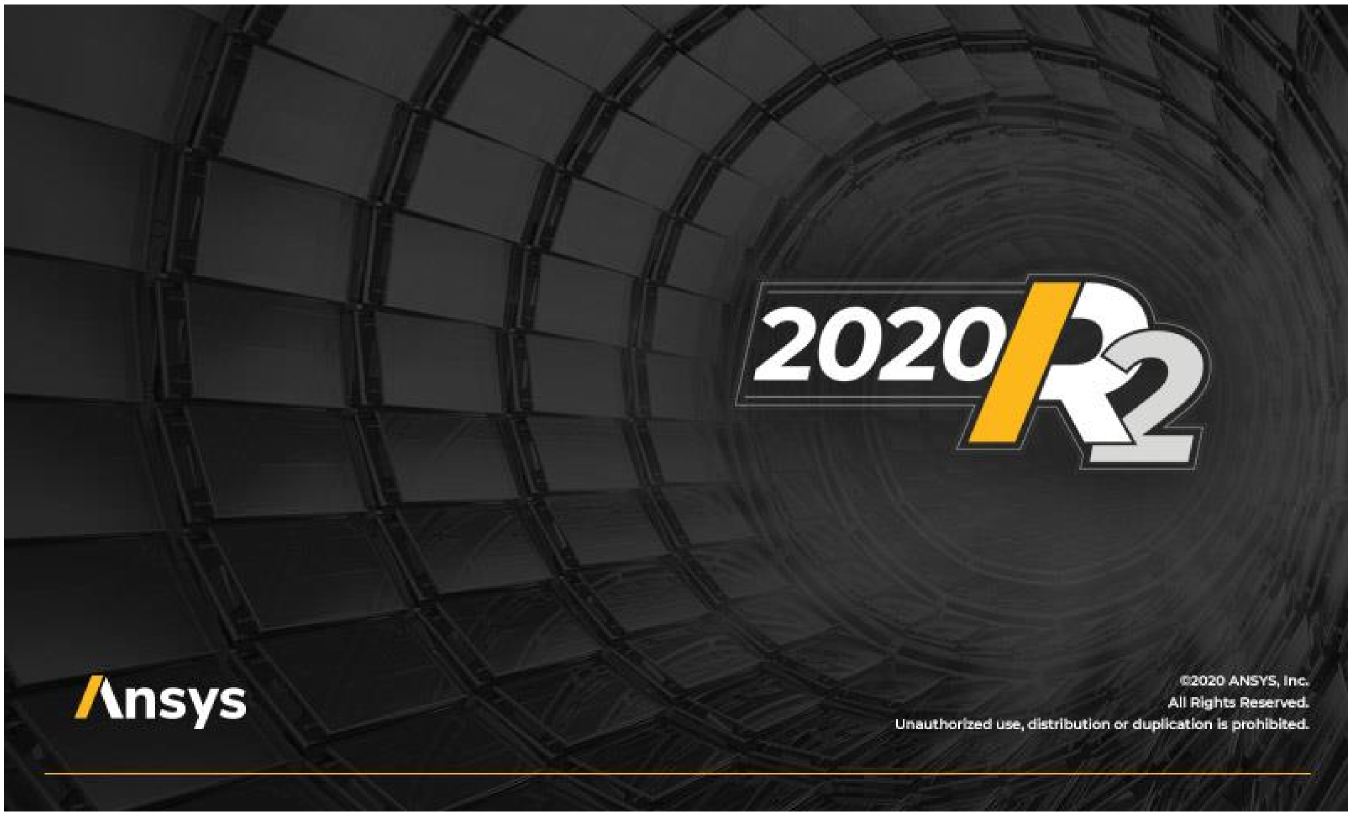

2020 R2
Ansys
©2020 ANSYS, Inc.
All Rights Reserved.
Unauthorized use, distribution or duplication is prohibited.

ANSYS ACT Customization Guide for Mechanical

ANSYS, Inc.
Southpointe
2600 ANSYS Drive
Canonsburg, PA 15317
ansysinfo@ansys.com
http://www.ansys.com
(T) 724-746-3304
(F) 724-514-9494

Release 2020 R2
July 2020

ANSYS, Inc. and ANSYS Europe, Ltd. are UL registered ISO 9001: 2015 companies.

Table of Contents

Introduction

This guide assumes that you are familiar with the general ACT usage information in the *ACT Developer's Guide*. This first section supplies ACT usage information specific to Mechanical:

Mechanical Scripting View

Debug Mode

Subsequent content is organized into primary sections as follows:

- Mechanical Feature Creation (p. 3): Provides examples of how to integrate ACT parameters into Mechanical models, create preprocessing and postprocessing capabilities, and connect to third-party solvers.

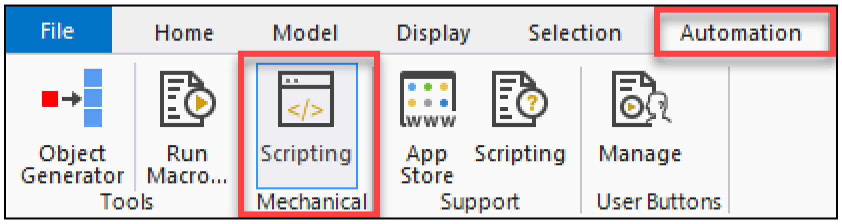
- Mechanical Wizards (p. 57): Describes Mechanical product wizards, using a supplied extension as an example.

Note:

For information on Mechanical ACT API changes and known issues and limitations that might affect your existing ACT extensions, see Mechanical API Migration Notes and Mechanical API Known Issues and Limitations in the *Scripting in Mechanical Guide*.

Mechanical Scripting View

In Mechanical, you can open and close the **Scripting** view from the ribbon's **Automation** tab. In the **Mechanical** group, clicking **Scripting** switches between opening and closing the view.



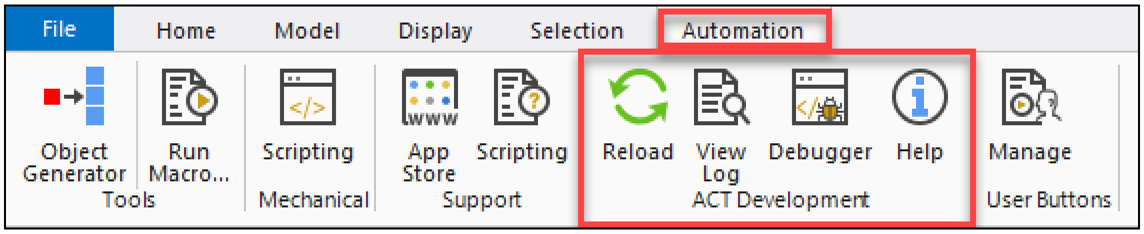
For information on using the Mechanical **Scripting** view, see Scripting Introduction in the *Scripting in Mechanical Guide*.

Note:

You can revert to the **ACT Console** by changing the scripting view preference under **File** > **Options** > **Mechanical** > **UI Options** > **New Scripting UI**. Mechanical must be restarted to see the scripting view change.

Debug Mode

The Mechanical ribbon's **Automation** tab displays the **ACT Development** group whenever debug mode is enabled. If Mechanical is already started when **Debug Mode** is enabled, you must exit Mechanical and then restart it. For more information, see Debug Mode in the *ANSYS ACT Developer's Guide*.



- Clicking the first button reloads ACT extensions.
- Clicking the second button switches between opening and closing the Extensions Log File.
- Clicking the third button opens the ACT Debugger.
- Clicking the fourth button opens the help panel for the ACT Start Page.

If you have ACT extensions loaded, in the lower left corner of the Mechanical **Scripting** view, a tab displays for each loaded extension. Tabs are only shown when debug mode is enabled for ACT extensions. For more information, see Scope Selection for ACT Extensions in the *Scripting in Mechanical Guide*.

Mechanical Feature Creation

In addition to supporting the common feature creation capabilities described in the *ANSYS ACT Developer's Guide*, Mechanical supports product-specific feature creation capabilities:

Note:

- All extensions referenced in this section are supplied. Mechanical requires BMP files for the images to display as toolbar buttons, while all other supported ANSYS products require PNG files.
- The callbacks **`<onstarteval>`** and **`<getvalue>`** have been replaced by the single callback **`<evaluate>`**. The callback **`<evaluate>`** simplifies the implementation of an ACT result, which now requires only a single IronPython function. While the callbacks **`<onstarteval>`** and **`<getvalue>`** are still supported, using the callback **`<evaluate>`** is recommended.

UI Customization in Mechanical

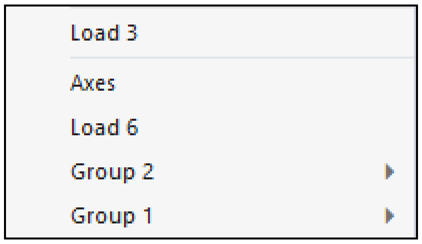
The attributes **`begingroup`** and **`contextgroup`** are available on **`<load>`**, **`<result>`**, and **`<object>`** tags. You can use these attributes to add separators and groups in your context menus.

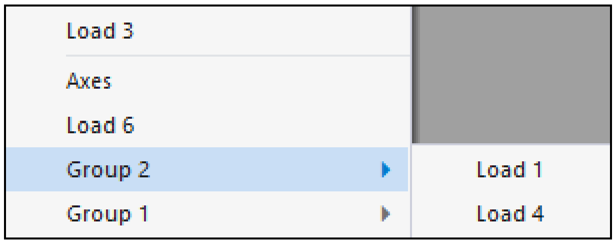
- Setting the attribute **`begingroup`** to true adds a separator before the entity in the context menu. The default value for **`begingroup`** is false.
- Specifying the name of a menu group for the attribute **`contextgroup`** causes the entity to appear in this group. When no group is specified, the entity is put at the root of the context menu.

