



## User-Defined Elements

User-defined elements:

- can be finite elements in the usual sense of representing a geometric part of the model;
- can be feedback links, supplying forces at some points as functions of values of displacement, velocity, etc. at other points in the model;
- can be used to solve equations in terms of nonstandard degrees of freedom;
- can be linear or nonlinear; and
- can access selected materials from the Abaqus material library.

This page discusses:

- [Assigning an Element Type Key to a User-Defined Element](#)
- [Invoking User-Defined Elements](#)
- [Defining the Active Degrees of Freedom at the Nodes](#)
- [Visualizing User-Defined Elements in Abaqus/CAE](#)
- [Defining a Linear User Element in Abaqus/Standard](#)
- [Defining a General User Element](#)

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

### Assigning an Element Type Key to a User-Defined Element

You must assign an element type key to a user-defined element. The element type key must be of the form  $Un$  in Abaqus/Standard and  $VUn$  in Abaqus/Explicit, where  $n$  is a positive integer that identifies the element type uniquely. For example, you can define element types  $U1$ ,  $U2$ ,  $U3$ ,  $VU1$ ,  $VU7$ , etc. In Abaqus/Standard  $n$  must be less than 10000; while in Abaqus/Explicit  $n$  must be less than 9000.

The element type key is used to identify the element in the element definition. For general

user elements the integer part of the identifier is provided in user subroutines [UEL](#), [UELMAT](#) and [VUEL](#) so that you can distinguish between different element types.

### Input File Usage:

```
\*USER\_ELEMENT, TYPE=element_type
```

## Invoking User-Defined Elements

User-defined elements are invoked in the same way as native Abaqus elements: you specify the element type, *Un* or *VUn*, and define element numbers and nodes associated with each element (see [Abaqus Model Definition](#)). User elements can be assigned to element sets in the usual way, for cross-reference to element property definitions, output requests, distributed load specifications, etc.

Material definitions ([Material Data Definition](#)) are relevant only to user-defined elements in Abaqus/Standard. If a material is assigned to a user-defined element ([Assigning an Abaqus Material to the User Element](#)), user subroutine [UELMAT](#) will be used to define the element response. User subroutine [UELMAT](#) allows access to selected Abaqus materials. If no material definition is specified, all material behavior must be defined in user subroutines [UEL](#) and [VUEL](#), based on user-defined material constants and on solution-dependent state variables associated with the element and calculated in the same subroutines. For linear user elements all material behavior must be defined through a user-defined stiffness matrix.

### Input File Usage:

Use the following options to invoke a user-defined element:

```
\*USER\_ELEMENT, TYPE=element_type
```

```
\*ELEMENT, TYPE=element_type
```

## Defining the Active Degrees of Freedom at the Nodes

Any number of user element types can be defined and used in a model. Each user element can have any number of nodes, at each of which a specified set of degrees of freedom is used by the element. The activated degrees of freedom should follow the Abaqus convention ([Conventions](#)). In Abaqus/Standard this is important because the convergence criteria are based on the degrees of freedom numbers. In Abaqus/Explicit the activated degrees of freedom must follow the Abaqus convention because these are the only degrees of freedom that can be updated.

Abaqus always works in the global system when passing information to or from a user element. Therefore, the user element's stiffness, mass, etc. should always be defined with respect to global directions at its nodes, even if local transformations ([Transformed Coordinate Systems](#)) are applied to some of these nodes.

You define the ordering of the variables on a user element. The standard and recommended ordering is such that the degrees of freedom at the first node appear first, followed by the degrees of freedom at the second node, etc. For example, suppose that the user-defined element type is a planar beam with three nodes. The element uses degrees of freedom 1, 2, and 6 ( $u_x$ ,  $u_y$ , and  $\phi_z$ ) at its first and last node and degrees of freedom 1 and 2 at its second (middle) node. In this case the ordering of variables on the element is:

Element variable number	Node	Degree of freedom
1	1	1
2	1	2
3	1	6
4	2	1
5	2	2
6	3	1
7	3	2
8	3	6

This ordering will be used in most cases. However, if you define an element matrix that does not have its degrees of freedom ordered in this way, you can change the ordering of the degrees of freedom as outlined below.

You specify the active degrees of freedom at each node of the element. If the degrees of freedom are the same at all of the element's nodes, you specify the list of degrees of freedom only once. Otherwise, you specify a new list of degrees of freedom each time the degrees of freedom at a node are different from those at previous nodes. Thus, different nodes of the element can use different degrees of freedom; this is especially useful when the element is being used in a coupled field problem so that, for example, some of its nodes have displacement degrees of freedom only, while others have displacement and temperature degrees of freedom. This method will produce an ordering of the element variables such that all of the degrees of freedom at the first node appear first, followed by the degrees of freedom at the second node, etc.

In Abaqus/Standard there are two ways to define element variable numbers that order the degrees of freedom on the element differently.

Since the user element can accept repeated node numbers when defining the nodal

connectivity for the element, the element can be declared to have one node per degree of freedom on the element. For example, if the element is a planar, 3-node triangle for stress analysis, it has three nodes, each of which has degrees of freedom 1 and 2. If all degrees of freedom 1 are to appear first in the element variables, the element can be defined with six nodes, the first three of which have degree of freedom 1, while nodes 4–6 have degree of freedom 2. The element variables would be ordered as follows:

Element variable number	Node	Degree of freedom
1	1	1
2	2	1
3	3	1
4	4	2
5	5	2
6	6	2

Alternatively, the user element variables can be defined so as to order the degrees of freedom on the element in any arbitrary fashion. You specify a list of degrees of freedom for the first node on the element. All nodes with a nodal connectivity number that is less than the next connectivity number for which a list of degrees of freedom is specified will have the first list of degrees of freedom. The second list of degrees of freedom will be used for all nodes until a new list is defined, etc. If a new list of degrees of freedom is encountered with a nodal connectivity number that is less than or equal to that given with the previous list, the previous list's degrees of freedom will be assigned through the last node of the element. This generation of degrees of freedom can be stopped before the last node on the element by specifying a nodal connectivity number with an empty (blank) list of degrees of freedom.

## Example

The above procedure continues using this new list to define additional degrees of freedom according to the new node and degrees of freedom. For example, consider a 3-node beam that has degrees of freedom 1, 2, and 6 at nodes 1 and 3 and degrees of freedom 1 and 2 at node 2 (middle node). To order degrees of freedom 1 first, followed by 2, followed by 6, the following input could be used:

\*USER ELEMENT

1

1, 2

1, 6

2,

3, 6

In this case the ordering of the variables on the element is:

Element variable number	Node	Degree of freedom
1	1	1
2	2	1
3	3	1
4	1	2
5	2	2
6	3	2
7	1	6
8	3	6

### Requirements for Activated Degrees of Freedom in Abaqus/Explicit

There are the following additional requirements with respect to activated degrees of freedom on a user element of type VUn:

- Only degrees of freedom 1 through 6, 8, and 11 can be activated because these are the only degrees of freedom numbers that can be updated in Abaqus/Explicit. (In Abaqus/Standard degrees of freedom 1 through 30 can be used.)
- If one translational degree of freedom is activated at a node, all translational degrees of freedom up to the specified maximum number of coordinates must be activated at that node; moreover, the translational degrees of freedom at the node must be in consecutive order.
- In three-dimensional analyses, if one rotational degree of freedom is activated at a node, all three rotational degrees of freedom must be activated in consecutive order.

