r$–$z$ plane only. For cohesive elements it is equal to the constitutive thickness. |
| `props(nprops)` | User-supplied material properties. |
| `stateOld (nblock, nstatev)` | State variables at each material point at the beginning of the increment. |

## Example: Damaged Elasticity Model

Included below is an example of user subroutine *VUSDFLD*. In this example a truss element is loaded in tension. A damaged elasticity model is introduced: the modulus decreases as a function of the maximum tensile strain that occurred during the loading history. The maximum tensile strain is stored as a solution-dependent state variable (see *Defining Solution-Dependent Field Variables*).

## Input file

```
*HEADING
 Damaged elasticity model with user subroutine vusdfld
*ELEMENT, TYPE=T2D2, ELSET=ONE
1, 1, 2
*NODE, NSET=NALL
   1,  0., 0.
   2, 10., 0.
*SOLID SECTION, ELSET=ONE, MATERIAL=ELASTIC
1.
*MATERIAL, NAME=ELASTIC
*ELASTIC, DEPENDENCIES=1
** Table of modulus values decreasing as a function
** of field variable 1.
2000., 0.3, 0., 0.00
1500., 0.3, 0., 0.01
1200., 0.3, 0., 0.02
1000., 0.3, 0., 0.04
*DENSITY
1.0e-6
*USER DEFINED FIELD
*DEPVAR
1
1,EPSMAX,"Maximum strain value"
*BOUNDARY
1, 1, 2
2, 2
*AMPLITUDE, NAME=LOAD1
0.0, 0.0, 1.0, 1.0
*AMPLITUDE, NAME=LOAD2
0.0, 0.0, 2.0, 1.0
*AMPLITUDE, NAME=UNLOAD
0.0, 1.0, 1.0, 0.0
*STEP, NLGEOM=NO
*DYNAMIC, EXPLICIT
 , 1.0
*CLOAD, AMPLITUDE=LOAD1
2, 1, 20.
*OUTPUT, FIELD, VARIABLE=PRESELECT
*OUTPUT, HISTORY, VARIABLE=PRESELECT
*ELEMENT OUTPUT, ELSET=ONE
S, E, SDV
*NODE OUTPUT, NSET=NALL
RF, CF, U
*END STEP
*STEP, NLGEOM=NO
*DYNAMIC, EXPLICIT
, 1.0
*CLOAD, AMPLITUDE=UNLOAD
2, 1, 20.
*END STEP
*STEP, NLGEOM=NO
*DYNAMIC, EXPLICIT
 , 2.0
*CLOAD, AMPLITUDE=LOAD2
2, 1, 40.
*END STEP
```

## User subroutine

```
      subroutine vusdfld(
c Read only -
     *   nblock, nstatev, nfieldv, nprops, ndir, nshr,
     *   jElem, kIntPt, kLayer, kSecPt,
     *   stepTime, totalTime, dt, cmname,
     *   coordMp, direct, T, charLength, props,
     *   stateOld,
c Write only -
     *   stateNew, field )
c
      include 'vaba_param.inc'
c
      dimension jElem(nblock), coordMp(nblock,*),
     *          direct(nblock,3,3), T(nblock,3,3),
     *          charLength(nblock), props(nprops),
     *          stateOld(nblock,nstatev),
     *          stateNew(nblock,nstatev),
     *          field(nblock,nfieldv)
      character*80 cmname
c
c     Local arrays from vgetvrm are dimensioned to
c     maximum block size (maxblk)
c
      parameter( nrData=6 )
      character*3 cData(maxblk*nrData)
      dimension rData(maxblk*nrData), jData(maxblk*nrData)
c
      jStatus = 1
      call vgetvrm( 'LE', rData, jData, cData, jStatus )
c
      if( jStatus .ne. 0 ) then
         call xplb_abqerr(-2,'Utility routine VGETVRM '//
     *      'failed to get variable.',0,zero,' ')
         call xplb_exit
      end if
c
      call setField( nblock, nstatev, nfieldv, nrData,
     *   rData, stateOld, stateNew, field)
c
      return
      end
      subroutine setField( nblock, nstatev, nfieldv, nrData,
     *   strain, stateOld, stateNew, field )
c
      include 'vaba_param.inc'
c
      dimension stateOld(nblock,nstatev),
     *   stateNew(nblock,nstatev),
     *   field(nblock,nfieldv), strain(nblock,nrData)
c
      do k = 1, nblock
c
c     Absolute value of current strain:
         eps = abs( strain(k,1) )
c
c     Maximum value of strain up to this point in time:
         epsmax = stateOld(k,1)
c
```

```
c     Use the maximum strain as a field variable
         field(k,1) = max( eps, epsmax )
c
c     Store the maximum strain as a solution dependent state
         stateNew(k,1) = field(k,1)
c
      end do
c
      return

      end
```

# VUSUPERELASHARDMOD: User subroutine to modify the material constants of the superelasticity model as a function of the plastic strain.

**Products:** Abaqus/Explicit

**References:**

- *Plasticity Model for Superelastic Materials*
- **SUPERELASTIC HARDENING*
- **SUPERELASTIC HARDENING MODIFICATIONS*

## Overview

User subroutine *VUSUPERELASHARDMOD*:

- is called at all material calculation points of elements for which the material definition includes user-defined hardening modifications; and
- can use field variables or state variables.

## User Subroutine Interface

```
      subroutine vusuperelashardmod(
C Read only -
     *     nblock,
     *     jElem, kIntPt, kLayer, kSecPt,
     *     stepTime, totalTime, dt, cmname,
     *     nstatev, nfieldv, nprops,
     *     props, tempOld, tempNew, fieldOld, fieldNew,
     *     stateOld,
     *     CAS, CSA, eqps, RefT, EmodARef,
     *     EnuARef, EmodSRef, EnuSRef, SsASRef, SfASRef, SsSARef,
     *     SfSARef,
C     Write only (modifiable) variables -
     *     EmodA, EnuA, EmodS, EnuS, SsAS, SfAS, SsSA, SfSA,
     *     stateNew )
C
      include 'vaba_param.inc'
C
      dimension props(nprops), tempOld(nblock), tempNew(nblock),
     1   fieldOld(nblock,nfieldv), fieldNew(nblock,nfieldv),
     2   stateOld(nblock,nstatev), CAS(nblock), CSA(nblock),eqps(nblock),
     3   RefT(nblock), EmodARef(nblock),EnuARef(nblock),EmodSRef(nblock),
     4   EnuSRef(nblock), SsASRef(nblock),SfASRef(nblock),SsSARef(nblock),

     5   SfSARef(nblock),EmodA(nblock),EnuA(nblock),EmodS(nblock),
     4   EnuS(nblock),SsAS(nblock),SfAS(nblock),SsSA(nblock),
     5   SfSA(nblock),stateNew(nblock,nstatev),jElem(nblock)
C
      character*80 cmname
C
      do 100 km = 1,nblock
        user coding
  100 continue
```

```
C
      return
      end
```

## Variables to Be Defined

| | |
|---|---|
| `EmodA(nblock)` | $E_A$, austenite's Young's modulus. |
| `EnuA(nblock)` | $\nu_A$, austenite's Poisson's ratio. |
| `EmodS(nblock)` | $E_M$, martensite's Young's modulus. |
| `EnuS(nblock)` | $\nu_M$, martensite's Poisson's ratio. |
| `SsAS(nblock)` | $\sigma_{tL}^{S}$, stress at which the transformation begins during loading in tension. |
| `SfAS(nblock)` | $\sigma_{tL}^{E}$, stress at which the transformation ends during loading in tension. |
| `SsSA(nblock)` | $\sigma_{tU}^{S}$, stress at which the reverse transformation begins during unloading in tension. |
| `SfSA(nblock)` | $\sigma_{tU}^{E}$, stress at which the reverse transformation begins during unloading in tension. |
| `stateNew(nblock,nstatev)` | Array containing the state variables at the material points at the end of the increment. The allocation of this array is described in *About User Subroutines and Utilities*. |

## Variables Passed in for Information

| | |
|---|---|
| `nblock` | Number of material points. |
| `jElem(nblock)` | Array of element numbers. |
| `kIntPt` | Integration point number. |
| `kLayer` | Layer number (for composite shells). |
| `kSecPt` | Section point number within the current layer. |
| `stepTime` | Value of time since the step began. |
| `totalTime` | Value of total time. The time at the beginning of the step is given by `totalTime-stepTime`. |
| `dt` | Time increment size. |
| `cmname` | Material name, left-justified. It is passed in as an uppercase character string. Some internal material models are given names starting with the "ABQ_" character string. To avoid conflict, "ABQ_" should not be used as the leading string for `cmname`. |
| `nstatev` | Number of user-defined state variables that are associated with this material type (see *About User Subroutines and Utilities*). |
| `nfieldv` | Number of user-defined external field variables. |
| `nprops` | User-specified number of user-defined material properties. |
| `tempOld(nblock)` | Temperatures at the material points at the beginning of the increment. |
| `tempNew(nblock)` | Temperatures at the material points at the end of the increment. |
| `fieldOld(nblock,nfieldv)` | Values of the user-defined field variables at the material points at the beginning of the increment. |

`fieldNew(nblock,nfieldv)` Values of the user-defined field variables at the material points at the end of the increment.

`stateOld(nblock,nstatev)` State variables at the material points at the beginning of the increment.

`CAS(nblock)`
$\left(\frac{\delta\sigma}{\delta T}\right)_L$, slope of the stress versus temperature curve for loading.

`CSA(nblock)`
$\left(\frac{\delta\sigma}{\delta T}\right)_U$, slope of the stress versus temperature curve for unloading.

`eqps(nblock)` Equivalent plastic strain at the material points.

`RefT(nblock)` Reference temperature.

`EmodARef(nblock)` $E_A$, reference value of austenite's Young's modulus.

`EnuARef(nblock)` $\nu_A$, reference value of austenite's Poisson's ratio.

`EmodSRef(nblock)` $E_M$, reference value of martensite's Young's modulus.

`EnuSRef(nblock)` $\nu_M$, reference value of martensite's Poisson's ratio.

`SsASRef(nblock)` $\sigma_{tL}^{S}$, reference value of stress at which the transformation begins during loading in tension.

`SfASRef(nblock)` $\sigma_{tL}^{E}$, reference value of stress at which the transformation ends during loading in tension.

`SsSARef(nblock)` $\sigma_{tU}^{S}$, reference value of stress at which the reverse transformation begins during unloading in tension.

`SfSARef(nblock)` $\sigma_{tU}^{E}$, reference value of stress at which the reverse transformation begins during unloading in tension.

# VUTEMP: User subroutine to prescribe temperatures at the nodes of a model.

**Products:** Abaqus/Explicit

**References:**

- *Predefined Fields*
- **TEMPERATURE*

## Overview

User subroutine *VUTEMP*:

- allows you to prescribe temperatures at the nodes of a model;
- can be called for blocks of nodes for which the temperature values are defined in the user subroutine; and
- ignores any temperature values specified directly outside the user subroutine.

## User Subroutine Interface

```
      SUBROUTINE VUTEMP(TEMP, NBLOCK, NCOMP, NFIELD,
     1                  KSTEP, JFLAGS, JNODEUID, TIME, FIELD,
     2                  COORDS, U, V, A)
C
      INCLUDE 'VABA_PARAM.INC'

C     indices for the time array TIME
      PARAMETER( i_utmp_Current   = 1,
     *           i_utmp_Increment = 2,
     *           i_utmp_Period    = 3,
     *           i_utmp_Total     = 4,
     *           n_utmp_Time      = 4 )

C     indices for the coordinate array COORDS
      PARAMETER( i_utmp_CoordX = 1,
     *           i_utmp_CoordY = 2,
     *           i_utmp_CoordZ = 3 )

C     indices for the displacement array U
      PARAMETER( i_utmp_SpaDisplX = 1,
     *           i_utmp_SpaDisplY = 2,
     *           i_utmp_SpaDisplZ = 3,
     *           i_utmp_RotDisplX = 4,
     *           i_utmp_RotDisplY = 5,
     *           i_utmp_RotDisplZ = 6,
     *           i_utmp_AcoPress  = 7,
     *           n_utmp_SolU      = 7 )

C     indices for the velocity array V
      PARAMETER( i_utmp_SpaVelX   = 1,
     *           i_utmp_SpaVelY   = 2,
     *           i_utmp_SpaVelZ   = 3,
     *           i_utmp_RotVelX   = 4,
     *           i_utmp_RotVelY   = 5,
     *           i_utmp_RotVelZ   = 6,
```

```
     *           i_utmp_DAcoPress = 7,
     *           n_utmp_SolV      = 7 )

C     indices for the acceleration array A
      PARAMETER( i_utmp_SpaAccelX  = 1,
     *           i_utmp_SpaAccelY  = 2,
     *           i_utmp_SpaAccelZ  = 3,
     *           i_utmp_RotAccelX  = 4,
     *           i_utmp_RotAccelY  = 5,
     *           i_utmp_RotAccelZ  = 6,
     *           i_utmp_DDAcoPress = 7,
     *           n_utmp_SolA       = 7 )

C     indices for JFLAGS
      PARAMETER( i_utmp_kInc   = 1,
     *           i_utmp_kPass  = 2,
     *           n_utmp_Flags  = 2 )
C
      DIMENSION TEMP(NBLOCK,NCOMP)
      DIMENSION FIELD(NBLOCK,NCOMP,NFIELD)
      DIMENSION JFLAGS(n_utmp_Flags), JNODEUID(NBLOCK), TIME(n_utmp_Time),

     *          COORDS(3,NBLOCK)
      DIMENSION U(n_utmp_SolU,NBLOCK), V(n_utmp_SolV,NBLOCK),
A(n_utmp_SolA,NBLOCK)
C

      user coding to define  TEMP

      RETURN
      END
```

## Variables to Be Defined

`TEMP(NBLOCK,NCOMP)` Array of temperature values at a collective number of nodes `NBLOCK` (see `NBLOCK` below).

For temperatures applied to nodes that are not part of pipe, beam, or shell elements, only one value of temperature is required (`NCOMP=1`), and the user subroutine is invoked in a single pass. For nodes that are part of pipe, beam, or shell elements, *VUTEMP* is invoked in two passes per increment for these elements, and the number of values to be returned depends on the mode of temperature input selected for the beam or shell section. The following cases are possible:

- Temperatures are given as values at the points on the shell or beam section. For a beam section, the number of values required is determined by the particular section type specified, as described in *Beam Cross-Section Library*. For a shell section, temperatures and field variables are given as values at *n* equally spaced points through each layer of a shell section. In the first pass `NCOMP` is passed in with the value of 1 to define the temperature values at the first point. The second pass is used to define temperatures at the remaining points.
- Temperatures for the shell or beam section are given as values at the origin of the cross-section together with gradients along the cross-section. The number of gradient values required is 2 for three-dimensional beams, 1 for two-dimensional beams, and 1 for shells. In the first pass `NCOMP` is passed in with the value of 1 to define the field variable values at the origin of the cross-section. The gradients are defined in the second pass.

Because temperatures can also be defined directly, it is important to understand the hierarchy used in situations with conflicting information (see *Predefined Fields*).

When the array TEMP is passed into user subroutine *VUTEMP*, it will contain the temperatures values from the previous increment.

## Variables Passed in for Information

| | |
|---|---|
| NBLOCK | User-specified number of nodes to be processed as a block in this call to *VUTEMP*. The value is equal to the total number of nodes given in a node set. |
| NCOMP | Maximum number of section values to be defined for any node in the model in the current pass. NCOMP is passed in with the value of 1 in the first pass to user subroutine *VUTEMP*. |
| NFIELD | Number of field variables passed in the array FIELD (see below) . |
| KSTEP | Current step number. |
| JFLAGS(i_utmp_kInc) | Increment number for step KSTEP. |
| JFLAGS(i_utmp_kPass) | This flag is equal to 1 for the first pass to user subroutine *VUTEMP* and is equal to 2 for the second pass. |
| JNODEUID(NBLOCK) | Array for user-defined node numbers. This array is dimensioned based on the size of NBLOCK, and the contained node numbers are identical to those defined in the input file. |
| TIME(n_utmp_Time) | Array for information of analysis time. You can retrieve any time information from this array by using the parameters given above. TIME(i_utmp_Current) stores the current analysis time, TIME(i_utmp_Increment) gives the time increment at this instance, TIME(i_ufld_Period) is the time period of the current step, and TIME(i_ufld_Total) is the total analysis time up to this point. |
| FIELD(NBLOCK,NCOMP,NFIELD) | Array containing the field variables in the analysis. |
| COORDS(3,NBLOCK) | Coordinates for nodes in the array JNODEUID. This array stores the current coordinates of nodes in which the order of coordinates stored corresponds to the order of nodes listed in the array JNODEUID. The coordinates can be retrieved by using the parameters given above. |
| U(N_UTMP_SOLU,NBLOCK), V(N_UTMP_SOLV,NBLOCK), and A(N_UTMP_SOLA,NBLOCK) | Arrays containing solution variables of displacements, rotations, and pressures and their corresponding temporal derivatives. The order in which these solutions are stored follows the order defined in the array JNODEUID. For a specific node, its solution variables can be retrieved by using the parameter indices given above. Depending on the degrees of freedom, some solution variables are not valid for a given node. The displacement values correspond to the current increment. However, the acceleration is from a configuration that is one increment behind, and the velocity is such that it is consistent with the displacement increment and the time increment between the two successive configurations. |

# VUTRS: User subroutine to define a reduced time shift function for a viscoelastic material.

**Products:** Abaqus/Explicit

**References:**

- *Time Domain Viscoelasticity*
- **TRS*
- **VISCOELASTIC*
- *Transient thermal loading of a viscoelastic slab*

## Overview

User subroutine *VUTRS*:

- can be used to define a temperature-time shift for a time domain viscoelastic analysis;
- will be called for all material points of elements for which a user-defined shift function is specified to define the time-temperature correspondence as part of the viscoelastic material definition;
- can use and update solution-dependent state variables; and
- can have incoming field variables redefined by user subroutine *VUSDFLD* to redefine any field variables before they are passed in.

## User Subroutine Interface

```
      subroutine vutrs(
c Read only variables -
     1   nblock, nstatev, nfieldv, nprops,
     2   stepTime, totalTime, dt,
     3   cmname, props, density, coordMp,
     4   tempOld, fieldOld, stateOld,
     5   tempNew, fieldNew,
c Write only variables -
     6   shift, stateNew )
c
      include 'vaba_param.inc'
c
      dimension props(nprops), density(nblock), coordMp(nblock,*),
     1   tempOld(nblock),tempNew(nblock),
     2   fieldOld(nblock,nfieldv), fieldNew(nblock,nfieldv),
     3   stateOld(nblock,nstatev), stateNew(nblock,nstatev),
     4   shift(nblock,2)
c
      character*80 cmname
c
      do 100 k=1, nblock
         user coding to define shift(k,1) and shift(k,2)
  100 continue
c
      return
      end
```

## Variables to Be Defined

`shift(nblock,2)` Array that defines the shift function, $A$ ($A > 0$), at the material points. For material point k, `shift(k,1)` defines the shift function at the beginning of the increment, and `shift(k,2)` defines the shift function at the end of the increment. Abaqus/Explicit will apply an averaging scheme to these values that assumes that the natural logarithm of the shift function can be approximated by a linear function over the increment.

If either `shift(k,1)` or `shift(k,2)` is less than or equal to zero, no time shift will be applied.

## Variables That Can Be Updated

`stateNew(nblock,nstatev)` Array containing the solution-dependent state variables at the material points. This array will be passed in containing the values of these variables at the start of the increment unless they are updated in user subroutine *VUSDFLD*, in which case the updated values are passed in. If any of the solution-dependent state variables are being used in conjunction with the viscoelastic behavior, they must be updated in this subroutine to their values at the end of the increment.

## Variables Passed in for Information

`nblock` Number of material points to be processed in this call to `VUTRS`.

`nstatev` Number of user-defined state variables that are associated with this material type (see *Allocating Space for Solution-Dependent State Variables*).

`nfieldv` Number of user-defined external field variables.

`nprops` User-specified number of user-defined material properties.

`stepTime` Value of time since the step began.

`totalTime` Value of total time. The time at the beginning of the step is given by `totalTime-stepTime`.

`dt` Time increment size.

`cmname` Material name, left justified. It is passed in as an uppercase character string. Some internal material models are given names starting with the "ABQ_" character string. To avoid conflict, "ABQ_" should not be used as the leading string for `cmname`.

`props(nprops)` User-supplied material properties.

`density(nblock)` Current density at the material points in the midstep configuration. This value might be inaccurate in problems where the volumetric strain increment is very small. If an accurate value of the density is required in such cases, the analysis should be run in double precision. Mass scaling does not affect this value of the density.

`coordMp(nblock,*)` Material point coordinates. It is the midplane material point for shell elements and the centroid for beam elements.

`tempOld(nblock)` Temperatures at each material point at the beginning of the increment.

`fieldOld(nblock,nfieldv)` Values of the user-defined field variables at each material point at the beginning of the increment.

`stateOld(nblock,nstatev)` State variables at each material point at the beginning of the increment.

`tempNew(nblock)` Temperatures at each material point at the end of the increment.

`fieldNew(nblock,nfieldv)` Values of the user-defined field variables at each material point at the end of the increment.

# VUTRSNETWORK: User subroutine to define a reduced time shift function for models defined within the parallel rheological framework.

**Products:** Abaqus/Explicit

**References:**

- *Parallel Rheological Framework*
- *Nonlinear large-strain viscoelasticity with hyperelasticity*
- *VUTRSNETWORK*
- **VISCOELASTIC*

## Overview

User subroutine *VUTRSNETWORK*:

- can be used to define a temperature-time shift for a nonlinear viscoelastic network for models defined using the parallel rheological framework;
- will be called for all material points of elements for which a user-defined shift function is specified to define the time-temperature correspondence as part of the viscoelastic material definition;
- can use and update solution-dependent state variables; and
- can be used in conjunction with user subroutine *VUSDFLD* to redefine any field variables before they are passed in.

## User Subroutine Interface

```
      subroutine vutrsnetwork (
c Read only variables -
     *   nblock, networkID, nstatev, nfieldv, nprops,
     *   stepTime, totalTime, dt,
     *   jElem, kIntPt, kLayer, kSecPt,
     *   cmname, props, density, coordMp,
     *   tempOld, fieldOld, stateOld,
     *   tempNew, fieldNew,
     *   nIarray, i_array, nRarray, r_array,
c Write only variables -
     *   shift, stateNew )
c
      include 'vaba_param.inc'
c
c     indices for shift I/O
      parameter( io_trs_shift_begin = 1,
     *           io_trs_shift_end   = 2 )
c
      dimension props(nprops), density(nblock), coordMp(nblock, *),
     *          jElem(nblock),
     *          i_array(nblock,nIarray),
     *          r_array(nblock,nRarray),
     *          tempOld(nblock), tempNew(nblock),
     *          fieldOld(nblock, nfieldv), fieldNew(nblock, nfieldv),
     *          stateOld(nblock, nstatev), stateNew(nblock, nstatev),
     *          shift(nblock, 2)
c
```

```
      character*80 cmname
c
      do 100 k=1, nblock
         user coding to define shift(k,io_trs_shift_begin)
         and shift(k,io_trs_shift_end)
  100 continue
c
      return
      end
```

## Variables to Be Defined

`shift(nblock,2)` Array that defines the shift function, $A$ ($A > 0$), at the material points. For material point k, `shift(k,io_trs_shift_begin)` defines the shift function at the beginning of the increment, and `shift(k,io_trs_shift_end)` defines the shift function at the end of the increment. Abaqus/Explicit will apply an averaging scheme to these values that assumes that the natural logarithm of the shift function can be approximated by a linear function over the increment.

If either `shift(k,io_trs_shift_begin)` or `shift(k,io_trs_shift_end)` is less than or equal to zero, no time shift will be applied.

## Variables That Can Be Updated

`stateNew(nblock,nstatev)` Array containing the solution-dependent state variables at the material points. This array will be passed in containing the values of these variables at the start of the increment unless they are updated in user subroutine *VUSDFLD*, in which case the updated values are passed in. If any of the solution-dependent state variables are being used in conjunction with the viscoelastic behavior, they must be updated in this subroutine to their values at the end of the increment.

## Variables Passed in for Information

`nblock` Number of material points to be processed in this call to *VUTRSNETWORK*.

`networkID` Network identification number, which identifies the network for which the shift function is defined.

`nstatev` Number of user-defined state variables that are associated with this material type (see *Allocating Space for Solution-Dependent State Variables*).

`nfieldv` Number of field variables.

`nprops` User-specified number of user-defined material properties.

`stepTime` Value of time since the step began.

`totalTime` Value of total time. The time at the beginning of the step is given by `totalTime-stepTime`.

`dt` Time increment size.

`jElem (nblock)` Array of element numbers.

`kIntPt` Integration point number.

`kLayer` Layer number (for composite shells).

`kSecPt` Section point number within the current layer.

| | |
|---|---|
| `cmname` | Material name, left justified. It is passed in as an uppercase character string. Some internal material models are given names starting with the "ABQ_" character string. To avoid conflict, "ABQ_" should not be used as the leading string for `cmname`. |
| `props(nprops)` | User-supplied material properties. |
| `density(nblock)` | Current density at the material points in the midstep configuration. This value might be inaccurate in problems where the volumetric strain increment is very small. If an accurate value of the density is required in such cases, the analysis should be run in double precision. Mass scaling does not affect this value of the density. |
| `coordMp(nblock,*)` | Material point coordinates. It is the midplane material point for shell elements and the centroid for beam elements. |
| `tempOld(nblock)` | Temperatures at each material point at the beginning of the increment. |
| `fieldOld(nblock,nfieldv)` | Values of the user-defined field variables at each material point at the beginning of the increment. |
| `stateOld(nblock,nstatev)` | State variables at each material point at the beginning of the increment. |
| `tempNew(nblock)` | Temperatures at each material point at the end of the increment. |
| `fieldNew(nblock,nfieldv)` | Values of the user-defined field variables at each material point at the end of the increment. |
| `nIarray` | Size of array `i_array`. |
| `i_array(nblock,nIarray)` | Array containing integer arguments. It is not currently used. |
| `nRarray` | Size of array `r_array`. |
| `r_array(nblock,nRarray)` | Array containing real arguments. It is not currently used. |

## Example: Williams-Landel-Ferry Shift Function

As an example of the coding of user subroutine *VUTRSNETWORK*, consider the William-Landel-Ferry model to define the shift function. In this case the shift function is expressed as (see *Thermorheologically Simple Temperature Effects*)

$$\log_{10}(A) = -\frac{C_1\,(\theta - \theta_0)}{C_2 + (\theta - \theta_0)},$$

where

$\theta$

is the temperature,

$\theta_0$

is the reference temperature, and

$C_1$ *and* $C_2$

are constants.

The user subroutine would be coded as follows:

```
      subroutine vutrsnetwork (
c Read only variables -
     *   nblock, networkID, nstatev, nfieldv, nprops,
```

```
     *   stepTime, totalTime, dt,
     *   jElem, kIntPt, kLayer, kSecPt,
     *   cmname, props, density, coordMp,
     *   tempOld, fieldOld, stateOld,
     *   tempNew, fieldNew,
     *   nIarray, i_array, nRarray, r_array,
c Write only variables -
     *   shift, stateNew )
c
      include 'vaba_param.inc'
c
c     indices for shift I/O
      parameter( io_trs_shift_begin = 1,
     *           io_trs_shift_end   = 2 )
c
      dimension props(nprops), density(nblock), coordMp(nblock, *),
     *          jElem(nblock),
     *          i_array(nblock,nIarray),
     *          r_array(nblock,nRarray),
     *          tempOld(nblock), tempNew(nblock),
     *          fieldOld(nblock, nfieldv), fieldNew(nblock, nfieldv),
     *          stateOld(nblock, nstatev), stateNew(nblock, nstatev),
     *          shift(nblock, 2)
c
      character*80 cmname
c
      parameter ( rln10 = 2.302585093d0 )
c
      theta0 = props(1)
      C1 = props(2)
      C2 = props(3)
      do 100 k=1, nblock
            dTemp1 = tempOld(k) - theta0
            shift(k,io_trs_shift_begin) =
     *           exp(-C1*rln10*dTemp1/(C2+dTemp1))
            dTemp2 = tempNew(k) - theta0
            shift(k,io_trs_shift_end) =
     *           exp(-C1*rln10*dTemp2/(C2+dTemp2))
  100 continue
c
      return
      end
```

# VUVISCOSITY: User subroutine to define the shear viscosity for equation of state models.

**Products:** Abaqus/Explicit

**References:**

- *Equation of State*
- **EOS*
- **VISCOSITY*
- *VUVISCOSITY*

## Overview

User subroutine *VUVISCOSITY*:

- is called at all material points of elements with an equation of state for which the material definition includes user-defined viscous shear behavior;
- can be used to define a material's isotropic viscous behavior;
- can use and update solution-dependent state variables; and
- can be used in conjunction with user subroutine *VUSDFLD* to redefine any field variables before they are passed in.