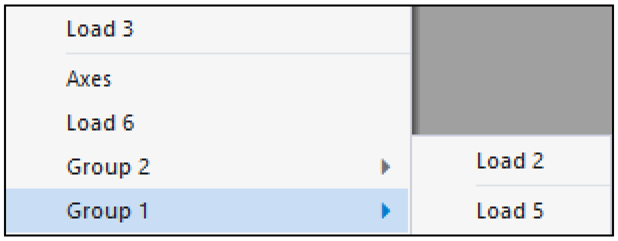
The following code demonstrate how to use these attributes:

```
<load name="Load1" version="1" caption="Load 1" icon="tload" issupport="false" isload="true" contextgroup="Group 2
    <property name="Geometry" control="scoping" />
</load>
<load name="Load2" version="1" caption="Load 2" icon="tload"  issupport="false" isload="true" color="#0000FF" cont
    <property name="Geometry" control="scoping" />
</load>
<load name="Load3" version="1" caption="Load 3" icon="tload" issupport="false" isload="true" color="#0000FF" begin
    <property name="Geometry" control="scoping" />
</load>
```

```
<load name="Load4" version="1" caption="Load 4" icon="tload" issupport="false" isload="true" contextgroup="Group 2
    <property name="Geometry" control="scoping" />
</load>
<load name="Load5" version="1" caption="Load 5" icon="tload" issupport="false" isload="true" color="#0000FF" begin
    <property name="Geometry" control="scoping" />
</load>
<load name="Load6" version="1" caption="Load 6" icon="tload" issupport="false" isload="true" color="#0000FF" conte
    <property name="Geometry" control="scoping" />
</load>
<load name="Load7" version="1" caption="Load 7" icon="tload" localize="true" isload="true" color="#0000FF">
    <property name="Geometry" control="scoping" />
</load>
```







Note:

All context groups defined for a given extension are at the bottom of the context menu section for the given extension. The context group order is determined by the order in which they are defined in the extension's XML file. For example, in the preceding images, Group 2 comes before Group 1 because Group 2 is defined before Group 1.

ACT-Based Property Parametrization in Mechanical

In Mechanical, it is possible to define ACT properties as either input parameters or output parameters. This section addresses how to implement a parameter on an ACT-based object in Mechanical. Specifically, it describes how to parameterize loads, results, and user objects. ACT objects in any of these categories behave and interact with parameters in exactly the same way.

The following topics provide examples and methods for integrating ACT-based parameters into your Mechanical model:

Defining ACT-Based Properties as Input Parameters in Mechanical

To define an ACT-based property as a parameter, add the attribute **`isparameter`** to the XML file and set it to true.

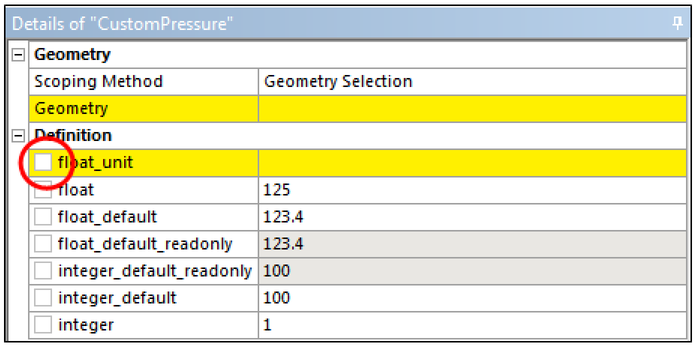
For example, the following code creates a property named **`float_unit`**. The property is a pressure and its value is a float.

```
<property name="float_unit" caption="float_unit" control="float" unit="Pressure" isparameter="true"/>
```

Note:

Other attributes such as **`default`** and **`isload`** can also be added. When adding a default value for a quantity, you must define both the value and the unit, placing the unit in brackets like this: **`default="12 [C]"`**

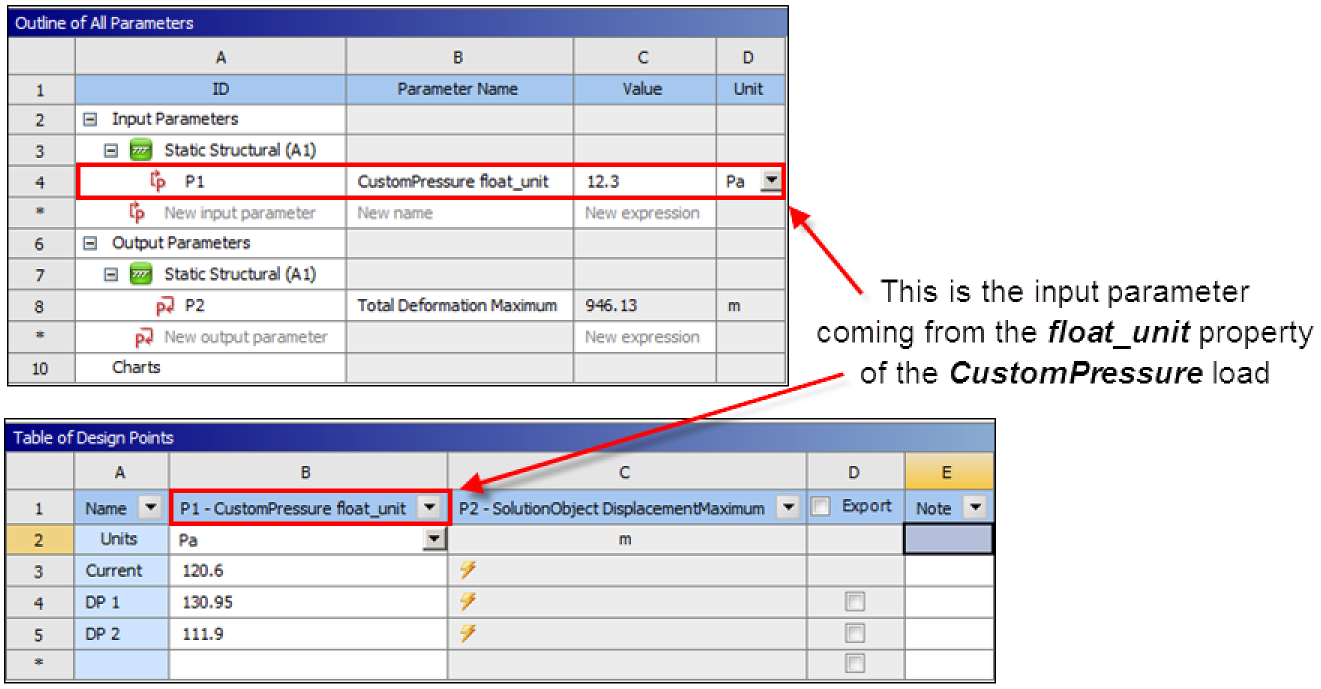
The attribute **`isparameter`** adds a check box to the **Details** view for the property, making it possible for this property to be selected for parameterization.



Details of "CustomPressure"

Geometry	
Scoping Method	Geometry Selection
Geometry	
Definition	
☐ float_unit	
☐ float	125
☐ float_default	123.4
☐ float_default_readonly	123.4
☐ integer_default_readonly	100
☐ integer_default	100
☐ integer	1

Once the property is selected for parametrization, it automatically displays in both the **Outline** and **Table** views for the **Parameter Set** bar.



Outline of All Parameters

	A	B	C	D
1	ID	Parameter Name	Value	Unit
2	⊟ Input Parameters			
3	⊟ Static Structural (A1)			
4	P1	CustomPressure float_unit	12.3	Pa
*	New input parameter	New name	New expression	
6	⊟ Output Parameters			
7	⊟ Static Structural (A1)			
8	P2	Total Deformation Maximum	946.13	m
*	New output parameter		New expression	
10	Charts			

This is the input parameter coming from the ***float_unit*** property of the ***CustomPressure*** load

Table of Design Points

	A	B	C	D	E
1	Name	P1 - CustomPressure float_unit	P2 - SolutionObject DisplacementMaximum	Export	Note
2	Units	Pa	m		
3	Current	120.6	⚡		
4	DP 1	130.95	⚡	☐	
5	DP 2	111.9	⚡	☐	
*				☐	

The following code is extracted from the XML file for this example. You can see how the definition of the property **`float_unit`** is incorporated into the file.

```
<load name="CustomPressure" version="1" caption="CustomPressure" icon="tload" support="false"
isload="true" color="#0000FF">
 <callbacks>
  <getsolvecommands order="1">writeNodeId</getsolvecommands>
 </callbacks>
 <property name="Geometry" caption="Geometry" control="scoping">
  <attributes selection_filter="face"/>
 </property>
 <property name="float_unit" caption="float_unit" control="float" unit="Pressure" isparameter="true"/>
</load>
```

Defining ACT-Based Properties as Output Parameters in Mechanical

It is also possible to define an ACT-based property as an output parameter.

The process for making a property available as a parameter is the same as for any input parameter. You add the attribute **`isparameter`** and set it to true. A check box allowing parameterization of the property then becomes available in the **Details** view for the property. To specify that the property is to be an output parameter, you must also add the attribute **`readonly`** to the XML file and set it to true.