For example, if you define a 4-node three-dimensional user element that has translations and rotations active at the first and fourth nodes, temperature only at the second node, and translations and temperature at the third node, the following input could be used:

#### \*USER ELEMENT

1, 2, 3, 4, 5, 6

2, 11

3, 1, 2, 3, 11

4, 1, 2, 3, 4, 5, 6

## Rotation Update in Geometrically Nonlinear Analyses

If all three rotational degrees of freedom (4, 5, and 6) are used at a node in a geometrically nonlinear analysis, Abaqus assumes that these rotations are finite rotations. In this case the incremental values of these degrees of freedom are not simply added to the total values: the quaternion update formulae are used instead. Similarly, the corrections are not simply added to the incremental values. The update procedure is described in [Rotation variables](#) and is mentioned in [Conventions](#).

To avoid the rotation update in a geometrically nonlinear analysis in Abaqus/Standard, you may define repeated node numbers in the nodal connectivity of the element such that at least one of the degrees of freedom 4, 5, or 6 is missing from the degree of freedom list at each node.

## Visualizing User-Defined Elements in Abaqus/CAE

Plotting of user elements is not supported in Abaqus/CAE. However, if the user elements contain displacement degrees of freedom, they can be overlaid with standard elements; and model plots of these standard elements can be displayed, allowing you to see the shape of the user elements. If deformed mesh plots of the user elements are required, the material properties of the overlaying standard elements must be chosen so that the solution is not changed by including them. If this technique is used, nodes of the user element will be tied to nodes of the standard elements. Therefore, degrees of freedom 1, 2, and 3 in the user element must correspond to the displacement degrees of freedom at the nodes of the standard elements.

## Defining a Linear User Element in Abaqus/Standard

Linear user elements can be defined only in Abaqus/Standard. In the simplest case a linear user element can be defined as a stiffness matrix and, if required, a mass matrix, as well as viscous and/or structural damping matrices. Stiffness and mass matrices can be read from a results file or defined directly. When damping is required, the linear user element must be defined directly. The damping matrices cannot be stored on a result file.

## Reading the Element Matrices from an Abaqus/Standard Results File

To read the element matrices from an Abaqus/Standard results file, you must have written the stiffness and/or mass matrices in a previous analysis to the results file as element matrix output ([Element Matrix Output in Abaqus/Standard](#)) or substructure matrix output ([Writing](#)

[the Recovery Matrix, Reduced Stiffness Matrix, Mass Matrix, Load Case Vectors, and Gravity Vectors to a File](#)). Damping matrices cannot be written to the results file.

You must specify the element number,  $n$ , or substructure identifier,  $Zn$ , to which the matrices correspond. For models defined in terms of an assembly of part instances ([Assembly Definition](#)), the element numbers written to the results file are internal numbers generated by Abaqus/Standard (see [About Output](#)). A map between these internal numbers and the original element numbers and part instance names is provided in the data file of the analysis from which the element matrix output was written.

In addition, for element matrix output you must specify the step number and increment number at which the element matrix was written. These items are not required if a substructure whose matrix was output during its generation is used.

#### **Input File Usage:**

[\\*USER ELEMENT](#), FILE=*name*, OLD ELEMENT= $n$  or  $Zn$ , STEP= $n$ , INCRE

### **Defining the Linear User Element by Specifying the Matrices Directly**

If you define the stiffness, mass, or damping matrices directly, you must specify the number of nodes associated with the element.

#### **Input File Usage:**

[\\*USER ELEMENT](#), LINEAR, NODES= $n$

### **Defining Whether or Not the Element Matrices Are Symmetric**

If the element matrices are not symmetric, you can request that Abaqus/Standard use its nonsymmetric equation solution capability (see [Defining an Analysis](#)).

#### **Input File Usage:**

[\\*USER ELEMENT](#), LINEAR, NODES= $n$ , UNSYMM

### **Defining the Mass, Stiffness, or Damping Matrix**

You define the element mass matrix, the element stiffness matrix, and the element viscous or structural damping matrix as needed. If the element is a heat transfer element, the "stiffness matrix" is the conductivity matrix and the "mass matrix" is the specific heat matrix.

You can define any applicable types of matrices simultaneously.

You can read the mass and/or stiffness matrices from a file or define them directly. When damping matrices are needed, all of the matrices must be defined directly. In either case Abaqus/Standard reads four values per line, using F20 format. This format ensures that the data are read with adequate precision. Data written in E20.14 format can be read under this

format.

Start with the first column of the matrix. Start a new line for each column. If you do not specify that the element matrix is unsymmetric, give the matrix entries from the top of each column to the diagonal term only: do not give the terms below the diagonal. If you specify that the element matrix is unsymmetric, give all terms in each column, starting from the top of the column.

### Input File Usage:

Use the following option to define the element mass matrix:

\*MATRIX, TYPE=MASS

Use the following option to define the element stiffness matrix:

\*MATRIX, TYPE=STIFFNESS

Use the following option to define the element viscous damping matrix:

\*MATRIX, TYPE=VISCOUS DAMPING

Use the following option to define the element structural damping matrix:

\*MATRIX, TYPE=STRUCTURAL DAMPING

Use the following option to read the element mass or stiffness matrix from a file:

\*MATRIX, TYPE=MASS or STIFFNESS, INPUT=*file\_name*

For example, if the matrix is symmetric, the following data lines should be used:

$A_{11}$   
 $A_{12}, A_{22}$   
 $A_{13}, A_{23}, A_{33}$   
 $A_{14}, A_{24}, A_{34}, A_{44}$   
 $A_{15}, A_{25}, A_{35}, A_{45}$   
 $A_{55}$   
 $A_{16}, A_{26}, A_{36}, A_{46}$   
 $A_{56}, A_{66}$   
Etc.

If the matrix is unsymmetric, the following data lines should be used:

$A_{11}, A_{21}, A_{31}, A_{41}$



$A_{51}, A_{61}, A_{71}, A_{81}$

...

...,  $A_{m1}$

$A_{12}, A_{22}, A_{32}, A_{42}$

Etc.

where  $m$  is the size of the matrix and  $A_{ij}$  is the entry in the matrix for row  $i$  column  $j$ .

## Geometrically Nonlinear Analysis

When a linear user element is used in a geometrically nonlinear analysis, the stiffness matrix provided will not be updated to account for any nonlinear effects such as finite rotations.

## Defining the Element Properties

You must associate a property definition with every user element, even though no property values (except Rayleigh and/or structural damping factors) are associated with linear user elements.

### Input File Usage:

Use the following option to associate a property definition with a user element set:

\*UEL PROPERTY, ELSET=*name*

## Defining Damping for Dynamic Analyses

You can define the Rayleigh damping factors for dynamic analyses ([Implicit Dynamic Analysis Using Direct Integration](#)) only for linear user elements. These damping factors can be defined for mode-based transient analysis, mode-based steady-state analysis, and direct steady-state analysis for both the linear and nonlinear formulations. The Rayleigh damping factors are defined as

$$[\mathbf{C}] = \alpha [\mathbf{M}] + \beta [\mathbf{K}],$$

where  $[\mathbf{C}]$  is the damping matrix,  $[\mathbf{M}]$  is the mass matrix,  $[\mathbf{K}]$  is the stiffness matrix, and  $\alpha$  and  $\beta$  are the user-specified damping factors. See [Material Damping](#) for more information on Rayleigh damping.

You can define the structural damping factor ( $s$ ) for direct and mode-based steady-state analyses. The damping matrix resulting from this factor is proportional to the user element stiffness matrix and is defined as

$$[\mathbf{S}] = s [\mathbf{K}].$$

### Input File Usage:

Use the following option to specify the Rayleigh damping factors:

\*UEL PROPERTY, ELSET=*name*, ALPHA= $\alpha$ , BETA= $\beta$

Use the following option to specify the structural damping factor:

\*UEL PROPERTY, ELSET=*name*, STRUCTURAL=*s*

## Defining Loads

You can apply point loads, moments, fluxes, etc. to the nodes of linear user-defined elements in the usual way using concentrated loads and concentrated fluxes ([Concentrated Loads](#) and [Thermal Loads](#)).

Distributed loads and fluxes cannot be defined for linear user-defined elements.