## User Subroutine Interface

```
      subroutine vuviscosity(
C Read only -
     *     nblock,
     *     jElem, kIntPt, kLayer, kSecPt,
     *     stepTime, totalTime, dt, cmname,
     *     nstatev, nfieldv, nprops,
     *     props, tempOld, tempNew, fieldOld, fieldNew,
     *     stateOld,
     *     shrRate,
C Write only -
     *     viscosity,
     *     stateNew )
C
      include 'vaba_param.inc'
C
      dimension props(nprops), tempOld(nblock), tempNew(nblock),
     1   fieldOld(nblock,nfieldv), fieldNew(nblock,nfieldv),
     2   stateOld(nblock,nstatev), eqps(nblock), eqpsRate(nblock),
     3   viscosity(nblock),
     4   stateNew(nblock,nstatev), jElem(nblock)
C
      character*80 cmname
C
      do 100 km = 1,nblock
        user coding
  100 continue
C
```

```
      return
      end
```

## Variables to Be Defined

| | |
|---|---|
| `viscosity(nblock)` | Array containing the viscosity at the material points. (Units of $FL^{-2}T$.) |
| `stateNew(nblock,nstatev)` | Array containing the state variables at the material points at the end of the increment. The allocation of this array is described in *Solution-Dependent State Variables*. |

## Variables Passed in for Information

| | |
|---|---|
| `nblock` | Number of material points to be processed in this call to `VUVISCOSITY`. |
| `jElem(nblock)` | Array of element numbers. |
| `kIntPt` | Integration point number. |
| `kLayer` | Layer number (for composite shells). |
| `kSecPt` | Section point number within the current layer. |
| `stepTime` | Value of time since the step began. |
| `totalTime` | Value of total time. The time at the beginning of the step is given by `totalTime-stepTime`. |
| `dt` | Time increment size. |
| `cmname` | Material name, left justified. It is passed in as an uppercase character string. Some internal material models are given names starting with the "ABQ_" character string. To avoid conflict, "ABQ_" should not be used as the leading string for `cmname`. |
| `nstatev` | Number of user-defined state variables that are associated with this material type (see *Allocating Space for Solution-Dependent State Variables*). |
| `nfieldv` | Number of user-defined external field variables. |
| `nprops` | User-specified number of user-defined material properties. |
| `tempOld(nblock)` | Temperatures at the material points at the beginning of the increment. |
| `tempNew(nblock)` | Temperatures at the material points at the end of the increment. |
| `fieldOld(nblock,nfieldv)` | Values of the user-defined field variables at the material points at the beginning of the increment. |
| `fieldNew(nblock,nfieldv)` | Values of the user-defined field variables at the material points at the end of the increment. |
| `stateOld(nblock,nstatev)` | State variables at the material points at the beginning of the increment. |
| `shrRate(nblock)` | Equivalent shear strain rate, $\dot{\gamma}$, at the material points. |

## Example: Cross Viscosity Model

As a simple example of the coding of subroutine `VUVISCOSITY`, consider the Cross viscosity model. The Cross model is commonly used when it is necessary to describe the low shear rate behavior of the viscosity. The viscosity is expressed as

$$\eta = \frac{\eta_0}{1 + (\lambda\dot{\gamma})^{1-n}},$$

where $\eta_0$ is the Newtonian viscosity, $n$ is the flow index in the power law regime, and $\lambda$ is a constant with units of time, such that $1/\lambda$ corresponds to the critical shear rate at which the fluid changes from Newtonian to power law behavior.

The subroutine would be coded as follows:

```
      subroutine vuviscosity (
C Read only -
     *     nblock,
     *     jElem, kIntPt, kLayer, kSecPt,
     *     stepTime, totalTime, dt, cmname,
     *     nstatev, nfieldv, nprops,
     *     props, tempOld, tempNew, fieldOld, fieldNew,
     *     stateOld,
     *     shrRate,
C Write only -
     *     viscosity,
     *     stateNew )
C
      include 'vaba_param.inc'
C
      dimension props(nprops),
     *  tempOld(nblock),
     *  fieldOld(nblock,nfieldv),
     *  stateOld(nblock,nstatev),
     *  shrRate(nblock),
     *  tempNew(nblock),
     *  fieldNew(nblock,nfieldv),
     *  viscosity(nblock),
     *  stateNew(nblock,nstatev)
C
      character*80 cmname
C
     parameter ( one = 1.d0 )
C
C     Cross viscosity
C
      eta0    = props(1)
      rlambda = props(2)
      rn      = props(3)
C
      do k = 1, nblock
         viscosity(k) = eta0/(one+(rlambda*shrRate(k))**(one-rn))
      end do
C
      return
      end
```

# VWAVE: User subroutine to define wave kinematics for an Abaqus/Aqua analysis.

**Products:** Abaqus/Explicit Abaqus/Aqua

**References:**

- *Abaqus/Aqua Analysis*
- **AQUA*
- **WAVE*
- **WIND*

## Overview

User subroutine *VWAVE*:

- will be called for a collection of points (typically load integration points) for which an Abaqus/Aqua load and a user-defined gravity wave are specified;
- can be used to define wave kinematics to provide unsteady contributions to fluid variables—such as velocity, acceleration, pressure, gradient of pressure along elevation, and the instantaneous free-surface elevation—as a function of time and space; and
- will be called twice within an increment for each Abaqus/Aqua load. The first call is used to obtain the instantaneous wave-surface elevation at the nodes of the elements on which the loads are applied. The second call is used to obtain desired fluid variables at the integration points for the load calculations.

## User Subroutine Interface

```
       subroutine vwave(
C Write  only -
     1 fWaveSurf, fUnsteadyVel, fFluidAcc,
     2 fUnsteadyPress, fUnsteadyDPressDZ,
C Read/Write -
     1 ScaleSteady, ScaleUnsteady,
C Read  only -
     1 kStep, kInc,
     2 nblock, ndim, nprops, naquaconst, nwindconst,
     3 nstatevar, nfieldvar, iElemType, iLoadType, sname,
     4 lUpdFluidVar, Coord, Velocity, StateVar, FieldVar,
     5 DirVec, AquaSteadyConstants, WindConstants,
     6 fSteadyVel, Props, dt, timeTotal, timeStep)
C
      include 'vaba_param.inc'
C
      parameter ( j_upd_FreeSurf = 0,
     1            j_upd_FluidVarBuoyancy = 1,
     2            j_upd_FluidVarDrag = 2,
     3            j_upd_FluidVarInertia = 3,
C     The types of distributed loads:
     1            j_lcr_PB  = 51,
     2            j_lcr_DragFDD = 53, j_lcr_DragWDD = 54,
     3            j_lcr_DragFDT = 55, j_lcr_DragFI  = 56,
     4            j_lcr_DragFD1 = 57, j_lcr_DragFD2 = 58,
```

```
     5            j_lcr_DragWD1 = 59, j_lcr_DragWD2 = 60,
     6            j_lcr_DragFI1 = 61, j_lcr_DragFI2 = 62,
C    The types of concentrated loads:
     1            j_ccr_TSB = 1002,     j_ccr_DragTFD = 1004,
     2            j_ccr_DragTWD = 1005, j_ccr_DragTSI = 1006 )
C
      dimension Props(nProps), Coord(nblock,ndim),
     1 Velocity(nblock,ndim), StateVar(nblock,nstatevar),
     2 FieldVar(nblock,nfieldvar), DirVec(nblock,ndim),
     3 fWaveSurf(nblock), fFluidAcc(nblock,ndim),
     4 fUnsteadyVel(nblock,ndim), fUnsteadyPress(nblock),
     5 fUnsteadyDPressDZ(nblock), fSteadyVel(nblock,ndim),
     6 AquaSteadyConstants(naquaconst), WindConstants(nwindconst)
C
      character*80 sname
C
C    The following if test and do loop structure illustrates
C    proper usage of this user subroutine.
C
      if (lUpdFluidVar .eq. j_upd_FreeSurf) then
C     This part is executed at the first call.
C
          do kn = 1,nblock
C            User coding to update fWaveSurf
C            optionally update StateVar
          end do
C
      else
C     This part is executed at the second call.
C
         if (lUpdFluidVar .eq. j_upd_FluidVarBuoyancy) then
C        Update variables for buoyancy loads (PB, TSB):
            do kn = 1,nblock
C              user coding to update fUnsteadyPress, fUnsteadyDPressDZ
            end do
C           optionally update multipliers ScaleSteady, ScaleUnsteady
         else if (lUpdFluidVar .eq. j_upd_FluidVarDrag) then
C        Update variables for drag loads (FDD, FDT, FD1, FD2, TFD):
             do kn = 1, nblock
C              User coding to update fUnsteadyVel
             end do
C            optionally update multipliers ScaleSteady, ScaleUnsteady
         else if (lUpdFluidVar .eq. j_upd_FluidVarInertia) then
C        Update variables for inertia loads (FI, FI1, FI2, TSI):
             do kn = 1, nblock
C                User coding to update fFluidAcc
             end do
C            optionally update multipliers ScaleSteady, ScaleUnsteady
         end if
C
      end if
C
      return
      end
```

## Variables to Be Defined

`fWaveSurf(nblock)` This array contains the instantaneous fluid free surface elevation at the elemental nodes and is calculated when the flag `lUpdFluidVar` has the value `j_upd_FreeSurf`. The incoming array contains the still free-surface elevation value for each node. The nodal values, as seen in the first call, are used to identify the wet portion of an element.

`fUnsteadyVel(nblock, ndim)` This array contains the unsteady part of the fluid velocity at load integration points and is calculated when the flag `lUpdFluidVar` has the value `j_upd_FluidVarDrag`. The incoming array contains zeros.

`fFluidAcc(nblock, ndim)` This array contains the fluid acceleration at load integration points and is calculated when the flag `lUpdFluidVar` has the value `j_upd_FluidVarInertia`. The incoming array contains zeros.

`fUnsteadyPress(nblock)` This array contains the unsteady part of the fluid pressure at load integration points and is calculated when the flag `lUpdFluidVar` has the value `j_upd_FluidVarBuoyancy`. The incoming array contains zeros. The steady part of the pressure is not stored but is calculated by Abaqus/Explicit at each load integration point based on the data provided under the **AQUA* option and is scaled by the `ScaleSteady` parameter.

`fUnsteadyDPressDZ(nblock)` This array contains the unsteady part of the gradient of fluid pressure along elevation at load integration points and is calculated when the flag `lUpdFluidVar` has the value `j_upd_FluidVarBuoyancy`. The incoming array contains zeros. Similar to steady pressures, the gradients are not stored but are calculated by Abaqus/Explicit at each load integration point based on the data provided in the fluid variable definition and scaled by the `ScaleSteady` parameter.

## Variables That Can Be Updated

The fluid variables—such as velocity, acceleration, pressure, and pressure gradient along elevation—used in load calculations are split in steady and unsteady parts. The following real variables scale each of those parts:

`ScaleSteady` This variable is used by Abaqus/Explicit to scale the steady part of the fluid variables. For drag loads this variable is the amplitude value provided for this purpose on the load data line as explained in *Abaqus/Aqua Analysis*. For all other loads its incoming value is one. The user can optionally update this variable.

`ScaleUnsteady` This variable is used by Abaqus/Explicit to scale the unsteady part of the fluid variables. The user is expected to define unscaled values for unsteady fluid variables. For drag loads this variable is the amplitude value provided for this purpose on the load data line, as explained in *Abaqus/Aqua Analysis*. For all other loads its incoming value is one. The user can optionally update this variable.

## Variables Passed in for Information

`kStep` Step number.

`kInc` Increment number.

`nblock` Number of nodes or load integration points, where the wave effects are to be computed.

`ndim` Dimension of the problem (two- or three-dimensional problem).

| | |
|---|---|
| `nprops` | The number of properties (real numbers) for the user-defined wave. |
| `naquaconst` | The number of fluid property constants in the fluid variable definition. |
| `nwindconst` | The number of constants included in the wind velocity profile. |
| `nstatevar` | The number of state variables for the user-defined wave, which is specified using the DEPVAR parameter. |
| `nfieldvar` | The number of field variables available on the element set on which the load is applied. At the first call the field variables will be available at nodes, and at the second call they will be available at the load integration points. |
| `iElemType` | An integer indicating the type of element these points belong to: 1 for line elements, 2 for surface elements, and 3 for solid elements. This integer is used to interpret the value of the direction vector (`DirVec`) provided at each point. |
| `iLoadType` | An integer indicating the type of Abaqus/Aqua load for which the wave kinematics are calculated. The parameters `j_lcr_PB` to `j_ccr_TSI` indicate the values by which the load types can be identified. |
| `sname` | The name of the element set on which the load is applied. |
| `lUpdFluidVar` | This integer flag is used to update different fluid variables. The flag acquires known values as declared in the list of parameters. This flag provides information regarding which variables are to be updated on each call. When the subroutine is called with `lUpdFluidVar` having a value of `j_upd_FreeSurf`, only the free surface elevation at each point is updated; otherwise, different sets of fluid variables are updated depending on different values of `lUpdFluidVar`, as explained in the subroutine structure above. |
| `Coord(nblock,ndim)` | This array contains the global coordinates of all points in their current configuration at the start of the time increment. |
| `Velocity(nblock,ndim)` | This array contains the structural velocities of all points at the mid-increment time level of the previous increment. |
| `StateVar(nblock,nstatevar)` | This array contains the user-defined solution-dependent state variables at all points. The state variables can optionally be updated during the first call, when the flag `lUpdFluidVar` has a value of `j_upd_FreeSurf`. At the second call the incoming values are interpolated values at the integration points and are read-only. |
| `FieldVar(nblock,nfieldvar)` | This array contains the field variables at the load integration points at the start of the time increment. |
| `DirVec(nblock,ndim)` | This array contains the direction vectors at the load integration points at the start of the time increment. For points lying on line elements, it is the tangent vector in the current configuration. For points lying on surface elements or solid faces, it is the surface-normal vector, pointing outward in the current configuration. The type of elements with which the points are associated can be found using the integer variable `iElemType`. The magnitude of this vector is the outer diameter for line elements under distributed loads (PB, FDD, WDD, FDT, FI), transitional |

| | |
|---|---|
| | section area for line elements under transitional distributed loads (FD1, FD2, WD1, WD2, FI1, FI2), and the nodal surface area for points under concentrated loads (TSB, TFD, TWD, TSI) or points lying on surface elements or solid faces. |
| `fSteadyVel(nblock,ndim)` | The incoming array contains the steady fluid velocity. |
| `AquaSteadyConstants(naquaconst)` | The user-specified fluid property constants in the fluid variable definition. |
| `WindConstants(nwindconst)` | The user-specified constants in the wind profile definition. In the absence of a wind profile, all values are zeros. |
| `Props(nprops)` | This real array contains properties for the user-defined wave, as listed on the data lines. |
| `dt` | The time increment. |
| `timeTotal` | The total time at the beginning of the increment over all steps. |
| `timeStep` | The step time at the beginning of the increment within the step. |

# General Utility Routines

A variety of utility routines are available to assist in the coding of user subroutines.

**In this section:**


- *Execution*
- *Model Data*
- *History Data*
- *Element Data*
- *Node Data*
- *Material Point Data*
- *Contact Data: Obtaining Contact Constraint Point Information in an Abaqus/Standard Analysis*

## Execution

Execution utility routines are available to assist in the coding of user subroutines.

### In this section:


- *Obtaining Abaqus Environment Variables*
- *Obtaining the Abaqus Job Name*
- *Obtaining the Abaqus Output Directory Name*
- *Obtaining Parallel Processes Information*
- *Printing Messages to the Message or Status File*
- *Terminating an Analysis*
- *Ensuring Thread Safety*
- *Allocatable Arrays*

# Obtaining Abaqus Environment Variables

**Products:** Abaqus/Standard Abaqus/Explicit

**References:**

- *About User Subroutines and Utilities*
- *Environment File Settings*
- *UWAVE and UEXTERNALDB*

## Overview

Utility routines GETENVVAR and VGETENVVAR can be called from any Abaqus/Standard or Abaqus/Explicit user subroutine, respectively, to obtain the value of an environment variable.

### Utility Routine Interface

```
character*256 ENVVAR
...
CALL GETENVVAR( 'ENVVARNAME',  ENVVAR, LENVVAR )
CALL VGETENVVAR( 'ENVVARNAME',  ENVVAR, LENVVAR )
...
```

### Variables to Be Provided to the Utility Routine

**ENVVARNAME** Environment variable name.

### Variables Returned from the Utility Routine

**ENVVAR** Character string to receive the value of the environment variable.

**LENVVAR** Length of the character string ENVVAR.

# Obtaining the Abaqus Job Name

**Products:** Abaqus/Standard Abaqus/Explicit

### References:

- *About User Subroutines and Utilities*
- *UWAVE and UEXTERNALDB*

## Overview

Utility routines `GETJOBNAME` and `VGETJOBNAME` can be called from any Abaqus/Standard or Abaqus/Explicit user subroutine, respectively, to obtain the name of the current job.

### Utility Routine Interface

```
character*256 JOBNAME
...
CALL GETJOBNAME( JOBNAME, LENJOBNAME )
CALL VGETJOBNAME( JOBNAME, LENJOBNAME )
...
```

### Variables Returned from the Utility Routine

**JOBNAME** Character string to receive the value of the job name.

**LENJOBNAME** Length of the character string `JOBNAME`.

# Obtaining the Abaqus Output Directory Name

**Products:** Abaqus/Standard Abaqus/Explicit

**References:**

- *About User Subroutines and Utilities*
- *UWAVE and UEXTERNALDB*

## Overview

Utility routines `GETOUTDIR` and `VGETOUTDIR` can be called from any Abaqus/Standard or Abaqus/Explicit user subroutine, respectively, to obtain the output directory of the current job.

### Utility Routine Interface

```
character*256 OUTDIR
...
CALL GETOUTDIR( OUTDIR, LENOUTDIR )
CALL VGETOUTDIR( OUTDIR, LENOUTDIR )
...
```

### Variables Returned from the Utility Routine

**`OUTDIR`** Character string to receive the value of the output directory name.

**`LENOUTDIR`** Length of the character string `OUTDIR`.

## Obtaining Parallel Processes Information

**Products:** Abaqus/Standard Abaqus/Explicit

### References:

- *About Parallel Execution*
- *VUSDFLD*

## Overview

Several different utility routines are available to provide detailed information about your parallel processes.

### GETNUMMPIRANKS (Obtain the Number of MPI Ranks)

Utility routine GETNUMMPIRANKS can be called from any Abaqus user subroutine. GETNUMMPIRANKS returns the number of MPI ranks or processes used for a parallel execution.

#### Utility Routine Interface

```
      CALL GETNUMMPIRANKS( NUMMPIRANKS )
      ...
```

#### Variables Returned from the Utility Routine

**NUMMPIRANKS** Number of MPI ranks specified for the analysis.

### GETRANK (Obtain the MPI Rank of the Current Process)

Utility routine GETRANK can be called from any Abaqus user subroutine. GETRANK returns the rank of the MPI process from which the function is called. For example, in a hybrid MPI and thread parallel execution scheme, multiple threads can all return the rank of their parent MPI process (see *Parallel Execution in Abaqus/StandardParallel Execution in Abaqus/Explicit*).

#### Utility Routine Interface

```
      CALL GETRANK( KRANKNUM )
      ...
```

#### Variables Returned from the Utility Routine

**KRANKNUM** Rank number. A rank number is either zero or a positive integer.

### GETNUMTHREADS (Obtain the Number of Active Threads for the Current Process)

Utility routine GETNUMTHREADS can be called from any Abaqus user subroutine. It returns the number of active threads in the current process. In a hybrid parallel execution mode, there will be several AbaqusMPI processes (ranks), each having multiple threads.



**Note:** The number of the active threads associated with an MPI process might be less than the **threads_per_mpi_process** parameter value.

### Utility Routine Interface

```
Fortran:
#include <SMAAspUserSubroutines.hdr>

         integer numThreads
         numThreads = GETNUMTHREADS()

C++:
         #include <SMAAspUserSubroutines.h>

         int numThreads = GETNUMTHREADS();
```

## GETTHREADID (Obtain the ID of the Current Thread Within the Current Process)

You can determine the ID of the thread you are in by calling the utility function GETTHREADID. The returned ID is an integer that Abaqus assigns to each of its active threads within the current process. The main thread will have the ID=0, and each subsequent thread will have an ID of 1, 2, 3, 4, ..., *N*. This function can be called from any Abaqus user subroutine and from both the Fortran and C/C++ codes.

### Utility Routine Interface

```
FOTRAN:
#include <SMAAspUserSubroutines.hdr>

         INTEGER myThreadID

         myThreadID = GETTHREADID()

C++:
         #include <SMAAspUserSubroutines.h>

         int myThreadID = GETTHREADID()
```

### Variables Returned from the Utility Routine

**myThreadID** Current thread ID, is either zero or a positive integer.

## GETCOMMUNICATOR (FORTRAN)

Utility function GETCOMMUNICATOR can be called from any Abaqus user subroutine. GETCOMMUNICATOR returns a communicator that Abaqus defines for its worker processes, similar to MPI_COMM_WORLD. In Fortran its type is an INTEGER. The communicator thus obtained can be used for subsequent MPI communication routines. In a nonparallel run, when the MPI subsystem is not initialized, communicators do not exist and GET_COMMUNICATOR() will return 0. Another way of testing is to call the MPI_Initialized(flag) function. This function will set the flag to 1 if MPI has been initialized.

### Utility Routine Interface

```
#include <SMAAspUserSubroutines.hdr>

         integer ABA_COMM_WORLD

         ABA_COMM_WORLD = GETCOMMUNICATOR()

         if (ABA_COMM_WORLD.ne.0) then
             ...do some parallel work, using MPI ...
         else
             ...do some work in a single process ...
         end if
```

### Variables Returned from the Utility Routine

**`INTEGER`**

Communicator handle identifier of type `INTEGER`. In a non-MPI run, the value returned will be zero.

## get_communicator (C++)

Utility function `get_communicator` can be called from any Abaqus user subroutine. `get_communicator` returns a communicator that Abaqus defines for its worker processes. In C++ this function is called `get_communicator()` and returns a value of the type `MPI_Comm`. The communicator thus obtained can be used for subsequent MPI communication routines. In a nonparallel run, when the MPI subsystem is not initialized, communicators do not exist and `get_communicator()` will return 0. Another way of testing is to call the `MPI_Initialized(flag)` function. This function will set the flag to 1 if MPI has been initialized.

### Utility Routine Interface

```
         #include <mpi.h>
         #include <SMAAspUserSubroutines.h>

         MPI_Comm ABA_COMM_WORLD = get_communicator()

         if (ABA_COMM_WORLD) {
            // ... do some parallel work, using MPI ...
         }
         else{
            // ...do some work in a single process ...
         }
```

### Variables Returned from the Utility Routine

**`MPI_Comm`**

Communicator handle of type `MPI_Comm`. In a non-MPI run, the value returned will be zero. The type `MPI_Comm` is defined in the `mpi.h` header file.

# Printing Messages to the Message or Status File

**Products:** Abaqus/Standard Abaqus/Explicit

### References:

- *About User Subroutines and Utilities*
- *UWAVE and UEXTERNALDB*
- *VUMAT*

## Overview

Utility routines `STDB_ABQERR` and `XPLB_ABQERR` can be called from any Abaqus/Standard or Abaqus/Explicit user subroutine, respectively, to issue an informational, a warning, or an error message to the message (`.msg`) file in Abaqus/Standard or the status (`.sta`) file in Abaqus/Explicit.

### Utility Routine Interface

```
DIMENSION INTV(*),REALV(*)
CHARACTER*8 CHARV(*)
...
CALL STDB_ABQERR(LOP,STRING,INTV,REALV,CHARV)
or
CALL XPLB_ABQERR(LOP,STRING,INTV,REALV,CHARV)
...
```

### Variables to Be Provided to the Utility Routine

**`LOP`**

Flag for the type of message to be issued.

Set `LOP` = 1 if an informational message is to be issued.

Set `LOP` = –1 if a warning message is to be issued.

Set `LOP` = –2 if an error message is to be issued and the analysis is to be continued.

Set `LOP` = –3 if an error message is to be issued and the analysis is to be stopped immediately.

**`STRING`**

A string of at most 500 characters long between single quotes containing the message to be issued. If the string needs to be written on more than one line, several one line long strings (between single quotes) should be concatenated using the double forward slash (//) operator.

Integer, real, and character variables can be referenced inside the message using the %I, %R, and %S inserts, respectively. The integer, real, or character variables are passed into the utility routine via the `INTV`, `REALV`, and `CHARV` variables, respectively. The variables are then output in the order they are stored in these arrays.

**`INTV`**

Array of integer variables to be output. The first %I in `STRING` will output `INTV`(1), the second `INTV`(2), and so on.

**`REALV`**

Array of real variables to be output. The first %R in `STRING` will output `REALV`(1), the second `REALV`(2), and so on.

**CHARV** Array of at most 8 character long variables to be output. The first %S in `STRING` will output `CHARV`(1), the second `CHARV`(2), and so on.

# Terminating an Analysis

**Products:** Abaqus/Standard Abaqus/Explicit

### References:

- *About User Subroutines and Utilities*
- *UMAT and UHYPER*
- *UWAVE and UEXTERNALDB*
- *VUMAT*

## Overview

Utility routines `XIT` and `XPLB_EXIT` can be called from within any Abaqus/Standard or Abaqus/Explicit user subroutine, respectively, (except `UEXTERNALDB`) to terminate an analysis.

`XIT` or `XPLB_EXIT` should be used instead of `STOP` to ensure that all files associated with the analysis are closed properly.

### Utility Routine Interface

```
CALL XIT
or
CALL XPLB_EXIT
```

# Ensuring Thread Safety

**Products:** Abaqus/Standard Abaqus/Explicit

### References:

- *About Parallel Execution*
- *Obtaining Parallel Processes Information*

## Overview

A number of primitives are provided to help code user subroutines for thread-parallel execution.

### Mutexinit, MutexLock, and Mutexunlock

In thread-parallel execution mutexes can be used to protect a common block or a common file from being updated by multiple threads at the same time. Abaqus provides 100 predefined mutexes for use in user subroutines. They are referenced simply by number (1–100).

Mutexes need to be initialized before they can be used. For example, `MutexInit(1)` initializes `mutex 1`. It is best to initialize mutexes at the very beginning of the analysis in user subroutines, such as user subroutines *UEXTERNALDB* and *VEXTERNALDB*.

Once initialized, mutexes can safeguard sensitive sections of the code against concurrent access. For example, `MutexLock(1)` and `MutexUnlock(1)` will respectively lock and unlock `mutex 1`.

Each mutex can protect a shared resource or a logical group of resources that are always accessed together. Different mutexes are provided so that users can protect a variety of shared resources or objects. For example, one mutex can protect a file and another mutex can protect common block variables. Thus, accessing a file can happen simultaneously with updating common block variables; but no two threads can write to the file at the same time, and no two threads can update the variables at the same time.

To make data transfer and accumulation easier and safer between user subroutines in a multi-threaded environment, Abaqus provides utility functions to create dynamic storage in the form of thread-local arrays, which are private to each thread, and global arrays, which are shared. Any number of arrays of any size can be created at run time. Global arrays are accessible from all user subroutines and all threads. Thread-local arrays are private and exist only within the scope of each thread. They are accessible to all user subroutines running in that thread but not across threads. Since they are not visible to neighboring threads, they do not need to be protected from concurrent access. They are designed as a thread-agnostic replacement for the COMMON BLOCKS and SAVE variables (see *Allocatable Arrays* for more information).

All other techniques commonly used in parallel programming can also be employed; for example, restricting all file operations only to thread 0. Oftentimes, these alternatives are preferable to using mutexes because they may provide better performance.

#### Utility Routine Interface

```
Fortran:
#include <SMAAspUserSubroutines.hdr>

       ! Initialization in UEXTERNALDB/VEXTERNALDB

       call MutexInit( 1 )       ! initialize Mutex #1

       ! Use in all other user subs after being initialized

        call MutexLock( 1 )       ! lock Mutex #1
```

```
        < critical section : update shared variables >

       call MutexUnlock( 1 )   ! unlock Mutex #1
C++:
       #include <SMAAspUserSubroutines.h>

       // Initialization in UEXTERNALDB/VEXTERNALD

       MutexInit( 1 );         // initialize Mutex #1

       // Use in all other user subs after being initialized

       MutexLock( 1 );         // lock Mutex #1

       < critical section : update shared variables >

       MutexUnlock( 1 );       // unlock Mutex #1

NOTE: IDs are arbitrary chosen by the user, from the pool of 1-100.
      Other threads, when encountering a locked mutex, will sleep.
      Once the first entering thread unlocks the mutex and leaves,
      other threads will be able to come in and execute the critical
      section (one at a
time).
```

# Allocatable Arrays

**Products:** Abaqus/Standard Abaqus/Explicit

**References:**

- *Ensuring Thread Safety*

## Overview

To facilitate data accumulation and transfer between user subroutines, you can use utility functions to create your own dynamic storage in the form of allocatable arrays. Thread-local and global arrays are supported. In addition to basic types, you can also vary the precision of real arrays according to the precision of Abaqus/Explicit and define arrays of user-defined data types.