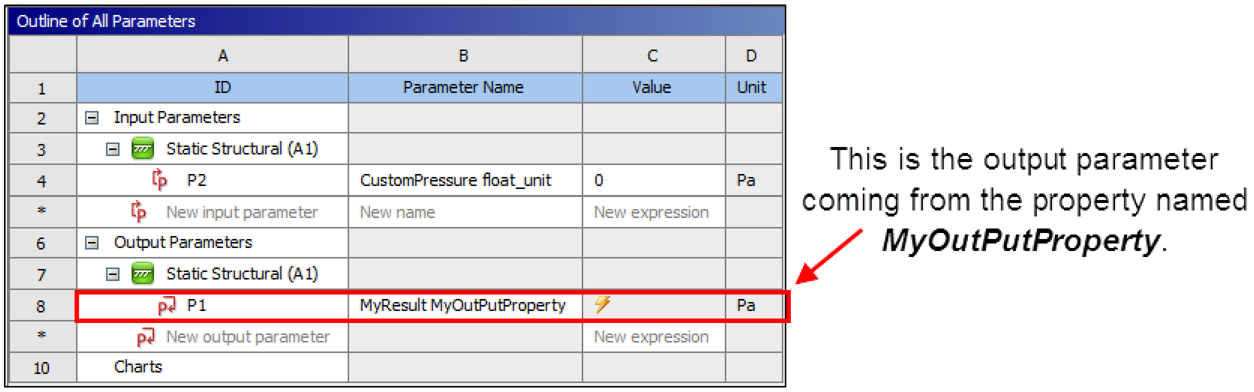
For example, the following code segment creates a property named **`MyOutPutProperty`**. As in the previous example, the property is a pressure and its value is a float. However, because it has the attribute **`readonly`** set to true, this property can be parametrized only as an output parameter.

```
<property name="MyOutPutProperty" caption="MyOutPutProperty" control="float" unit="Pressure"
readonly="true" isparameter="true"/>
```

Note:

Other attributes such as **`default`** and **`isload`** can be added as well. When adding a default value for a quantity, you must define both the value and the unit, placing the unit in brackets like this: **`default="12 [C]"`**.

Again, in Mechanical, the check box provided by the ACT attribute to parameterize the property must be selected for the corresponding output to be automatically generated in the **Outline** and **Table** views for the **Parameter Set** bar.



Outline of All Parameters

	A	B	C	D
1	ID	Parameter Name	Value	Unit
2	⊟ Input Parameters			
3	⊟ Static Structural (A1)			
4	P2	CustomPressure float_unit	0	Pa
*	New input parameter	New name	New expression	
6	⊟ Output Parameters			
7	⊟ Static Structural (A1)			
8	P1	MyResult MyOutPutProperty	⚡	Pa
*	New output parameter		New expression	
10	Charts			

This is the output parameter coming from the property named *MyOutPutProperty*.

In addition, the minimum and maximum values of an ACT result object are available to become output parameters by default. This capability is not managed by the ACT extension but takes advantage of the Mechanical environment.

Defining Parameters under Results in a Third-Party Solver for Mechanical

The process for defining parameters under **Results** in Mechanical is identical to the process for parametrizing properties (p. 4) in Mechanical. Results parameters can be either inputs or outputs, depending on whether the attribute **`readonly`** is set to true or false.

Preprocessing Capabilities in Mechanical

You can use ACT to create custom boundary conditions or loads:

- Adding a Custom Load
- Defining Objects and Loads that Do Not Affect the Solution State
- Creating Objects and Loads that Have Body Views
- Coupling Two Sets of Nodes

Adding a Custom Load

While toolbar examples in the *ANSYS ACT Developer's Guide* show how to extend the interface of an ANSYS product, they do not show how to perform meaningful operations. Here, the supplied extension `DemoLoad` shows how to add a button to Mechanical that adds a generic load to the project.

Creating the Extension for Adding a Custom Load

The file `DemoLoad.xml` follows.

```
<extension version="1" minorversion="0" name="DemoLoad">
  <guid shortid="DemoLoad">7dfd2b34-dfe1-469d-b244-1c7e5c222e54</guid>
  <script src="main.py" />
  <interface context="Mechanical">
    <images>images</images>
    <toolbar name="Loads" caption="Loads">
      <entry name="DemoLoad" icon="tload">
        <callbacks>
           <onclick>CreateDemoLoad</onclick>
        </callbacks>
      </entry>
    </toolbar>
  </interface>

  <simdata context="Mechanical">

    <load name="DemoLoad" version="1" caption="DemoLoad" icon="tload" color="#00FFFF">

      <callbacks>
        <getnodalvaluesfordisplay>GetNodalValuesForDisplay_DL</getnodalvaluesfordisplay>
        <getprecommands>GetPreCommands_DL</getprecommands>
        <getsolvecommands order="1">GetSolveCommands_DL</getsolvecommands>
      </callbacks>

      <property name="Geometry" caption="Geometry" control="scoping">
        <attributes selection_filter="edge" />
      </property>
      <property name="Text" caption="Text" control="text"></property>
      <property name="FileName" caption="FileName" control="fileopen"></property>
      <property name="SelectStatic" caption="Select (static)" control="select">
        <attributes options="Option 1,Option 2,Option 3" />
      </property>
      <property name="SelectDynamic" caption="Select (dynamic)" control="select">
        <callbacks>
          <onactivate>StringOptions_DL</onactivate>
        </callbacks>
      </property>
      <property name="Double" caption="Double" unit="Length" control="float"
default="1 [m]"></property>


    </load>

  </simdata>

</extension>
```

In this file, the element **`<script>`** references the IronPython script `main.py`. The element **`<interface>`** defines the toolbar and buttons. The callback function **`CreateDemoLoad`** creates and adds the load to the simulation environment.

The element **`<simdata>`** defines the load. The attributes for the child element **`<load>`** provide the name, version, caption, icon, and color that apply to the load. The attribute **`color`** is defined in a hexadecimal format. This color is used to contrast the load when it is displayed on the model. Its child element **`<callbacks>`** defines three callbacks: **`<getnodalvaluesfordisplay>`** and two occurrences of **`<getcommands>`**.

- The callback **`<getnodalvaluesfordisplay>`** specifies the name of the function that is invoked when the load is displayed in the product.

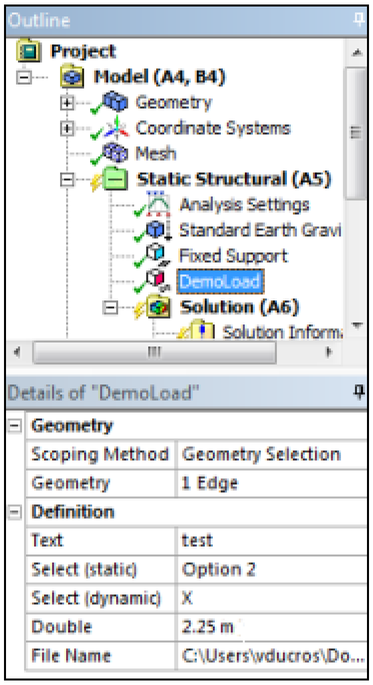
- The two occurrences of the callback **`<getcommands>`** specify the names of the functions that are invoked to insert specific solver commands in the solver input file.
  - For the first occurrence, the attribute **`location`** is set to **`pre`**, indicating that the solver commands defined in the function **`GetPreCommands_DL`** are to be inserted before the standard loads and boundary conditions defined in the product. For the ANSYS solver, the new commands are inserted in the context of the /PREP7 preprocessor after the mesh has been written.
  - For the second occurrence, the attribute **`location`** is set to **`solve`**, indicating that the solver commands defined in the function **`GetSolveCommands_DL`** are to be inserted just before the SOLVE command. Consequently, the related function is responsible for generating the APDL commands that describe the load within the ANSYS input file.

Note:

For the callback **`<getcommands>`**, the attribute **`location`** can be set to any of these values: **`init`**, **`pre`**, **`post`**, **`solve`**, and **`preload`**.

In the callback definition, you define the properties to apply to the definition of the load. These properties are displayed in the **Details** view in Mechanical, where you provide the necessary values to complete the load definition. In the IronPython script, you see how the load's properties can be retrieved and modified.

The following figure shows how each property defined in the XML file appears in the **Details** view.



Geometry	
Scoping Method	Geometry Selection
Geometry	1 Edge
Definition	
Text	test
Select (static)	Option 2
Select (dynamic)	X
Double	2.25 m
File Name	C:\Users\vducros\Do...

For the properties **Select (static)** and **Select (dynamic)**, the attribute **`control`** is set to **`select`**.

- The first property populates the list of options from the XML file.

- The second property defines the callback **`<onactivate>`**. This callback, which is invoked when the control is activated, populates the available options for the property. The definition for the second property provides full control of the options to expose in the drop-down menu and makes the list dependent on the current status of the project. Many different situations that can impact the content of the list can be addressed, as long as they are implemented in the callback **`<onactivate>`**.

The next property, **Double**, does not require the definition of a specific callback. Instead, the attribute **`unit`** is set to **`Length`** to indicate that the property is a physical quantity. The attribute **`default`** is set to **`1 [m]`**. This default value appears in the **Details** view each time a new load is created.