## Defining a General User Element

General user elements are defined in user subroutines [UEL](#) and [UELMAT](#) in Abaqus/Standard and in user subroutine [VUEL](#) in Abaqus/Explicit. *The implementation of user elements in user subroutines is recommended only for advanced users.*

## Defining the Number of Nodes Associated with the Element

You must specify the number of nodes associated with a general user element. You can define “internal” nodes that are not connected to other elements.

### Input File Usage:

\*USER ELEMENT, NODES=*n*

## Defining Whether or Not the Element Matrices Are Symmetric in Abaqus/Standard

If the contribution of the element to the Jacobian operator matrix of the overall Newton method is not symmetric (i.e., the element matrices are not symmetric), you can request that Abaqus/Standard use its nonsymmetric equation solution capability (see [Defining an Analysis](#)).

### Input File Usage:

\*USER ELEMENT, NODES=*n*, UNSYMM

## Defining the Maximum Number of Coordinates Needed at Any Nodal Point

You can define the maximum number of coordinates needed in user subroutines [UEL](#), [UELMAT](#), or [VUEL](#) at any node point of the element. Abaqus assigns space to store this many coordinate values at all of the nodes associated with elements of this type. The default maximum number of coordinates at each node is 1.

Abaqus will change the maximum number of coordinates to be the maximum of the user-specified value or 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 a maximum number of coordinates of 1 and the active degrees of freedom of the user element are 2, 3, and 6, the maximum number of coordinates will be changed to 3. If you specify a maximum number of coordinates of 2 and the active degrees of freedom of the user element are 11 and 12, the maximum number of coordinates will remain as 2.

### Input File Usage:

[\\*USER ELEMENT](#), COORDINATES= $n$

## Defining the Element Properties

You can define the number of properties associated with a particular user element and then specify their numerical values.

### Specifying the Number of Property Values Required

Any number of properties can be defined to be used in forming a general user element. You can specify the number of integer property values required,  $n$ , and the number of real (floating point) property values required,  $m$ ; the total number of values required is the sum of these two numbers. The default number of integer property values required is 0 and the default number of real property values required is 0.

Integer property values can be used inside user subroutines [UEL](#), [UELMAT](#), and [VUEL](#) as flags, indices, counters, etc. Examples of real (floating point) property values are the cross-sectional area of a beam or rod, thickness of a shell, and material properties to define the material behavior for the element.

### Input File Usage:

[\\*USER ELEMENT](#), I PROPERTIES= $n$ , PROPERTIES= $m$

### Specifying the Numerical Values of Element Properties

You must associate a user element property definition with each user-defined element, even if no property values are required. The property values specified in the property definition are passed into user subroutines [UEL](#), [UELMAT](#), and [VUEL](#) each time the subroutine is called for the user elements that are in the specified element set.

### Input File Usage:

Use the following option to associate a property definition with a user element set:

\*UEL PROPERTY, ELSET=*name*

To define the property values, enter all floating point values on the data lines first, followed immediately by the integer values. Eight values should be entered on all data lines except the last one, which may have fewer than eight values.

### Assigning an Abaqus Material to the User Element

If the Abaqus material library is accessed from a user element, a material must be defined and assigned to the user element.

#### Input File Usage:

Use the following option to associate a material with the user element:

\*UEL PROPERTY, MATERIAL=*name*

If this option is used, user subroutine [UELMAT](#) must be used to define the contribution of the element to the model. Otherwise, user subroutine [UEL](#) must be used.

### Assigning an Orientation Definition

If the Abaqus material library is accessed from a user element, you can associate a material orientation definition ([Orientations](#)) with the user element. The orientation definition specifies a local coordinate system for material calculations in the element. The local coordinate system is assumed to be uniform in a given element and is based on the coordinates at the element centroid.

#### Input File Usage:

Use the following option to associate an orientation definition with a user element:

\*UEL PROPERTY, ORIENTATION=*name*

### Specifying the Element Type

If the Abaqus material library is accessed from a user element, the element type must be specified.

#### Input File Usage:

Use the following option to define a three-dimensional element in a

stress/ displacement or a heat transfer analysis:

[\\*USER ELEMENT](#), TENSOR=THREED

Use the following option to define a two-dimensional element in a heat transfer analysis:

[\\*USER ELEMENT](#), TENSOR=TWOD

Use the following option to define a plane strain element in a stress/ displacement analysis:

[\\*USER ELEMENT](#), TENSOR=PSTRAIN

Use the following option to define a plane stress element in a stress/ displacement analysis:

[\\*USER ELEMENT](#), TENSOR=PSTRESS

### Specifying the Number of Integration Points

If the Abaqus material library is accessed from a user element, the number of integration points must be specified.

#### Input File Usage:

Use the following option to specify the number of integration points:

[\\*USER ELEMENT](#), INTEGRATION=*n*

### Defining the Number of Solution-Dependent Variables That Must Be Stored within the Element

You can define the number of solution-dependent state variables that must be stored within a general user element. The default number of variables is 1.

Examples of such variables are strains, stresses, section forces, and other state variables (for example, hardening measures in plasticity models) used in the calculations within the element. These variables allow quite general nonlinear kinematic and material behavior to be modeled. These solution-dependent state variables must be calculated and updated in user subroutines [UEL](#), [UELMAT](#), and [VUEL](#).

As an example, suppose the element has four numerical integration points, at each of which you wish to store strain, stress, inelastic strain, and a scalar hardening variable to define the material state. Assume that the element is a three-dimensional solid, so that there are six components of stress and strain at each integration point. Then, the number of solution-dependent variables associated with each such element is  $4 \times (6 \times 3 + 1) = 76$ .

## Input File Usage:

\*USER ELEMENT, VARIABLES=*n*

## Defining the Contribution of the Element to the Model in User Subroutine

### UEL

For a general user element in Abaqus/Standard, user subroutine UEL may be coded to define the contribution of the element to the model. Abaqus/Standard calls this routine each time any information about a user-defined element is needed. At each such call Abaqus/Standard provides the values of the nodal coordinates and of all solution-dependent nodal variables (displacements, incremental displacements, velocities, accelerations, etc.) at all degrees of freedom associated with the element, as well as values, at the beginning of the current increment, of the solution-dependent state variables associated with the element.

Abaqus/Standard also provides the values of all user-defined properties associated with this element and a control flag array indicating what functions the user subroutine must perform. Depending on this set of control flags, the subroutine must define the contribution of the element to the residual vector, define the contribution of the element to the Jacobian (stiffness) matrix, update the solution-dependent state variables associated with the element, form the mass and damping matrix, and so on. Often, several of these functions must be performed in a single call to the routine. In mode-based analyses the user element is formulated only once and projected onto the eigensystem.

## Formulation of an Element with User Subroutine UEL

The element's principal contribution to the model during general analysis steps is that it provides nodal forces  $F^N$  that depend on the values of the nodal variables  $u^M$  and on the solution-dependent state variables  $H^\alpha$  within the element:

$$F^N = F^N(u^M, H^\alpha, \text{geometry, attributes, predefined field variables, ...})$$

Here we use the term “force” to mean that quantity in the variational statement that is conjugate to the basic nodal variable: physical force when the associated degree of freedom is physical displacement, moment when the associated degree of freedom is a rotation, heat flux when it is a temperature value, and so on. The signs of the forces in  $F^N$  are such that external forces provide positive nodal force values and “internal” forces caused by stresses, internal heat fluxes, etc. in the element provide negative nodal force values. For example, in the case of mechanical equilibrium of a finite element subject to surface tractions  $\mathbf{t}$  and body forces  $\mathbf{f}$  with stress  $\boldsymbol{\sigma}$ , and with interpolation  $\delta \mathbf{u} = \mathbf{N}^N \delta u^N$ ,  $\delta \boldsymbol{\epsilon} = \boldsymbol{\beta}^N \delta u^N$ ,

$$F^N = \int_S \mathbf{N}^N \cdot \mathbf{t} dS + \int_V \mathbf{N}^N \cdot \mathbf{f} dV - \int_V \boldsymbol{\beta}^N : \boldsymbol{\sigma} dV.$$

In general procedures Abaqus/Standard solves the overall system of equations by Newton's method:

<i>Solve</i>	$\widetilde{K}^{NM} c^M = R^M,$
<i>Set</i>	$u^N = u^N + c^N,$
<i>Iterate</i>	

where  $R^N$  is the residual at degree of freedom  $N$  and

$$\widetilde{K}^{NM} = -\frac{dR^N}{du^M}$$

is the Jacobian matrix.