### `SMALocalIntArrayCreate`, `SMALocalFloatArrayCreate`

You can create any number of thread-local or global arrays. You give each array an identifier (an arbitrary positive integer) at the time of its creation. You create an array in one user subroutine and reference it in another simply by its identifier. The arrays persist in memory until you explicitly delete them or until the analysis terminates.

## Thread-local arrays

A thread-local array is a mechanism to allocate storage that is local to a thread and does not need any locking for access. In a multi-threaded environment the thread safety of these arrays stems from their design and usage: they are deliberately not shared and, thus, do not need to be protected from competing threads. In fact, one thread cannot reference a local array of another thread. They can be accessed concurrently without any locking and, thus, are faster than global arrays.

Thread-local arrays are unique in each thread. They are nonintersecting and nonoverlapping in memory, with each thread starting out with its own private copy of an array. For example, Thread 0 can have a local array with ID 1 and Thread 4 can have a local array with ID 1. Those two arrays are different and separate from each other. Similarly, it is possible to have an integer array with ID 1 and a float array with ID 1. Again, they are two different arrays. It is not possible to cross-reference these arrays across different threads. However, all user subroutines running in one thread can access all arrays of that thread. In a thread-agnostic way, these arrays are shared between user subroutines but not among threads. These routines are meant as a thread-safe replacement for COMMON BLOCKs and SAVE variables.

The following utility subroutines are available to operate on thread-local arrays:

- `SMALocalIntArrayCreate`, `SMALocalFloatArrayCreate`: to create or resize a local array.
- `SMALocalIntArrayAccess`, `SMALocalFloatArrayAccess`: to locate an existing local array.
- `SMALocalIntArrayDelete`, `SMALocalFloatArrayDelete`: to delete a local array.
- `SMALocalIntArraySize`, `SMALocalFloatArraySize`: to get the size of the array.

These utility routines are accessible from both Fortran and C/C++. The details of their interfaces are described below.

## Global arrays

Global arrays are visible and accessible from all threads in an executable. To prevent race conditions, protect the creation and the write access to these arrays with mutexes (mutual exclusion locks). You can

have each thread execute global array creation under a mutex protection. However, only the first thread to arrive will create the global array; the later threads will simply connect to the array already created. In addition, using mutexes on every write access will incur a performance penalty. In some situations it is possible to avoid unnecessary locking by restricting all threads to operate on nonintersecting ranges of a global array. Another alternative is to use thread-local arrays.

The following utility routines are available to operate on global arrays:

- `SMAIntArrayCreate`, `SMAFloatArrayCreate`: to create or resize a global array.
- `SMAIntArrayAccess`, `SMAFloatArrayAccess`: to locate an existing global array.
- `SMAIntArrayDelete`, `SMAFloatArrayDelete`: to delete a global array.
- `SMAIntArraySize`, `SMAFloatArraySize`: to get the size of the global array.

These arrays are global and accessible from all threads within a process but not across different MPI processes. To share data between separate MPI processes, MPI facilities must be used. Abaqus supports the full use of MPI within user subroutines.

### Utility Routine Interface

```
Fortran:

      INTEGER*8 SMALocalIntArrayCreate(ID,SIZE,INITVAL)
      INTEGER*8 SMALocalFloatArrayCreate(ID,SIZE,INITVAL)

Example:

#include <SMAAspUserSubroutines.hdr>

      integer a(100)
      pointer(ptra,a)

      real*8 b(*)
      pointer(ptrb,b)

      ! create a local array with ID=1 and SIZE=100
      ptra = SMALocalIntArrayCreate(1,100)
      a(1) = 11  ! use as a native Fortran array
      a(2) = 22  ! use as a native Fortran array

      ! create a local float array with ID=1 and SIZE=100, and
      ! initial value = -1.0
      ptrb = SMALocalFloatArrayCreate(1,100,-1.0)

C++:

       #include <SMAAspUserSubroutines.h>

       // Create a local integer array of with ID=1 and size=100
       int* a = SMALocalIntArrayCreate(1,100);

       // Create a local float array of with ID=1, size=20, and
       // initial value = -1.0
       real* b = SMALocalFloatArrayCreate(1,100,-1.0);

NOTE: Float Arrays can store both SINGLE PRECISION
      and DOUBLE PRECISION numbers. Internally,
      memory is allocated in units of 64 bits (double/real*8).

NOTE: To resize an array, simply call Create() with the same ID,
```

```
      but give it a new SIZE parameter. If the new size is larger,
      the old data are copied over to the new array. No data are lost
      during resizing.

For example:

      ! resize array with ID=1 to 300 integers
      ptra = SMALocalIntArrayCreate(1,300)

NOTE: In Create() functions, there is an optional third
      argument -- initial value. If not supplied, all Int
      arrays are initialized with INT_MAX ( 2,147,483,647 ).
      All Float Arrays are initialized with Signaling NANs.
      The values of INT_MAX and signaling NANs are accessible
      via the 'SMAAspNumericLimits.h' and 'SMAAspNumericLimit.hdr'
      header files.
```

### Variables to Be Provided to the Utility Routine

**ID**
ID of the array (an integer), chosen by the user at the time of creation. Using this ID, an array can be opened in any other user subroutine.

**SIZE**
Size of the array as the number of ints or doubles. The maximum size for thread-local arrays is INT_MAX (2,147,483,647).

**INITVAL**
Initial value for each item in the array. If this argument is not supplied, in the case of an integer a large value is used; in the case of a float NAN is used.

### Variables Returned from the Utility Routine

**INTEGER*8 ( address )**
Returns a pointer to the array created. This pointer can be associated with a native Fortran array or native C/C++ array. Each thread will receive a different pointer. Each thread will create and hold its own array. For example, `Array(1)` in Thread 0 is separate from `Array(1)` in Thread 4. These arrays are nonoverlapping and nonintersecting in any way.

## SMALocalIntArrayAccess, SMALocalFloatArrayAccess

### Utility Routine Interface

```
Fortran interface:

      INTEGER*8 SMALocalIntArrayAccess(ID)
      INTEGER*8 SMALocalFloatArrayAccess(ID)

Example:

#include <SMAAspUserSubroutines.hdr>

      integer a(100)
      pointer(ptra,a)

C Locate local Array(1) and associate a native array pointer with it
```

```
      ptra = SMALocalIntArrayAccess(1)

      a(1) = 11 ! use as a native Fortran array
      a(2) = 22 ! use as a native Fortran array



C++ interface:

      #include <SMAAspUserSubroutines.h>

      // Locate and open array with ID=1
      int* a =  SMALocalArrayIntAccess(1);

      a[1] = 11;  // use as a native array
      a[2] = 22;  // use as a native array

NOTE: If a request is made to access an array that has
      not been created, the function will return 0.
```

### Variables to Be Provided to the Utility Routine

**ID** ID of the array (an integer), chosen by the user at the time of creation. Using this ID, an array can be opened in any other user subroutine.

### Variables Returned from the Utility Routine

**INTEGER*8 ( address )** Returns a pointer to the array created. This pointer can be associated with a native Fortran array or native C/C++ array. Each thread will receive a different pointer. Each thread, as it passes through this code, will create and hold its own array. For example, `Array(1)` in Thread 0 is a separate array from `Array(1)` in Thread 4. These arrays are nonoverlapping and nonintersecting in any way.

## SMALocalIntArraySize, SMALocalFloatArraySize

### Utility Routine Interface

```
Fortran interface:

      INTEGER*4 SMALocalIntArraySize(ID)
      INTEGER*4 SMALocalFloatArraySize(ID)

Example:

#include <SMAAspUserSubroutines.hdr>

      integer a_size, d_size

C Get the size of Array(1) as the number of INTEGERs
      a_size = SMALocalIntArraySize(1)

! Get the size of Array(1) as the number of REALs
```

```
      d_size = SMALocalFloatArraySize(1)

      do k=1,a_size
          ...
      end do

C++:

      #include <SMAAspUserSubroutines.h>

      // Lookup the size of Array(1) as the number of ints

      int a_size = SMALocalIntArraySize(1);

      // Lookup the size of Array(1) as the number of doubles

      int d_size = SMALocalFloatArraySize(1);

      for(int i=1; i<=size; i++) {
          ...
      }
```

### Variables to Be Provided to the Utility Routine

**ID** ID of the array (an integer), chosen by the user at the time of creation. Using this ID, an array can be opened in any other user subroutine.

### Variables Returned from the Utility Routine

**INTEGER*4** Size of the array.

## SMALocalIntArrayDelete, SMALocalFloatArrayDelete

### Utility Routine Interface

```
Fortran interface:

      subroutine SMALocalIntArrayDelete(ID)
      subroutine SMALocalFloatArrayDelete(ID)

Example:

#include <SMAAspUserSubroutines.hdr>

      call SMALocalIntArrayDelete(1) ! Delete Array(1)

C++ interface:

      #include <SMAAspUserSubroutines.h>

      SMALocalIntArrayDelete(1);  // Delete Array(1)

NOTE: Deletion of arrays is optional. All storage allocated
      for these arrays will be freed when Abaqus threads
      terminate (at the very end of the analysis). It is,
      however, a good programming practice to delete all
```

```
      allocations explicitly, especially when they are
      no longer needed, as this will free up memory for
      something else.
```

### Variables to Be Provided to the Utility Routine

**ID** ID of the array (an integer), chosen by the user at the time of creation. Using this ID, an array can be opened in any other user subroutine.

## SMAIntArrayCreate, SMAFloatArrayCreate

### Utility Routine Interface

```
Fortran interface:

      INTEGER*8 SMAIntArrayCreate(ID,SIZE,INITVAL)
      INTEGER*8 SMAFloatArrayCreate(ID,SIZE,INITVAL)

Example:

#include <SMAAspUserSubroutines.hdr>

      integer a(100)
      pointer(ptra,a)
      double precision b(100)
      pointer(ptrb,b)

      ! create a global array with ID=1, SIZE=100, and
      ! INITVAL=-1.0
      ptra = SMAIntArrayCreate(1,100,-1)

      a(1) = 11  ! use as a native Fortran array
      a(2) = 22  ! use as a native Fortran array

      ! create a global array with ID=2, SIZE=100, and
      ! INITVAL=-1.0
      ptrb = SMAFloatArrayCreate(2,100,-1.0)

C++ interface:

       #include <SMAAspUserSubroutines.h>

       // Create an integer array of with ID=1, size=100,
       // and initial value=-1.0
       int* a = SMAIntArrayCreate(1,100,-1);

       // Create a float array of with ID=2, size=20,
       // and initial value=-1.0
       Real* b = SMAFloatArrayCreate(2,20,-1.0);

NOTE: Float Arrays can store both SINGLE PRECISION and
      DOUBLE PRECISION numbers. Internally, they
      allocate storage in 64-bit units (double/real*8).

NOTE: To resize an array, simply call Create() with the same ID,
      but give it a new SIZE parameter. If the size has increased,
      the old data will be copied over to the new array.
      No data is lost during resizing.
```

```
For example:

      ! resize array with ID=1 to 300 integers
      ptra = SMAIntArrayCreate(1,300,-1)
```

### Variables to Be Provided to the Utility Routine

**ID** ID of the array (an integer), chosen by the user at the time of creation. Using this ID, an array can be opened in any other user subroutine.

**SIZE** Size of the array as the number of ints or doubles. The maximum size is INT_MAX.

**INITVAL** Optional initial value for each item of the array. If this argument is not supplied, integer arrays are filled with INT_MAX and float arrays are filled with NANs.

### Variables Returned from the Utility Routine

**INTEGER*8 ( address )** Returns a pointer to the array created. This pointer can be associated with a native Fortran array or native C/C++ array. All threads will see the same address when they try to access this array through its ID.

## SMAIntArrayAccess, SMAFloatArrayAccess

### Utility Routine Interface

```
Fortran interface:

      INTEGER*8 SMAIntArrayAccess(ID)
      INTEGER*8 SMAFloatArrayAccess(ID)

Example:

#include <SMAAspUserSubroutines.hdr>

      integer a(100)
      pointer(ptra,a)

C Locate Array(1) and associate a native array pointer with it

      ptra = SMAIntArrayAccess(1)

      a(1) = 11 ! use as a native Fortran array
      a(2) = 22 ! use as a native Fortran array

C++ interface:

      #include <SMAAspUserSubroutines.h>

      // Locate and open array with ID=1
      int* a =  SMAIntArrayAccess(1);

      a[1] = 11;  // use as a native array
```

```
      a[2] = 22;  // use as a native array


NOTE: If a request is made to access an array which has
      not been created, the function will return 0.
```

### Variables to Be Provided to the Utility Routine

**ID** ID of the array (an integer), chosen by the user at the time of creation. Using this ID, an array can be opened in any other user subroutine.

### Variables Returned from the Utility Routine

**INTEGER*8 ( address )** Returns a pointer to the array, or 0 if an array with the requested ID does not exist. This pointer can be associated with a native Fortran or C/C++ array.

## SMAIntArraySize, SMAFloatArraySize

### Utility Routine Interface

```
Fortran interface:

      INTEGER SMAIntArraySize(ID)
      INTEGER SMAFloatArraySize(ID)

Example:

#include  <SMAAspUserSubroutines.hdr>

      integer a_size, d_size

C Get the size of Array(1) as the number of INTEGERs
      a_size = SMAIntArraySize(1)

! Get the size of Array(1) as the number of REALs

      d_size = SMAFloatArraySize(1)

      do k=1,a_size
          ...
      end do

C++ interface:

      #include <SMAAspUserSubroutines.h>

      // Lookup the size of Array(1) as the number of INTS

      int a_size = SMAIntArraySize(1);

      // Lookup the size of Array(1) as the number of doubles

      int d_size = SMAFloatArraySize(1);

      for(int i=1; i<=d_size; i++) {
```

```
        ...
    }
```

### Variables to Be Provided to the Utility Routine

**ID** ID of the array (an integer), chosen by the user at the time of creation. Using this ID, an array can be opened in any other user subroutine.

### Variables Returned from the Utility Routine

**INTEGER*4** Size of the array.

## SMAIntArrayDelete, SMAFloatArrayDelete

### Utility Routine Interface

```
Fortran:

#include <SMAAspUserSubroutines.hdr>

      call SMAIntArrayDelete(1) ! Delete global Array(1)

C++:

      #include <SMAAspUserSubroutines.h>

      SMAIntArrayDelete(1);  // Delete global Array(1)

NOTE: Deletion of arrays is optional. All storage allocated
      for these arrays will be freed when Abaqus terminates
      (at the very end of the analysis). It is, however, a good
      programming practice to delete all allocations explicitly,
      especially when they are no longer needed, as this will
      free up memory for use somewhere else.
```

### Variables to Be Provided to the Utility Routine

**ID** ID of the array (an integer), chosen by the user at the time of creation. Using this ID, an array can be opened in any other user subroutine.

## Allocatable Global Arrays of Variable Precision

The usage of real arrays is exactly the same as that of integer and floating point arrays except for the handling of precision. The precision of real arrays varies, changing along with the precision of Abaqus/Explicit. In single precision the values of real arrays are 32-bits long, and in double precision their values are 64-bits. For this automatic switching to work in Fortran, the type of such an array should not be declared explicitly. Abaqus relies on the implicit naming to alternate between single precision and double precision. In C/C++ the type of the native array should be `Real*`. The `typedef` declaration changes between `float` and `double` depending on the precision of Abaqus/Explicit. The precision does not change during a run; it is determined at the beginning of the analysis and remains the same until the end.

When you create real arrays, you give each array an identifier. Arrays can be created in one user subroutine and operated on in another simply by referencing this identifier. You need not capture the pointer to the array and pass it between routines. The arrays persist in memory from the moment they are created until you delete them explicitly or until the analysis ends. The arrays do not disappear when any particular user subroutine terminates. They are accessible from all user subroutines and all threads. Each MPI process is separate in memory from other MPI processes and has its own arrays. There is no cross-referencing of these arrays across MPI processes.

These arrays can be resized dynamically as needed. A call to `Create()` on an existing array but with a different size resizes the array. If the new size is larger than the previous size, there is no loss of data and the previous contents are carried over.

### Utility Routine Interface

```
Fortran:

#include <vaba_param.inc>
#include <SMAAspUserSubroutines.hdr>

      ! Note: we do not explicitly declare the type of 'ra', we
      ! rely on rules of implicit typing: it will become real*4
      ! or real*8 depending on the precision of Abaqus

      dimension  ra(*)
      pointer(ptrra,ra)

      integer     sz

      rinitval = -1.0e36        ! again, implicit typing

      ! Creating an array

      ! ID=1, SIZE=10, no initializer
      ptrra = SMARealArrayCreate(1, 10)
      ! ID=2, SIZE=10, rinitval used to initialize
      ptrra = SMARealArrayCreate(2, 10, rinitval)
      ! ID=3, SIZE=10, initial value is -3.3d0
      ptrra = SMARealArrayCreate(3, 10, -3.3d0)
      ! ID=4, SIZE=10, initial value is -3.3
      ptrra = SMARealArrayCreate(4, 10, -3.3)

      ! Use ( from another subroutine )

      ptrra = SMARealArrayAccess(1)

      if (ptrra.eq.0) then
           write(*,*) '### Array',i, 'does not exist'
      end if

      ! Use as a native array in Fortran

      ra(1) = 11.11
      ra(2) = 22.22

      ! Looping

      ! Find out the current size of the array #1
      sz = SMARealArraySize(1)

      do k=1,sz
           write(*,*) k, '=', ra(k)
```

```
      end do

      ! Resizing

      ptrra = SMARealArrayCreate(1, 1000, -1.0)

             ! Array #1 is resized; the original 10 entries
             ! are intact and carried over to the new array;
             ! all new entries are set to -1.0

      ! Deletion

      call SMARealArrayDelete(1)
      call SMARealArrayDelete(2)
      call SMARealArrayDelete(3)
      call SMARealArrayDelete(4)


C/C++:

      #include <omi_for_types.h>
      #include <SMAAspUserSubroutines.h>

      Real* ra = 0;   // Type 'Real' switches precision with Explicit
      int   sz = 0;

      // Examples of Array Creation

      // ID=1, SIZE=10, no initializer used
      ra = SMARealArrayCreate(1, 10);
      // ID=2, SIZE=10, initial value = -1.0
      ra = SMARealArrayCreate(2, 10, -1.0);

      // Access from another User Subroutine

      ra = SMARealArrayAccess(1);

      if ( ra == 0 ) {
          fprintf(stderr,

          "*** Error: array %d does not exist ***\n", 1 );
      }

      // Looping over the entries

      //  obtain the current size of array #1
      sz = SMARealArraySize(1);

      for (int i=0; i<sz; i++) {
          sum = sum + ra[i];
      }

      // Deletion

      SMARealArrayDelete(1);
      SMARealArrayDelete(2);
```

### Variables to Be Provided to the Utility Routine

**ID**
ID of the array (an integer), chosen by the user at the time of creation. Using this ID, an array can be opened in any other user subroutine.

**SIZE**
Size of the array as the number of items. The maximum size is INT_MAX (2,147,483,647).

**INITVAL**
Initial value for each item of the array. If the argument is not supplied, zero is used as the initial value.

### Variables Returned from the Utility Routine

**INTEGER*8 (address)**
Returns a pointer to the array created. This pointer can be associated with a native Fortran array or a native C/C++ array. These arrays are global. All threads will see and access exactly the same global array with a given ID.

## Allocatable Global Arrays of User-Defined Types

The usage and syntax of arrays of structures are exactly the same as those of integer, floating point, and real arrays. These arrays are designed to store any user-defined types or classes, defined either in Fortran or in C/C++. The only information an array needs to know about these structures is their memory size. Most compilers provide the `sizeof()` operator, which returns the size of any object in memory in bytes. This size is one additional argument to the routines that operate on arrays of structures.

When you create arrays of structures, you give each array an identifier. Arrays can be created in one user subroutine and operated on in another simply by referencing this identifier. You need not capture the pointer to the array and pass it between routines. The arrays persist in memory from the moment they are created until you delete them explicitly or until the analysis ends. The arrays do not disappear when any particular user subroutine terminates. They are accessible from all user subroutines and from all threads. Each MPI process is separate in memory from other MPI processes and has its own arrays. There is no cross-referencing of these arrays across MPI processes.

These arrays can be resized dynamically as needed. A call to `Create()` on an existing array but with a different size resizes the array. If the new size is larger than the previous size, there is no data loss and the previous contents are carried over.

### Utility Routine Interface

```
Fortran:

! Include a user module called, for example, 'mod',
! which defines some user structure 'UserStruct'

       use mod

#include <aba_param.inc>      ! include this for Abaqus/Standard
#include <vaba_param.inc>     ! include this for Abaqus/Explicit

#include <SMAAspUserSubroutines.hdr>

       type(UserStruct)::  us(10)
       type(UserStruct)::  structs(10)
       type(UserStruct)::  initval,s
```

```
      pointer(ptrstructs, structs)

      integer:: size1, size2, size3, size4
      integer(kind=8) :: arraySize


      ! Create an initializer for the values of the array
      !(optional)

      initval%a = 100
      initval%b = 200
      initval%c = 300


      ! Different ways of obtaining the size of a structure

      size1 = storage_size( us(1) ) / 8     ! returns the size
                                            ! in bits
      size2 = sizeof( us(1) )
      size3 = storage_size( initval ) / 8   ! returns the size
                                            ! in bits
      size4 = sizeof( initval )


      ! Creating an array

      write(*,*) 'Array without initializers:'

      ptrstructs = SMAStructArrayCreate(1, 10, sizeof(initval))

      write(*,*) 'Array with  initializers:'

      ptrstructs = SMAStructArrayCreate(2, 10, sizeof(initval),
     &                                  initval)


      ! Use ( from another subroutine )

      ptrstructs = SMAStructArrayAccess(2)

      if (ptrstructs.eq.0) then
          write(*,*) '### Array 2 does not exist'
      end if


     ! Use as a native array in Fortran

     structs(5).a = -51
     structs(5).b = -52
     structs(5).c = -53

     structs(10).a = 111
     structs(10).b = 222
     structs(10).c = 333


     ! Looping over the entries

     arraySize = SMAStructArraySize(2)
```

```
      do k=1,arraySize
            s = structs(k); call PrintStruct(s)
      end do


      ! Resize an array without using initializer

      ptrstructs = SMAStructArrayCreate(2,  100, sizeof(initval))

      arraySize = SMAStructArraySize(2)


      ! Resize array 2 with initializer

      ptrstructs = SMAStructArrayCreate(2,   200, sizeof(initval),
     &                                  initval)

      arraySize = SMAStructArraySize(2)


      ! Deletion

      call SMAStructArrayDelete(1)
      call SMAStructArrayDelete(2)
```

C/C++:

```
      #include <omi_for_types.h>
      #include <SMAAspUserSubroutines.h>

      // Include the definition of a user-defined type,
      // for example, A

      #include <A.h>


      // Create an (optional) initializer for user structs

      A init = { -1, -2, -3 };

      // Creating arrays

      // no initializer
      SMAStructArrayCreate(1, 10, sizeof(A));
      // with initializer
      SMAStructArrayCreate(2, 10, sizeof(A), &init;);

      // Accessing arrays (from another subroutine)

      A* array = (A*) SMAStructArrayAccess(1);


      // Modifying values in the array

      A* s1 = &array[5]; // We use a pointer to modify the value in
                         // the array itself. Without a pointer, s1
                         // will contain a copy of the entry in
                         // the array, and any modifications to
                         // this copy will not affect the value in
                         // the original array.
```

```
      s1->a = -111;
      s1->b = -222;
      s1->c = -333;

      // Looping over the entries

      size_t sz = SMAStructArraySize(1);

      printf("Array 1: \n");
      for (size_t i=0; i < sz; i++) {
          PrintStruct(i, &array[i]);
      }

      // Deletion

      SMAStructArrayDelete(1);
      SMAStructArrayDelete(2);
```

### Variables to Be Provided to the Utility Routine

**ID**
ID of the array (an integer), chosen by the user at the time of creation. Using this ID, an array can be opened in any other user subroutine.

**NUM_ITEMS**
Size of the array as the number of items. The maximum size is INT_MAX (2,147,483,647).

**ITEM_SIZE**
Size of one item (struct) in bytes.

**INITVAL**
Initial value for each item (struct) in the array. If this value is not supplied, the memory is simply zeroed out.

### Variables Returned from the Utility Routine

**INTEGER*8 (address)**
Returns a pointer to the array created. This pointer can be associated with a native Fortran array or a native C/C++ array. These arrays are global. All threads will see and access exactly the same global array with a given ID.

## Model Data

Utility routines to access model data are available to assist in the coding of user subroutines.

### In this section:


- *Obtaining Part Information*
- *Obtaining Physical Constants in an Abaqus/Standard Analysis*
- *Accessing Abaqus Table Collections*
- *Accessing Abaqus Event Series Data*
- *Evaluating Abaqus Neural Networks*

# Obtaining Part Information

**Products:** Abaqus/Standard Abaqus/Explicit

**References:**

- *About User Subroutines and Utilities*
- *Assembly Definition*
- *Pure bending of a cylinder: CAXA elements*

## Overview

Several utility routines are available to allow you to obtain information about your part instances.

Utility routines `GETPARTINFO` and `VGETPARTINFO` can be called from any Abaqus/Standard or Abaqus/Explicit user subroutine, respectively, to retrieve the part instance name and original node or element number corresponding to an internal node or element number. Utility routines `GETINTERNAL` and `VGETINTERNAL` can be called from any Abaqus/Standard or Abaqus/Explicit user subroutine, respectively, to retrieve the internal node or element number corresponding to a part instance name and original node or element number. The part file (*jobname*`.prt`) must be available. The expense of calling these routines is not trivial, so minimal use of them is recommended.

### `GETPARTINFO` and `VGETPARTINFO` (Obtain Part Instance Information Given Global Node/Element Number)

#### Utility Routine Interface

```
      CHARACTER*80 CPNAME
      ...
      CALL GETPARTINFO(INTNUM, JTYP, CPNAME, LOCNUM, JRCD)
      or
      CALL VGETPARTINFO(INTNUM, JTYP, CPNAME, LOCNUM, JRCD)
```

#### Variables to Be Provided to the Utility Routine

**`INTNUM`**
The internal (global) node or element number to be looked up.

**`JTYP`**
An integer flag indicating whether `INTNUM` is a node or element number. Set `JTYP=0` to look up a node number, and set `JTYP=1` to look up an element number.

#### Variables Returned from the Utility Routine

**`CPNAME`**
The name of the part instance that contains `INTNUM`. An empty part instance name indicates that the node or element is at the assembly level and is not included in any part instance.

**`LOCNUM`**
The part-local node or element label corresponding to `INTNUM`.

**`JRCD`**
Return code (0–no error, 1–error).

## GETINTERNAL and VGETINTERNAL (Obtain Global Node/Element Number Given Part Instance Information )

### Utility Routine Interface

```
      CHARACTER*80 CPNAME
      ...
      CALL GETINTERNAL(CPNAME, LOCNUM, JTYP, INTNUM, JRCD)
      or
      CALL VGETINTERNAL(CPNAME, LOCNUM, JTYP, INTNUM, JRCD)
```

### Variables to Be Provided to the Utility Routine

**CPNAME**
The name of the part instance that contains LOCNUM.

**LOCNUM**
The part-local node or element label to be looked up.

**JTYP**
An integer flag indicating whether LOCNUM is a node or element label. Set JTYP=0 to look up a node number, and set JTYP=1 to look up an element number.

### Variables Returned from the Utility Routine

**INTNUM**
The internal (global) node or element label corresponding to LOCNUM in part instance CPNAME.

**JRCD**
Return code (0–no error, 1–error).

# Obtaining Physical Constants in an Abaqus/Standard Analysis

**Products:** Abaqus/Standard

**References:**

- **PHYSICAL CONSTANTS*

## Overview

Utility routine `GETPHYSICALCONSTANT` can be called from any Abaqus/Standard user subroutine to obtain values of physical constants defined in an Abaqus/Standard analysis.

### Utility Routine Interface

```
include 'aba_phcon_param.inc'
...
call GETPHYSICALCONSTANT (jKey, value, jError)
```

### Variables to Be Provided to the Utility Routine

**`jKey`**

Key identifier of the physical constant requested.

| Key identifier (`jKey`) | Physical constant |
|---|---|
| `j_phcon_Abs0` | Absolute zero |
| `j_phcon_Boltzmann` | Ludwig Boltzmann constant |
| `j_phcon_SplRef` | Reference pressure used for computing the sound pressure level |
| `j_phcon_StefanB` | Stefan-Boltzmann constant |
| `j_phcon_UnivGas` | Universal gas constant |
| `j_phcon_Avogadro` | Avogadro number |
| `j_phcon_Faraday` | Faraday constant |
| `j_phcon_eCharge` | Elementary charge |

### Variables Returned from the Utility Routine

**`value`**

Value of the physical constant requested.

**`jError`**

Return code (0 indicates no error, 1 indicates that the physical constant requested is not defined).

# Accessing Abaqus Table Collections

**Products:** Abaqus/Standard Abaqus/Explicit

**References:**

- **TABLE COLLECTION*
- **PARAMETER TABLE*
- **PROPERTY TABLE*
- *About Table Collections, Parameter Tables, and Property Tables*

## Overview

These utility routines can be used to activate the table collection and to access data from parameter tables and interpolate data that were defined in property tables.