Defining Functions for Adding a Custom Load

The IronPython script `main.py` follows.

```
import os

def init(context):
    ExtAPI.Log.WriteMessage("Init DemoLoads...")

def CreateDemoLoad(analysis):
    analysis.CreateLoadObject("DemoLoad")

def StringOptions_DL(load,prop):
    prop.Options.Clear()
    prop.Options.Add("X")
    prop.Options.Add("Y")
    prop.Options.Add("Z")

def GetPreCommands_DL(load, solverData, stream):
    stream.Write("/COM, **************************************************" + "\n")
    stream.Write("/COM, Load properties from DemoLoad getcommands pre event" + "\n")
    stream.Write("/COM, Text Property = " + load.Properties["Text"].ValueString + "\n")
    stream.Write("/COM, SelectDynamic Property = " + load.Properties["SelectDynamic"].ValueString + "\n")
    stream.Write("/COM, SelectStatic Property = " + load.Properties["SelectStatic"].ValueString + "\n")
    stream.Write("/COM, Double Property = " + load.Properties["Double"].ValueString + "\n")
    stream.Write("/COM, **************************************************" + "\n")

def GetSolveCommands_DL(load, solverData, stream):
    stream.Write("/COM, **************************************************" + "\n")
    stream.Write("/COM, Load properties from DemoLoad getcommands solve event" + "\n")
    stream.Write("/COM, Text Property = " + load.Properties["Text"].ValueString + "\n")
    stream.Write("/COM, SelectDynamic Property = " + load.Properties["SelectDynamic"].ValueString + "\n")
    stream.Write("/COM, SelectStatic Property = " + load.Properties["SelectStatic"].ValueString + "\n")
    stream.Write("/COM, Double Property = " + load.Properties["Double"].ValueString + "\n")
    stream.Write("/COM, **************************************************" + "\n")

def GetNodalValuesForDisplay_DL(load, nodeIds):
    dval = load.Properties["Double"].Value
    coordselect = load.Properties["SelectDynamic"].ValueString
    mesh = load.Analysis.MeshData
    values = []
    for id in nodeIds:
        node = mesh.NodeById(id)
        dispval = float(0.0)
        if coordselect == "X":
            dispval = node.X * float(dval)
        elif coordselect == "Y":
            dispval = node.Y * float(dval)
        elif coordselect == "Z":
            dispval = node.Z * float(dval)
        else:
            dispval = float(0.0)
        values.Add(dispval)
    return values
```

The functions **`GetNodalValuesForDisplay_DL`** and **`GetSolveCommands_DL`** are critical to the behavior and application of the load. **`GetNodalValuesForDisplay_DL`** is called each time the graphic is refreshed. The required input arguments are **`load`** and **`nodeIds`**, where **`load`** is the load object for the load and **`nodeIds`** is a list of node identifiers.

In this script, **`load.Properties["Double"].Value`** queries for the value of the property **`"Double"`**. The argument **`nodeIds`** contains a list of node numbers on which one value has to be returned by the function. For every node in the list, the value of the property **`"Double"`** is assigned in the values array representing the output of the function. This output is subsequently treated by the graphics engine of the product so that the visualization on the FE model is available.

The function **`GetSolveCommands_DL`** is intentionally simplified for this example. The prototype of this function is made of two input arguments, the load object and the filename in which the new specific solver commands are written. The output file is only a temporary file. The content is rewritten in the final solver input file to ensure that the specific commands related to the customized load are merged with all the other commands already defined by the standard features of the product.

Adding Content Menu Options

In the XML file, you can create a specific set of context menu options in the element **`<callbacks>`** for the load. Each menu option is defined in a child element **`<action>`**.

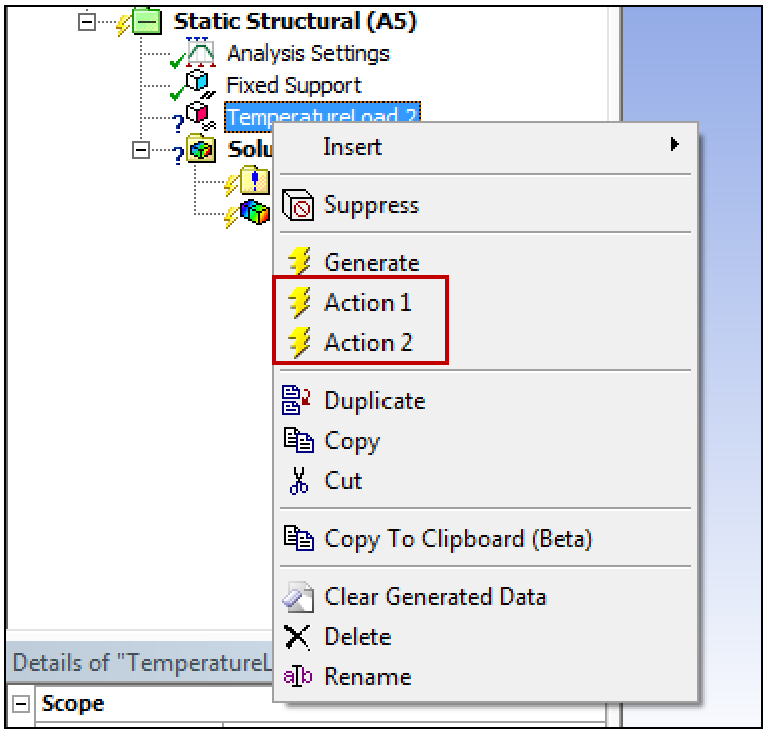
```
<callbacks>

    <ongenerate>generate</ongenerate>
    <oncleardata>clearData</oncleardata>

    <action name="a1" caption="Action 1" icon="update">action1 </action>
    <action name="a2" caption="Action 2" icon="update">action2 </action>

</callbacks>
```

The element **`<action>`** takes three attributes:

- The attribute **`name`** identifies the action.
- The attribute **`caption`** is the text to display in the context menu.
- The attribute **`icon`** specifies the name of the image file to display as the icon. You must place the BMP file that Mechanical requires in the directory **`images`** specified in the extension's XML file.

The element **`<action>`** defines the function to invoke when the associated context menu option is selected.

The following figure shows two additional context menu options, **Action 1** and **Action 2**.



This figure also displays a **Generate** context menu option. This option derives from the standard action **`generate`** provided by Mechanical. For this reason, the declaration of the option **Generate** differs from the declaration of the options **Action1** and **Action2**. The option **Generate** is always associated with the option **Clear Generated Data**.

These options allow you to create a load that can mimic a standard imported load. The callback associated with the option **Generate** replaces the standard function integrated in Mechanical.

The feature is activated when you define the callback **`<ongenerate>`** for the load. The callback **`<ongenerate>`** is invoked each time the **Generate** context menu option is selected. It is also invoked during a solve if the state of the load is set to "not solved."

As for the standard imported load object, the callback **`<ongenerate>`** is called only if the mesh is already generated.

```
<callbacks>

    <ongenerate>generate</ongenerate>
    <oncleardata>clearData</oncleardata>

    <action name="a1" caption="Action 1" icon="update">action1 </action>
    <action name="a2" caption="Action 2" icon="update">action2 </action>

</callbacks>
```

The associated IronPython code looks like this:

```
def generate(load, fct):
    pct = 0
    fct(pct,"Generating data...")

    propEx = load.PropertyByName("Expression")
    exp = propEx.Value
    if exp=="":
        return False
    try:
        vexp = compile(exp,'','eval')
    except:
        return False

    values = SerializableDictionary[int,float]()
    nodeIds = []

    propGeo = load.PropertyByName("Geometry")
    refIds = propGeo.Value
    mesh = ExtAPI.DataModel.MeshDataByName("Global")
    for refId in refIds:
        meshRegion = mesh.MeshRegionById(refId)
        nodeIds += meshRegion.NodeIds

    nodeIds = list(set(nodeIds))

    for i in range(0,nodeIds.Count):
        id = nodeIds[i]
        node = mesh.NodeById(id)
        x = node.X
        y = node.Y
        z = node.Z
        v = 0.
        try:
            v = eval(vexp)
        finally:
            values.Add(id,v)
        new_pct = (int)((i*100.)/nodeIds.Count)
        if new_pct!=pct:
            pct = new_pct
            stopped = fct(pct,"Generating data...")
            if stopped:
                return False

    propEx.SetAttributeValue("Values",values)
    fct(100,"Generating data...")

    return True

def clearData(load):
    ExtAPI.Log.WriteMessage("ClearData: "+load.Caption)
    propEx = load.PropertyByName("Expression")
    propEx.SetAttributeValue("Values",None)
```

The callback **`<ongenerate>`** takes two arguments: the load object and a function to manage a progress bar. The function also takes two arguments: the message to display and the value of the progress bar, which is between 0 and 100.

During the process, the generated data is stored using an attribute on the property **`Expression`**. For more information, see Extension Data Storage.

The callback **`<oncleardata>`** takes one argument: the load object. This callback is invoked each time the mesh is cleared or when the **Clear Generated Data** context menu option is selected.

Defining Objects and Loads that Do Not Affect the Solution State

You can create custom objects or loads that don't affect the solution state. Any interactions with the object or load, such as inserting, suppressing, editing properties, or deleting, will keep the solution up-to-date. The feature can be used by setting the **`affectSolution`** attribute to false on the **`<object>`** or **`<load>`** tag:

```
<object name="objectName" version="1" caption="Object" icon="group" affectsSolution="false" > </object>
```

For example, if a load with the attribute **`affectSolution="false"`** is inserted in a solved analysis, the solution will stay up-to-date. If this attribute is not set, it will default to true. This feature also extends to the child entities. You can define a child entity with the same attribute and it will not affect the solution state.

Tip:

In **`Template2-GenericObjectCreation`**, you can see an example where this callback is implemented. To download all Mechanical templates, on the App Developer Resources page, click the **ACT Templates** tab and then click **ACT Templates for Mechanical**.

Creating Objects and Loads that Have Body Views

To show body views for custom objects or loads, you define the callback **`onDrawBodyViews`**. This callback allows entities to show body views in the same fashion as for native connection objects such as joints, contact region, and so on. With this callback, you can set the properties of a single-body view or a two-body view. Properties include the pane name, scoping, and whether to show the body view.