During such iterations you must define  $F^N$ , which is the element's contribution to the residual,  $R^N$ , and

$$-\frac{dF^N}{du^M},$$

which is the element's contribution to the Jacobian  $\widetilde{K}^{NM}$ . By writing the total derivative  $-dF^N/du^M$ , we imply that the element's contribution to  $\widetilde{K}^{NM}$  should include all direct and indirect dependencies of the  $F^N$  on the  $u^M$ . For example, the  $H^\alpha$  will generally depend on  $u^M$ ; therefore,  $-dF^N/du^M$  will include terms such as

$$-\frac{\partial F^N}{\partial H^\alpha} \frac{\partial H^\alpha}{\partial u^M}.$$

### Use in Transient Analysis Procedures

In procedures such as transient heat transfer and dynamic analysis, the problem also involves time integration of rates of change of the nodal degrees of freedom. The time integration schemes used by Abaqus/Standard for the various procedures are described in more detail in the [Introduction: general](#). For example, in transient heat transfer analysis, the backward difference method is used:

$$\dot{u}_{t+\Delta t} = \frac{1}{\Delta t} (u_{t+\Delta t} - u_t).$$

Therefore, if  $F^N$  depends on  $u^M$  and  $\dot{u}^M$  (as would be the case if the user element includes thermal energy storage), the Jacobian contribution should include the term

$$-\frac{\partial F^N}{\partial \dot{u}^M} \left( \frac{d\dot{u}}{du} \right)_{t+\Delta t},$$

where  $(d\dot{u}/du)_{t+\Delta t}$  is defined from the time integration procedure as  $1/\Delta t$ .

In all cases where Abaqus/Standard integrates first-order problems in time, the  $\dot{u}^M$  are never stored because they are readily available as  $\Delta u^M / \Delta t$ , where  $\Delta u^M = u_{t+\Delta t}^M - u_t^M$ . However, for direct, implicit integration of dynamic systems (see [Implicit dynamic analysis](#)) Abaqus/Standard requires storage of  $\dot{u}^M$  and  $\ddot{u}^M$ . These values are, therefore, passed into subroutine [UEL](#). If the user element contains effects that depend on these time derivatives (damping and inertial effects), its Jacobian contribution will include

$$-\frac{\partial F^N}{\partial u^M} - \frac{\partial F^N}{\partial \dot{u}^M} \left( \frac{d\dot{u}}{du} \right)_{t+\Delta t} - \frac{\partial F^N}{\partial \ddot{u}^M} \left( \frac{d\ddot{u}}{du} \right)_{t+\Delta t}.$$

For the Hilber-Hughes-Taylor scheme

$$\begin{aligned} \left( \frac{d\dot{u}}{du} \right)_{t+\Delta t} &= \frac{\gamma}{\beta \Delta t}, \\ \left( \frac{d\ddot{u}}{du} \right)_{t+\Delta t} &= \frac{1}{\beta \Delta t^2}, \end{aligned}$$

where  $\beta$  and  $\gamma$  are the (Newmark) parameters of the integration scheme. For backwark Euler time integration, the same expressions apply with  $\beta$  and  $\gamma$  equal to unity. The term  $-\partial F^N / \partial \dot{u}^M$  is the element's damping matrix, and  $-\partial F^N / \partial \ddot{u}^M$  is its mass matrix.

The Hilber-Hughes-Taylor scheme writes the overall dynamic equilibrium equations as



$$-M^{NM}\ddot{u}_{t+\Delta t} + (1 + \alpha) G_{t+\Delta t}^N - \alpha G_t^N = 0,$$

where  $G^N$  is the total force at degree of freedom  $N$ , excluding d'Alembert (inertia) forces.  $G^N$  is often referred to as the "static residual." Therefore, if a user element is to be used with Hilber-Hughes-Taylor time integration, the element's contribution  $F^N$  to the overall residual must be formulated in the same way. Since Abaqus/Standard provides information only at the time point at which [UEL](#) is called, this implies that each time [UEL](#) is called the  $H_\alpha$  array must be used to recover  $G_t^N$  (and  $G_{t^-}^N$  if half-increment residual calculations are required, where  $t^-$  indicates  $G^N$  from the beginning of the previous increment) and used to store  $G_{t+\Delta t}$  (and  $G_t^N$  if half-increment residual calculations are required) for use in the next increment. This complication can be avoided if the numerical damping control parameter,  $\alpha$ , for the dynamic step is set to zero; i.e., if the trapezoidal rule is used for integration of the dynamic equations (see [Implicit Dynamic Analysis Using Direct Integration](#) for details). This complication is also avoided with the backward Euler time integration operator because dynamic equilibrium is enforced at the end of the step.

If solution-dependent state variables ( $H^\alpha$ ) are used in the element, a suitable time integration method must be coded into subroutine [UEL](#) for these variables. Any of the  $u^N$  associated with the element that are not shared with standard Abaqus/Standard elements may be integrated in time by any suitable technique. If, in such cases, it is necessary to store values of  $u^N$ ,  $\dot{u}^N$ , etc. at particular points in time, the solution-dependent state variable array,  $H_\alpha$ , can be used for this purpose. Abaqus/Standard will still compute and store values of  $\dot{u}^N$  and  $\ddot{u}^N$  using the formulae associated with whatever time integrator it is using, but these values need not be used. To ensure accurate, stable time integration, you can control the size of the time increment used by Abaqus/Standard.

### Constraints Defined with Lagrange Multipliers

Introduction of constraints with Lagrange multipliers should be avoided since Abaqus/Standard cannot detect such variables and avoid eigensolver problems by proper ordering of the equations.

## Defining the Contribution of the Element to the Model in User Subroutine

### UELMAT

Alternatively, for a general user element in Abaqus/Standard, user subroutine [UELMAT](#) may be coded to define the contribution of the element to the model. User subroutine [UELMAT](#) is an enhanced version of user subroutine [UEL](#); consequently, all the information provided for user subroutine [UEL](#) is also valid for user subroutine [UELMAT](#). The enhancement allows you to access some of the material models from the Abaqus material library from [UELMAT](#). [UELMAT](#) works only with a subset of procedures for which [UEL](#) is available:

- static;
- direct-integration dynamic;
- frequency extraction;
- steady-state uncouple heat transfer; and
- transient uncouple heat transfer.

User subroutine [UEL](#) will be called if an Abaqus material model is assigned to a user element (see [Assigning an Abaqus Material to the User Element](#) above); otherwise, user subroutine [UEL](#) will be called.

## Accessing Abaqus Materials from User Subroutine [UEL](#)

Abaqus allows you to access some of the material models from the Abaqus material library from user subroutine [UEL](#). The material models are accessed through the utility routines `MATERIAL_LIB_MECH` and `MATERIAL_LIB_HT` ([Accessing Abaqus Thermal Materials](#) and [Accessing Abaqus Materials](#)). Each time user subroutine [UEL](#) is called with the flags set to values that require computation of the right-hand-side vector and the element Jacobian, the material library must be called for each integration point, where the number of integration points is specified in the element definition ([Specifying the Number of Integration Points](#)). The material models that are accessible from user subroutine [UEL](#) are:

- linear elastic model;
- hyperelastic model;
- Ramberg-Osgood model;
- classical metal plasticity models (Mises and Hill);
- extended Drucker-Prager model;
- modified Drucker-Prager/Cap plasticity model;
- porous metal plasticity model;
- elastomeric foam material model; and
- crushable foam plasticity model.