These utility routines can be invoked from any Abaqus/Standard or Abaqus/Explicit user subroutine except *UEXTERNALDB* when called at the start of the analysis.

### setTableCollection

Activate table collection, required in all user subroutines except *UMDFLUX*, *UEPACTIVATIONVOL*, *UELEMDRESP*, and material-related user subroutines.

#### Utility Routine Interface

```
Fortran:
character*80 tcName
...
call setTableCollection(tcName, jError)
or
C++:
#include <SMAAspUserSubroutines.h>

void setTableCollectionC(const char* tcName, int &jError);
```

#### Variables to Be Provided to the Utility Routine

**tcName** The name of the table collection that will be accessed.

#### Variables Returned from the Utility Routine

**jError** Return code (0 indicates no error, nonzero indicates an output request error).

### getParameterTable

Access parameters defined in the parameter table.

### Utility Routine Interface

```
Fortran:
include 'aba_tcs_param.inc'

character*80 parameterTableLabel, cParams(maxParams)
dimension iParamsDataType(maxParams), iParams(maxParams), rParams(maxParams)
...
call getParameterTable(parameterTableLabel, numParams, iParamsDataType,
iParams, rParams, cParams, jError)
or
C++:
#include <SMAAspUserSubroutines.h>
#include <aba_tcs_param.h>

void getParameterTableC(const char* parameterTableLabel, int &numParams,
int *iParamsDataType,
     int *iParams, double *rParams, std::vector<std::string> &cParams, int
 &jError);
```

### Variables to Be Provided to the Utility Routine

**parameterTableLabel**
Label of the parameter table to be accessed. A parameter table with this label must be defined in the active table collection.

### Variables Returned from the Utility Routine

**numParams**
Number of parameters defined in a row of this parameter table.

**iParamsDataType(numParams)**
Array containing integer flags for data types of parameters: `j_tcsI_PDT_int`, `j_tcsI_PDT_float`, or `j_tcsI_PDT_string`, which are defined in the `aba_tcs_param.inc` include file. `maxParams` must be equal to or greater than `numParams`.

**iParams(numParams*numRows)**
Array containing values of `INTEGER` parameters. `numRows` is the number of rows in the parameter table. You can get this value from the `queryParameterTable` utility. The parameters are ordered in row-by-row format.

**rParams(numParams*numRows)**
Array containing values of `FLOAT`-type parameters. The parameters are ordered in row-by-row format.

**cParams(numParams*numRows)**
Array containing values of `STRING`-type parameters. The parameters are ordered in row-by-row format.

**jError**
Return code (0 indicates no error, nonzero indicates an output request error).

## getParameterTableRow

Access a row of parameters defined in the parameter table.

### Utility Routine Interface

```
Fortran:
include 'aba_tcs_param.inc'

character*80 parameterTableLabel, cParams(maxParams)
dimension iParamsDataType(maxParams), iParams(maxParams), rParams(maxParams)

call getParameterTableRow(parameterTableLabel, jRow, numParams,
iParamsDataType, iParams,
rParams, cParams, jError)
or
C++:
#include <SMAAspUserSubroutines.h>
#include <aba_tcs_param.h>

void getParameterTableRowC(const char* parameterTableLabel, int jRow, int
&numParams, int *iParamsDataType,
     int *iParams, double *rParams, std::vector<std::string> &cParams, int
 &jError);
```

### Variables to Be Provided to the Utility Routine

**parameterTableLabel**
Label of the parameter table to be accessed. A parameter table with this label must be defined in the active table collection.

**jRow**
Row of the parameter table to be accessed.

### Variables Returned from the Utility Routine

**numParams**
Number of parameters defined in a row of this parameter table.

**iParamsDataType(numParams)**
Array containing integer flags for data types of parameters `j_tcsI_PDT_int`, `j_tcsI_PDT_float`, or `j_tcsI_PDT_string`, which are defined in the `aba_tcs_param.inc` include file. `maxParams` must be equal to or greater than `numParams`.

**iParams(numParams)**
Array containing values of `INTEGER` parameters in `jRow` row of this parameter table.

**rParams(numParams)**
Array containing values of `FLOAT`-type parameters in `jRow` row of this parameter table.

**cParams(numParams)**
Array containing values of `STRING`-type parameters in `jRow` row of this parameter table.

**jError** Return code (0 indicates no error, nonzero indicates an output request error).

## getPropertyTable

Perform interpolation of properties defined in the property table in Abaqus/Standard.

### Utility Routine Interface

```
Fortran:
character*80 propertyTableLabel
dimension field(*)
dimension rIndepVars(maxIndepVars), rProps(maxProps),
dPropDVar(maxProps*maxVars)
...
call getPropertyTable(propertyTableLabel, rIndepVars, temp, field,
                                    numProps, rProps, dPropDVar, numDervs,
 jError)
or
C++:
#include <SMAAspUserSubroutines.h>

void getPropertyTableC(const char* propertyTableLabel, double *rIndepVars,
 double *temp, double *field,
     int &numProps, double *rProps, double *dPropDVar, int &numDervs, int
&jError);
```

### Variables to Be Provided to the Utility Routine

**propertyTableLabel** Label of the property table to be accessed. A property table with this label must be defined in the active table collection.

**rIndepVars(maxIndepVars)** Array containing values of independent variables. `maxIndepVars` must be equal to or greater than the number of independent variables in the table.

**temp** Temperature.

**field(*)** Array of values of predefined field variables.

**numDervs** Flag to indicate if the calculation of derivatives is required (0 indicates that no calculation of derivatives is required, 1 indicates that calculation of derivatives is required with respect to the first variable, and 2 or higher indicates that calculation of derivatives is required with respect to all variables).

### Variables Returned from the Utility Routine

**numProps** Number of properties defined in this property table.

**rProps(maxProps)**
Array containing interpolated values of properties. `maxProps` must be equal to or greater than `numProps`.

**dPropDVar(maxProps*maxVars)**
Array containing derivatives of properties with respect to independent variables, temperature, or field variables. `maxVars` must be equal to or greater than the sum of independent variables, temperature, and field variables in the table. The derivatives are returned in the following order:

- Derivatives of the first property, second property, … `numProps` with respect to the first variable.
- Derivatives of the first property, second property, … `numProps` with respect to the second variable.
- …

**jError**
Return code (0 indicates no error, nonzero indicates an output request error).

## vGetPropertyTable

Perform interpolation of properties defined in the property table in Abaqus/Explicit.

### Utility Routine Interface

```
character*80 propertyTableLabel
dimension temp(nBlock), field(nBlock,*)
dimension rIndepVars(maxBlock*maxIndepVars), rProps(maxBlock*maxProps),
          dPropDIVar(maxBlock*maxProps*maxIndepVars),
dPropdTemp(maxBlock*maxProps)
...
call vGetPropertyTable(propertyTableLabel, nBlock, rIndepVars, temp, field,

                       numProps, rProps, dPropDIVar, dPropDTemp, jDervs,
jError)
```

### Variables to Be Provided to the Utility Routine

**propertyTableLabel**
Label of the property table to be accessed. A property table with this label must be defined in the active table collection.

**nBlock**
Number of points at which properties are evaluated in this call to `vGetPropertyTable`.

**rIndepVars(maxIndepVars)**
Array containing values of independent variables. `maxIndepVars` must be equal to or greater than the number of independent variables in the table.

**temp(nBlock)**
Temperature.

**field(nBlock,*)**
Array of values of predefined field variables.

**jDervs**

Flag to indicate if the calculation of derivatives is required (0 indicates that no calculation of derivatives is required, 1 indicates that calculation of derivatives is required with respect to all independent variables and temperature).

### Variables Returned from the Utility Routine

**numProps**

Number of properties defined in this property table.

**rProps(maxBlock*maxProps)**

Array containing interpolated values of properties. `maxBlock` must be equal to or greater than `nBlock`. `maxProps` must be equal to or greater than `numProps`.

**dPropDIVar(maxBlock*maxProps*maxIndepVars)**

Array containing derivatives of properties with respect to independent variables. The derivatives are returned in the following order:

- `nBlock` derivatives of the first property with respect to the first independent variable, `nBlock` derivatives of the first property with respect to the second independent variable, and so on.
- `nBlock` derivatives of the second property with respect to the first independent variable, `nBlock` derivatives of the second property with respect to the second independent variable, and so on.
- …

**dPropdTemp(maxBlock,maxProps)**

Array containing derivatives of properties with respect to temperature. The derivatives are returned in the following order:

- `nBlock` derivatives of the first property with respect to temperature.
- `nBlock` derivatives of the second property with respect to temperature.
- …

**jError**

Return code (0 indicates no error, nonzero indicates an output request error).

## vGetPropertyTablePtr

Perform interpolation of properties defined in the property table in Abaqus/Explicit. You pass independent variables as pointers, which offers better performance compared to `vGetPropertyTable` when you pass more than one independent variable.

### Utility Routine Interface

```
character*80 propertyTableLabel
integer*8 mIndepVars(maxIndepVars)
dimension temp(nBlock), field(nBlock,*)
dimension rProps(maxBlock*maxProps),
dPropDIVar(maxBlock*maxProps*maxIndepVars), dPropdTemp(maxBlock*maxProps)
...
call SMASetPointer(mIndVarPtrs(1), eqps)
call SMASetPointer(mIndVarPtrs(2), eqpsRate)
call vGetPropertyTablePtr(propertyTableLabel, nBlock, mIndepVars, temp,
field,
                     numProps, rProps, dPropDIVar, dPropDTemp, jDervs,
jError)
```

### Variables to Be Provided to the Utility Routine

**propertyTableLabel**
Label of the property table to be accessed. A property table with this label must be defined in the active table collection.

**nBlock**
Number of points at which properties are evaluated in this call to `vGetPropertyTablePtr`.

**mIndepVars(maxIndepVars)**
Array, declared as an 8-byte integer that contains pointers to arrays with independent variables initialized with the `SMASetPointer` utility routine. `maxIndepVars` must be equal to or greater than the number of independent variables in the table.

**temp(nBlock)**
Temperature.

**field(nBlock,*)**
Array of values of predefined field variables.

**jDervs**
Flag to indicate if the calculation of derivatives is required (0 indicates that no calculation of derivatives is required, 1 indicates that calculation of derivatives is required with respect to all independent variables and temperature).

### Variables Returned from the Utility Routine

**numProps**
Number of properties defined in this property table.

**rProps(maxBlock*maxProps)**
Array containing interpolated values of properties. `maxBlock` must be equal to or greater than `nBlock`. `maxProps` must be equal to or greater than `numProps`.

**dPropDIVar(maxBlock*maxProps*maxIndepVars)**

Array containing derivatives of properties with respect to independent variables. The derivatives are returned in the following order:

- `nBlock` derivatives of the first property with respect to the first independent variable, `nBlock` derivatives of the first property with respect to the second independent variable, and so on.
- `nBlock` derivatives of the second property with respect to the first independent variable, `nBlock` derivatives of the second property with respect to the second independent variable, and so on.
- …

**dPropdTemp(maxBlock*maxProps)**

Array containing derivatives of properties with respect to temperature. The derivatives are returned in the following order:

- `nBlock` derivatives of the first property with respect to temperature.
- `nBlock` derivatives of the second property with respect to temperature.
- …

**jError**

Return code (0 indicates no error, nonzero indicates an output request error).

## queryTableCollectionSize

Query the size of the table collection database.

### Utility Routine Interface

```
Fortran:
include 'aba_tcs_param.inc'

dimension jSize(n_tcsI_TCDB_size)
...
call queryTableCollectionSize(jSize, jError)
or
C++:
#include <SMAAspUserSubroutines.h>
#include <aba_tcs_param.h>

void queryTableCollectionSizeC(int *jSize, int &jError);
```

### Variables Returned from the Utility Routine

**jSize(i_tcsI_TCDB_numTCs)**

Number of table collections that are defined.

| | |
|---|---|
| **jSize(i_tcsI_TCDB_maxParams)** | Maximum number of parameters in any parameter table. |
| **jSize(i_tcsI_TCDB_maxParamRows)** | Maximum number of rows in any parameter table. |
| **jSize(i_tcsI_TCDB_maxProps)** | Maximum number of properties in any property table. |
| **jError** | Return code (0 indicates no error, nonzero indicates an output request error). |

## queryTableCollectionNames

Query the names of the table collections.

### Utility Routine Interface

```
Fortran:
character*80 tcNames(maxTCs)
...
call queryTableCollectionNames(tcNames, numTCs)
or
C++:
#include <SMAAspUserSubroutines.h>

void queryTableCollectionNamesC(std::vector<std::string> &tcNames, int
&numTCs);
```

### Variables Returned from the Utility Routine

| | |
|---|---|
| **tcNames(maxTCs)** | Array containing names of all defined table collections. `maxTCs` must be equal to or greater than `numTCs`. |
| **numTCs** | Number of table collections. |

## queryTableCollection

Query properties of the active table collection.

### Utility Routine Interface

```
Fortran:
include 'aba_tcs_param.inc'

character*80 cTableColl(n_tcsC_TC_size)
dimension jTableColl(n_tcsI_TC_size)
...
call queryTableCollection(jTableColl, cTableColl, jError)
or
C++:
#include <SMAAspUserSubroutines.h>
#include <aba_tcs_param.h>
```

```
void queryTableCollectionC(int *jTableColl, std::vector<std::string>
&cTableColl,
     int &jError);
```

### Variables Returned from the Utility Routine

**jTableColl(i_tcsI_TC_numParamTbls)** Number of parameter tables in the active table collection.

**jTableColl(i_tcsI_TC_numPropTbls)** Number of property tables in the active table collection.

**jTableColl(i_tcsI_TC_maxParams)** Maximum number of parameters in any parameter table in the active table collection.

**jTableColl(i_tcsI_TC_maxParamRows)** Maximum number of rows in any parameter table in the active table collection.

**jTableColl(i_tcsI_TC_maxProps)** Maximum number of properties in any property table in the active table collection.

**cTableColl(i_tcsC_TC_name)** Name of the active table collection.

**jError** Return code (0 indicates no error, nonzero indicates an output request error).

## queryParameterTable

Query the number of parameters in a parameter table.

### Utility Routine Interface

```
Fortran:
character*80 parameterTableLabel
...
call queryParameterTable(parameterTableLabel, numParams, numRows, jError)
or
C++:
#include <SMAAspUserSubroutines.h>

void queryParameterTableC(const char *parameterTableLabel, int &numParams,

     int &numRows, int &jError);
```

### Variables to Be Provided to the Utility Routine

**parameterTableLabel** Label of the parameter table to be queried. A parameter table with this label must be defined in the active table collection.

### Variables Returned from the Utility Routine

**numParams** Number of parameters in the parameter table.

**numRows** Number of rows in the parameter table, which is a multiple of the parameter number declared for the parameter table type of this table.

**jError** Return code (0 indicates no error, nonzero indicates an output request error).

## queryPropertyTable

Query properties of a property table.

### Utility Routine Interface

```
Fortran:
include 'aba_tcs_param.inc'

character*80 propertyTableLabel, cPropTable(n_tcsC_PRT_size)
dimension jPropTable(n_tcsI_PRT_size), rPropTable(n_tcsR_PRT_size)
...
call queryPropertyTable(propertyTableLabel, jPropTable, rPropTable,
cPropTable, jError)
or
C++:
#include <SMAAspUserSubroutines.h>
#include <aba_tcs_param.h>

void queryPropertyTableC(const char *propertyTableLabel, int *jPropTable,
double *rPropTable,
     std::vector<std::string> &cPropTable, int &jError);
```

### Variables to Be Provided to the Utility Routine

**propertyTableLabel** Label of the property table to be queried. A property table with this label must be defined in the active table collection.

### Variables Returned from the Utility Routine

**jPropTable(i_tcsI_PRT_numProps)** Number of properties in the property table.

**jPropTable(i_tcsI_PRT_numIndVars)** Number of independent variables in the property table.

**jPropTable(i_tcsI_PRT_numFieldVars)** Number of field variables in the property table.

**jPropTable(i_tcsI_PRT_tempDep)** 0 indicates that properties in the property table are not temperature dependent, 1 indicates that properties are temperature dependent.

| | |
|---|---|
| **jPropTable(i_tcsI_PRT_extrapolation)** | 0 indicates constant extrapolation of properties in the property table, 1 indicates linear extrapolation of properties. |
| **jPropTable(i_tcsI_PRT_regularize)** | 0 indicates that property table data is not regularized, 1 indicates that property table data is regularized. |
| **jPropTable(i_tcsI_PRT_origin)** | 0 indicates that property values at point 0 of the first independent variable might be not preserved, 1 indicates that property values at point 0 of the first independent variable are preserved. |
| **rPropTable(i_tcsR_PRT_rTol)** | Tolerance used to regularize the property table data. |
| **jError** | Return code (0 indicates no error, nonzero indicates an output request error). |

# Accessing Abaqus Event Series Data

**Products:** Abaqus/Standard

**References:**

- **EVENT SERIES TYPE*
- **EVENT SERIES*
- *About Event Series*

## Overview

These utility routines can be used to access event series data from Abaqus/Standard user subroutines.

These utility routines can be invoked from Abaqus/Standard user subroutines *UMDFLUX* and *UEPACTIVATIONVOL* to access a slice of event series data. A slice is a series of events from a current time increment. You can query the properties of event series (`getEventSeriesSliceProperties`), get the sequence of all events in the slice (`getEventSeriesSliceLG`), or get the list of segments that are within the provided distance to a center point (`getEventSeriesSliceLGLocationPath`).

### `getEventSeriesSliceProperties`

Query properties of a slice of the event series data.

#### Utility Routine Interface

```
include 'aba_evs_param.inc'

character*80 evsName, cProps(n_evsC_size)
dimension iProps(n_evsI_size), rProps(n_evsR_size)
...
call getEventSeriesSliceProperties(evsName, iProps, rProps, cProps)
```

#### Variables to Be Provided to the Utility Routine

**`evsName`** The name of the event series that will be queried.

#### Variables Returned from the Utility Routine

**`iProps(i_evsI_numFields)`** The number of fields in the event series.

**`iProps(i_evsI_numEvents)`** The number of events in the slice of the event series.

**`iProps(i_evsI_time)`** 1 for STEP TIME, 2 for TOTAL TIME.

**`rProps(i_evsR_tStart)`** Time of the first event in the slice.

**`rProps(i_evsR_tEnd)`** Time of the last event in the slice.

### getEventSeriesSliceLG

Get the sequence of all events in the slice.

#### Utility Routine Interface

```
include 'aba_evs_param.inc'

character*80 evsName
dimension  evsTime(maxEvents), evsCoord(3,maxEvents),
evsField(numFields,maxEvents)
...
call getEventSeriesSliceLG(evsName, numEvsPoints, evsTime, evsCoord,
evsField)
```

#### Variables to Be Provided to the Utility Routine

**evsName**
The name of the event series that will be accessed.

#### Variables Returned from the Utility Routine

**numEvents**
The number of events in the slice of event series data.

**evsTime(maxEvents)**
Array containing time of events in the slice. `maxEvents` must be equal to or greater than `numEvents`.

**evsCoord(3,maxEvents)**
Array containing *X*-, *Y*-, and *Z*-coordinates of locations of events.

**evsField(numFields,maxEvents)**
Array containing field values of events. `numFields` is the number of fields in the event series.

### getEventSeriesSliceLGLocationPath

Get the list of path segments of the slice events that are within the provided distance to a center point.

#### Utility Routine Interface

```
include 'aba_evs_param.inc'

character*80 evsName
dimension  xCenter(3), evsPathTime(2,maxEvents), evsPathCoord(3,2,maxEvents),

 evsPathField(nFieldSize,2,maxEvents)
...
call getEventSeriesSliceLGLocationPath(evsName, xCenter, radius,
nFieldSize, numSegments, evsPathTime, evsPathCoord, evsPathField)
```

#### Variables to Be Provided to the Utility Routine

**evsName**
The name of the event series that will be accessed.

**xCenter(3)** Coordinates of the center point.

**radius** Distance to the center point.

**nFieldSize** Leading dimension of `evsPathValue`. Must be equal to or greater than the number of fields in the event series.

### Variables Returned from the Utility Routine

**numSegments** The number of path segments of the slice events that are within the provided distance to a center point.

**evsPathTime(2,maxSegments)** Array containing the start and end time of event segments. `maxSegments` must be equal to or greater than `numSegments`.

**evsPathCoord(3,2,maxEvents)** Array containing *X*-, *Y*-, and *Z*-coordinates at the start and end of locations of event segments.

**evsPathField(nFieldSize,2,maxEvents)** Array containing the start and end field values of event segments.

# Evaluating Abaqus Neural Networks

**Products:** Abaqus/Standard

**References:**

- *About Neural Networks*
- **NEURAL NETWORK*
- **NEURAL NETWORK DATA*

## Overview

These utility routines can be used to evaluate defined neural networks from Abaqus/Standard user subroutines.

These utility routines can be invoked from any Abaqus/Standard user subroutine to evaluate the output and derivatives of a feed-forward neural network that was defined in the input file.

### `nntk_ffd_eval_func`

Evaluate the output of a defined feed-forward neural network.

#### Utility Routine Interface

```
Fortran usage:

include 'aba_nntk.hdr'

integer iRetCode
...
iRetCode =  nntk_ffd_eval_func ( nnet_id, x_vals, y_vals )

or C/C++ usage:

#include <aba_nntk.h>

int nntk_ffd_eval_funcC(const int& nnet_id, const double* x_vals, double*
y_vals);
```

#### Variables to Be Provided to the Utility Routine

**`nnet_id`**

The integer identifier of the neural network to be evaluated.

**`x_vals`**

Array containing the input values $(X_1, X_2, \ldots, X_N)$ at which the neural network will be evaluated. The size of this input array is assumed to be the same as the neural network input size $(N)$ specified in the input file.

#### Variables Returned from the Utility Routine

**`y_vals`**

Array in which the neural network output values $(Y_1, Y_2, \ldots, Y_M)$ will be stored. The size of this output array is assumed to be the same as the neural network output size $(M)$ specified in the input file.

**iRetCode**
Integer return code value. The evaluation utility returns 0 if no errors are detected; otherwise, it returns a nonzero value.

## nntk_ffd_eval_deriv

Evaluate the Jacobian matrix (derivatives of the outputs with respect to inputs) for a defined feed-forward neural network.

### Utility Routine Interface

```
Fortran usage:

include 'aba_nntk.hdr'

integer iRetCode
...
iRetCode =  nntk_ffd_eval_deriv ( nnet_id, x_vals, jac_vals )

or C/C++ usage:

#include <aba_nntk.h>

int nntk_ffd_eval_derivC(const int& nnet_id, const double* x_vals, double*
 jac_vals);
```

### Variables to Be Provided to the Utility Routine

**nnet_id**
The integer identifier of the neural network to be evaluated.

**x_vals**
Array containing the input values $(X_1, X_2, \ldots, X_N)$ at which the Jacobian matrix of the neural network will be evaluated. The size of this input array is assumed to be the same as the neural network input size $(N)$ specified in the input file.

### Variables Returned from the Utility Routine

**jac_vals**
Array in which the Jacobian matrix entries will be stored. The size of this output array should be equal to the produc neural network output size $(M)$ and the neural network input size $(N)$. The Jacobian matrix entries are written in the fo order:

$\partial Y_1/\partial X_1, \partial Y_1/\partial X_2, \ldots, \partial Y_1/\partial X_N, \partial Y_2/\partial X_1, \partial Y_2/\partial X_2, \ldots, \partial Y_2/\partial X_N, \ldots, \partial Y_M/\partial X_1, \partial Y_M/\partial X_2, \ldots, \partial$

**iRetCode**
Integer valued return code. The evaluation utility returns 0 if no errors are detected; otherwise, it returns a nonzero v

## nntk_ffd_eval_func_deriv

Evaluate the output and the Jacobian matrix (derivatives of the outputs with respect to inputs) for a defined feed-forward neural network.

### Utility Routine Interface

```
Fortran usage:

include 'aba_nntk.hdr'

integer iRetCode
...
iRetCode =  nntk_ffd_eval_func_deriv ( nnet_id, x_vals, y_vals, jac_vals )


or C/C++ usage:

#include <aba_nntk.h>

int nntk_ffd_eval_func_derivC(const int& nnet_id, const double* x_vals,
double* y_vals, double* jac_vals);
```

### Variables to Be Provided to the Utility Routine

**nnet_id**
The integer identifier of the neural network to be evaluated.

**x_vals**
Array containing the input values $(X_1, X_2, \ldots, X_N)$ at which the output and the Jacobian matrix of the neural network will be evaluated. The size of this input array is assumed to be the same as the neural network input size $(N)$ specified in the input file.

### Variables Returned from the Utility Routine

**y_vals**
Array in which the neural network output values $(Y_1, Y_2, \ldots, Y_M)$ will be stored. The size of this output array is as to be the same as the neural network output size $(M)$ specified in the input file.

**jac_vals**
Array in which the Jacobian matrix entries will be stored. The size of this output array should be equal to the produc neural network output size $(M)$and the neural network input size $(N)$. The Jacobian matrix entries are written in the fo order:

$\partial Y_1/\partial X_1, \partial Y_1/\partial X_2, \ldots, \partial Y_1/\partial X_N, \partial Y_2/\partial X_1, \partial Y_2/\partial X_2, \ldots, \partial Y_2/\partial X_N, \ldots, \partial Y_M/\partial X_1, \partial Y_M/\partial X_2, \ldots,$

**iRetCode**
Integer valued return code. The evaluation utility returns 0 if no errors were detected; otherwise, it returns a nonzero

## History Data

Utility routines to access history data are available to assist in the coding of user subroutines.

### In this section:


- *Setting a User-Defined Value*
- *Obtaining Sensor Information*

## Setting a User-Defined Value

**Products:** Abaqus/Standard

**References:**

## Overview

Utility routine SETALLUSER can be called from within Abaqus/Standard user subroutines used in heat transfer analyses to define a user value that is accumulated incrementally during the analysis.
The accumulated value can be requested as output variable ALLUSER.

### Utility Routine Interface

```
      CALL SETALLUSER(userValue)
```

### Variables to Be Provided to the Utility Routine

**userValue** User-defined value to be accumulated with each utility call.

# Obtaining Sensor Information

**Products:** Abaqus/Standard Abaqus/Explicit

### References:

- *UAMP*
- *VUAMP*
- *Crank mechanism*

## Overview

Given the user-defined name for a sensor, utility routines can be used to obtain the sensor ID or the sensor value using a computationally efficient searching technique.

Utility routines `IGETSENSORID` and `GETSENSORVALUE` can be called only from Abaqus/Standard user subroutine *UAMP*. Utility routines `IVGETSENSORID` and `VGETSENSORVALUE` can be called only from Abaqus/Explicit user subroutine *VUAMP*.

### Utility Routine Interface

```
      character*80 mySensorName
      ...

      iMySensorID =  IGETSENSORID(mySensorName, jSensorLookUpTable )
      iMySensorID = IVGETSENSORID(mySensorName, jSensorLookUpTable )
      dMySensorValue = sensorValues(iMySensorID)
      ...
      dMySensorValue =   GETSENSORVALUE(mySensorName,
C                        jSensorLookUpTable, sensorValues )
      dMySensorValue =  VGETSENSORVALUE(mySensorName,
C                       jSensorLookUpTable, sensorValues )
      ...
```

### Variables to Be Provided to the Utility Routine

**`mySensorName`**
User-defined character string, uppercase, left justified.

**`jSensorLookUpTable`**
Pointer to an object containing a binary tree look up table for sensors. The calling user subroutine provides this variable.

**`sensorValues`**
Array containing the latest sensor values for all sensors in the model.

### Variables Returned from the Utility Routine

**`iMySensorID`**
Index in the `sensorValues` array for this sensor name.

**`dMySensorValue`**
Sensor value for this sensor name.

## Element Data

Utility routines to access element data are available to assist in the coding of user subroutines.

### In this section:


- *Obtaining Material Point Information from User Subroutine UVAREL*
- *Obtaining the Element Orientation Name from User Subroutine UVAREL*
- *Obtaining the Material Name from User Subroutine UVAREL*
- *Obtaining Element Section Information from User Subroutine UVAREL*
- *Obtaining Wave Kinematic Data in an Abaqus/Aqua Analysis*

# Obtaining Material Point Information from User Subroutine UVAREL

**Products:** Abaqus/Standard

**References:**

- *UVAREL*
- *Using Abaqus/Standard Output Variable Identifiers*

## Overview

Utility routine getElMPtVar can be called from user subroutine UVAREL to obtain material point information within an element.

### Utility Routine Interface

```
Fortran:
        character*3 flgray(15)
        dimension array(15),jarray(15)

        call getElMPtVar('Var', kPt, kLayer, kSpt, array, jArray, flgray,
jError, abqInternal)
or
C++:
        #include <SMAAspUserSubroutines.h>

        void getElMPtVarC(const char* Var, int kPt, int kLayer, int kSpt,
double* array, int* jArray,
                std::vector<std::string> &flgray, int &jError, double*
abqInternal);
```

### Variables to Be Provided to the Utility Routine

**Var**
Output variable key from the table in *Element Integration Point Variables*. The applicable keys are listed in the output table as being available for output at the element integration points; for example, S for stress.