The return value from this callback determines whether to show body views. The callback gets passed in three arguments: a user load, view one, and view two, respectively.

- View one corresponds to the top pane in the body views as seen in the default pane layout.
- View two corresponds to the bottom pane.

A view has two properties that can be set on it:

- Name of the pane
- Selection to display in the pane

An example follows of the IronPython code:

```
def OnDrawBodyViews(load, view1, view2):
    view1.Name = "View 1"
    view2.Name = "View 2"

    view1.Selection = load.Properties["Scoping1"].Value
    view2.Selection = load.Properties["Scoping2"].Value

    return True
```

To show a single body view, you do not set a pane name value for view two:

```
def OnDrawBodyViewsBodyToGround(load, view1, view2):
    view1.Name = "Body 1"
    view1.Selection = load.Properties["Scoping1"].Value
    return True
```

Note:

Scopings supported by this feature are the same as those supported by native connection objects:

- Faces
- Bodies
- Vertices
- Edges
- Element Faces

Tip:

In **`Template12-BodyView`**, you can see an example where this callback is implemented. To download all Mechanical templates, on the App Developer Resources page, click the **ACT Templates** tab and then click **ACT Templates for Mechanical**.

Coupling Two Sets of Nodes

The supplied extension **`Coupling`** creates a tool for coupling two sets of nodes related to two edges. This extension demonstrates how you can develop your own preprocessing feature, such as a custom load, to address a specific need.

Creating the Extension for Coupling Two Sets of Nodes

The file `Coupling.xml` follows.

```
<extension version="1" name="Coupling">
<guid shortid="Coupling">e0d5c579d-0263-472a-ae0e-b3cbb9b74b6c</guid>
  <script src="main.py" />
  <interface context="Mechanical">
    <images>images</images>
    <toolbar name="Coupling" caption="Coupling">
      <entry name="Coupling" icon="support">

        <callbacks>
          <onclick>CreateCoupling</onclick>
        </callbacks>
      </entry>
    </toolbar>
  </interface>

  <simdata context="Mechanical">

  <load name="Coupling" version="1" caption="Coupling" icon="support" issupport="true"
          color="#FF0000">
   <callbacks>
    <getsolvecommands>SolveCmd</getsolvecommands>
```

```
     <onshow>ShowCoupling</onshow>
     <onhide>HideCoupling</onhide>
    </callbacks>

    <property name="Source" caption="Source" control="scoping">
     <attributes selection_filter="edge" />
     <callbacks>
      <isvalid>IsValidCoupledScoping</isvalid>
      <onvalidate>OnValidateScoping</onvalidate>
     </callbacks>
    </property>

    <property name="Target" caption="Target" control="scoping">
     <attributes selection_filter="edge" />
     <callbacks>
      <isvalid>IsValidCoupledScoping</isvalid>
      <onvalidate>OnValidateScoping</onvalidate>
     </callbacks>
    </property>

    <property name="Reverse" caption="Reverse" control="select" default="No">
     <attributes options="No,Yes" />
     <callbacks>
      <onvalidate>OnValidateReverse</onvalidate>
     </callbacks>
    </property>
   </load>
  </simdata>
</extension>
```

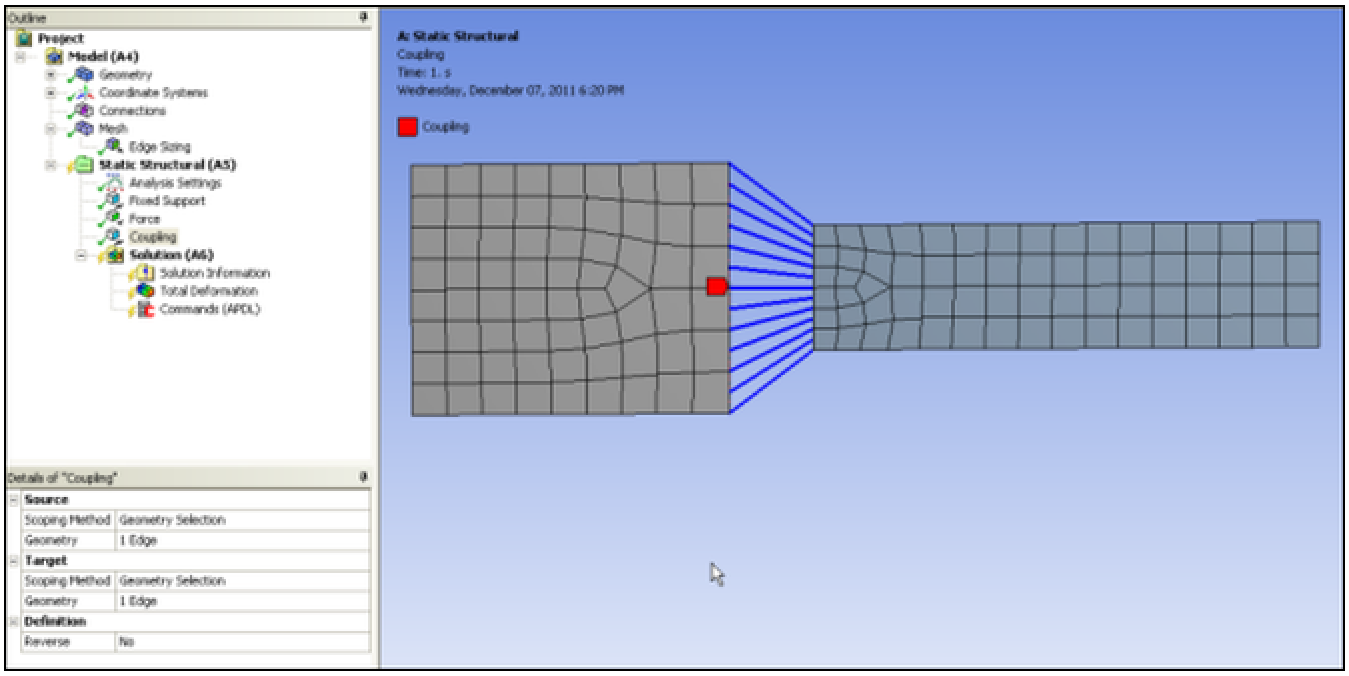
As in the earlier custom load example (p. 7), the element **`<interface>`** adds a toolbar and a toolbar button to Mechanical. The callback function **`<CreateCoupling>`** is invoked when the toolbar button is clicked.

The element **`<simdata>`** encapsulates the information that defines the support. The value for the attribute **`issupport`** is set to true. The attribute **`issupport`** is of particular importance because it tells Mechanical which type of boundary condition to apply. Three result level callback functions are declared.

- The function **`SolveCmd`** is registered and called for an event that gets fired when the solver input is being written.
- Both the functions **`ShowCoupling`** and **`HideCoupling`** are registered and called for events used to synchronize tree view selections with content in the graphics pane.

The details needed to define the inputs and behavior of this special load consist of three properties, **`Source`**, **`Target`**, and **`Reverse`**, along with their behavioral callbacks.

The following figure shows how a fully defined coupling appears in Mechanical.



Details of "Coupling"

Source	
Scoping Method	Geometry Selection
Geometry	1 Edge
Target	
Scoping Method	Geometry Selection
Geometry	1 Edge
Definition	
Reverse	No