## Defining the Contribution of the Element to the Model in User Subroutine [VUEL](#)

For a general user element in Abaqus/Explicit, user subroutine [VUEL](#) must be coded to define the contribution of the element to the model. Abaqus/Explicit calls this routine each time any information about a user-defined element is needed. At each such call Abaqus/Explicit provides the values of the nodal coordinates and of all solution-dependent nodal variables

(displacements, velocities, accelerations, etc.) at all degrees of freedom associated with the element, as well as values of the solution-dependent state variables associated with the element at the beginning of the current increment. The incremental displacements are those obtained in a previous increment. Abaqus/Explicit also provides the values of all user-defined properties associated with this element and a control flag array indicating what functions the user subroutine must perform. Depending on this set of control flags, the subroutine must define the contribution of the element to the internal or external force/flux vector, form the mass/capacity matrix, update the solution-dependent state variables associated with the element, and so on.

The element's principal contribution to the model is that it provides nodal forces  $F^J$  that depend on the values of the nodal variables  $u^M$ , the rate of nodal variables  $\dot{u}^M$ , and on the solution-dependent state variables  $H^\alpha$  within the element:

$$F^J = F^J \left( u^M, \dot{u}^M, H^\alpha, \text{geometry, attributes, predefined field variables} \right)$$

In addition, the element mass matrix  $M^{NJ}$  can be defined. Optionally, you can also define the external load contribution from the element due to specified distributed loading. In each increment Abaqus/Explicit solves for the accelerations at the end of the increment using

$$\ddot{u}_{(i)}^N = (M^{NJ})^{-1} \left( P_{(i)}^J - F_{(i)}^J \right),$$

where  $P^J$  is the applied load vector. The solution (velocity, displacement) is then integrated in time using the central difference method

$$\dot{u}_{\left(i+\frac{1}{2}\right)}^N = \dot{u}_{\left(i-\frac{1}{2}\right)}^N + \frac{\Delta t_{(i+1)} + \Delta t_{(i)}}{2} \ddot{u}_{(i)}^N,$$

$$u_{(i+1)}^N = u_{(i)}^N + \Delta t_{(i+1)} \dot{u}_{\left(i+\frac{1}{2}\right)}^N.$$

For coupled temperature/displacement elements the temperatures are computed at the beginning of the increment using

$$\dot{\theta}_{(i)}^N = (C^{NJ})^{-1} \left( \mathbf{P}_{(i)}^J - F_{(i)}^J \right),$$

where  $C^{NJ}$  is the lumped capacitance matrix,  $P^J$  is the applied nodal source, and  $F^J$  is the internal flux vector. The temperature is integrated in time using the explicit forward-difference integration rule,

$$\theta_{(i+1)}^N = \theta_{(i)}^N + \Delta t_{(i+1)} \cdot \dot{\theta}_{(i)}^N.$$

More details can be found in [Explicit Dynamic Analysis](#) and [Fully Coupled Thermal-Stress Analysis](#). The signs of the forces defined in  $F^J$  are such that external forces provide positive nodal force values and “internal” forces caused by stresses, damping effects, internal heat fluxes, etc. in the element provide negative nodal force values. Internal forces due to bulk viscosity are dependent on the scaled mass of the element. The necessary information (bulk viscosity constants and mass scaling factors) is passed into the user subroutine for this purpose.

### Requirements for Defining the Mass Matrix

As explained in [Explicit Dynamic Analysis](#), what makes the explicit time integration method efficient is that the mass inversion process is extremely effective. This is due to the fact that most of the nonzero entries in the mass matrix are located on the diagonal positions. The only exception is for rotational degrees of freedom in three-dimensional analyses in which case at each node an anisotropic rotary inertia (symmetric  $3 \times 3$  tensor) can be defined. In these cases some of the nonzero entries in the mass matrix may be off-diagonal; but the inversion process is local and, hence, very effective. The mass matrix defined in user subroutine [VUEL](#) must adhere to these requirements as illustrated in detail in [VUEL](#). If you specify a zero mass matrix or skip the definition of the mass matrix altogether, Abaqus/Explicit issues an error message.

The definition of a realistic mass matrix is not mandatory, but it is strongly recommended. If you choose to not define a realistic mass matrix using the user subroutine, you must provide realistic mass, rotary inertia, heat capacity, etc. at all nodes and at all degrees of freedom associated with the user element. This can be accomplished by various means, such as by defining mass and rotary inertia elements at the nodes or by connecting the user element to other elements for which density, heat capacity, etc. was specified.

Mass is computed only once at the beginning of the analysis. Consequently, the mass of the user element cannot be changed arbitrarily during the analysis. If necessary, mass scaling is applied accordingly to ensure the requested time incrementation.

## Definition of the Stable Time Increment

Since the central difference operator is conditionally stable, the time increments in Abaqus/Explicit must be somewhat smaller than the stable time increment. You must provide an accurate estimate for the stable time increment associated with the user element. This scalar value is highly dependent on the element formulation, and sophisticated coding may be required inside the user subroutine to obtain a reliable estimate. A conservative estimate will reduce the time increment size for the entire analysis and, hence, lead to longer analysis times.

## Defining Loads

You can apply point loads, moments, fluxes, etc. to the nodes of general user-defined elements in the usual way, using concentrated loads and concentrated fluxes ([Concentrated Loads](#) and [Thermal Loads](#)).

You can also define distributed loads and fluxes for general user-defined elements ([Distributed Loads](#) and [Thermal Loads](#)). These loads require a load type key. For user-defined elements, you can define load type keys of the forms  $Un$  and, in Abaqus/Standard,  $UnNU$ , where  $n$  is any positive integer.

If the load type key is of the form  $Un$ , the load magnitude is defined directly and follows the standard Abaqus conventions with respect to its amplitude variation as a function of time. In Abaqus/Standard, if the load key is of the form  $UnNU$ , all of the load definition will be accomplished inside subroutine [UEL](#) and [UELMAT](#). Each time Abaqus/Standard calls subroutine [UEL](#) or [UELMAT](#), it tells the subroutine how many distributed loads/fluxes are currently active. For each active load or flux of type  $Un$  Abaqus/Standard gives the current magnitude and current increment in magnitude of the load. The coding in subroutine [UEL](#) or [UELMAT](#) must distribute the loads into consistent equivalent nodal forces and, if necessary, provide their contribution to the Jacobian matrix—the “load stiffness matrix.”

In Abaqus/Explicit only load keys of the form  $Un$  can be used, and they can be used only for distributed loads (however, thermal fluxes can be defined in the coding in subroutine [VUEL](#)). Each time Abaqus/Explicit calls subroutine [VUEL](#), it tells the subroutine which load number is currently active and the current magnitude of the load. The coding in subroutine [VUEL](#) must distribute the loads into consistent equivalent nodal forces.

## Defining Output

All quantities to be output must be saved as solution-dependent state variables. In Abaqus/Standard the solution-dependent state variables can be printed or written to the results file using output variable identifier SDV ([Abaqus/Standard Output Variable Identifiers](#)).

The components of solution-dependent state variables that belong to a user element are not available in Abaqus/CAE. You can write output to separate files in a table format that can be accessed in Abaqus/CAE to produce history output.

Currently element output to the output database is not supported for user-defined elements.

## Defining Wave Kinematic Data

A utility routine GETWAVE is provided in user subroutine [UEL](#) to access the wave kinematic data defined for an Abaqus/Aqua analysis ([Abaqus/Aqua Analysis](#)). This utility is discussed in [Obtaining Wave Kinematic Data in an Abaqus/Aqua Analysis](#), where the arguments to GETWAVE and the syntax for its use are defined.