**kPt**
Integration point number.

**kLayer**
Composite layer number.

**kSpt**
Section point number within the current layer.

**abqInternal**
Variable that must be passed into utility routine getElMPtVar. The calling user subroutine provides this variable.

### Variables Returned from the Utility Routine

**array**
Real array containing individual components of the output variable.

**jArray** Integer array containing individual components of the output variable.

**flgray** Character array containing flags corresponding to the individual components. Flags will contain either YES, NO, or N/A (not applicable).

**jError** Return code (0 – no error, 1 – output request error or all components of the output request are zero).

## Available Output Variable Keys

Only output variable keys that are valid for material point output are available for use with `getElMPtVar`. In general, if a key corresponds to a collective output variable (rather than to an individual component), it can be used with `getElMPtVar`. For example, S for the stress tensor can be used, but any individual component of stress (for example, S11) cannot be used. The collective output variable keys stand out from their individual components because they include the word "yes" after the category .fil in the tables in *Using Abaqus/Standard Output Variable Identifiers*. *Supported Element Types and Output Variable Keys* lists output variable keys that can and cannot be used with `getElMPtVar`.

Utility routine `getElMPtVar` returns `array`, `jArray`, and `flgray`, which correspond to the real-valued components, integer-valued components, and the flags associated with the request `Var`, respectively. If any array component is not applicable for a given request, its value is returned as the initialized value: 0.0 in `array`, 0 in `jArray`, and N/A in `flgray`. `getElMPtVar` returns the error flag `jError`=1 any time a request key is not recognized, the request is not valid (such as requesting transverse shear stress for a shell element that uses thin shell theory), or all the output components requested are zero; otherwise, `jError`=0.

## Ordering of Returned Components

The components for a request are written as follows:

- Single index components (and requests without components) are returned in positions 1, 2, 3, etc.
- Double index components are returned in the order 11, 22, 33, 12, 13, 23 for symmetric tensors, followed by 21, 31, 32 for unsymmetric tensors (deformation gradient).

Therefore, the stresses for a plane stress element are returned as `array(1)`=S11, `array(2)`=S22, `array(3)`=0.0, and `array(4)`=S12. Three values are always returned for principal value requests, with the minimum value first and the maximum value third, regardless of the dimensionality of the analysis.

The description of the output variable (see *Abaqus/Standard Output Variable Identifiers*) determines which components are retrieved with `getElMPtVar`.

## Analysis Time for which Values Are Returned

When you request a material point quantity with the utility routine, `getElMPtVar` always returns the requested quantities with the values at the end of the current increment.

## Accessing State-Dependent Variables

If you use `getElMPtVar` to access state-dependent variables (output variable key SDV) and more than 15 state-dependent variables have been defined in the analysis, you must change the dimension statement for `array` and `jArray` so that you dimension these arrays to the maximum number of state-dependent variables.

## Supported Element Types and Output Variable Keys

This utility routine can be called from user subroutine *UVAREL*, which is supported only for continuum solid, solid shell, conventional shell, continuum shell, and beam elements. All other element types are unsupported.

You cannot use output variable SVOL with `getElMPtVar`.

# Obtaining the Element Orientation Name from User Subroutine UVAREL

**Products:** Abaqus/Standard

**References:**

- *UVAREL*

## Overview

Utility routine `getUVarElOrientationName` can be called from user subroutine `UVAREL` to obtain the name of the element orientation. In the case of elements with composite sections, the utility routine will return the section orientation associated with the element.

### Utility Routine Interface

```
character*80 cOriName
...
call getUVarElOrientationName(kLayer, cOriName, jError, abqInternal)
...
```

### Variables to Be Provided to the Utility Routine

**kLayer**
Composite layer number. 1 for elements without a composite section (currently not used).

**abqInternal**
Variable that must be passed into the `getUVarElOrientationName` utility routine. The calling user subroutine provides this variable.

### Variables Returned from the Utility Routine

**cOriName**
Character string with the name of the orientation.

**jError**
Return code (0 – no error, 1 – error).

# Obtaining the Material Name from User Subroutine UVAREL

**Products:** Abaqus/Standard

**References:**

- *UVAREL*

## Overview

Utility routine getUVarElMaterialName can be called from user subroutine UVAREL to obtain the name of the material.
In the case of elements with composite sections, the utility routine can be called to obtain the material name for each layer.

### Utility Routine Interface

```
character*80 cMatName
...
call getUVarElMaterialName(kLayer, cMatName, jError, abqInternal)
...
```

### Variables to Be Provided to the Utility Routine

**kLayer**
Composite layer number. 1 for elements without a composite section.

**abqInternal**
Variable that must be passed into the getUVarElMaterialName utility routine. The calling user subroutine provides this variable.

### Variables Returned from the Utility Routine

**cMatName**
Character string with the name of the material.

**jError**
Return code (0 – no error, 1 – error).

# Obtaining Element Section Information from User Subroutine UVAREL

**Products:** Abaqus/Standard

**References:**

- *UVAREL*
- *Using Abaqus/Standard Output Variable Identifiers*

## Overview

Utility routine getSecVar can be called from user subroutine UVAREL to access element section information.

### Utility Routine Interface

```
Fortran:
        character*3 flgray(15)
        dimension array(15),jarray(15)

        call getSecVar('Var', kPt, array, jArray, flgray, jError,
abqInternal)
or
C++:
        #include <SMAAspUserSubroutines.h>

       void getSecVarC(const char* Var, int kPt, double* array, int* jArray,

                        std::vector<std::string> &flgray, int &jError,
double* abqInternal);
```

### Variables to Be Provided to the Utility Routine

**Var**
Output variable key from the table in *Element Section Variables*. The applicable keys are listed in the output table as the element section variables; for example, SF for section force.

**kPt**
Integration point number.

**abqInternal**
Variable that must be passed into utility routine getSecVar. The calling user subroutine provides this variable.

### Variables Returned from the Utility Routine

**array**
Real array containing individual components of the output variable.

**jArray**
Integer array containing individual components of the output variable.

**flgray**
Character array containing flags corresponding to the individual components. Flags will contain either YES, NO, or N/A (not applicable).

**jError** Return code (0 – no error, 1 – output request error or all components of the output request are zero).

## Available Output Variable Keys

Only output variable keys that are valid for output are available for use with `getSecVar`. In general, if a key corresponds to a collective output variable (rather than to an individual component), it can be used with `getSecVar`. For example, SF for the section force can be used, but any individual component of section force (for example, SF1) cannot be used. The collective output variable keys are distinguished from their individual components by the fact that they have the word "yes" after the category .fil in the tables in *Using Abaqus/Standard Output Variable Identifiers*.

Utility routine `getSecVar` returns `array`, `jArray`, and `flgray`, which correspond to the real-valued components, integer-valued components, and the flags associated with the request `Var`, respectively. If any array component is not applicable for a given request, its value is returned as the initialized value: 0.0 in `array`, 0 in `jArray`, and N/A in `flgray`. `getSecVar` returns the error flag `jError`=1 from `getSecVar` any time a request key is not recognized, the request is not valid (such as requesting transverse shear stress for a shell element that uses thin shell theory), or all the output components requested are zero; otherwise, `jError`=0.

The following output variable keys are available for use with `getSecVar`:

- COORD
- SF
- SSAVG
- STH

## Analysis Time for which Values Are Returned

When you request a section output quantity with the utility routine, `getSecVar` always returns the requested quantities with the values at the end of the current increment.

## Supported Element Types

This capability pertains to section quantities; therefore, you cannot use it for most of the element types that do not require section integration.

Utility routine `getSecVar` supports only the following element types:

- Membrane elements
- Shell elements
- Beam elements

# Obtaining Wave Kinematic Data in an Abaqus/Aqua Analysis

**Products:** Abaqus/Aqua

**References:**

- *UEL*
- *UWAVE*
- *Abaqus/Aqua Analysis*
- **AQUA*
- *UEL*

## Overview

Utility routines `GETWAVE`, `GETWAVEVEL`, `GETWINDVEL`, and `GETCURRVEL` are provided to access the fluid kinematic data for an Abaqus/Aqua analysis.

These routines can be used only from within user subroutine *UEL*.

### `GETWAVE` (Get Wave Kinematics)

#### Utility Routine Interface

```
      PARAMETER(MWCOMP=number of wave components)
      DIMENSION WAMP(MWCOMP),WPERD(MWCOMP),WXLAMB(MWCOMP),
     1 WPHI(MWCOMP),WOFF(3),WANG(2,MWCOMP)
      ...
      CALL GETWAVE(MWCOMP,NWCOMP,WAMP,WPERD,WXLAMB,WPHI,WOFF,WANG,
     1 ELEVB,ELEVS,JWTYPE,JRCD)
```

#### Variables Returned from the Utility Routine

**`NWCOMP`**
Number of wave components (always 1 for Stokes wave theory).

**`WAMP`**
Array containing the amplitude of the wave components.

**`WPERD`**
Array containing the period of the wave components.

**`WXLAMB`**
Array containing the wavelength of the wave components.

**`WPHI`**
Array containing the phase angle of the wave components.

**`WOFF`**
Used only for gridded wave data (`JWTYPE`=2), when `WOFF` gives the position of the origin of the gridded coordinate system with respect to the global system.

**`WANG(2,*)`**
For Stokes fifth-order wave theory `WANG(1,1)` and `WANG(2,1)` are the direction cosines of wave travel. For Airy wave theory `WANG(1,K1)` and `WANG(2,K1)` are the direction cosines of the direction of travel of the `K1`th wave. For gridded wave data `WANG(1,1)` and `WANG(2,1)` are the direction cosines of the wave data grid. In all cases these direction cosines are with respect to the global coordinate system.

**ELEVB**

User-defined elevation of the seabed.

**ELEVS**

User-defined elevation of the still water surface.

**JWTYPE**

Integer flag indicating the wave type, as follows:

| | |
|---|---|
| JWTYPE=0 | Airy wave theory |
| JWTYPE=1 | Stokes fifth-order wave theory |
| JWTYPE=2 | Wave data obtained from gridded values |

**JRCD**

The error flag JRCD is returned from GETWAVE as 0 if all the wave kinematic data are read correctly and as −1 if an error occurred (for instance, NWCOMP is greater that MWCOMP).

## GETWAVEVEL, GETWINDVEL, and GETCURRVEL (Get Wave, Wind, and Current Velocities)

### Utility Routine Interface

```
      CALL GETWAVEVEL (NDIM, X, V, A, LERROR, NOEL, XINTERMED)
      CALL GETWINDVEL (NDIM, X, V, NOEL, XINTERMED)
      CALL GETCURRVEL (NDIM, X, V, NOEL, XINTERMED)
```

### Variables to Be Provided to the Utility Routine

**NDIM**

Dimensionality of the element. It should be set to 2 for two-dimensional cases (for example, beams in a plane) and 3 for three-dimensional cases (for example, beams in space).

**X(1..NDIM)**

Global coordinates of the point.

### Variables Returned from the Utility Routine

**V(1..NDIM)**

Velocity components in the global coordinate system.

**A(1..NDIM)**

Wave acceleration components in the global coordinate system. This variable is returned by GETWAVEVEL only.

**LERROR**

For gridded wave data LERROR is returned as 0 if the current point is within the grid or above the crest; it is returned as 1 if the point is outside the bounds of the grid. For Airy and Stokes waves LERROR is always returned as 0. If LERROR is returned as 1, the global coordinates of the nearest grid point are returned in X. LERROR is returned by GETWAVEVEL only.

**NOEL**

Element number.

**XINTERMED(NDIM)** An array containing the intermediate configuration coordinates of the load integration point. For nonstochastic analysis this array is not used. In a stochastic analysis the wave field is based upon this configuration. Additional details are found in *UWAVE*.

## Node Data

Utility routines to access node data are available to assist in the coding of user subroutines.

### In this section:


- *Obtaining Node Point Information*
- *Obtaining Node to Element Connectivity*

# Obtaining Node Point Information

**Products:** Abaqus/Standard

**References:**

- *UMESHMOTION*
- *Erosion of material (sand production) in an oil wellbore*

## Overview

Utility routine GETVRN can be called from user subroutine UMESHMOTION to access node point information.

### Utility Routine Interface

```
DIMENSION ARRAY(15),JGVBLOCK(*)
...
CALL GETVRN(NODE,'VAR',ARRAY,JRCD,JGVBLOCK,LTRN)
```

### Variables to Be Provided to the Utility Routine

**NODE** Node number.

**VAR** Output variable key from the table in *Nodal Variables*. The applicable keys are listed in the output table as being available for results file output at nodes; e.g., U for displacement.

**JGVBLOCK** Variable that must be passed into the GETVRN utility routine. The variable is available in user subroutine *UMESHMOTION* for this purpose.

**LTRN** Variable indicating the coordinate system the nodal quantity is to be returned in. A value of 0 specifies that the results are to be returned in the global coordinate system, regardless of any transformation applied at the node. A value of 1 specifies that the results are to be returned in the local transformed system.

### Variables Returned from the Utility Routine

**ARRAY** Real array containing individual components of the output variable.

**JRCD** Return code (0 – no error, 1 – output request error or all components of output request are zero).

### Available Output Variable Keys

Only nodal output variable keys that are valid for results file output in the current step are available for use with GETVRN. In general, if a key corresponds to a collective output variable, rather than an individual component, it can be used with GETVRN. For example, U for displacement can be used, whereas any individual component of displacement, say U1, cannot be used. The collective output variable keys are distinguished from their individual components by the fact that they have the word "yes" after the category **.fil** in the tables in *Nodal Variables*.

You will be returned ARRAY, which corresponds to the real-valued components associated with the request VAR. If any array component is not applicable for a given request, its value will be returned as the initialized value: 0.0. The error flag JRCD=1 is returned from GETVRN any time a request key is not recognized, the request is not valid (such as requesting pore pressure for a node not associated with a pore pressure or acoustic element, or requesting a variable not available for the current procedure), or all of the output components requested are zero; otherwise, JRCD=0.

### Ordering of Returned Components

The components for a vector request are returned in positions 1, 2, 3, etc.

### Analysis Time for Which Values Are Returned

GETVRN returns values at the end of the current increment.

# Obtaining Node to Element Connectivity

**Products:** Abaqus/Standard

**References:**

- *UMESHMOTION*
- *Obtaining Material Point Information Averaged at a Node*
- *Erosion of material (sand production) in an oil wellbore*

## Overview

Utility routine GETNODETOELEMCONN can be called from user subroutine UMESHMOTION to retrieve a list of elements connected to a specified node.

### Utility Routine Interface

```
PARAMETER ( MAXNELEMS = 100 )
DIMENSION JELEMLIST(MAXNELEMS),JELEMTYPE(MAXNELEMS),JGVBLOCK(*)
...
NELEMS = MAXNELEMS
CALL GETNODETOELEMCONN(NODE, NELEMS, JELEMLIST, JELEMTYPE,
     JRCD, JGVBLOCK)
```

### Variables to Be Provided to the Utility Routine

**NODE**
User node number.

**NELEMS**
You must set NELEMS to the maximum allowable length of the JELEMLIST and JELEMTYPE arrays. This value corresponds to the maximum expected number of elements attached to an adaptive mesh constraint node in your model. GETNODETOELEMCONN will assume that your JELEMLIST and JELEMTYPE arrays are NELEMS long. In the event that the actual element connectivity exceeds NELEMS, no result will be returned and the return code JRCD will indicate an error. An NELEMS value of 100 is typically more than adequate for common meshes. NELEMS is modified by GETNODETOELEMCONN and should not be a Fortran parameter-statement constant.

**JGVBLOCK**
Variable that must be passed into the GETNODETOELEMCONN utility routine. This variable is available in user subroutine *UMESHMOTION* for this purpose.

### Variables Returned from the Utility Routine

**JELEMLIST**
Array of element numbers for elements connected to NODE. The list will contain elements only in adaptive mesh domains active in the step as well as any contact elements associated with the domain. The number of entries in this array corresponds to the returned value of NELEMS.

**JELEMTYPE** Array of element type designators describing the element types corresponding to each element entry in JELEMLIST. The number of entries in this array corresponds to the returned value of NELEMS.

JELEMTYPE entries:

1 indicates a solid element.

2 indicates a contact element.

**NELEMS** Actual length of the JELEMLIST and JELEMTYPE arrays.

**JRCD** Return code (0 indicates no error, 1 indicates an output request error). An output request error indicates either that the requested variable is not available or that your NELEMS parameter setting is smaller than the element connectivity list at this node.

# Material Point Data

Utility routines to access material point data are available to assist in the coding of user subroutines.

### In this section:


- *Obtaining Material Point Data in Abaqus/Standard*
- *Obtaining Material Point Data in Abaqus/Explicit*

## Obtaining Material Point Data in Abaqus/Standard

Utility routines to access material point data are available to assist in the coding of user subroutines.

### In this section:

- *Obtaining Material Point Information in an Abaqus/Standard Analysis*
- *Decoding Material and Orientation Names*
- *Obtaining Constituent-Level Material Point Information*
- *Obtaining Material Point Information Averaged at a Node*
- *Obtaining Stress Invariants, Principal Stress/Strain Values and Directions, and Rotating Tensors*
- *Accessing Abaqus Materials*
- *Accessing Abaqus Thermal Materials*

## Obtaining Material Point Information in an Abaqus/Standard Analysis

**Products:** Abaqus/Standard

**References:**

- *UVARM*
- *USDFLD*
- *UDMGINI*
- *Using Abaqus/Standard Output Variable Identifiers*
- *Damage and failure of a laminated composite plate*
- *USDFLD*
- *UVARM*

## Overview

Utility routine GETVRM can be called from either user subroutine UVARM, UDMGINI, or USDFLD to access material integration point information.

### Utility Routine Interface

```
Fortran:
         DIMENSION ARRAY(15), JARRAY(15)
         CHARACTER*3 FLGRAY(15)

         CALL GETVRM('VAR',ARRAY,JARRAY,FLGRAY,JRCD,JMAC,JMATYP,MATLAYO,
                     LACCFLA)
or
C++:
        #include <SMAAspUserSubroutines.h>

        void getVrmC(const char* var, double* array, int* jarray,
std::vector<std::string> &flgray,
          int &jrcd, int* jmac, int* jmtyp, int* matlayo, int* laccfla);
```

### Variables to Be Provided to the Utility Routine

**VAR**
Output variable key from the table in *Element Integration Point Variables*. The applicable keys are listed in the output table as being available for results file output at the element integration points; for example, S for stress.

**JMAC**
Variable that must be passed into the GETVRM utility routine. The calling user subroutine provides this variable.

**JMATYP**
Variable that must be passed into the GETVRM utility routine. The calling user subroutine provides this variable.

**MATLAYO**
Variable that must be passed into the GETVRM utility routine. The calling user subroutine provides this variable.

**LACCFLA** Variable that must be passed into the GETVRM utility routine. The calling user subroutine provides this variable.

### Variables Returned from the Utility Routine

**ARRAY** Real array containing individual components of the output variable.

**JARRAY** Integer array containing individual components of the output variable.

**FLGRAY** Character array containing flags corresponding to the individual components. Flags will contain either YES, NO, or N/A (not applicable).

**JRCD** Return code (0 – no error, 1 – output request error or all components of the output request are zero).

### Available Output Variable Keys

Only output variable keys that are valid for results file output are available for use with GETVRM. In a static perturbation procedure, only those output variables for which perturbation values are available can be accessed. In general, if a key corresponds to a collective output variable, rather than an individual component, it can be used with GETVRM. For example, S for the stress tensor can be used, whereas any individual component of stress, say S11, cannot be used. The collective output variable keys are distinguished from their individual components in that they have the word "yes" after the category **.fil** in the tables in *Using Abaqus/Standard Output Variable Identifiers*. Output variable keys that cannot be used with GETVRM are listed later in this section.

You will be returned ARRAY, JARRAY, and FLGRAY, which correspond to the real-valued components, integer-valued components, and the flags associated with the request VAR, respectively. If any array component is not applicable for a given request, its value will be returned as the initialized value: 0.0 in ARRAY, 0 in JARRAY, and N/A in FLGRAY. The error flag JRCD=1 is returned from GETVRM any time a request key is not recognized, the request is not valid (such as requesting transverse shear stress for a shell element that uses thin shell theory), or all the output components requested are zero; otherwise, JRCD=0.

### Ordering of Returned Components

The components for a request are written as follows. Single index components (and requests without components) are returned in positions 1, 2, 3, etc. Double index components are returned in the order 11, 22, 33, 12, 13, 23 for symmetric tensors, followed by 21, 31, 32 for unsymmetric tensors (deformation gradient). Thus, the stresses for a plane stress element are returned as ARRAY(1)=S11, ARRAY(2)=S22, ARRAY(3)=0.0, and ARRAY(4)=S12. Three values are always returned for principal value requests, the minimum value first and maximum value third, regardless of the dimensionality of the analysis.

The description of the output variable (see *Abaqus/Standard Output Variable Identifiers*) determines which components are retrieved with GETVRM.

### Analysis Time for Which Values Are Returned

When a material point quantity is requested with utility routine GETVRM, the time in the increment at which the values are returned will depend on which user subroutine calls it. GETVRM returns values at the end of the current increment to user subroutine *UVARM*, whereas it returns values at the beginning of the current increment to user subroutine *USDFLD*.

### Equilibrium State for Values Returned

User subroutine *UVARM* might call GETVRM multiple times for each increment, as Abaqus/Standard iterates to a converged solution. Values returned from GETVRM calls preceding the final iteration for the increment will not represent the converged solution.

### Example

If you specify the identifier PEQC for use with a jointed material or a material with the cap plasticity model, ARRAY is returned with the individual equivalent plastic strain components PEQC1, PEQC2, PEQC3, and PEQC4. Since there are no integer output variables associated with this identifier, JARRAY is returned with default values of 0. The FLGRAY array contains either YES or NO flags indicating whether each component is actively yielding.

If you specify the identifier PE for a material with plasticity except with the cap plasticity model, ARRAY is returned with plastic strain components PE11, PE22, PE33, PE12, PE13, PE23, the equivalent plastic strain PEEQ, and the plastic strain magnitude PEMAG. Since there are no integer values associated with this request, JARRAY is 0. The FLGRAY array contains N/A for the first six components, either YES or NO in the seventh component (corresponding to PEEQ) indicating whether the material is currently yielding, and N/A in the eighth component.

If you specify the identifier PE for a material with the cap plasticity model, ARRAY is returned with plastic strain components PE11, PE22, PE33, PE12, PE13, PE23, the cap position PBCAP, and the plastic strain magnitude PEMAG. Since there are no integer values associated with this request, JARRAY is 0. The FLGRAY array contains N/A for all eight components.

If you specify the identifier CE, ARRAY is returned with its seventh, eighth, and ninth components equal to CEEQ, CESW, and CEMAG, respectively.

If you specify the identifier VE, ARRAY is returned with its seventh component equal to VEEQ.

If you specify the identifier HFL, ARRAY is returned with the magnitude HFLM and the components HFL1, HFL2, and HFL3 as described in *Abaqus/Standard Output Variable Identifiers*.

### Accessing State-Dependent Variables

If GETVRM is used to access state-dependent variables (output variable key SDV) and more than 15 state-dependent variables have been defined in the analysis, the dimension statement for ARRAY and JARRAY must be changed so that these arrays are dimensioned to the maximum number of state-dependent variables.

### Accessing Parallel Rheological Framework Variables

The components of the output variables CEEQNET, CEEQSSNET, and SDEFRESNET specific to the parallel rheological framework (*Parallel Rheological Framework*) are returned in the order of increasing network identification numbers in the real-valued output array ARRAY. In addition, the identification numbers of the corresponding networks associated with the values in ARRAY are included in the integer-valued array JARRAY. The size of ARRAY and JARRAY for these request variables is as follows:

| Variable identifier (VAR) | Size of ARRAY and JARRAY |
|---|---|
| CEEQNET | Total number of viscous networks |
| CEEQSSNET | Number of viscous networks using the modified Darveaux model |
| SDEFRESNET | Number of viscous networks using the Anand model |

For example, consider a PRF model with four viscous networks with creep laws:

| Network identification number | Creep law |
|---|---|
| 1 | Hyperbolic-Sine model |
| 2 | Anand model |
| 3 | Modified Darveaux Model |
| 4 | Anand model |

The components of ARRAY and JARRAY for requests CEEQNET, CEEQSSNET, and SDEFRESNET, are:

| Component | ARRAY | JARRAY |
|---|---|---|
| 1 | CEEQNET1 | 1 |
| 2 | CEEQNET2 | 2 |
| 3 | CEEQNET3 | 3 |
| 4 | CEEQNET4 | 4 |

| Component | ARRAY | JARRAY |
|---|---|---|
| 1 | CEEQSSNET3 | 3 |

| Component | ARRAY | JARRAY |
|---|---|---|
| 1 | SDEFRESNET2 | 2 |
| 2 | SDEFRESNET4 | 4 |

If GETVRM is used to access state-dependent variables output variables specific to the parallel rheological framework (output variable key CEEQNET, CEEQSSNET, SDEFRESNET) and more than 15 networks have been defined in a PRF material, the dimension statement for ARRAY and JARRAY might need to be modified so that these arrays are dimensioned appropriately.

### Unsupported Element Types, Procedures, and Output Variable Keys

Since this capability pertains to material point quantities, it cannot be used for most of the element types that do not require a material definition. The following element types are, therefore, not supported:

- DASHPOT*x*
- SPRING*x*
- CONN*x*D*x*
- FRAME*x*D
- JOINTC
- JOINT*x*D
- DRAG*x*D
- PSI*xx*
- ITS*xxx*
- MASS
- ROTARYI
- all acoustic elements
- all contact elements
- all hydrostatic fluid elements

If used with user subroutine *UVARM*, this capability is available for static perturbation procedures but not for any other linear perturbation procedures (*General and Perturbation Procedures*):

- *Eigenvalue Buckling Prediction*,
- *Natural Frequency Extraction*,
- *Transient Modal Dynamic Analysis*,
- *Mode-Based Steady-State Dynamic Analysis*,
- *Direct-Solution Steady-State Dynamic Analysis*,
- *Subspace-Based Steady-State Dynamic Analysis*,
- *Response Spectrum Analysis*, and
- *Random Response Analysis*.

The following output variable keys are not available for use with GETVRM:

- SVOL
- TSHR
- CTSHR
- COORD
- UVARPT

## Decoding Material and Orientation Names

**Products:** Abaqus/Standard

## Overview

Utility routines `getMaterialName` and `getOrientationName` can be called from Abaqus/Standard user subroutine `UELEMDRESP` to obtain the name of the material and orientation.

### Utility Routine Interface

```
character*80 charName
...
call getMaterialName(rName, charName)
call getOrientationName(rName, charName)
...
```

### Variables to Be Provided to the Utility Routine

**`rName`** Encoded name of the material or orientation that is made available in user subroutine *UELEMDRESP*.

### Variables Returned from the Utility Routine

**`charName`** Character string with the name of the material or orientation.

## Obtaining Constituent-Level Material Point Information

**Products:** Abaqus/Standard

**References:**

- *UDAMAGEMF*
- **MEAN FIELD DAMAGE*

## Overview

Utility routine GETVRM_MF can be called from user subroutine UDAMAGEMF to obtain material information at the constituent level.

### Utility Routine Interface

```
      SUBROUTINE GETVRM_MF(VAR,KPHASE,JPHASE2MAT,ARRAY,JARRAY,FLGRAY,
     1 JRCD,JMAC,JMATYP,SECLAYO,LACCFLG)
C
      INCLUDE 'ABA_PARAM.INC'
C
      DIMENSION ARRAY(15), JARRAY(15)
      CHARACTER*3 FLGRAY(15)

      user coding to obtain values of  VAR


      RETURN
      END
```

### Variables to Be Provided to the Utility Routine

**VAR**
Output variable key from the table in *Element Integration Point Variables*. The applicable keys are listed in the output table as being available for results file output at the element integration points; for example, S for stress.

**KPHASE**
Variable representing the ID of the constituent whose information is to be obtained through the GETVRM_MF utility routine. This ID can be retrieved by looking up the name or micro-level properties of the constituent in the calling user subroutine. If this variable is set to zero, the material point information of the composite is returned.

**JPHASE2MAT**
Variable that must be passed into the GETVRM_MF utility routine. The calling user subroutine provides this variable.

**JMAC**
Variable that must be passed into the GETVRM_MF utility routine. The calling user subroutine provides this variable.

**JMATYP**
Variable that must be passed into the GETVRM_MF utility routine. The calling user subroutine provides this variable.

**SECLAYO**
Variable that must be passed into the GETVRM_MF utility routine. The calling user subroutine provides this variable.

**LACCFLG** Variable that must be passed into the GETVRM_MF utility routine. The calling user subroutine provides this variable.

### Variables Returned from the Utility Routine

**ARRAY** Real array containing individual components of the output variable.

**JARRAY** Integer array containing individual components of the output variable.

**FLGRAY** Character array containing flags corresponding to the individual components. Flags will contain either YES, NO, or N/A (not applicable).