Defining Functions for Coupling Two Sets of Nodes

The IronPython script `main.py` for this extension follows.

```
import graphics

def CreateCoupling(analysis):
    analysis.CreateLoadObject("Coupling")

#-------------------------------
#   Callbacks
#-------------------------------

def OnValidateReverse(load, prop):
    ShowCoupling(load)

def OnValidateScoping(load, prop):
    ShowCoupling(load)

def IsValidScoping(load, prop):

    if not prop.Controller.isvalid(load, prop):
        return False

    selection = prop.Value
    if selection == None: return False
    if selection.Ids.Count != 1: return False
    return True

def IsValidCoupledScoping(load, prop):

    sProp = load.Properties["Source"]
    tProp = load.Properties["Target"]

    if not IsValidScoping(load, sProp):
        return False
    if not IsValidScoping(load, tProp):
        return False

    sIds = sProp.Value.Ids
    tIds = tProp.Value.Ids
    try:
        mesh = load.Analysis.MeshData
        sNum = mesh.MeshRegionById(sIds[0]).NodeCount
        tNum = mesh.MeshRegionById(tIds[0]).NodeCount
        if sNum == 0 or tNum == 0: return False
    except:
        return False


    return sNum == tNum

#-------------------------------
#   Show / Hide
#-------------------------------

graphicsContext = {}
def getContext(entity):
     global graphicsContext
     if entity.Id in graphicsContext : return graphicsContext[entity.Id]
     else : return None

def setContext(entity, context):
     global graphicsContext
     graphicsContext[entity.Id] = context

def delContext(entity):
    context = getContext(entity)
    if context != None : context.Visible = False
    context = None
    setContext(entity, None)

def ShowCoupling(load):
    delContext(load)
    ctxCoupling = ExtAPI.Graphics.CreateAndOpenDraw3DContext()

    sourceColor = load.Color
    targetColor = 0x00FF00
    lineColor   = 0x0000FF

    sProp = load.Properties["Source"] ; sSel = sProp.Value
    tProp = load.Properties["Target"] ; tSel = tProp.Value

    ctxCoupling.LineWeight = 1.5
    if sSel != None:
        ctxCoupling.Color = sourceColor
        for id in sSel.Ids:
            graphics.DrawGeoEntity(ExtAPI, load.Analysis.GeoData, id, ctxCoupling)
    if tSel != None:
        ctxCoupling.Color = targetColor
        for id in tSel.Ids:
            graphics.DrawGeoEntity(ExtAPI, load.Analysis.GeoData, id, ctxCoupling)

    if IsValidSelections(load):

        ctxCoupling.Color = lineColor
        ctxCoupling.LineWeight = 1.5

        mesh = load.Analysis.MeshData
        sList, tList = GetListNodes(load)

        for sId, tId in zip(sList, tList):
            sNode = mesh.NodeById(sId)
            tNode = mesh.NodeById(tId)
            ctxCoupling.DrawPolyline([sNode.X,sNode.Y,sNode.Z,tNode.X,tNode.Y,tNode.Z])

    ctxCoupling.Close()
    ctxCoupling.Visible = True
    setContext(load, ctxCoupling)

def HideCoupling(load):
    delContext(load)

#-------------------------------
#   Commands
#-------------------------------

def SolveCmd(load, s):
    s.WriteLine("! Coupling - CP")
    sList, tList = GetListNodes(load)
    for sId, tId in zip(sList, tList):
        s.WriteLine("CP,NEXT,ALL,{0},{1}", sId, tId)

#-------------------------------
#   Utils
#-------------------------------

def IsValidSelections(load):
    return load.Properties["Source"].IsValid and load.Properties["Target"].IsValid

def GetListNodes(load):

    if IsValidSelections(load):

        sProp = load.Properties["Source"] ; sIds = sProp.Value.Ids
        tProp = load.Properties["Target"] ; tIds = tProp.Value.Ids

        geometry = ExtAPI.DataModel.GeoData
        mesh = load.Analysis.MeshData

        sList = GetSubListNodes(geometry, mesh, sIds[0])
        tList = GetSubListNodes(geometry, mesh, tIds[0])

        rev = False
        r = load.Properties["Reverse"].Value
        if r == "Yes": rev = True

        sList = sorted(sList, key=sList.get)
        tList = sorted(tList, key=tList.get, reverse=rev)

    return (sList, tList)

def GetSubListNodes(geometry, mesh, refId):

    entity = geometry.GeoEntityById(refId)
    region = mesh.MeshRegionById(refId)

    result = {}
    pt = System.Array.CreateInstance(System.Double, 3)

    for nodeId in region.NodeIds:
        node = mesh.NodeById(nodeId)
        pt[0], pt[1], pt[2] = (node.X, node.Y, node.Z)
        result[nodeId] = entity.ParamAtPoint(pt)

    return result
```

This script defines a callback function named **`<CreateCoupling>`**. When activated by clicking the **Coupling** toolbar button, this callback creates the load **`Coupling`**. The callback invokes the function **`CreateLoadObject`** for the current analysis. The function **`SolveCmd`** is invoked when the solver input is being generated. **`SolveCmd`** invokes **`GetListNodes`** to obtain two lists of node IDs corresponding to the edges **`Target`** and **`Source`**. These node IDs are then used to write APDL CP commands to the solver input. **`GetListNodes`** is also invoked by the callback function **`<ShowCoupling>`**. In **`<ShowCoupling>`**, the interface **`IGraphics`** is used to create a graphics context. Using the object returned, the inter-nodal lines are drawn to provide a visual representation of the coupling.

The graphics context associated with this custom load and the validation of the user inputs provide for managing more varied situations than the earlier custom load example (p. 7). This explains why this example requires more functions and sub-functions.

Postprocessing Capabilities in Mechanical

You can use ACT to create custom results. For shell and layer elements, you can access existing stress results and create new custom stress results:

Creating a Custom Result (Von Mises Stress)

The supplied extension **Mises** reproduces the result for the averaged von Mises equivalent stress. While this result is already available in the standard results that can be selected in Mechanical, this extension demonstrates how to define a custom result.

Creating the Extension for the Custom Result (Von Mises Stress)

The file **Mises.xml** follows.

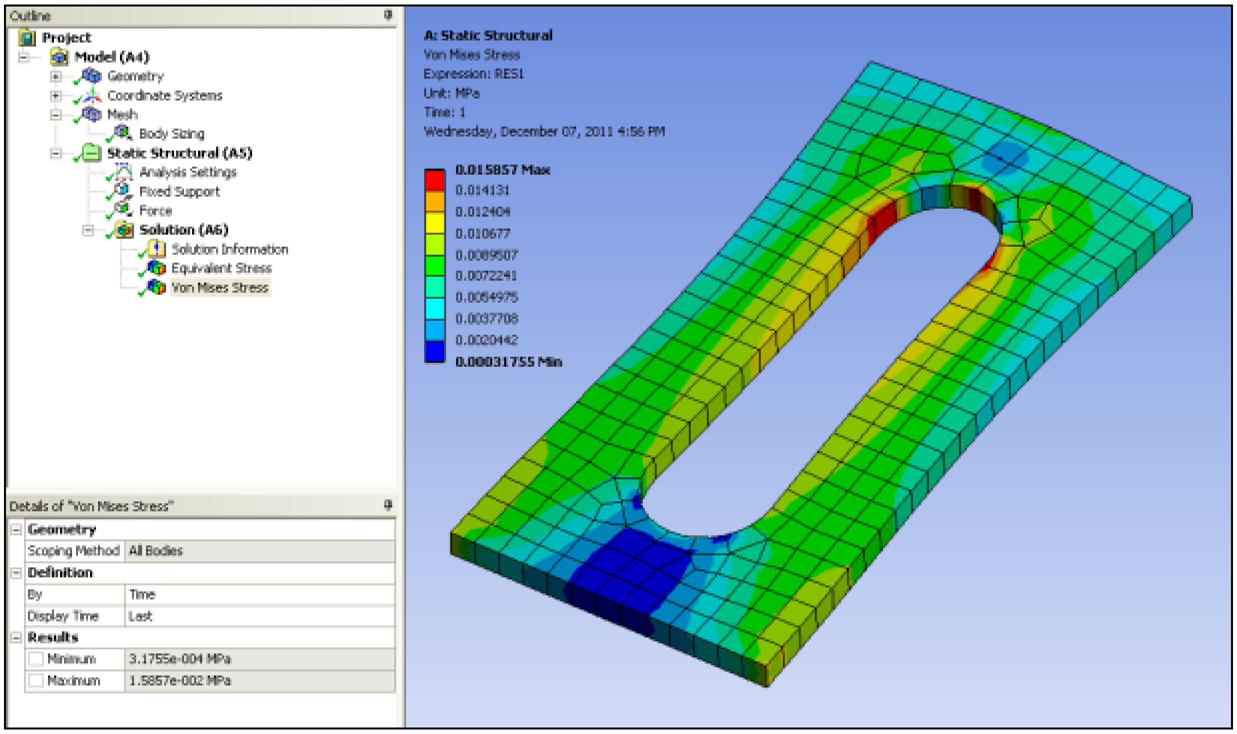
```
<extension version="1" name="Mises">

    <guid shortid="Mises">a1844c3c-b65c-444c-a5ad-13215a9f0413</guid>

 <script src="main.py" />

 <interface context="Mechanical">

  <images>images</images>

  <toolbar name="Von Mises Stress" caption="Von Mises Stress">
   <entry name="Von Mises Stress" icon="result">
    <callbacks>
     <onclick>Create_Mises_Result</onclick>
    </callbacks>
   </entry>
  </toolbar>

 </interface>

 <simdata context="Mechanical">

  <result name="Von Mises Stress" version="1" caption="Von Mises Stress" unit="Stress" icon="result" location=

   <callbacks>
    <onstarteval>Mises_At_Nodes_Eval</onstarteval>
    <getvalue>Mises_At_Nodes_GetValue</getvalue>
   </callbacks>

   <property name="Geometry" caption="Geometry" control="scoping"></property>
  </result>

 </simdata>

</extension>
```

In this file, the element **`<interface>`** adds a toolbar and a toolbar button Mechanical. The callback function **`<Create_Mises_Result>`** is invoked when the toolbar button is clicked.

The element **`<simdata>`** encapsulates the information needed for the custom result. One element **`result`** is declared. The **`type`** attribute on the **`result`** element is used to assign the type of result desired. This can currently be set to either **`scalar`** or **`vector`**. Results of tensor type are not supported.

The callbacks **`onstarteval`** and **`getvalue`** invoke functions for computing and storing the custom result **`Von Mises Stress`**. These callbacks are invoked when custom result values are queried for graphical display. The property **`Geometry`** defines one property to add in the **Details** pane of the custom result. For this example, the property integrates a scoping method in the custom result.

The following figure shows how the toolbar **Von Mises Stress** and the result **Von Mises Stress** are added to Mechanical.