## Use in Contact

Only node-based surfaces ([Node-Based Surface Definition](#)) can be created on user-defined elements. Hence, these elements can be used to define only secondary surfaces in a contact analysis. In Abaqus/Explicit the user elements will not be included in the general contact algorithm automatically. Node-based surfaces can be defined using these nodes and then included in the general contact definition.

## Import of User Elements

User elements cannot be imported from an Abaqus/Standard analysis into an Abaqus/Explicit analysis or vice versa. Equivalent user elements can be defined in both products to overcome this limitation. However, the state variables associated with these elements will not be communicated.



## UEL

**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.

The following topics are discussed:

- [Wave kinematic data](#)
- [User subroutine interface](#)
- [Variables to be defined](#)
- [Variables that can be updated](#)
- [Variables passed in for information](#)
- [UEL conventions](#)
- [Usage with general nonlinear procedures](#)
- [Usage with linear perturbation procedures](#)
- [Nondiagonal damping in linear perturbation procedures](#)
- [Example: Structural and heat transfer user element](#)

**Products:** Abaqus/Standard

## 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,NPREDEF,  
3  LFLAGS,MLVARX,DDL MAG,MDLOAD,PNEWDT,JPROPS,NJPROP,PERIOD)  
  
C  
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,*),DDL MAG(MDLOAD,*),  
4  PREDEF(2,NPREDEF,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 `K1`th 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 `K1`th 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  $PNEWDT \times DTIME$ , 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  $PNEWDT \times DTIME$ , 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 `K1`th coordinate of the `K2`th 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:

<code>U(K1)</code>	Total values of the variables. If this is a linear perturbation step, it is the value in the base state.
<code>DU(K1,KRHS)</code>	Incremental values of the variables for the current increment for right-hand-side <code>KRHS</code> . If this is an eigenvalue extraction step, this is the eigenvector magnitude for eigenvector <code>KRHS</code> . For steady-state dynamics, <code>KRHS = 1</code> denotes real components of perturbation displacement and <code>KRHS = 2</code> denotes imaginary components of perturbation displacement.
<code>V(K1)</code>	Time rate of change of the variables (velocities, rates of rotation). Defined for implicit dynamics only ( <code>LFLAGS(1) = 11</code> or <code>12</code> ).
<code>A(K1)</code>	Accelerations of the variables. Defined for implicit dynamics only ( <code>LFLAGS(1) = 11</code> or <code>12</code> ).

#### JDLTYP

An array containing the integers used to define distributed load types for the element. Loads of type `Un` are identified by the integer value `n` in `JDLTYP`; loads of type `UnNU` 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 `Un`. For distributed loads of type `UnNU`, 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 `Un` applied in the base state. Base state loading of type `UnNU` must be dealt with inside [UEL](#). `ADLMAG(K1,2)`, `ADLMAG(K1,3)`, etc. are currently not used.

#### DDL MAG

For general nonlinear steps `DDL MAG` contains the increments in the magnitudes of the distributed loads that are currently active on this element for distributed loads of type `Un`. `DDL MAG(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 `UnNU`, the load magnitude is defined in [UEL](#); therefore, the corresponding entries in `DDL MAG` are zero. For linear perturbation steps `DDL MAG(K1,K2)` contains the perturbation in the magnitudes of the distributed loads that are currently active on this element for distributed loads of type `Un`. `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 `UnNU` 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,  $\kappa_1$ , 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,  $\kappa_2$ , 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,  $\kappa_3$ , indicates the local node number on the element.

PREDEF( $\kappa_1, 1, \kappa_3$ ) Temperature.

PREDEF( $\kappa_1, 2, \kappa_3$ ) First predefined field variable.

PREDEF( $\kappa_1, 3, \kappa_3$ ) Second predefined field variable.

Etc. Any other predefined field variable.

PREDEF( $\kappa_1, \kappa_2, \kappa_3$ ) Total or incremental value of the  $\kappa_2$ th predefined field variable at the  $\kappa_3$ th node of the element.

PREDEF( $1, \kappa_2, \kappa_3$ ) Values of the variables at the end of the current increment.

PREDEF( $2, \kappa_2, \kappa_3$ ) 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)  $\alpha$

[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 [AMATRIX](#).

[LFLAGS](#)(3)=2 Define the current stiffness matrix ([AMATRIX](#) =  $K^{NM} = -\partial F^N / \partial u^M$  or  $-\partial G^N / \partial u^M$ ) only.

[LFLAGS](#)(3)=3 Define the current damping matrix ([AMATRIX](#) =  $C^{NM} = -\partial F^N / \partial \dot{u}^M$  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 ([AMATRIX](#) =  $M^{NM} = -\partial F^N / \partial \ddot{u}^M$ ) 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 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  $Un$  ([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  $u^N$  (and, in direct-integration dynamic steps,  $\dot{u}^N$  and  $\ddot{u}^N$ ) enter user subroutine [UEL](#) 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)

- $F^N = F^N(u^M, H^\alpha, p^\beta, t)$ , 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)

- $F^N = F^N(u^M, H^\alpha, p^\beta)$ , 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\(2\)](#) =  $\Delta \lambda (\partial F^N / \partial \lambda)$  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  $M^{NM} = M^{NM}(u^M, \dot{u}^M, H^\alpha, p^\beta, t, \dots)$  and  $G^N = G^N(u^M, \dot{u}^M, H^\alpha, p^\beta, t, \dots)$ ; 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 [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  $\max |F^N_{1/2}| < tolerance$  (where *tolerance* is specified in the dynamic step definition). The half-increment residual is defined as

$$F_{1/2}^N = -M^{NM} \ddot{u}_{t+\Delta t/2} + (1 + \alpha) G_{t+\Delta t/2}^N - \frac{\alpha}{2} (G_t^N + G_{t^-}^N),$$

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  $\text{RHS} = F_{1/2}^N$ . To evaluate  $M^{NM}$  and  $G_{t+\Delta t/2}^N$ , you must calculate  $H_{t+\Delta t/2}^\alpha$ . 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,  $\dot{u}_{t+\Delta t}^M = \Delta u^M / \Delta t$ , where  $\Delta u^M = u_{t+\Delta t}^M - u_t^M$ .
- 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,  $\dot{u}_{t+\Delta t}^M = \Delta u^M / \Delta t$ , where  $\Delta u^M = u_{t+\Delta t}^M - u_t^M$ .
- 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  $\text{AMATRX} = K^{NM} + (1/\Delta t) C^{NM}$ , where  $C^{NM}$  is the heat capacity matrix and  $\text{RHS} = F^N$ , 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  $\text{AMATRX} = K^{NM} + (1/\Delta t) C^{NM}$ , where  $C^{NM}$  is the heat capacity matrix and  $\text{RHS} = F^N$ , 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  $\text{AMATRX} = K^{NM} + (1/\Delta t) C^{NM}$ , where  $C^{NM}$  is the heat capacity matrix and  $\text{RHS} = F^N$ ; 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  $\tilde{u}^M$ ,  $\tilde{H}^\alpha$ , 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  $\text{AMATRX} = K^{NM}$  and  $\text{RHS} = \tilde{P}^N$ .
- LFLAGS(3)=100: You must compute perturbations of the internal variables,  $\tilde{H}^\alpha$ , and define  $\text{RHS} = \tilde{P}^N - K^{NM} \tilde{u}^M$  for output purposes.

### Eigenfrequency extraction analysis (LFLAGS(1)=41)

- $F^N = -M^{NM} \ddot{u} + G^N(u^M + \tilde{u}^M, \dots) = -M^{NM} \ddot{u} + (\partial G^N / \partial u^M) \tilde{u}^M$ .
- 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  $\text{AMATRX} = K^{NM}$ .
- LFLAGS(3)=4: Define  $\text{AMATRX} = M^{NM}$ .