**JRCD** Return code (0 – no error, 1 – output request error or all components of the output request are zero).

## Obtaining Material Point Information Averaged at a Node

**Products:** Abaqus/Standard

**References:**

- *UMESHMOTION*
- *Using Abaqus/Standard Output Variable Identifiers*
- *Erosion of material (sand production) in an oil wellbore*

## Overview

Utility routine GETVRMAVGATNODE can be called from user subroutine UMESHMOTION to access material integration point information averaged at a node.

The results variables available from GETVRMAVGATNODE are nearly the same as those available from GETVRM; the exceptions follow from the restriction that, since GETVRMAVGATNODE will average results, it will operate only on real-valued results. Results values represented as integers or as flags are not supported.

### Utility Routine Interface

```
DIMENSION ARRAY(15), JELEMLIST(NELEMS)
...
CALL GETVRMAVGATNODE(NODE,JTYP,'VAR',ARRAY,JRCD,JELEMLIST,NELEMS,
JMATYP,JGVBLOCK)
```

### Variables to Be Provided to the Utility Routine

**NODE**
Node number.

**JTYP**
An integer flag indicating how the material point information is averaged. Set JTYP=0 to extrapolate results, using element shape functions, and to average results at the node. Set JTYP=1 to perform a volume-weighted average of results.

**VAR**
Output variable key from the table in *Element Integration Point Variables*. The applicable keys are listed in the output table as being available for results file output at the element integration points; for example, S for stress. One exception is the integration point coordinates variable COORD, which cannot be passed into the utility routine; you should use utility routine GETVRN instead to obtain nodal coordinates.

**JELEMLIST**
Array of element numbers for elements connected to NODE for which you want material point quantities considered in the average result. Results from each element in the list that contain the node will be extrapolated to that node and averaged. JELEMLIST can be obtained from utility routine GETNODETOELEMCONN.

**NELEMS**
Length of JELEMLIST.

**JGVBLOCK**
Variable that must be passed into the GETVRMAVGATNODE utility routine. This variable is available in user subroutine *UMESHMOTION* for this purpose.

**JMATYP** Variable that must be passed into the GETVRMAVGATNODE utility routine. This variable is available in user subroutine *UMESHMOTION* for this purpose.

### Variables Returned from the Utility Routine

**ARRAY** Real array containing individual components of the output variable.

**JRCD** Return code (0 – no error, 1 – output request error or all components of output request are zero).

### Available Output Variable Keys

Only output variable keys that are valid for results file output are available for use with GETVRMAVGATNODE. In general, if a key corresponds to a collective output variable, rather than an individual component, it can be used with GETVRMAVGATNODE. For example, S for the stress tensor can be used, whereas any individual component of stress, say S11, cannot be used. The collective output variable keys are distinguished from their individual components by the fact that they have the word "yes" after the category **.fil** in the tables in *Using Abaqus/Standard Output Variable Identifiers*. Output variable keys that cannot be used with GETVRMAVGATNODE are listed later in this section.

You will be returned ARRAY with components associated with the request VAR. If any array component is not applicable for a given request, its value will be returned as the initialized value: 0.0 in ARRAY. The error flag JRCD=1 is returned from GETVRMAVGATNODE any time a request key is not recognized, the request is not valid, or all the output components requested are zero; otherwise, JRCD=0.

### Ordering of Returned Components

The components for a request are written as follows. Single index components (and requests without components) are returned in positions 1, 2, 3, etc. Double index components are returned in the order 11, 22, 33, 12, 13, 23 for symmetric tensors, followed by 21, 31, 32 for unsymmetric tensors (deformation gradient). Thus, the stresses for a plane stress element are returned as ARRAY(1)=S11, ARRAY(2)=S22, ARRAY(3)=0.0, and ARRAY(4)=S12. Three values are always returned for principal value requests, the minimum value first and the maximum value third, regardless of the dimensionality of the analysis.

The description of the output variable (see *Using Abaqus/Standard Output Variable Identifiers*) determines which components are retrieved with GETVRMAVGATNODE.

### Analysis Time for Which Values Are Returned

GETVRMAVGATNODE returns values at the end of the current increment to user subroutine *UMESHMOTION*.

### Accessing State-Dependent Variables

If GETVRMAVGATNODE is used to access solution-dependent state variables (output variable key SDV) and more than 15 solution-dependent state variables have been defined in the analysis, the dimension statement for ARRAY must be changed so that these arrays are dimensioned to the maximum number of solution-dependent state variables.

### Unsupported Element Types and Output Variable Keys

Since this capability pertains to material point quantities, it cannot be used for most of the element types that do not require a material definition. The following element types are, therefore, not supported:

- DASHPOT*x*
- SPRING*x*
- CONN*x*D*x*
- FRAME*x*D
- JOINTC
- JOINT*x*D
- DRAG*x*D
- PSI*xx*
- ITS*xxx*
- MASS
- ROTARYI
- all acoustic elements
- all hydrostatic fluid elements

The following output variable keys are not available for use with `GETVRMAVGATNODE`:

- SVOL
- TSHR
- CTSHR

### Example: Obtaining Plastic Strain Results

To illustrate the use of `GETVRMAVGATNODE`, consider a case where the identifier PE is specified and `JELEMLIST` lists four three-dimensional elements, two of which have plastic yield behavior defined and two of which do not. `ARRAY` will be returned with the individual plastic strain components PE11, PE22, PE33, PE12, PE13, and PE23; the equivalent plastic strain PEEQ (except for the cap plasticity model, for which this is the cap position PBCAP); and the plastic strain magnitude PEMAG. The result returned in `ARRAY` will be an average reflecting extrapolations of plastic strain results to `NODE` from only the two elements that have plastic yield behavior defined.

### Example: Obtaining Contact Results

A second illustration is relevant to the modeling of wear with *`UMESHMOTION`*. Consider a case where `JELEMLIST` is obtained from `GETNODETOELEMCONN` and where the identifier CSTRESS is specified. If `NODE` is associated with a contact pair secondary surface, `JELEMLIST` will contain the internal element identifier for the contact element associated with the secondary node pairing. `ARRAY` will be returned with the individual contact stress components CPRESS, CSHEAR1, and CSHEAR2. Similarly, if CDISP is specified, `ARRAY` will be returned with the individual contact stress components CDISP, CSLIP1, and CSLIP2, where CDISP corresponds to the penetration distance (CDISP is negative where contact surfaces are separated).

## Obtaining Stress Invariants, Principal Stress/Strain Values and Directions, and Rotating Tensors

**Products:** Abaqus/Standard

**References:**

- *UMAT*
- *Calculation of principal stresses and strains and their directions: FPRIN*

## Overview

Utility routines are available for calculating stress invariants, principal stress/strain values, and principal stress/strain directions from the relevant tensors, as well as for transforming tensors to a new basis.

These utility routines are available for Abaqus/Standard user subroutines that store stress and strain components according to the convention presented in *Conventions*. They are most commonly called from user subroutine *UMAT*.

### SINV (Calculate Stress Invariants)

*Utility Routine Interface*

```
      CALL SINV(STRESS,SINV1,SINV2,NDI,NSHR)
```

*Variables to Be Provided to the Utility Routine*

**STRESS** A stress tensor.

**NDI** Number of direct components.

**NSHR** Number of shear components.

*Variables Returned from the Utility Routine*

**SINV1** First invariant.

$$\mathtt{SINV1} = \frac{1}{3}\,\mathrm{trace}\,\boldsymbol{\sigma},$$

where $\boldsymbol{\sigma}$ is the stress tensor.

**SINV2** Second invariant.

$$\mathtt{SINV2} = \sqrt{\frac{3}{2}\mathbf{S} : \mathbf{S}},$$

where $\mathbf{S}$ is the deviatoric stress tensor, defined as

$$\mathbf{S} = \boldsymbol{\sigma} - \frac{1}{3}\,\mathrm{trace}\,\boldsymbol{\sigma}\mathbf{I}.$$

### SPRINC (Calculate Principal Values)

*Utility Routine Interface*

```
      CALL SPRINC(S,PS,LSTR,NDI,NSHR)
```

*Variables to Be Provided to the Utility Routine*

**S**
Stress or strain tensor.

**LSTR**
An identifier. LSTR=1 indicates that S contains stresses; LSTR=2 indicates that S contains strains.

**NDI**
Number of direct components.

**NSHR**
Number of shear components.

*Variables Returned from the Utility Routine*

**PS(I), I=1,2,3**
The three principal values in an unsorted order.

### SPRIND (Calculate Principal Values and Directions)

*Utility Routine Interface*

```
      CALL SPRIND(S,PS,AN,LSTR,NDI,NSHR)
```

*Variables to Be Provided to the Utility Routine*

**S**
A stress or a strain tensor.

**LSTR**
An identifier. LSTR=1 indicates that S contains stresses; LSTR=2 indicates that S contains strains.

**NDI**
Number of direct components.

**NSHR**
Number of shear components.

*Variables Returned from the Utility Routine*

**PS(I), I=1,2,3**
The three principal values in an unsorted order.

**AN(I,J), J=1,2,3**
The direction cosines of the principal directions corresponding to PS(I).

### ROTSIG (Rotate a Tensor)

#### *Utility Routine Interface*

```
      CALL ROTSIG(S,R,SPRIME,LSTR,NDI,NSHR)
```

#### *Variables to Be Provided to the Utility Routine*

**S**
A stress or strain tensor.

**NDI**
Number of direct components.

**NSHR**
Number of shear components.

**R**
Rotation matrix.

**LSTR**
An identifier. `LSTR` = **1** indicates that `S` contains stresses; `LSTR` = **2** indicates that `S` contains strains.

#### *Variables Returned from the Utility Routine*

**SPRIME**
The rotated stress or strain tensor.

#### *Typical Usage*

In user subroutine *UMAT* it is often necessary to rotate tensors during a finite-strain analysis. The matrix `DROT` that is passed into *UMAT* represents the incremental rotation of the material basis system in which the stress and strain are stored. For an elastic-plastic material that hardens isotropically, the elastic and plastic strain tensors must be rotated to account for the evolution of the material directions. In this case `S` is the elastic or plastic strain tensor and `R` is the incremental rotation `DROT`.

## Accessing Abaqus Materials

**Products:** Abaqus/Standard

**References:**

- *UELMAT*
- *UELMAT*

## Overview

Utility routine `MATERIAL_LIB_MECH` returns the stress and the material Jacobian at the element material point.

The routine can be called only from Abaqus/Standard user subroutine *UELMAT*.

### Utility Routine Interface

```
     dimension stress(*),ddsdde(ntens,*),stran(*),dstran(*),
    * defGrad(3,3),predef(npredf),dpredef(npredf),coords(3)
     ...

     call material_lib_mech(materiallib,stress,ddsdde,stran,dstran,
    *       npt,dvdv0,dvmat,dfgrd,predef,dpredef,npredf,celent,coords)
     ...
```

### Variables to Be Provided to the Utility Routine

**materiallib** Variable containing information about the Abaqus material. This variable is passed into user subroutine *UELMAT*.

**stran** Strain at the beginning of the increment.

**dstran** Strain increment.

**npt** Integration point number.

**dvdv0** Ratio of the current volume to the reference volume at the integration point.

**dvmat** Volume at the integration point.

**dfgrd** Array containing the deformation gradient at the end of the increment.

**predef** Array of interpolated values of predefined field variables at the integration point at the start of the increment.

**dpredef** Array of increments of predefined field variables.

**npredf** Number of predefined field variables, including temperature.

**celent** Characteristic element length.

**coords**

An array containing the coordinates of this point. These are the current coordinates if geometric nonlinearities are accounted for during the step (see *Defining an Analysis*); otherwise, the array contains the original coordinates of the point.

### Variables Returned from the Utility Routine

**stress**

Stress tensor at the end of the increment.

**ddsdde**

Jacobian matrix of the constitutive model, $\partial \Delta\boldsymbol{\sigma} / \partial \Delta\boldsymbol{\varepsilon}$, where $\Delta\boldsymbol{\sigma}$ are the stress increments and $\Delta\boldsymbol{\varepsilon}$ are the strain increments. `ddsdde(i,j)` defines the change in the *i*th stress component at the end of the time increment caused by an infinitesimal perturbation of the *j*th component of the strain increment array.

## Accessing Abaqus Thermal Materials

**Products:** Abaqus/Standard

**References:**

- *UELMAT*
- *UELMAT*

## Overview

Utility routine MATERIAL_LIB_HT returns heat fluxes, internal energy time derivative, volumetric heat generation rate, and their derivatives at the element material point.

The routine can be called only from Abaqus/Standard user subroutine *UELMAT*.

### Utility Routine Interface

```
     dimension predef(npredef),dpredef(npredef),dtemdx(*),
    *          rhodUdg(*),flux(*),dfdt(*),dfdg(ndim,*),drpldt(*),     *
    coords(3)
     ...

     call material_lib_ht(materiallib,rhoUdot,rhodUdt,rhodUdg,
    *       flux,dfdt,dfdg,rpl,drpldt,npt,dvmat,predef,
    *       dpredef,npredf,temp,dtemp,dtemdx,celent,coords)
     ...
```

### Variables to Be Provided to the Utility Routine

**materiallib** — Variable containing information about the Abaqus material. This variable is passed into user subroutine *UELMAT*.

**npt** — Integration point number.

**dvmat** — Volume at the integration point.

**predef** — Array of interpolated values of predefined field variables at the integration point at the start of the increment.

**dpredef** — Array of increments of predefined field variables.

**npredf** — Number of predefined field variables, including temperature.

**temp** — Temperature at the integration point at the start of the increment, $\theta$.

**dtemp** — Increment of temperature.

**dtemdx** — Spatial gradients of temperature, $\partial\theta/\partial\mathbf{x}$, at the end of the increment.

**celent** — Characteristic element length.

**coords** The array containing the original coordinates of this point.

### Variables Returned from the Utility Routine

**rhoUdot** Time derivative of the internal thermal energy per unit mass, $U$, multiplied by density at the end of increment.

**rhodUdt** Variation of internal thermal energy per unit mass with respect to temperature multiplied by density evaluated at the end of the increment.

**rhodUdg** Variation of internal thermal energy per unit mass with respect to the spatial gradients of temperature, $\partial U/\partial(\partial\theta/\partial\mathbf{x})$, multiplied by density at the end of the increment.

**flux** Heat flux vector, $\mathbf{f}$, at the end of the increment.

**dfdt** Variation of the heat flux vector with respect to temperature, $\partial\mathbf{f}/\partial\theta$, evaluated at the end of the increment.

**dfdg** Variation of the heat flux vector with respect to the spatial gradients of temperature, $\partial\mathbf{f}/\partial(\partial\theta/\partial\mathbf{x})$, at the end of the increment

**rpl** Volumetric heat generation per unit time at the end of the increment.

**drpldt** Variation of `rpl` with respect to temperature.

## Obtaining Material Point Data in Abaqus/Explicit

Utility routines to access material point data are available to assist in the coding of user subroutines.

### In this section:


- *Obtaining Material Point Information in an Abaqus/Explicit Analysis*
- *Obtaining Principal Stress/Strain Values and Directions*

## Obtaining Material Point Information in an Abaqus/Explicit Analysis

**Products:** Abaqus/Explicit

**References:**

- *VUSDFLD*
- *Damage and failure of a laminated composite plate*

## Overview

Utility routine VGETVRM can be called from VUSDFLD to access selected output variables at the material points for the current block of elements processed by VUSDFLD.

### Utility Routine Interface

```
   include 'vaba_param.inc'
   parameter( nrData=6 )  ! maximum number of components of variables to
be requested (6 for symmetric tensor)
  character*3 cData(maxblk*nrData)  ! parameter maxblk is the maximum value
 of nblock
   dimension rData(maxblk*nrData), jData(maxblk*nrData)
   ...
   call vgetvrm( 'VAR', rData, jData, cData, jStatus )
```

### Variables to Be Provided to the Utility Routine

**VAR** Output variable key. The available material point variables are given in *Available Output Variable Keys* below.

### Variables Returned from the Utility Routine

**rData** Real array containing individual components of the output variable.

**jData** Integer array containing individual components of the output variable.

**cData** Character array containing flags corresponding to the individual components. Flags will either be YES, NO, or N/A (not applicable).

**jStatus** Return code (0: output request successful; 1: output request not available).

### Available Output Variable Keys

The following output variable keys are supported:

- S: All stress components.
- SP: All principal stresses.
- MISES: Mises equivalent stress.
- PRESS: Equivalent pressure stress.
- TRIAX: Stress triaxiality.

- LODE: Lode angle term.
- LE: All logarithmic strain components.
- LEP: All principal logarithmic strains.
- ER: All logarithmic strain rate components.
- ERP: All principal logarithmic strain rates.
- THE: All thermal strain components.
- PE: All plastic strain components.
- PEP: All principal plastic strains.
- PEEQ: Equivalent plastic strain.
- PEEQR: Equivalent plastic strain rate.
- PEEQT: Equivalent plastic strain in uniaxial tension, defined as $\int \dot{\bar{\varepsilon}}_t^{pl} dt$.
- PEEQTSHR: Equivalent plastic strain for paperboard out-of-plane plasticity. Available only for the paperboard material model.
- PEQC: All equivalent plastic strains for models that have more than one yield/failure surface.
- PBCAP: The cap position for the cap plasticity model.
- CE: All creep strain components.
- CEP: All principal creep strains.
- CEEQ: Equivalent creep strain.
- CEEQNET: The equivalent creep strain in all the networks. Available only for the parallel rheological framework.
- CEEQSSNET: Accumulated steady-state equivalent creep strain in all the networks that use the modified Darveaux creep law. Available only for the parallel rheological framework.
- SDEFRESNET: Deformation resistance in all the networks that use the Anand creep law. Available only for the parallel rheological framework.
- EEQUT: Equivalent uniaxial tensile total strain. Available only for the superelasticity model.
- MVF: Fraction of martensite. Available only for the superelasticity model.
- SEQUT: Equivalent uniaxial tensile total stress. Available only for the superelasticity model.
- TE: All transformation strain components. Available only for the superelasticity model.
- TEEQ: Equivalent transformation strain. Available only for the superelasticity model.
- ALPHA: All total kinematic hardening shift tensor components.
- ALPHAP: All principal values of total kinematic hardening shift tensor.
- TEMP: Temperature.
- SDEG: Scalar stiffness degradation variable.
- DUCTCRT: Ductile damage initiation criterion.
- JCCRT: Johnson-Cook damage initiation criterion.
- HCCRT: Hosford-Coulomb damage initiation criterion.
- SHRCRT: Shear damage initiation criterion.
- FLDCRT: Forming limit diagram (FLD) damage initiation criterion.
- FLSDCRT: Forming limit stress diagram (FLSD) damage initiation criterion.
- MSFLDCRT: Müschenborn-Sonne forming limit diagram (MSFLD) damage initiation criterion.

- MKCRT: Marciniak-Kuczynski (M-K) damage initiation criterion.
- MSTRESSCRT: Maximum stress-based damage initiation criterion.
- MSTRAINCRT: Maximum strain-based damage initiation criterion.
- TSAIWUCRT: Stress-based Tsai-Wu damage initiation criterion.
- TSAIWUECRT: Strain-based Tsai-Wu damage initiation criterion.
- EREQ: Strain rate used to evaluate strain-rate-dependent damage initiation criteria.
- POR: Pore fluid pressure.
- EVF: Volume fraction. Available only for Eulerian elements.
- RD: Relative density. Available only for cap plasticity and porous metal plasticity.
- VVF: Void volume fraction. Available only for cap plasticity and porous metal plasticity.
- UMULMAX: The maximum deviatoric strain energy density for rubberlike materials. The maximum strain energy density for elastomeric foams. Available only when the Mullins effect is modeled.

A requested output variable must be valid for the material model for the request to be successful.

The returned arrays `rData`, `jData`, and `cData` correspond to the real-valued variable, integer-valued variable, and string variable, respectively, that can be associated with the request output variable key `VAR`. If any output variable component is not applicable for a given request, its value will be returned as the initialized value: 0.0 in `rData`, 0 in `jData`, and N/A in `cData`. The association of the integer-valued variable and the string variable with the request output variable is not supported. The error flag `jStatus` is returned with a value of 1 if a request key is not recognized, not valid, or not supported; otherwise, `jStatus` is returned with a value of 0.

### Component Ordering in Symmetric Tensors

For symmetric tensors such as the stress and strain tensors, there are `ndir+nshr` components, where `ndir` and `nshr` are the number of direct and shear components, respectively, that are passed into user subroutine *VUSDFLD*. The component order is given as a natural permutation of the indices of the tensor. The direct components are first and then the indirect components, beginning with the 12-component. For example, a stress tensor contains `ndir` direct stress components and `nshr` shear stress components, which are returned as:

| Component | 2D Case | 3D Case |
|---|---|---|
| 1 | $\sigma_{11}$ | $\sigma_{11}$ |
| 2 | $\sigma_{22}$ | $\sigma_{22}$ |
| 3 | $\sigma_{33}$ | $\sigma_{33}$ |
| 4 | $\sigma_{12}$ | $\sigma_{12}$ |
| 5 | | $\sigma_{23}$ |
| 6 | | $\sigma_{31}$ |

The shear strain components returned from utility subroutine `VGETVRM` are tensor components and not engineering components.

### Analysis Time for Which Values Are Returned

Utility subroutine `VGETVRM` returns values of the requested variable that correspond to the beginning of the current increment.

### Example

To illustrate the use of `VGETVRM`, if the identifier PE is specified for a material with plasticity, `rData` will be returned with plastic strain components PE11, PE22, PE33, PE12, PE23, and PE13. `jData` will be 0 and `cData` array will have N/A for all components.

### Accessing Parallel Rheological Framework Variables

The components of output variables CEEQNET, CEEQSSNET, and SDEFRESNET specific to the parallel rheological framework (*Parallel Rheological Framework*) are returned in the order of increasing network identification numbers in the real-valued output array `rData`. In addition, the identification numbers of the corresponding networks associated with the values in `rData` are included in the integer-valued array `jData`. The size of `rData` and `jData` for these request variables is as follows:

| Variable identifier (VAR) | Size of `rData` and `jData` |
|---|---|
| CEEQNET | Total number of viscous networks |
| CEEQSSNET | Number of viscous networks using the modified Darveaux model |
| SDEFRESNET | Number of viscous networks using the Anand model |

Depending on the request and the number of networks, the maximum number of components of variables to be requested, `nrData`, might need be modified so that these arrays are dimensioned appropriately.

For example, consider a PRF model with four viscous networks with creep laws:

| Network identification number | Creep law |
|---|---|
| 1 | Hyperbolic-Sine model |
| 2 | Anand model |
| 3 | Modified Darveaux Model |
| 4 | Anand model |

Then, the components of `rData` and `jData` for requests CEEQNET, CEEQSSNET, and SDEFRESNET, are:

| Component | `rData` | `jData` |
|---|---|---|
| 1 | CEEQNET1 | 1 |
| 2 | CEEQNET2 | 2 |
| 3 | CEEQNET3 | 3 |
| 4 | CEEQNET4 | 4 |

| Component | `rData` | `jData` |
|---|---|---|
| 1 | CEEQSSNET3 | 3 |

| Component | `rData` | `jData` |
|---|---|---|
| 1 | SDEFRESNET2 | 2 |
| 2 | SDEFRESNET4 | 4 |

### Unsupported Element Types, Procedures, and Output Variable Keys

Since this capability pertains to material point quantities, it cannot be used for most of the element types that do not require a material definition. The following element types are, therefore, not supported:

- DASHPOT*x*
- SPRING*x*
- CONN*x*D*x*
- MASS
- ROTARYI
- all acoustic elements

## Obtaining Principal Stress/Strain Values and Directions

**Products:** Abaqus/Explicit

**References:**

- *VUMAT*

## Overview

Utility routines are available for calculating principal stress/strain values and principal stress/strain directions from the relevant tensors.

These utility routines are available for Abaqus/Explicit user subroutines that store stress and strain components according to the convention presented in *Conventions*. They are most commonly called from user subroutine *VUMAT*.

### VSPRINC (Calculate Principal Values)

*Utility Routine Interface*

```
      call vsprinc( nblock, s, eigVal, ndir, nshr )
```

*Variables to Be Provided to the Utility Routine*

**s(nblock,ndir+nshr)** Stress or strain symmetric tensor.

**nblock** Number of material points to be processed in this call to VSPRINC.

**ndir** Number of direct components in the symmetric tensor.

**nshr** Number of shear components in the symmetric tensor.

*Variables Returned from the Utility Routine*

**eigVal(nblock,I), I=1,2,3** The three principal values in an unsorted order.

### VSPRIND (Calculate Principal Values and Directions)

*Utility Routine Interface*

```
      call vsprind( nblock, s, eigVal, eigVec, ndir, nshr )
```

*Variables to Be Provided to the Utility Routine*

**s(nblock,ndir+nshr)** Stress or strain symmetric tensor.

**nblock** Number of material points to be processed in this call to VSPRIND.

**ndir** Number of direct components in the symmetric tensor.

**nshr**
Number of shear components in the symmetric tensor.

*Variables Returned from the Utility Routine*

**eigVal(nblock,I), I=1,2,3**
The three principal values in an unsorted order.

**eigVec(nblock,J,I), J=1,2,3**
The direction cosines of the principal directions corresponding to the `I`$^{th}$ principal value, `eigVal(:,I)`.

# Contact Data: Obtaining Contact Constraint Point Information in an Abaqus/Standard Analysis

**Products:** Abaqus/Standard

**References:**

- *FRIC*
- *FRIC_COEF*
- *GAPCON*
- *GAPELECTR*
- *UINTER*

## Overview

Utility routine GETVRC can be called from user subroutines FRIC, FRIC_COEF, GAPCON, GAPELECTR, and UINTER to access contact variables for individual contact constraints. For these user subroutines GETVRC returns values at the beginning of the current increment.

## Utility Routine Interface

```
PARAMETER(NDATASIZE=10)
DIMENSION ARRAY(NDATASIZE), JARRAY(NDATASIZE)
CHARACTER*3 FLGRAY(NDATASIZE)
...
CALL GETVRC('VAR', ARRAY, JARRAY, FLGRAY, NDATASIZE, JRCD)
```

## Variables to Be Provided to the Utility Routine

**VAR** The supported list of key identifiers for various contact variables and their meanings and sizes are listed in *Table 1*; for example, the identifier key for contact stress for a contact constraint is CSTRESS.

**NDATASIZE** The common integer size for the input buffers that are passed in to be filled with data. If the input buffer size for a particular variable is smaller than the size required, an error is issued.

## Variables Returned from the Utility Routine

**ARRAY** Input buffer for retrieving double precision components of the requested variable.

**JARRAY** Input buffer for retrieving integer components of the requested variable when applicable.

**FLGRAY** Input buffer for retrieving character flags when applicable. Flags will contain either YES, NO, or N/A (not applicable).

**JRCD** Return code (0 indicates no error, 1 indicates an output request error or all components of the output request are zero).

## Available Variable Keys

*Table 1* describes the supported key identifiers for various contact variables.

**Table 1: Variable keys for contact variables.**

| Variable identifier (VAR) | Description | Type of data | Size |
|---|---|---|---|
| **Contact stress and strain variables** | | | |
| CSTRESS | Contact stress | Real | Number of friction directions + 1 |
| CDISP | Contact strains; first entry is the penetration distance and is negative if constraint is open or -1e+36 if no pairing main is found; the remaining entries are the accumulated tangential slip values | Real | Number of friction directions + 1 |
| CSLIPEQ | Equivalent contact slip | Real | 1 |
| CSLIP_PLEQ | Equivalent plastic contact slip | Real | 1 |
| CSLIP_PL | Plastic slip | Real | Number of friction directions |
| CNAREA | Contact constraint area | Real | 1 (or 2 only in a coupled analysis if temperatures or pore pressures are active for the contact constraint; the second entry corresponds to area associated with temperatures or pore pressures) |
| **Contact constraint energy densities with units of energy per unit area** | | | |
| ELSEDEN | Recoverable “elastic” energy density due softened contact pressure versus overclosure behavior or penalty enforcement of contact constraints in normal and tangential directions | Real | 1 |
| ELPDDEN | Frictional dissipation energy density (frictionless contact does not contribute to ELPDDEN) | Real | 1 |
| ELSDDEN | Viscous dissipation energy density due to contact stabilization in normal and tangential directions | Real | 1 |
| ELVDDEN | Viscous dissipation energy density due to contact damping | Real | 1 |
| ELCDDEN | Cohesive contact damage stabilization energy density | Real | 1 |
| ELJDDEN | Joule heating energy density due to contact resistance to electric current through a contact interface | Real | 1 |
| ELDDDEN | Dissipated energy density due to cohesive contact damage | Real | 1 |
| **Contact constraint time integrated variables** | | | |

| Variable identifier (VAR) | Description | Type of data | Size |
|---|---|---|---|
| ELSJDT | Time integrated heat flux per unit area due to electrical current | Real | 1 |
| ELSJDTA | Time integrated heat flux due to electrical current | Real | 1 |
| ELHTL | Time integrated heat flux per unit area leaving the secondary surface | Real | 1 |
| ELHTLA | Time integrated heat flux leaving the secondary surface | Real | 1 |
| ELSFDRT | Time integrated heat flux per unit area due to frictional dissipation | Real | 1 |
| ELSFDRTA | Time integrated heat flux due to frictional dissipation | Real | 1 |
| ELPTL | Time integrated pore fluid volume flux per unit area leaving the secondary surface | Real | 1 |
| ELPTLA | Time integrated pore fluid volume flux leaving the secondary surface | Real | 1 |
| ELECDT | Time integrated electrical current per unit area | Real | 1 |
| ELECDTA | Time integrated electrical current | Real | 1 |

# Toolpath-Mesh Intersection Utility Routines

Toolpath-mesh intersection utility routines are used to activate the toolpath-mesh intersection module and to access data computed by the module.