Details of "Von Mises Stress"

Geometry	
Scoping Method	All Bodies
Definition	
By	Time
Display Time	Last
Results	
Minimum	3.1755e-004 MPa
Maximum	1.5857e-002 MPa

Defining Functions for the Custom Result (Von Mises Stress)

The IronPython script `main.py` for this extension follows.

```
import units
import math

context = ExtAPI.Context
```

```
if context == "Mechanical":
    link = {}
    #kHex20
    link.Add(ElementTypeEnum.kHex20,{ 0:[3,1,4], 1:[0,2,5], 2:[1,3,6], 3:[2,0,7], 4:[5,0,7], 5:[1,4,6], 6:[5,7
    #kHex8
    link.Add(ElementTypeEnum.kHex8,{ 0:[3,1,4], 1:[0,2,5], 2:[1,3,6], 3:[2,0,7], 4:[5,0,7], 5:[1,4,6], 6:[5,7,
    #kPyramid13
    link.Add(ElementTypeEnum.kPyramid13,{ 0:[3,1,4], 1:[0,2,4], 2:[1,3,4], 3:[2,0,4], 4:[0,1,2,3], 5:[0,1], 6:
    #kPyramid5
    link.Add(ElementTypeEnum.kPyramid5,{ 0:[3,1,4], 1:[0,2,4], 2:[1,3,4], 3:[2,0,4], 4:[0,1,2,3]})
    #kQuad4
    link.Add(ElementTypeEnum.kQuad4, { 0:[3,1], 1:[0,2], 2:[1,3], 3:[2,0]})
    #kQuad8
    link.Add(ElementTypeEnum.kQuad8, { 0:[3,1], 1:[0,2], 2:[1,3], 3:[2,0], 4:[0,1], 5:[1,2], 6:[2,3], 7:[3,0]}
    #kTet10
    link.Add(ElementTypeEnum.kTet10,{ 0:[2,1,3], 1:[0,2,3], 2:[1,0,3], 3:[0,1,2], 4:[0,1], 5:[1,2], 6:[2,0], 7
    #kTet4
    link.Add(ElementTypeEnum.kTet4, { 0:[2,1,3], 1:[0,2,3], 2:[1,0,3], 3:[0,1,2]})
    #kTri3
    link.Add(ElementTypeEnum.kTri3, { 0:[2,1], 1:[0,2], 2:[1,0]})
    #kTri6
    link.Add(ElementTypeEnum.kTri6, { 0:[2,1], 1:[0,2], 2:[1,0], 3:[0,1], 4:[1,2], 5:[2,0]})
    #kWedge15
    link.Add(ElementTypeEnum.kWedge15,{ 0:[2,1,3], 1:[0,2,4], 2:[1,0,5], 3:[5,4,0], 4:[3,5,1], 5:[4,3,2], 6:[0
    #kWedge6
    link.Add(ElementTypeEnum.kWedge6,{ 0:[2,1,3], 1:[0,2,4], 2:[1,0,5], 3:[5,4,0], 4:[3,5,1], 5:[4,3,2]})


def Create_Mises_Result(analysis):
    analysis.CreateResultObject("Von Mises Stress")


mises_stress = {}

# This function evaluates the specific result (i.e. the Von-Mises stress) on each element required by the geom
# The input data "step" represents the step on which we have to evaluate the result
def Mises_At_Nodes_Eval(result, step):
    global mises_stress

    ExtAPI.Log.WriteMessage("Launch evaluation of the Mises result at nodes: ")
    # Reader initialization
    reader = result.Analysis.GetResultsData()
    istep = int(step)
    reader.CurrentResultSet = istep
    # Get the stress result from the reader
    stress = reader.GetResult("S")
    stress.SelectComponents(["X", "Y", "Z", "XY", "XZ", "YZ"])
    unit_stress = stress.GetComponentInfo("X").Unit
    conv_stress = units.ConvertUnit(1.,unit_stress,"Pa","Stress")

    # Get the selected geometry
    propGeo = result.Properties["Geometry"]
    refIds = propGeo.Value.Ids

    nodal_stress = [0.] * 6

    # Get the mesh of the model
    mesh = result.Analysis.MeshData

    # Loop on the list of the selected geometrical entities
    for refId in refIds:
        # Get mesh information for each geometrical entity
        meshRegion = mesh.MeshRegionById(refId)
        nodeIds = meshRegion.NodeIds

        # Loop on the nodes related to the current geometrical refId
        for nodeId in nodeIds:

            for i in range(6):
                nodal_stress[i] = 0.
            elementIds = mesh.NodeById(nodeId).ConnectedElementIds
            num = 0
            # Loop on the elements related to the current node
            for elementId in elementIds:
                # Get the stress tensor related to the current element
                tensor = stress.GetElementValues(elementId)

                element = mesh.ElementById(elementId)
                # Look for the position of the node nodeId in the element elementId
                # cpt contains this position
                cpt = element.NodeIds.IndexOf(nodeId)

                # for corner nodes, cpt is useless.
                # The n corner nodes of one element are always the first n nodes of the list
                if cpt < element.CornerNodeIds.Count:
                    for i in range(6):
                        nodal_stress[i] = nodal_stress[i] + tensor[6*cpt+i]
                else:
                # For midside nodes, cpt is used and the link table provides the two neighbouring corner nodes
                    itoadd = link[element.Type][cpt]
                    for ii in itoadd:
                        for i in range(6):
                            nodal_stress[i] = nodal_stress[i] + tensor[6*ii+i] / 2.

                num += 1

            # The average stress tensor is computed before to compute the Von-Mises stress
            # num is the number of elements connected with the current node nodeId
            for i in range(6):
                nodal_stress[i] *= conv_stress / num

            # Von-Mises stress computation
            vm_stress = Mises(nodal_stress)
            # Result storage
            mises_stress[nodeId] = [vm_stress]


# This function returns the values array. This array will be used by Mechanical to make the display available
def Mises_At_Nodes_GetValue(result,nodeId):
    global mises_stress
    try:    return mises_stress[nodeId]
    except: return [System.Double.MaxValue]


# This function computes the Von-Mises stress from the stress tensor
# The Von-Mises stess is computed based on the three eigenvalues of the stress tensor
def Mises(tensor):

    # Computation of the eigenvalues
    (S1, S2, S3) = EigenValues(tensor)
    return sqrt( ( (S1-S2)*(S1-S2) + (S2-S3)*(S2-S3) + (S1-S3)*(S1-S3) ) / 2. )


# This function computes the three eigenvalues of one [3*3] symetric tensor
EPSILON = 1e-4
def EigenValues(tensor):

    a = tensor[0]
    b = tensor[1]
    c = tensor[2]
    d = tensor[3]
    e = tensor[4]
    f = tensor[5]

    if ((abs(d)>EPSILON) or (abs(e)>EPSILON) or (abs(f)>EPSILON)):
        # Polynomial reduction
        A = -(a+b+c)
        B = a*b+a*c+b*c-d*d-e*e-f*f
        C = d*d*c+f*f*a+e*e*b-2*d*e*f-a*b*c

        p = B-A*A/3
        q = C-A*B/3+2*A*A*A/27
        R = sqrt(fabs(p)/3)
        if q < 0: R = -R

        z = q/(2*R*R*R)
        if z < -1. : z = -1.
        elif z > 1.: z = 1.
        phi = acos(z)

        S1 = -2*R*cos(phi/3)-A/3
        S2 = -2*R*cos(phi/3+2*math.pi/3)-A/3
        S3 = -2*R*cos(phi/3+4*math.pi/3)-A/3
    else:
        S1 = a
        S2 = b
        S3 = c

    return (S1, S2, S3)
```

This script defines a callback function named **`<Create_Mises_Result>`**. When activated by clicking the **Von Mises Stress** toolbar button, the callback creates the result **Von Mises Stress**. The callback invokes the function **`Create_Mises_Result`** on the interface **`simDataMgr`**.

In addition, the script defines the functions used by the extension. Near the top of the script is the list definition for the node adjacency tables. Named **`link`**, the list contains hash maps. Each hash map has a numeric hash key that corresponds to the type returned by the property **`IElementType`**. The values for the hash map are in arrays. The names of the arrays match the local node numbers for a typical element of the key (element-type). Finally, the content of each array provides the adjacent local node numbers for the corresponding elements node. This list of arrays makes the extension compatible with all the different element topologies that Mechanical potentially creates during the mesh generation.

The XML file for this extension specifies one other callback in the script:

```
<onstarteval>Mises_At_Nodes_Eval</onstarteval>
```

The callback **`<onstarteval>`** is associated with the function **`Mises_At_Nodes_Eval`**. It is invoked as the result of an event thrown by Mechanical. This function computes the appropriate stress results and stores them. The comments in the script describe the algorithm used. It returns the nodal results set stored for the specified node ID. This callback is used by the graphics system to display the results.

The script also demonstrates the use of utility sub-functions. Using sub-functions to perform common repetitive task helps to make the script modular and more understandable. The utility sub-functions used here are **`Mises`** and **`EigenValues`**.

- **`Mises`** is called by the function **`Mises_At_Nodes_Eval`** and returns the computed equivalent stress for a given stress tensor (SX,SY,SZ,SXY,SXZ,SYZ).
- **`EigenValues`** is called by the function **`Mises`** to compute the Eigen vectors for a given stress tensor.

Creating a Custom Result (Absolute Principal Stress)

The supplied extension `DemoResult.xml` shows how to add a toolbar with a single button to create a custom result that computes the worst value of the principal stresses for the scoped geometry entities.

Creating the Extension for the Custom Result (Absolute Principal Stress)

The file `DemoResult.xml` follows.