### Direct steady-state analysis (LFLAGS(1)=95)

- LFLAGS(3)=2: Define stiffness matrix  $\text{AMATRX} = K^{NM}$ .
- LFLAGS(3)=4: Define mass matrix  $\text{AMATRX} = M^{NM}$ .
- LFLAGS(3)=3: Define damping matrix  $\text{AMATR} = C^{NM}$ .
- LFLAGS(7)=1: Define viscous damping matrix  $\text{AMATRX} = C_{viscous}^{NM}$ .
- LFLAGS(7)=2: Define structural damping matrix  $\text{AMATRX} = C_{structural}^{NM}$ .
- 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  $\text{AMATRX} = K^{NM}$ .
- LFLAGS(3)=4: Define mass matrix  $\text{AMATRX} = M^{NM}$ .
- LFLAGS(3)=3: Define damping matrix  $\text{AMATR} = C^{NM}$ .
- LFLAGS(7)=1: Define viscous damping matrix  $\text{AMATRX} = C_{viscous}^{NM}$ .
- LFLAGS(7)=2: Define structural damping matrix  $\text{AMATRX} = C_{structural}^{NM}$ .
- 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  $\text{AMATRX} = C_{viscous}^{NM}$ . To provide structural damping matrix you must also set LFLAGS(7)=2 and define  $\text{AMATRX} = C_{structural}^{NM}$ .

## 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=UTRUS
101, 5
*UEL PROPERTY, ELSET=UTRUS
```



```
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 `Un` and `UnNU` 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,DDL MAG,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      DDL MAG(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
C      AREA = PROPS(1)
C      E = PROPS(2)
C      ANU = PROPS(3)
C      RHO = PROPS(4)
C
C      ALEN = ABS(COORDS(1,2)-COORDS(1,1))
C      AK = AREA*E/ALEN
C      AM = HALF*AREA*RHO*ALEN
C
C      DO K1 = 1, NDOFEL
C        SRESID(K1) = ZERO
C        DO KRHS = 1, NRHS
C          RHS(K1,KRHS) = ZERO
C        END DO
C        DO K2 = 1, NDOFEL
C          AMATRX(K2,K1) = ZERO
C        END DO
C      END DO
C
C      IF (LFLAGS(3).EQ.1) THEN
C        Static or nonlinear dynamic analysis
C        Normal incrementation
C        IF (LFLAGS(1).EQ.1 .OR. LFLAGS(1).EQ.2) THEN
C          *STATIC
C          AMATRX(1,1) = AK
C          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*DDL MAG(KDLOAD,1))*DU(4,1)
      IF (NRHS.EQ.2) THEN
        Riks
        RHS(4,2) = RHS(4,2)+DDL MAG(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

```