These utility routines are invoked from Abaqus/Standard user subroutines. Three types of utility routines are available:

- Setup utility routines that are invoked at the beginning of an analysis step from user subroutines *UEPACTIVATIONSETUP* and *UMDFLUXSETUP*.
- A compute utility routine that is invoked at the beginning of an increment from user subroutines *UEPACTIVATIONSETUP* and *UMDFLUXSETUP*.
- Data retrieval utility routines that are invoked at the beginning of an increment after the call to a compute routine from user subroutines *UEPACTIVATIONSETUP* and *UMDFLUXSETUP* and from within the element loop from user subroutines *UEPACTIVATIONVOL*, *UEPACTIVATIONFACET*, and *UMDFLUX*.

All toolpath-mesh intersection utility routine names use the prefix “Ptk."

### In this section:


- *Setup Utility Routines*
- *Compute Utility Routine*
- *Data Retrieval Utility Routines*

# Setup Utility Routines

**Products:** Abaqus/Standard

**References:**

- *UEPACTIVATIONSETUP*
- *UMDFLUXSETUP*

## Overview

Toolpath-mesh intersection utility routines are used to activate the toolpath-mesh intersection module and to access data computed by the module.
Setup utility routines are invoked at the beginning of an analysis step from user subroutines *UEPACTIVATIONSETUP* and *UMDFLUXSETUP*.

### PtkSetMeshAndEventSeries

Construct a singleton object for the toolpath-mesh intersection module, and set mesh and event series data.

#### Utility Routine Interface

```
#include <PtkUtilitySubs.hdr>
#include <aba_ptk_enums.inc>

      character *80 cElSetName, cEventSeries
      call PtkSetMeshAndEventSeries (cElSetName, cEventSeries)
```

#### Variables to Be Provided to the Utility Routine

| | |
|---|---|
| **cElSetName** | Name of the element set. |
| **cEventSeries** | Name of the event series describing the toolpath. |

### PtkSetEventSeriesProperties

Define the shape of a toolpath and the algorithm to be used in intersection computations. Refer to *Figure 1* for box toolpath descriptions.

#### Utility Routine Interface

```
#include <PtkUtilitySubs.hdr>
#include <aba_ptk_enums.inc>

      character *80 cEventSeries
      dimension jPtkEvtOps (nPtkEventOpt_ArraySize)
      dimension jPtkEvtProps (nPtkIntEventProperties_ArraySize)
      dimension rPtkEvtProps (nPtkRealEventProperties_ArraySize)

      call PtkSetEventSeriesProperties (cEventSeries, cPtkID, iEventType,

     *                             jPtkEvtOps, jPtkEvtProps, rPtkEvtProps)
```

## Variables to Be Provided to the Utility Routine

**cEventSeries** Name of the event series.

**cPtkID** Tag name assigned to the event series.

**iEventType** Integer flag indicating the type of the toolpath. Set `iEventType` to `iPtkEventSeriesShape_Point`, `iPtkEventSeriesShape_InfiniteLine`, or `iPtkEventSeriesShape_Box`.

**jPtkEvtOps** Integer array of size `nPtkEventOpt_ArraySize`.

| | |
|---|---|
| `jPtkEvtOps(iPtkEventOpt_Algorithm)` | Indicates the algorithm to be used for the toolpath-mesh intersection. For a box toolpath, set `JPtkEvtOps(iPtkEventOpt_Algorithm)` to `iPtkEventSeriesAlgorithm_SubElement` or `iPtkEventSeriesAlgorithm_SubSegment`. For point or infinite line toolpaths, set `JPtkEvtOps(iPtkEventOpt_Algorithm)` to `iPtkEventSeriesAlgorithm_Undefined`. |
| `jPtkEvtOps(iPtkEventOpt_AbsoluteOffsets)` | Indicates whether components of the offset vector **F** to position the box with respect to the path are normalized by the box lengths in the segment local $x_l$-, $y_l$-, and $z_l$-directions or by absolute values. Set `jPtkEvtOps(iPtkEventOpt_AbsoluteOffsets)`=1 to interpret components of the offset vector as absolute values. Components of the offset vector must be between –1 and 1 if they are normalized by box lengths. (Unused for point or infinite line toolpaths). |
| `jPtkEvtOps(iPtkEventOpt_SegmentTransition)` | Set `jPtkEvtOps(iPtkEventOpt_SegmentTransition)`=1 to activate transitioning between segments in a box toolpath. Transitioning between segments results in no overlap of the path swept by the box. (Unused for point or infinite line toolpaths). |
| `jPtkEvtOps(iPtkEventOpt_Conservative)` | Set `jPtkEvtOps(iPtkEventOpt_Conservative)`=1 to indicate that if part of a path segment for a point toolpath or part of a subsegment for a box toolpath using the subsegment approach is outside the external mesh boundary, fields of part of segments or subsegments that are inside the mesh boundary are incremented proportionally to conserve the total field. (Unused for infinite line toolpaths). |

**jPtkEvtProps** Integer array of size `nPtkIntEventProperties_ArraySize)`.

| | |
|---|---|
| `jPtkEvtProps(iPtkIntEventProperties_BoxSubDiv1)` | Number of subdivisions along the segment local $x_l$-direction for a box toolpath using the subsegment method. (Unused for point or infinite line toolpaths). |
| `jPtkEvtProps(iPtkIntEventProperties_BoxSubDiv2)` | Number of subdivisions along the segment local $y_l$-direction for a box toolpath using the subsegment method. (Unused for point or infinite line toolpaths). |
| `jPtkEvtProps(iPtkIntEventProperties_BoxSubDiv3)` | Number of subdivisions along the segment local $z_l$-direction for a box toolpath using the subsegment method. (Unused for point or infinite line toolpaths). |

**rPtkEvtProps** Real array of size `nPtkRealEventProperties_ArraySize`.

`rPtkEvtProps(iPtkRealEventProperties_Offset1)` Component of the offset vector $\mathbf{F}$ with respect to the segment local $x_l$-direction for a box toolpath. (Unused for point or infinite line toolpaths).

`rPtkEvtProps(iPtkRealEventProperties_Offset2)` Component of the offset vector $\mathbf{F}$ with respect to the segment local $y_l$-direction for a box toolpath. (Unused for point or infinite line toolpaths).

`rPtkEvtProps(iPtkRealEventProperties_Offset3)` Component of the offset vector $\mathbf{F}$ with respect to the segment local $z_l$-direction for a box toolpath. (Unused for point or infinite line toolpaths.)

`rPtkEvtProps(iPtkRealEventProperties_NormalVector1)` Component of a vector $\mathbf{v}$ defining the segment local $\mathbf{z_1}$–$\mathbf{x_1}$ plane for a box toolpath. The $x_l$-direction is along the segment. Component must be given with respect to the global coordinate system. (Unused for point or infinite line toolpaths.)

`rPtkEvtProps(iPtkRealEventProperties_NormalVector2)` Component of a vector $\mathbf{v}$ defining the segment local $\mathbf{z_1}$–$\mathbf{x_1}$ plane for a box toolpath. The $y_l$-direction is $\mathbf{y_1} = \mathbf{v} \times \mathbf{x_1}$. Component must be given with respect to the global coordinate system. (Unused for point or infinite line toolpaths.)

`rPtkEvtProps(iPtkRealEventProperties_NormalVector3)` Component of a vector $\mathbf{v}$ defining the segment local $\mathbf{z_1}$–$\mathbf{x_1}$ plane for a box toolpath. The $z_l$-direction is $\mathbf{z_1} = \mathbf{x_1} \times \mathbf{y_1}$. Component must be given with respect to the global coordinate system. (Unused for point or infinite line toolpaths.)

`rPtkEvtProps(iPtkRealEventProperties_BoxLength1)` Length of box along the segment local $x_l$-direction for a box toolpath. (Unused for point or infinite line toolpaths.)

`rPtkEvtProps(iPtkRealEventProperties_BoxLength2)` Length of box along the segment local $y_l$-direction for a box toolpath. (Unused for point or infinite line toolpaths.)

`rPtkEvtProps(iPtkRealEventProperties_BoxLength3)` Length of box along the segment local $z_l$-direction for a box toolpath. (Unused for point or infinite line toolpaths.)

## Figures

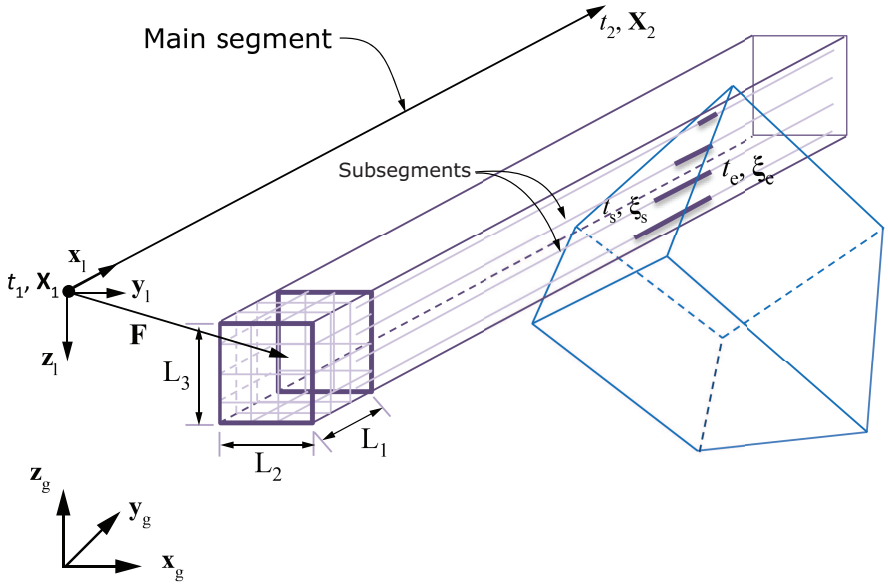



**Figure 1: Box toolpath-mesh intersection using the subsegment approach.**

### `PtkGetEventSeriesNumDivisionPoints`

Retrieve number of subdivision points in a box toolpath using the subsegment algorithm for intersection computations.

### Utility Routine Interface

```
#include <PtkUtilitySubs.hdr>
#include <aba_ptk_enums.inc>

       character *80 cEventSeries
       call ptkGetEventSeriesNumDivisionPoints(cEventSeries,numSubDiv)
```

### Variables to Be Provided to the Utility Routine

**`cEventSeries`** Name of the event series describing the toolpath.

### Variables Returned from the Utility Routine

**`numSubDiv`** Number of subdivisions.

### `PtkGetEventSeriesDivisionCoordinates`

Retrieve coordinates of subdivision points in the segment local coordinate system defined by $\mathbf{x}_1$, $\mathbf{y}_1$, and $\mathbf{z}_1$ (see *Figure 1*) in a box toolpath using the subsegment algorithm.

### Utility Routine Interface

```
#include <PtkUtilitySubs.hdr>
#include <aba_ptk_enums.inc>

       character *80 cPtkID
       call ptkGetEventSeriesDivisionCoordinates(cPtkID,subDivPtCoords)
```

### Variables to Be Provided to the Utility Routine

**cPtkID** Tag name of the event series defined in `PtkSetEventSeriesProperties`.

### Variables Returned from the Utility Routine

**subDivPtCoords** Real array containing coordinates of the subdivision points. Array size is 3 x `numSubDiv`.

## PtkSetEventSeriesDivisionPointWeights

Set weights to subdivision points of a box toolpath using the subsegment algorithm.

### Utility Routine Interface

```
#include <PtkUtilitySubs.hdr>
#include <aba_ptk_enums.inc>

       character *80 cPtkID
       call ptkSetEventSeriesDivisionPointWeights(cPtkID, subDivPtWts)
```

### Variables to Be Provided to the Utility Routine

**cPtkID** Tag name of the event series defined in `PtkSetEventSeriesProperties`.

**subDivPtWts** Real array containing weights of subdivision points. Array size is `numSubDiv`.

## PtkSetMeshAndScanStrategies

Construct a singleton object for the toolpath mesh intersection module, and set mesh and scan pattern data. Refer to *Figure 2* for scan pattern descriptions and to *Figure 3* for scan pattern orientations.

### Utility Routine Interface

```
#include <PtkUtilitySubs.hdr>
#include <aba_ptk_enums.inc>

       character *80 cEventSeries, cPtkID
       dimension rPtkScanData(*), iPtkScanData(*), iPtkScanOpts(*)

       call PtkSetMeshAndScanStrategies(cElsetName, cPtkID, iPtkScanOpts,
      *                                 iPtkScanData, rPtkScanData)
```

## Variables to Be Provided to the Utility Routine

**cElsetName**

Name of the element set.

**cPtkID**

User given name of the scan strategy.

**iPtkScanOpts**

Integer array of size `iPtkScanPatternOpt_Size`.

| | |
|---|---|
| `iPtkScanOpts (iPtkScanPatternOpt_LayerByLayerActivation)` | Indicates which algorithm to use for activation. Set to 1 for sweep activation. Set to 0 for layer-by-layer activation. |

**iPtkScanData**

Variable length integer array.

| | |
|---|---|
| `iPtkScanData(iPtkIntScanPatternProps_numRegions)` | Number of scan regions: `nRegions`. |
| For $n^{th}$ region, n = 1 to `nRegions`: | |
| `iPtkScanData(iPtkIntScanPatternProps_variableDataStart + (size of data for (n-1) previous regions ) + iPtkIntScanRegion_numIslands)` | Number of scan patches/islands in the pattern: `nIsland`. |
| For $p^{th}$ patch, p = 1 to `nPatches`: | |
| `iPtkScanData(iPtkIntScanPatternProps_variableDataStart + (size of data for (n-1) previous regions ) + (size of (p-1) previous patches +iPtkIntScanIsland_StrainID)` | Strain tensor ID (see how to define an eigenstrain library below) assigned to $p^{th}$ scan patch. |

**rPtkScanData**

Variable length real array.

| | |
|---|---|
| `rPtkScanData (iPtkRealScanPatternProps_sliceHeight)` | Height $h$ of the slice/layers in the scan pattern. |
| `rPtkScanData(iPtkRealScanPatternProps_ori11)` | Component of vector $\mathbf{I}$ along the global $x_g$-direction. |
| `rPtkScanData(iPtkRealScanPatternProps_ori12)` | Component of vector $\mathbf{I}$ along the global $y_g$-direction. |
| `rPtkScanData(iPtkRealScanPatternProps_ori13)` | Component of vector $\mathbf{I}$ along the global $z_g$-direction. |
| `rPtkScanData(iPtkRealScanPatternProps_ori21)` | Component of vector $\mathbf{J}$ along the global $x_g$-direction. |
| `rPtkScanData(iPtkRealScanPatternProps_ori22)` | Component of vector $\mathbf{J}$ along the global $y_g$-direction. |
| `rPtkScanData(iPtkRealScanPatternProps_ori23)` | Component of vector $\mathbf{J}$ along the global $z_g$-direction. |
| `rPtkScanData(iPtkRealScanPatternProps_ori31)` | Component of vector $\mathbf{K}$ along the global $x_g$-direction. |
| `rPtkScanData(iPtkRealScanPatternProps_ori32)` | Component of vector $\mathbf{K}$ along the global $y_g$-direction. |
| `rPtkScanData(iPtkRealScanPatternProps_ori33)` | Component of vector $\mathbf{K}$ along the global $z_g$-direction. $\mathbf{I} - \mathbf{J} - \mathbf{K}$ must be orthonormal, and they form the build axis system. |
| `rPtkScanData(iPtkRealScanPatternProps_origin1)` | $x$-coordinate of the origin of build axis system. |
| `rPtkScanData(iPtkRealScanPatternProps_origin2)` | $y$-coordinate of the origin of build axis system. |
| `rPtkScanData(iPtkRealScanPatternProps_origin3)` | $z$-coordinate of the origin of build axis system. |
| `rPtkScanData(iPtkRealScanPatternProps_startTime)` | Print start time. |

| | |
|---|---|
| `rPtkScanData(iPtkRealScanPatternProps_endTime)` | Print end time. |
| For $n^{th}$ region, n = 1 to `nRegions`: | |
| `rPtkScanData (iPtkRealScanPatternProps_variableDataStart + (size of data for (n-1) previous regions ) + iPtkRealScanRegion_rotation)` | Layer-to-layer rotation angle in degrees. |
| `rPtkScanData(iPtkRealScanPatternProps_variableDataStart variableDataStart + (size of data for (n-1) previous regions ) + iPtkRealScanRegion_xMin : iPtkRealScanPatternProps_variableDataStart variableDataStart + (size of data for (n-1) previous regions ) + iPtkRealScanRegion_zMax)` | Array of length six. Defines ($x_{min}$, $y_{min}$, $z_{min}$) and ($x_{max}$, $y_{max}$, $z_{max}$) of the scanning region with respect to the build system $\mathbf{I} - \mathbf{J} - \mathbf{K}$ |
| For $i^{th}$ patch, i = 1 to `nIsland`: | |
| `rPtkScanData(iPtkRealScanPatternProps_variableDataStart + (size of data for (n-1) previous regions ) + (size of (i-1) previous islands + iPtkRealScanIsland_localRotation)` | Local rotation of $\varphi$ $i^{th}$ patch. |
| `rPtkScanData(iPtkRealScanPatternProps_variableDataStart + (size of data for (n-1) previous regions ) + (size of (i-1) previous islands ) + iPtkRealScanIsland_xMin : iPtkRealScanPatternProps_variableDataStart + (size of data for (n-1) previous regions ) + (size of (i-1) previous islands ) + iPtkRealScanIsland_yMax)` | Real array of size four. ($x_{min}$, $y_{min}$) and ($x_{max}$, $y_{max}$) of $i^{th}$ patch. |

## Figures

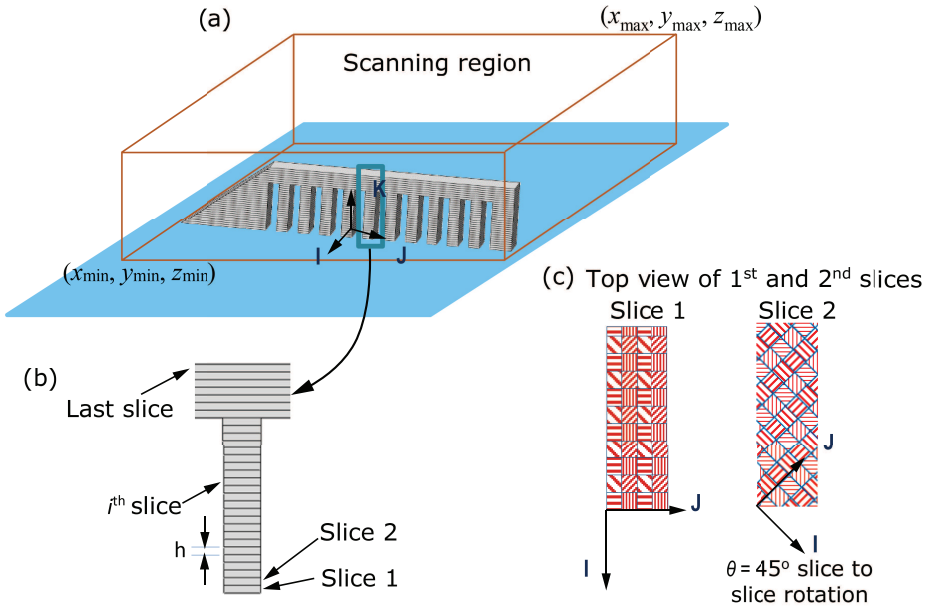



**Figure 1: Scan pattern.**

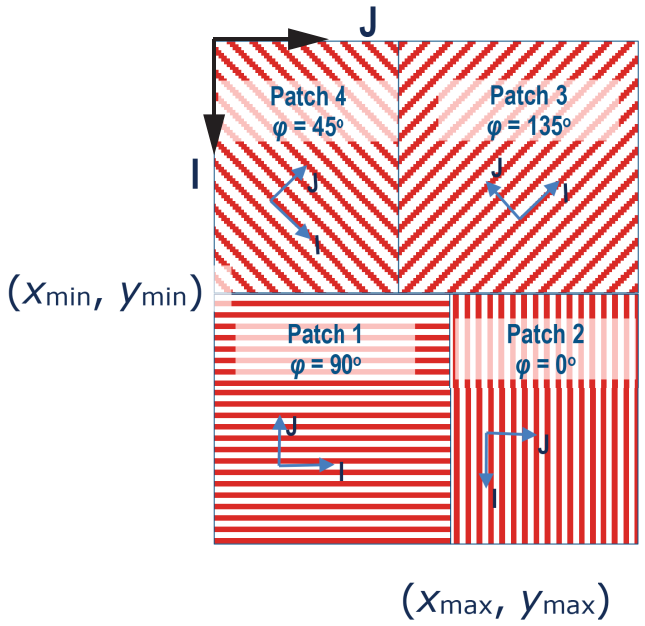



**Figure 1: Scan pattern orientation.**

## PtkSetEigLibrary

Create an eigenstrain library.

### Utility Routine Interface

```
#include <PtkUtilitySubs.hdr>
#include <aba_ptk_enums.inc>

      character *80 cEigLibID
      dimension iEigLib(iSMAPtkStrainLibOptions_Size)
      dimension rEigLib(6, numEigenStrainsInLib)
      character*80 cEigLib(numEigenStrainsInLib)

      call PtkSetEigLibrary (cEigLibID, iEigLib, rEigLib, cEigLib)
```

### Variables to Be Provided to the Utility Routine

**cEigLibID**

User-defined name for the eigenstrain library.

**iEigLib (iSMAPtkStrainLibOptions_Size)**

Integer array of size `iSMAPtkStrainLibOptions_Size`.

| | |
|---|---|
| `iEigLib (iSMAPtkStrainLibOptions_numStrains)` | Total number of strains in the library `numEigenStrainsInLib`. |

**rEigLib**

Real array of size 6 x `numEigenStrainsInLib`. Set to six components of eigenstrains for each of `numEigenStrainsInLib` strains.

**cEigLib**

`Character*80` array of size `numEigenStrainsInLib`. Set to a tag name of `Character*80` for each of `numEigenStrainsInLib` strains. These tags are used in diagnostic messages from the toolpath-mesh intersection module.

## PtkAssignEigLibToScanStrategies

Associate an eigenstrain library with a scan pattern.

### Utility Routine Interface

```
#include <PtkUtilitySubs.hdr>
#include <aba_ptk_enums.inc>

      character *80 cEigLibID, cPtkID
      call PtkAssignEigLibToScanStrategies (cEigLibID,cPtkID)
```

### Variables to Be Provided to the Utility Routine

**cEigLibID**

Name for the eigenstrain library defined in `PtkSetEigLibrary`.

**cPtkID**

Name of the scan strategy defined in `PtkSetMeshAndScanStrategies`.

## PtkAssignEigLibToES

Associate an eigenstrain library with an event series describing a toolpath.

### Utility Routine Interface

```
#include <PtkUtilitySubs.hdr>
#include <aba_ptk_enums.inc>

      character *80 cEigLibID, cPtkID
      call PtkAssignEigLibToES (cEigLibID,cPtkID)
```

### Variables to Be Provided to the Utility Routine

**cEigLibID**

Name for the eigenstrain library defined in `PtkSetEigLibrary`.

**cPtkID**

Tag name associated with an event series describing the toolpath.

# Compute Utility Routine

**Products:** Abaqus/Standard

**References:**

- *UEPACTIVATIONSETUP*
- *UMDFLUXSETUP*

## Overview

Toolpath-mesh intersection utility routines are used to activate the toolpath-mesh intersection module and to access data computed by the module.
The compute utility routine is invoked at the beginning of an increment from user subroutines *UEPACTIVATIONSETUP* and *UMDFLUXSETUP*.

### PtkCompute

Compute geometric intersection between a mesh and a toolpath within an analysis increment.

#### Utility Routine Interface

```
#include <PtkUtilitySubs.hdr>
#include <aba_ptk_enums.inc>

      character *80 cPtkID
      call PtkCompute (cPtkID,tStart,tEnd)
```

#### Variables to Be Provided to the Utility Routine

**cPtkID** Tag name of an event series defined in `PtkSetEventSeriesProperties` or name of a scan strategy defined in `PtkSetMeshAndScanStrategies`.

**tStart** Value of step (or total) time at the start of the increment when the event series is defined in terms of step (or total) time.

**tEnd** Value of step (or total) time at the end of the increment when the event series is defined in terms of step (or total) time.

# Data Retrieval Utility Routines

**Products:** Abaqus/Standard

**References:**

- *UEPACTIVATIONSETUP*
- *UMDFLUXSETUP*
- *UEPACTIVATIONVOL*
- *UEPACTIVATIONFACET*
- *UMDFLUX*
- *Toolpath-Mesh Intersection Module*

## Overview

Toolpath-mesh intersection utility routines are used to activate the toolpath-mesh intersection module and to access data computed by the module.
The data retrieval utility routines are invoked at the beginning of an increment after the call to a compute routine from user subroutines *UEPACTIVATIONSETUP* and *UMDFLUXSETUP* and from within the element loop from user subroutines *UEPACTIVATIONVOL*, *UEPACTIVATIONFACET*, and *UMDFLUX*.

### PtkGetScanPatternData

Retrieve scan pattern properties.

#### Utility Routine Interface

```
#include <PtkUtilitySubs.hdr>
#include <aba_ptk_enums.inc>

       character *80 cEigLibID, cPtkID
       dimension patternData(iPtkRealScanPatternData_Size)

       call ptkGetScanPatternData (cPtkID, rPatternData)
```

#### Variables to Be Provided to the Utility Routine

**cPtkID** Name of the scan strategy defined in `PtkSetMeshAndScanStrategies`.

#### Variables Returned from the Utility Routine

**rPatternData** Real array of size `iPtkRealScanPatternData_Size`.

`patternData(iPtkRealScanPatternData_sliceTime)` Time to complete activation of element in a slice.

### PtkGetNumIntersectedElements

Retrieve the number of elements that are intersected by a toolpath or a scan pattern.

### Utility Routine Interface

```
#include <PtkUtilitySubs.hdr>
#include <aba_ptk_enums.inc>

      character *80 cPtkID
      pointer(ptr_elemListHandler, elemListHandler)
      ptr_elemListHandler =
     *          PtkGetNumIntersectedElements(cPtkID,nIntersectedElems)
```

### Variables to Be Provided to the Utility Routine

**cPtkID** Tag name of an event series defined in `PtkSetEventSeriesProperties` or name of a scan strategy defined in `PtkSetMeshAndScanStrategies`.

### Variables Returned from the Utility Routine

**ptr_elemListHandler** A pointer to the list of element labels.

**nIntersectedElems** Number of intersected elements.

## PtkGetIntersectedElements

Retrieve list of labels of elements that are intersected by a toolpath.

### Utility Routine Interface

```
#include <PtkUtilitySubs.hdr>
#include <aba_ptk_enums.inc>

      character*80 cEventSeries
      PtkGetIntersectedElements (elemListHandler, jIntersectedElems)
```

### Variables to Be Provided to the Utility Routine

**elemListHandler** A pointer to the list of element labels retrieved by a previous call to `PtkGetNumIntersectedElements`.

### Variables Returned from the Utility Routine

**jIntersectedElems** List of element labels that are intersected by the toolpath. Integer array of size `nIntersectedElems`.

## PtkGetDataAccess

Retrieve number of records of geometric intersections for a given element.

### Utility Routine Interface

```
#include <PtkUtilitySubs.hdr>
#include <aba_ptk_enums.inc>

       pointer(ptr_iEventRecordHandler,iEventRecordHandler)
       character*80 cPtkID
       ptr_iEventRecordHandler=PtkGetDataAccess(cPtkID,
     *                                   noel, nRecords, nRecordLength,
     *                                  nFieldLength, jPtkShape, jPtkAlgo)
```

### Variables to Be Provided to the Utility Routine

**cPtkID** Tag name of an event series defined in `PtkSetEventSeriesProperties` or name of a scan strategy defined in `PtkSetMeshAndScanStrategies`.

**Noel** Element label.

### Variables Returned from the Utility Routine

**ptr_iEventRecordHandler** Pointer to the record of geometric intersections.

**nRecords** Number of geometric intersections within the element.

**nRecordLength** Length of the geometric intersection records.

**nFieldLength** Length of the field records.

**jPtkShape** Shape of the toolpath (`iPtkEventSeriesShape_Point`, `iPtkEventSeriesShape_InfiniteLine`, `iPtkEventSeriesShape_Box`, or `iPtkEventSeriesShape_ScanPattern`).

**jPtkAlgo** Algorithm used in geometric intersection of the mesh with the toolpath (`iPtkEventSeriesAlgorithm_SubElement`, `iPtkEventSeriesAlgorithm_SubSegment`, or `iPtkEventSeriesAlgorithm_Undefined`).

## PtkGetData

Retrieve records of geometric intersections for a given element.

### Utility Routine Interface

```
#include <PtkUtilitySubs.hdr>
#include <aba_ptk_enums.inc>

       call PtkGetData(iEventRecordHandler, Record, Field)
```

## Variables to Be Provided to the Utility Routine

**`iEventRecordHandler`** Pointer to the record of geometric intersections retrieved by calling `PtkGetDataAccess`.

## Variables Returned from the Utility Routine

**`Record`** Real array of size `nRecordLength`. See *Table 1* for a description of the record contents.

**`Field`** Real array of size `nFieldLength`. `Field` from the event series definition associated with intersecting segments, subsegments, or subelements. For subsegments, the associated field is found by multiplying the field of the main segment by the weight assigned with the subsegment.

## Description of Intersection Records

Quantities computed by the toolpath-mesh intersection method are stored in records. A record represents quantities from one intersection. For point and infinite line toolpaths, one intersection represents the intersection between one segment and a given element. For a box toolpath, one record represents the intersection between one subsegment and an element or one subelement that is inside the path. For toolpath descriptions, see the figures in *Toolpath-Mesh Intersection Module*.

The array `Record` returned by the routine `PtkGetData` contains records of intersections for a given element. Quantities stored in the array `Record` are listed in the tables below. To obtain quantities for the $i^{th}$ intersection, the index of the `Record` array must be offset by `(i-1) x record size`. Record sizes for different toolpaths are also listed in the tables below. All parameters are defined in the header file `aba_ptk_enums.inc`.

### Table 1: Intersection record for point toolpath. Size of one record is `nPtkPointEventRecord_Size`.

| Record array | Quantities |
|---|---|
| `Record(iPtkPointEventRecord_CoordStart1)` | *x*-coordinate of the start point of intersection with respect to the element reference coordinate. |
| `Record(iPtkPointEventRecord_CoordStart2)` | *y*-coordinate of the start point of intersection with respect to the element reference coordinate. |
| `Record(iPtkPointEventRecord_CoordStart3)` | *z*-coordinate of the start point of intersection with respect to the element reference coordinate. |
| `Record(iPtkPointEventRecord_CoordEnd1)` | *x*-coordinate of the end point of intersection with respect to the element reference coordinate. |
| `Record(iPtkPointEventRecord_CoordEnd2)` | *y*-coordinate of the end point of intersection with respect to the element reference coordinate. |
| `Record(iPtkPointEventRecord_CoordEnd3)` | *z*-coordinate of the end point of intersection with respect to the element reference coordinate. |
| `Record(iPtkPointEventRecord_timeStart)` | Start times of intersection. |
| `Record(iPtkPointEventRecord_timeEnd)` | End times of intersection. |

**Table 2: Intersection record for infinite line toolpath. Size of one record is `nPtkInfiniteLineEventRecord_Size`.**

| Record array | Quantities |
|---|---|
| `Record(iPtkInfiniteLineEventRecord_VolFraction)` | Volume fraction $v_f$. |
| `Record(iPtkInfiniteLineEventRecord_ExposedArea)` | Area $A$ of intersection. |
| `Record(iPtkInfiniteLineEventRecord_EACenter1)` | $x$-coordinate of the center X[a] of the intersection. |
| `Record(iPtkInfiniteLineEventRecord_EACenter2)` | $y$-coordinate of the center X[a] of the intersection. |
| `Record(iPtkInfiniteLineEventRecord_EACenter3)` | $z$-coordinate of the center X[a] of the intersection. |
| `Record(iPtkInfiniteLineEventRecord_ExposedATime)` | Time at which toolpath passes through the center of the intersection. |
| `Record(iPtkInfiniteLineEventRecord_AreaFracF1 : iPtkInfiniteLineEventRecord_AreaFracF6)` | Ratio of part of area of an element facet (for facets 1 to 6) below z-plane to the total area of the facet. |
| `Record(iPtkInfiniteLineEventRecord_AreaFracAddedF1 : iPtkInfiniteLineEventRecord_AreaFracAddedF6)` | Side facet (for facets 1 to 6) area between two $z$-planes. |
| `Record(iPtkInfiniteLineEventRecord_F1CenterAdded1)`<br>`Record(iPtkInfiniteLineEventRecord_F1CenterAdded2)`<br>`Record(iPtkInfiniteLineEventRecord_F1CenterAdded3)`<br>…<br>`Record(iPtkInfiniteLineEventRecord_F6CenterAdded3)` | Center of the facet area between two $z$-planes with respect to the element reference coordinate system. |
| `Record(iPtkInfiniteLineEventRecord_AreaFracDt1)`<br>…<br>`Record(iPtkInfiniteLineEventRecord_AreaFracDt6)` | Time duration between the toolpath crossing the center of the facet fraction to the end of the increment. |

**Table 3: Intersection record for box toolpath using the subsegment approach. Size of one record is `nPtkBoxSubSegmentEventRecord_Size`.**

| Record array | Quantities |
|---|---|
| `Record(iPtkBoxSubSegmentEventRecord_CoordStart1)`<br>`Record(iPtkBoxSubSegmentEventRecord_CoordStart2)`<br>`Record(iPtkBoxSubSegmentEventRecord_CoordStart3)`<br>`Record(iPtkBoxSubSegmentEventRecord_CoordEnd1)`<br>`Record(iPtkBoxSubSegmentEventRecord_CoordEnd2)`<br>`Record(iPtkBoxSubSegmentEventRecord_CoordEnd3)` | Start and end coordinates of intersection with respect to the element reference coordinate system. |
| `Record(iPtkBoxSubSegmentEventRecord_timeStart)`<br>`Record(iPtkBoxSubSegmentEventRecord_timeEnd)` | Start and end times of intersection. |

**Table 4: Intersection record for box toolpath using the subelement approach. Size of one record is `nPtkBoxSubElemEventRecord_Size`.**

| Record array | Quantities |
|---|---|
| `Record(iPtkBoxSubElemEventRecord_subVol)` | Volume of a subelement. |
| `Record(iPtkBoxSubElemEventRecord_elemVol)` | Volume of the parent element. |

| Record array | Quantities |
|---|---|
| `Record(iPtkBoxSubElemEventRecord_subCsiCenter1)` | *x*-coordinate of the subelement in element reference coordinate system. |
| `Record(iPtkBoxSubElemEventRecord_subCsiCenter2)` | *y*-coordinate of the subelement in element reference coordinate system. |
| `Record(iPtkBoxSubElemEventRecord_subCsiCenter3)` | *z*-coordinate of the subelement in element reference coordinate system. |
| `Record(iPtkBoxSubElemEventRecord_timeStart)` | Times when the front of the box passes through the center of the subelement. |
| `Record(iPtkBoxSubElemEventRecord_timeEnd)` | Times when the rear of the box passes through the center of the subelement. |

### Table 5: Intersection record for scan pattern. Size of one record is `nPtkScanPatternRecord_Size`.

| Record array | Quantities |
|---|---|
| `Record(iPtkScanPatternRecord_VolFraction)` | Volume fraction $v_f$. |
| `Record(iPtkScanPatternRecord_RegionID)` | ID of scanning region containing the center of the slice. Integer IDs are assigned to scanning regions internally starting from the 1st region. |
| `Record(iPtkScanPatternRecord_IslandID)` | ID of patch containing the center of the slice. Integer IDs are assigned to patches internally starting from the 1st patch. |
| `Record(iPtkScanPatternRecord_Orient11 : iPtkScanPatternRecord_Orient33)` | Three orthonormal vectors (nine real values) defining the local orientation of the strain patch that contains the center of the slice. |
| `Record(iPtkScanPatternRecord_Eig11 : iPtkScanPatternRecord_Eig23)` | Six components of eigenstrain (11-22-33-12-13-23) with respect to the local orientation. |

# Index

Several indexes are available for user subroutines.

### In this section:


- *User Subroutines Index*
- *User subroutine functions listing*

# User Subroutines Index

This index categorizes each user subroutine according to its primary function.

The following tables list the topics alphabetically.

### Table 1: Abaqus/Standard user subroutines.

| Function | Related user subroutines |
|---|---|
| Amplitudes, User-defined | *UAMP* |
| Boundary Conditions | *DISP*, *UDEMPOTENTIAL* |
| Constraints | *MPC* |
| Contact Behavior | *FRIC*, *FRIC_COEF*, *GAPCON*, *GAPELECTR*, *UINTER*, *UMIXMODEFATIGUE*, *UXFEMCRACK* |
| Contact Surfaces | *RSURFU* |
| Design Response | *UELEMDRESP* |
| Eigenstrain | *UEIGENSTRAIN* |
| Element Activation | *UEPACTIVATIONFACET*, *UEPACTIVATIONSETUP*, *UEPACTIVATIONVOL* |
| Element Output | *UVAREL*, *UVARM* |
| Elements, User-Defined | *UEL*, *UELMAT* |
| Fields, Predefined | *UFIELD*, *UMASFL*, *UPRESS*, *USDFLD*, *UTEMP* |
| Fluid Pipe Section Behavior | *UFLUIDCONNECTORLOSS*, *UFLUIDCONNECTORVALVE*, *UFLUIDPIPEFRICTION* |
| Intercalation and Aging Currents for a Battery Simulation Using Porous Electrode Theory | *UECHEMECDBV*, *UECHEMECDDISS*, *UECHEMECDLPL*, *UECHEMECDSEI*, *UECHEMECDSEIREFORM*, *UEREF* |
| Initial Conditions | *HARDINI*, *SDVINI*, *SIGINI*, *UPOREP*, *VOIDRI* |
| Interfacing with External Resources | *UEXTERNALDB*, *URDFIL* |
| Internal Source Generation, User-Defined | *UINTFLUXGEN* |
| Loads, Distributed | *DLOAD*, *UTRACLOAD* |
| Loads, Electromagnetic | *DECURRENT*, *UDECURRENT*, *UDSECURRENT* |
| Loads, Thermal | *FILM*, *HETVAL*, *UMDFLUX*, *UMDFLUXSETUP* |
| Material Properties | *CREEP*, *UANISOHYPER_INV*, *UANISOHYPER_STRAIN*, *UCOHESIVEOFFSET*, *UCREEPNETWORK*, *UCURE*, *UDAMAGEMF*, *UDMGINI*, *UEXPAN*, *UFLUID*, *UFLUIDLEAKOFF*, *UHARD*, *UHYPEL*, *UHYPER*, *UHYPER_STRETCH*, *UMULLINS*, *USUPERELASHARDMOD*, *UTRS*, *UTRSNETWORK*, *UXFEMNONLOCALWEIGHT* |
| Materials, User-Defined | *UCOUPLEDCONSTRESP*, *UMAT*, *UMATHT* |
| Motion, Prescribed | *UMESHMOTION*, *UMOTION* |
| Orientation | *ORIENT* |
| Pore Fluid Flow | *DFLOW*, *DFLUX*, *FLOW* |
| Random Response | *UCORR*, *UPSD* |
| Shell Section Behavior | *UGENS* |
| Slurry Transportation | *USETTLING* |

| Function | Related user subroutines |
|---|---|
| Wave Kinematics | *UWAVE* |

**Table 2: Abaqus/Explicit user subroutines.**

| Function | Related user subroutines |
|---|---|
| Amplitudes, User-defined | *VUAMP* |
| Boundary Conditions | *VDISP* |
| Contact Behavior | *VFRIC*, *VFRIC_COEF*, *VFRICTION*, *VUINTER*, *VUINTERACTION* |
| Elements, User-defined | *VUEL* |
| Fields, Predefined | *VUFIELD*, *VUSDFLD*, *VUTEMP* |
| Fluid Exchange, User-defined | *VUFLUIDEXCH*, *VUFLUIDEXCHEFFAREA* |
| Initial Conditions | *VSDVINI* |
| Interfacing with External Resources | *VEXTERNALDB* |
| Loads, Distributed | *VDLOAD* |
| Loads, Thermal | *VDFLUX*, *VHETVAL* , *VFILM* |
| Material Properties | *VFABRIC*, *VUANISOHYPER_INV*, *VUANISOHYPER_STRAIN*, *VUCHARLENGTH*, *VUCREEPNETWORK*, *VUEOS*, *VUEXPAN*, *VUHARD*, *VUHYPER*, *VUHYPER_STRETCH*, *VUMULLINS*, *VUSUPERELASHARDMOD*, *VUTRS*, *VUTRSNETWORK*, *VUVISCOSITY* |
| Materials, User-defined | *VUMAT*, *VUMATHT* |
| Shell Section Behavior | *VUGENS* |
| Wave Kinematics | *VWAVE* |

# User subroutine functions listing

The following tables describe the function of each available user subroutine.

### Abaqus/Standard User Subroutines

| Name | Function |
|---|---|
| *CREEP* | User subroutine to define time-dependent, viscoplastic behavior (creep and swelling). |
| *DECURRENT* | User subroutine to define electrical current density magnitude as a function of position, time, element number, field variables, etc. for coupled thermal-electrical, coupled temperature-displacement electrical, coupled thermal-electrochemical, and coupled temperature-displacement electrochemical analyses. |
| *DFLOW* | User subroutine to define nonuniform pore fluid velocity in a consolidation analysis. |
| *DFLUX* | User subroutine to define nonuniform distributed flux in a heat transfer or mass diffusion analysis. |
| *DISP* | User subroutine to specify prescribed boundary conditions. |
| *DLOAD* | User subroutine to specify nonuniform distributed loads. |
| *FILM* | User subroutine to define nonuniform film coefficient and associated sink temperatures for heat transfer analysis. |
| *FLOW* | User subroutine to define nonuniform seepage coefficient and associated sink pore pressure for consolidation analysis. |
| *FRIC* | User subroutine to define frictional behavior for contact surfaces. |
| *FRIC_COEF* | User subroutine to define the frictional coefficient for contact surfaces. |
| *GAPCON* | User subroutine to define conductance between contact surfaces or nodes in a fully coupled temperature-displacement analysis, coupled thermal-electrical-structural analysis, or pure heat transfer analysis. |
| *GAPELECTR* | User subroutine to define electrical conductance between surfaces in a coupled thermal-electrical or a coupled thermal-electrical-structural analysis. |
| *HARDINI* | User subroutine to define initial equivalent plastic strain and initial backstress tensor. |

| Name | Function |
|---|---|
| *HETVAL* | User subroutine to provide internal heat generation in heat transfer analysis. |
| *MPC* | User subroutine to define multi-point constraints. |
| *ORIENT* | User subroutine to provide an orientation for defining local material directions or local directions for kinematic coupling constraints or local rigid body directions for inertia relief. |
| *RSURFU* | User subroutine to define a rigid surface. |
| *SDVINI* | User subroutine to define initial solution-dependent state variable fields. |
| *SIGINI* | User subroutine to define an initial stress field. |
| *UAMP* | User subroutine to specify amplitudes. |
| *UANISOHYPER_INV* | User subroutine to define anisotropic hyperelastic material behavior using the invariant formulation. |
| *UANISOHYPER_STRAIN* | User subroutine to define anisotropic hyperelastic material behavior based on Green strain. |
| *UCOHESIVEOFFSET* | User subroutine to define the cohesive offset for pore pressure cohesive elements. |
| *UCORR* | User subroutine to define cross-correlation properties for random response loading. |
| *UCOUPLEDCONSTRESP* | User subroutine to define a fully coupled constitutive response based on the different physics that is being modeled. |
| *UCREEPNETWORK* | User subroutine to define time-dependent behavior (creep) for models defined within the parallel rheological framework. |
| *UCURE* | User subroutine to define cure kinetics |
| *UDAMAGEMF* | User subroutine to define damage evolution at the macro-level for a composite modeled with mean-field homogenization. |
| *UDECURRENT* | User subroutine to define nonuniform volume current density in an eddy current or magnetostatic analysis. |
| *UDEMPOTENTIAL* | User subroutine to define nonuniform magnetic vector potential on a surface in an eddy current or magnetostatic analysis. |
| *UDMGINI* | User subroutine to define the damage initiation criterion. |

| Name | Function |
|---|---|
| *UDSECURRENT* | User subroutine to define nonuniform surface current density in an eddy current or magnetostatic analysis. |
| *UECHEMECDBV* | User subroutine to define the Butler-Volmer intercalation current to be used for elements that utilize porous electrode theory in an electrochemical analysis. |
| *UECHEMECDDISS* | User subroutine to define the electrode dissolution aging current to be used for elements that utilize porous electrode theory in an electrochemical analysis. |
| *UECHEMECDLPL* | User subroutine to define the lithium plating (LPL) aging current to be used for elements that utilize porous electrode theory in an electrochemical analysis. |
| *UECHEMECDSEI* | User subroutine to define the Solid Electrolyte Interface (SEI) aging current to be used for elements that utilize porous electrode theory in an electrochemical analysis. |
| *UECHEMECDSEIREFORM* | User subroutine to define the Solid Electrolyte Interface (SEI) reformation current to be used for elements that utilize porous electrode theory in an electrochemical analysis. |
| *UEIGENSTRAIN* | User subroutine to define the eigenstrain in a material from different physics fields. |
| *UEL* | User subroutine to define an element. |
| *UELEMDRESP* | User subroutine to define an element design response and its derivatives. |
| *UELMAT* | User subroutine to define an element with access to materials. |
| *UEPACTIVATIONFACET* | User subroutine to specify the area fraction over which convective or radiative cooling is applied when using element activation. |
| *UEPACTIVATIONSETUP* | User subroutine to control interaction with the toolpath-mesh intersection module and to specify the list of elements that can be activated. |
| *UEPACTIVATIONVOL* | User subroutine to specify material volume fraction added during element activation. |
| *UEREF* | User subroutine to define the Open Circuit Potential (OCP) to be used for elements that utilize porous electrode theory in an electrochemical analysis. |
| *UEXPAN* | User subroutine to define incremental thermal or field strains. |

| Name | Function |
|---|---|
| *UEXTERNALDB* | User subroutine to manage user-defined external databases and calculate model-independent history information. |
| *UFIELD* | User subroutine to specify predefined field variables. |
| *UFLUID* | User subroutine to define fluid density and fluid compliance for hydrostatic fluid elements. |
| *UFLUIDCONNECTORLOSS* | User subroutine to define the loss coefficient for fluid flow in fluid pipe connector elements. |
| *UFLUIDCONNECTORVALVE* | User subroutine to define the valve opening to control flow in fluid pipe connector elements. |
| *UFLUIDLEAKOFF* | User subroutine to define the fluid leak-off coefficients for pore pressure cohesive elements. |
| *UFLUIDPIPEFRICTION* | User subroutine to define the frictional coefficient for fluid flow in fluid pipe elements. |
| *UGENS* | User subroutine to define the mechanical behavior of a shell section. |
| *UHARD* | User subroutine to define the yield surface size and hardening parameters for isotropic plasticity or combined hardening models. |
| *UHYPEL* | User subroutine to define a hypoelastic stress-strain relation. |
| *UHYPER* | User subroutine to define a hyperelastic material. |
| *UHYPER_STRETCH* | User subroutine to define a hyperelastic material in terms of principal stretches. |
| *UINTER* | User subroutine to define surface interaction behavior for contact surfaces. |
| *UINTFLUXGEN* | User subroutine to define the internal heat generation in a material from different physics fields. |
| *UMASFL* | User subroutine to specify prescribed mass flow rate conditions for a convection/diffusion heat transfer analysis. |
| *UMAT* | User subroutine to define a material's mechanical behavior. |
| *UMATHT* | User subroutine to define a material's thermal behavior. |
| *UMDFLUX* | User subroutine for specifying moving or stationary nonuniform heat flux in a heat transfer analysis. |

| Name | Function |
|---|---|
| *UMDFLUXSETUP* | User subroutine to communicate with the toolpath-mesh intersection module and to specify the list of elements that are subjected to the moving flux load. |
| *UMESHMOTION* | User subroutine to specify mesh motion constraints during adaptive meshing. |
| *UMIXMODEFATIGUE* | User subroutine to specify a user-defined mixed-mode form of the Paris law in a fatigue crack growth analysis. |
| *UMOTION* | User subroutine to specify motions during cavity radiation heat transfer analysis or steady-state transport analysis. |
| *UMULLINS* | User subroutine to define damage variable for the Mullins effect material model. |
| *UPOREP* | User subroutine to define initial fluid pore pressure. |
| *UPRESS* | User subroutine to specify prescribed equivalent pressure stress conditions. |
| *UPSD* | User subroutine to define the frequency dependence for random response loading. |
| *URDFIL* | User subroutine to read the results file. |
| *USDFLD* | User subroutine to redefine field variables at a material point. |
| *USETTLING* | User subroutine to define the settling velocity field in slurry transport analysis. |
| *USUPERELASHARDMOD* | User subroutine to modify the material constants of the superelasticity model as a function of the plastic strain. |
| *UTEMP* | User subroutine to specify prescribed temperatures. |
| *UTRACLOAD* | User subroutine to specify nonuniform traction loads. |
| *UTRS* | User subroutine to define a reduced time shift function for a viscoelastic material. |
| *UTRSNETWORK* | User subroutine to define a reduced time shift function for models defined within the parallel rheological framework. |
| *UVAREL* | User subroutine to generate element output. |
| *UVARM* | User subroutine to generate element output. |

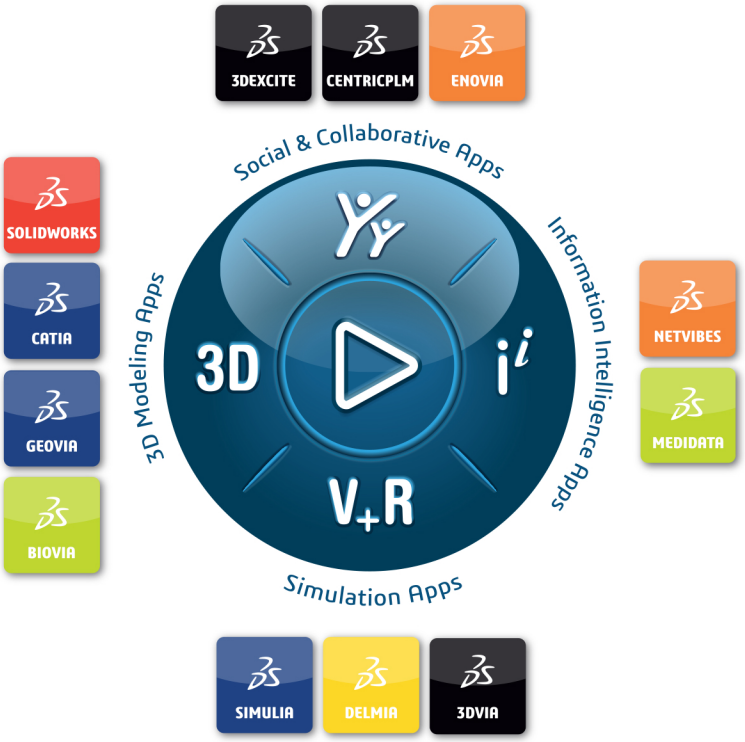
| Name | Function |
|---|---|
| *UWAVE* | User subroutine to define wave kinematics for an analysis. |
| *UXFEMCRACK* | User subroutine to define the interactions between the cracked element surfaces for enriched elements using a small sliding formulation. |
| *UXFEMNONLOCALWEIGHT* | User subroutine to define the weight function used to compute the average stress/strain to determine the crack propagation direction. |
| *VOIDRI* | User subroutine to define initial void ratios. |

### Abaqus/Explicit User Subroutines

| Name | Function |
|---|---|
| *VDFLUX* | User subroutine to specify nonuniform distributed fluxes in an explicit dynamic coupled temperature-displacement analysis. |
| *VDISP* | User subroutine to specify prescribed boundary conditions. |
| *VDLOAD* | User subroutine to specify nonuniform distributed loads. |
| *VEXTERNALDB* | User subroutine that gives control to the user at key moments of the analysis so that data can be exchanged dynamically among user subroutines and with external programs or files. |
| *VFABRIC* | User subroutine to define fabric material behavior. |
| *VFILM* | User subroutine to define nonuniform film coefficients and sink temperatures for heat transfer analysis. |
| *VFRIC* | User subroutine to define frictional behavior for contact surfaces. |
| *VFRIC_COEF* | User subroutine to define the frictional coefficient for contact surfaces. |
| *VFRICTION* | User subroutine to define frictional behavior for contact surfaces. |
| *VHETVAL* | User subroutine to provide internal heat generation in a heat transfer analysis. |
| *VSDVINI* | User subroutine to define initial solution-dependent state variable fields. |
| *VUAMP* | User subroutine to specify amplitudes. |

| Name | Function |
|---|---|
| *VUANISOHYPER_INV* | User subroutine to define anisotropic hyperelastic material behavior using the invariant formulation. |
| *VUANISOHYPER_STRAIN* | User subroutine to define anisotropic hyperelastic material behavior based on Green strain. |
| *VUCHARLENGTH* | User subroutine to define characteristic element length at a material point. |
| *VUCREEPNETWORK* | User subroutine to define time-dependent behavior (creep) for models defined within the parallel rheological framework. |
| *VUEL* | User subroutine to define an element. |
| *VUEOS* | User subroutine to define equation of state material model. |
| *VUEXPAN* | User subroutine to define thermal strain increments. |
| *VUFIELD* | User subroutine to specify predefined field variables. |
| *VUFLUIDEXCH* | User subroutine to define the mass flow rate/heat energy flow rate for fluid exchange. |
| *VUFLUIDEXCHEFFAREA* | User subroutine to define the effective area for fluid exchange. |
| *VUGENS* | User subroutine to define the mechanical behavior of a shell section. |
| *VUHARD* | User subroutine to define the yield surface size and hardening parameters for isotropic plasticity or combined hardening models. |
| *VUHYPER* | User subroutine to define isotropic hyperelastic material behavior using the invariant formulation. |
| *VUHYPER_STRETCH* | User subroutine to define isotropic hyperelastic material behavior using the stretch formulation. |
| *VUINTER* | User subroutine to define the interaction between contact surfaces. |
| *VUINTERACTION* | User subroutine to define the contact interaction between surfaces with the general contact algorithm. |
| *VUMAT* | User subroutine to define a material's mechanical behavior. |
| *VUMATHT* | User subroutine to define a material's thermal behavior. |
| *VUMULLINS* | User subroutine to define damage variable for the Mullins effect material model. |

| Name | Function |
| --- | --- |
| *VUSDFLD* | User subroutine to redefine field variables at a material point. |
| *VUSUPERELASHARDMOD* | User subroutine to modify the material constants of the superelasticity model as a function of the plastic strain. |
| *VUTEMP* | User subroutine to prescribe temperatures at the nodes of a model. |
| *VUTRS* | User subroutine to define a reduced time shift function for a viscoelastic material. |
| *VUTRSNETWORK* | User subroutine to define a reduced time shift function for models defined within the parallel rheological framework. |
| *VUVISCOSITY* | User subroutine to define the shear viscosity for equation of state models. |
| *VWAVE* | User subroutine to define wave kinematics for an analysis. |




## Our **3D**EXPERIENCE® platform powers our brand applications, serving 12 industries, and provides a rich portfolio of industry solution experiences.

Dassault Systèmes is a catalyst for human progress. We provide business and people with collaborative virtual environments to imagine sustainable innovations. By creating virtual twin experiences of the real world with our **3D**EXPERIENCE platform and applications, our customers can redefine the creation, production and life-cycle-management processes of their offer and thus have a meaningful impact to make the world more sustainable. The beauty of the Experience Economy is that it is a human-centered economy for the benefit of all – consumers, patients and citizens.

Dassault Systèmes brings value to more than 300,000 customers of all sizes, in all industries, in more than 150 countries. For more information, visit **www.3ds.com**.

**Europe/Middle East/Africa**
Dassault Systèmes
10, rue Marcel Dassault
CS 40501
78946 Vélizy-Villacoublay Cedex
France

**Asia-Pacific**
Dassault Systèmes
17F, Foxconn Building,
No. 1366, Lujiazui Ring Road
Pilot Free Trade Zone, Shanghai 200120
China

**Americas**
Dassault Systèmes
175 Wyman Street
Waltham, Massachusetts
02451-1223
USA

DASSAULT SYSTEMES

©2026 Dassault Systèmes. All rights reserved. **3DEXPERIENCE**, the 3DS logo, the Compass icon, IFWE, 3DEXCITE, 3DVIA, BIOVIA, CATIA, CENTRIC PLM, DELMIA, ENOVIA, GEOVIA, MEDIDATA, NETVIBES, OUTSCALE, SIMULIA and SOLIDWORKS are commercial trademarks or registered trademarks of Dassault Systèmes, a European company (Societas Europaea) incorporated under French law, and registered with the Versailles trade and companies registry under number 322 306 440, or its subsidiaries in the United States and/or other countries.