```

  *DYNAMIC
  ALPHA = PARAMS(1)
  BETA = PARAMS(2)
  GAMMA = PARAMS(3)

```

```

  DADU = ONE/(BETA*DTIME**2)
  DVDU = GAMMA/(BETA*DTIME)

```

```

  dynt6 = GAMMA/(BETA*DTIME)
  dynt4 = ONE + ALPHA

```

```

  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

```

```

  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

```

```

  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)

```

```

  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

```

```

  ENERGY(2) = HALF*AK*(U(4)-U(1))*(U(4)-U(1))
END IF

```

```

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

```



## \*USER ELEMENT

This option is used to introduce a linear or a general user-defined element. It must precede any reference to this user element on an [\\*ELEMENT](#) option.

This page discusses:

- [Introducing a linear user-defined element \(Abaqus/Standard only\)](#)
- [Introducing a general user-defined element](#)

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

**Type:** Model data

**Level:** Part, Part instance, Model

**Abaqus/CAE:** Actuator/sensor interactions can be defined in the [Interaction module](#).

## Introducing a linear user-defined element (Abaqus/Standard only)

### Required parameters:

#### TYPE

Set this parameter equal to the element type key used to identify this element on the [\\*ELEMENT](#) option. The format of this type key must be  $Un$  in Abaqus/Standard, where  $n$  is a positive integer less than 10000. To use this element type, set  $TYPE=Un$  on the [\\*ELEMENT](#) option.

### Optional parameters:

#### FILE

Set this parameter equal to the name of the results file (with no extension) from which the data are to be read. See [Input Syntax Rules](#) for the syntax of such file names.

This parameter can be used only if the user-defined element type is linear and its stiffness and/or mass matrices are to be read from the Abaqus/Standard results file of a previous analysis (in which they were written by using the [\\*ELEMENT MATRIX OUTPUT](#) or [\\*SUBSTRUCTURE MATRIX OUTPUT](#) options). When this parameter is used, all values are taken from the results file. For example, if the stiffness or mass being read from the results file is not symmetric, the UNSYMM parameter will be invoked automatically.

If this parameter is omitted, the data will be read from a standard input file.

## INTEGRATION

This parameter applies only to Abaqus/Standard analyses.

Set this parameter equal to the number of integration points to be used in Gauss integration. This parameter must be used in conjunction with the TENSOR parameter.

## TENSOR

This parameter applies only to Abaqus/Standard analyses.

Include this parameter to specify the element type. This parameter must be used in conjunction with the INTEGRATION parameter.

Set TENSOR=THREED to specify that it is a three-dimensional element in a stress/displacement or heat transfer analysis.

Set TENSOR=TWOD to specify that it is a two-dimensional element in a heat transfer analysis.

Set TENSOR=PSTRAIN to specify that it is a plane strain element in a stress/displacement analysis.

Set TENSOR=PSTRESS to specify that it is a plane stress element in a stress/displacement analysis.

## Required parameters if the FILE parameter is included:

### OLD ELEMENT

Set this parameter equal to the element number that was assigned to the element whose matrices are being read. This parameter can also be set to a substructure identifier to read a substructure matrix from an Abaqus/Standard results file.

### STEP

Set this parameter equal to the step number in which the element matrix was written. This parameter is not required if using a substructure whose matrix was output during its generation.

## INCREMENT

Set this parameter equal to the increment number in which the element matrix was written. This parameter is not required if using a substructure whose matrix was output during its generation.

### Required parameters if the FILE parameter is omitted:

#### LINEAR

Include this parameter to indicate that the behavior of the element type is linear and is defined by a stiffness matrix and/or a mass matrix. The [\\*MATRIX](#) option is required to define the element's behavior.

#### NODES

Set this parameter equal to the number of nodes associated with an element of this type.

### Optional parameters if the FILE parameter is omitted:

#### COORDINATES

Abaqus/Standard assigns space to store the coordinate values at each node in user subroutine [UEL](#). The default number of coordinate values is equal to the largest active degree of freedom of the user element with a maximum of 3. Use the COORDINATES parameter to increase the number of coordinate values.

#### UNSYMM

Include this parameter if the element matrices are not symmetric. This parameter will cause Abaqus/Standard to use its unsymmetric equation solution capability.

The presence or absence of this parameter determines the form in which the matrices must be provided for reading.

### Data lines if the FILE parameter is omitted:

#### First line:

1. Enter the list of active degrees of freedom at the first node of the element (as determined by the connectivity list). The rule in [Conventions](#) regarding which degrees of freedom can be used for displacement, rotation, temperature, etc. must be conformed to.

#### Second line if the active degrees of freedom are different at subsequent nodes:

1. Enter the position in the connectivity list (node position on the element) where the new list of active degrees of freedom first applies.

2. Enter the new list of active degrees of freedom.

Repeat the second data line as often as necessary.

## Introducing a general user-defined element

### Required parameters:

#### TYPE

Set this parameter equal to the element type key used to identify this element on the [\\*ELEMENT](#) option. The format of this type key must be  $Un$  in Abaqus/Standard and  $VUn$  in Abaqus/Explicit, where  $n$  is a positive integer less than 10000. To use this element type, set TYPE= $Un$  (or  $VUn$ ) on the [\\*ELEMENT](#) option.

#### NODES

Set this parameter equal to the number of nodes associated with an element of this type.

### Optional parameters:

#### COORDINATES

Set this parameter equal to the maximum number of coordinates needed in user subroutine [UEL](#) in Abaqus/Standard and user subroutine [VUEL](#) in Abaqus/Explicit at any node point of the element. Abaqus assigns space to store the coordinate values at all the nodes associated with elements of this type. The default is COORDINATES=1.

Abaqus will change the value of COORDINATES to be the maximum of the user-specified value of the COORDINATES parameter or the value of the largest active degree of freedom of the user element that is less than or equal to 3. For example, if COORDINATES=1 and the active degrees of freedom of the user element are 2, 3, and 6, the value of COORDINATES will be changed to 3. If COORDINATES=2 and the active degrees of freedom of the user element are 11 and 12, the value of COORDINATES will remain as 2.

#### I PROPERTIES

Set this parameter equal to the number of integer property values needed as data in user subroutine [UEL](#) (or [VUEL](#)) to define such an element. The default is I PROPERTIES=0.

#### PROPERTIES

Set this parameter equal to the number of real (floating point) property values needed as data in user subroutine [UEL](#) (or [VUEL](#)) to define such an element. The default is PROPERTIES=0.

## UNSYMM

This parameter applies only to Abaqus/Standard analyses.

Include this parameter if the element matrices are not symmetric. This parameter will cause Abaqus/Standard to use its unsymmetric equation solution capability.

## VARIABLES

Set this parameter equal to the number of solution-dependent state variables that must be stored within the element. Its value must be greater than 0. The default is VARIABLES=1.

### Data lines to define a general user-defined element:

#### First line:

1. Enter the list of active degrees of freedom at the first node of the element (as determined by the connectivity list). The rule in [Conventions](#) regarding which degrees of freedom can be used for displacement, rotation, temperature, etc. must be conformed to.

#### Second line if the active degrees of freedom are different at subsequent nodes:

1. Enter the position in the connectivity list (node position on the element) where the new list of active degrees of freedom first applies.
2. Enter the new list of active degrees of freedom.

Repeat the second data line as often as necessary.





## \*UEL PROPERTY

This option is required to define the properties of a user element, where the element set must be provided. User element properties provided on the data line are relevant only for nonlinear user elements. The damping parameters provided via parameters on this option are relevant for all procedure types except for nonlinear user elements given by the user subroutine in direct-integration transient dynamic analyses.

This page discusses:

- [Required parameters](#)
- [Optional parameters](#)
- [Optional parameters \(relevant for mode based procedures, direct steady-state dynamic analyses, and direct-integration transient dynamic analysis with linear user elements\)](#)
- [There are no data lines required to define the properties of linear user elements](#)
- [Data lines to define the properties of nonlinear user elements if the PROPERTIES and/or I PROPERTIES parameters are used on the \\*USER ELEMENT option with a value of one or more](#)

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

**Type:** Model data

**Level:** Part, Part instance, Model

**Abaqus/CAE:** Actuator/sensor interaction properties can be defined in the [Interaction module](#).

### Required parameters:

#### ELSET

Set this parameter equal to the name of the element set containing the user elements for which these property values are being defined.

### Optional parameters:

#### MATERIAL

This parameter applies only to Abaqus/Standard analyses.

Set this parameter equal to the name of the material to be used with these elements.

## ORIENTATION

This parameter applies only to Abaqus/Standard analyses.

Set this parameter equal to the name of an orientation definition ([Orientations](#)) to be used to define a local coordinate system for material calculations in the elements in this set.

**Optional parameters (relevant for mode based procedures, direct steady-state dynamic analyses, and direct-integration transient dynamic analysis with linear user elements):**

### ALPHA

Set this parameter equal to the Rayleigh mass damping factor,  $\alpha$ .

### BETA

Set this parameter equal to the Rayleigh stiffness damping factor,  $\beta$ .

## STRUCTURAL

Set this parameter equal to the structural stiffness proportional damping factor,  $s$ .

**There are no data lines required to define the properties of linear user elements.**

**Data lines to define the properties of nonlinear user elements if the PROPERTIES and/or I PROPERTIES parameters are used on the \*USER ELEMENT option with a value of one or more:**

### First line:

1. Enter the values of the element properties. Enter all floating point values first, followed immediately by the integer values.

Repeat this data line as often as necessary. Eight values per line are used for both real and integer values.



## About User Subroutines and Utilities

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](#).

This page discusses:

- [Including User Subroutines in a Model](#)
- [Managing External Databases in Abaqus and Exchanging Information with Other Software](#)
- [Writing a User Subroutine](#)
- [Models Defined in Terms of an Assembly of Part Instances](#)
- [Solution-Dependent State Variables](#)
- [Element Solution-Dependent Variables](#)
- [Alphanumeric Data](#)
- [Precision in Abaqus/Explicit](#)
- [Vectorization in Abaqus/Explicit](#)
- [Parallelization](#)
- [User Subroutine Calls](#)
- [Utility Routines](#)

## 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:

**abaqus job=***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](#)

- [UEL](#)
- [UEXPAN](#)
- [UGENS](#)
- [UHARD](#)
- [UHYPER](#)
- [UINTER](#)
- [UMAT](#)
- [UMATHT](#)
- [UMULLINS](#)
- [USDFLD](#)
- [UTRS](#)
- [VFABRIC](#)
- [VFRIC](#)
- [VFRICTION](#)
- [VHETVAL](#)
- [VUANISOHYPER\\_INV](#)
- [VUANISOHYPER\\_STRAIN](#)
- [VUFLUIDEXCH](#)
- [VUHARD](#)
- [VUINTER](#)
- [VUINTERACTION](#)
- [VUMAT](#)
- [VUMATHT](#)
- [VUMULLINS](#)
- [VUSDFLD](#)
- [VUTRS](#)
- [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](#), [VFRICITION](#), 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 [VFRICITION](#):

[\\*FRICTION](#), *USER=FRICTION, DEPVAR=number of variables*

For subroutine [VUFLUIDEXCH](#):

[\\*FLUID EXCHANGE PROPERTY](#), TYPE=USER, DEPVAR=*number of variab*

For subroutine [VUINTERACTION](#):

[\\*SURFACE INTERACTION](#), USER=INTERACTION, DEPVAR=*number of var*

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 SDVn 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](#), [VFRICITION](#), [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

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.



## Ensuring Thread Safety

A number of primitives are provided to help code user subroutines for thread-parallel execution.

- [Mutexinit, MutexLock, and Mutexunlock](#)

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

### 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

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.

This page discusses:

- [SMALocalIntArrayCreate, SMALocalFloatArrayCreate](#)
- [SMALocalIntArrayAccess, SMALocalFloatArrayAccess](#)
- [SMALocalIntArraySize, SMALocalFloatArraySize](#)
- [SMALocalFloatArrayDelete, SMALocalFloatArrayDelete](#)
- [SMAIntArrayCreate, SMAFloatArrayCreate](#)
- [SMAIntArrayAccess, SMAFloatArrayAccess](#)
- [SMAIntArraySize, SMAFloatArraySize](#)
- [SMAFloatArrayDelete, SMAFloatArrayDelete](#)
- [Allocatable Global Arrays of Variable Precision](#)
- [Allocatable Global Arrays of User-Defined Types](#)

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

### **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
ptr = SMAALocalIntArrayCreate(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.

## SMALocalFloatArrayDelete, 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
ptrb = 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
ptr = 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(ptr,a)
```

C Locate Array(1) and associate a native array pointer with it

```
ptr = 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.

## **SMAFloatArrayDelete, 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.