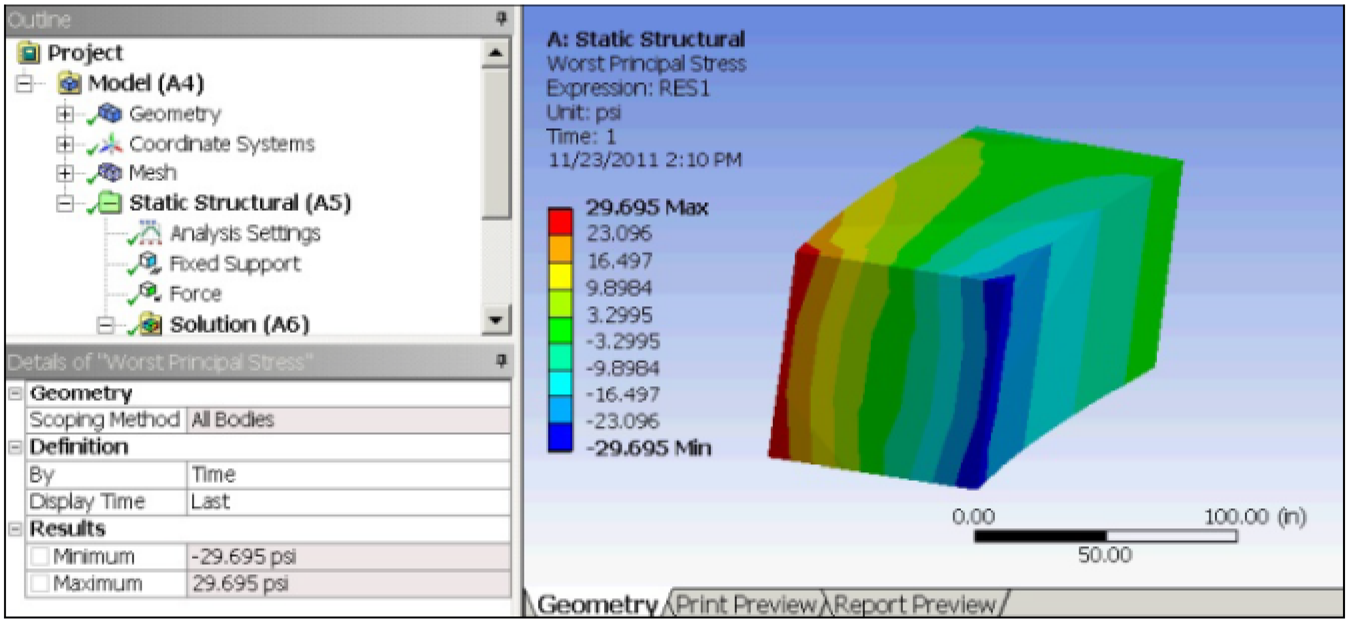
```
<extension version="1" name="DemoResult">

  <guid shortid="DemoResult">1ed8a677-3f81-49eb-9f31-41364844c769</guid>

  <script src="demoresult.py" />

  <interface context="Mechanical">

    <images>images</images>

    <callbacks>
      <oninit>init</oninit>
    </callbacks>

    <toolbar name="Stress Results" caption="Extension: Worst Principal Stress">
      <entry name="Worst Principal Stress" icon="result">
        <callbacks>
          <onclick>Create_WPS_Result</onclick>
        </callbacks>
      </entry>
    </toolbar>

  </interface>

  <simdata context="Mechanical">

    <result name="Worst Principal Stress" version="1" caption="Worst Principal Stress" unit="Stress" icon="res

      <callbacks>
        <onstarteval>WPS_Eval</onstarteval>
        <getvalue>WPS_GetValue</getvalue>
      </callbacks>

      <property name="Geometry" caption="Geometry" control="scoping"></property>

    </result>

  </simdata>

</extension>
```

In this file, the element **`<script>`** references the IronPython script `demoresult.py`. The element **`<interface>`** defines a toolbar and a button. In the element **`<callbacks>`**, the function **`Create_WPS_Result`** creates and adds the result to the simulation environment.

The element **`<simdata>`** defines the result. The attributes in the child element **`<result>`** provide the name, version, caption, unit, and icon that apply to the result. The child element **`<callbacks>`** defines the single callback **`<onstarteval>`**. This callback specifies the function to invoke to compute the result when Mechanical requests an evaluation. The output of this function is sent directly to Mechanical, which displays the result.

In the callback definition, you define the properties to apply to the result definition. These properties display in the **Details** view of Mechanical when a result is selected in the **Outline** view.

The **Details** view shows each property with the corresponding names and result values.



A: Static Structural
Worst Principal Stress
Expression: RES1
Unit: psi
Time: 1
11/23/2011 2:10 PM

Details of "Worst Principal Stress"

Geometry	
Scoping Method	All Bodies
Definition	
By	Time
Display Time	Last
Results	
Minimum	-29.695 psi
Maximum	29.695 psi

Defining Functions for the Custom Result (Absolute Principal Stress)

The IronPython script `demoresult.py` follows.

```
import units

wps_stress = {}
eigenvalues = {}

def init(context):
    ExtAPI.Log.WriteMessage("Init Demoresult Extension...")

def Create_WPS_Result(analysis):
    ExtAPI.Log.WriteMessage("Launch Create_WPS_Result...")
    analysis.CreateResultObject("Worst Principal Stress")

# This function evaluates the specific result (i.e. the Absolute principal stress) on each element required by
# The input data "step" represents the step on which you have to evaluate the result
def WPS_Eval(result,step):
    global wps_stress
    analysis = result.Analysis
    ExtAPI.Log.WriteMessage("Launch evaluation of the WPS result: ")
    # Reader initialization
    reader = analysis.GetResultsData()
    reader.CurrentResultSet = step
    # Get the stress result from the reader
    stress = reader.GetResult("S")

    # Define which component of the stress tensor you want to work with.
    stress.SelectComponents(["X","Y","Z","XY","XZ","YZ"])
    # Unit sytem management:
    # First get the unit system that was used during the resolution
    # Second compute the coefficient to be used so that the final result will be returned in the SI unit syste
    unit_stress = stress.GetComponentInfo("X").Unit
    conv_stress = units.ConvertUnit(1.,unit_stress,"Pa","Stress")

    # Get the selected geometry
    propGeo = result.Properties["Geometry"]
    refIds = propGeo.Value.Ids
    # Get the mesh of the model
    mesh = analysis.MeshData
    #Loop on the list of the selected geometrical entities
    for refId in refIds:
        # Get mesh information for each geometrical entity
        meshRegion = mesh.MeshRegionById(refId)
        elementIds = meshRegion.ElementIds
        # Loop on the elements related to the current geometrical entity
        for elementId in elementIds:
            # Get the stress tensor related to the current element
            tensor = stress.GetElementValues(elementId)
            # Get element information
            element = mesh.ElementById(elementId)
            nodeIds = element.CornerNodeIds

            wps_stress[elementId] = []
            # Loop on the nodes of the current element to compute the Von-Mises stress on the element nodes
            for i in range(0,nodeIds.Count):
                local_wps = WPS(tensor[6*i:6*(i+1)])

                #  Final stress result has to be returned in theSI unit system
                local_wps = local_wps * conv_stress
                wps_stress[elementId].Add(local_wps)

# This function returns the values array. This array will be used by Mechanical to make the display available
def WPS_GetValue(result,elementId):
    global wps_stress
    if elementId in wps_stress:
        values = wps_stress[elementId]
    else:
        values = []
        mesh = result.Analysis.MeshData
        element = mesh.ElementById(elementId)
        nodeIds = element.CornerNodeIds
        for id in nodeIds:
            values.Add(System.Double.MaxValue)
    return values

# This function computes the absolute principal stress from the stress tensor
# The Von-Mises stess is computed based on the three eigenvalues of the stress tensor
def WPS(tensor):

    # Computation of the eigenvalues
    eigenvalues = EigenValues(tensor)

    # Extraction of the worst principal stress
    wplocal_stress = eigenvalues[0]
    if abs(eigenvalues[1])>abs(wplocal_stress):
        wplocal_stress = eigenvalues[1]
    if abs(eigenvalues[2])>abs(wplocal_stress):
        wplocal_stress = eigenvalues[2]

    # Return the worst value of the three principal stresses S1, S2, S3
    return wplocal_stress


# This function computes the three eigenvalues of one [3*3] symetric tensor
EPSILON = 1e-4
def EigenValues(tensor):
    global eigenvalues

    eigenvalues = []

    a = tensor[0]
    b = tensor[1]
    c = tensor[2]
    d = tensor[3]
    e = tensor[4]
    f = tensor[5]
    if ((abs(d)>EPSILON) or (abs(e)>EPSILON) or (abs(f)>EPSILON)):
        # Polynomial reduction
        A = -(a+b+c)
        B = a*b+a*c+b*c-d*d-e*e-f*f
        C = d*d*c+f*f*a+e*e*b-2*d*e*f-a*b*c

        p = B-A*A/3
        q = C-A*B/3+2*A*A*A/27
        if (q<0):
            R = -sqrt(fabs(p)/3)
        else:
            R = sqrt(fabs(p)/3)

        phi = acos(q/(2*R*R*R))

        S1=-2*R*cos(phi/3)-A/3
        S2=-2*R*cos(phi/3+2*3.1415927/3)-A/3
        S3=-2*R*cos(phi/3+4*3.1415927/3)-A/3
    else:
        S1 = a
        S2 = b
        S3 = c

    eigenvalues.Add(S1)
    eigenvalues.Add(S2)
    eigenvalues.Add(S3)

    return eigenvalues
```

The functions **`Create_WPS_Result`** and **`WPS_Eval`** are critical to the behavior and application of the result.

- The function **`Create_WPS_Result`** creates the result and adds it to the simulation environment. This function uses IronPython dot notation, which allows you to chain objects with methods that return objects to each other. In the expression **`analysis.CreateResultObject"Absolute Principal Stress"`**, the **`IAnalysis`** object analysis is given in the argument of the callback. The object **`IAnalysis`** calls the method **`CreateResultObject`**. From this method, the XML is interpreted and the internal mechanisms are set into motion to add the details and register the callbacks that define the results framework. For more comprehensive descriptions of the API objects, methods, and properties, see the ANSYS ACT API Reference Guide.

- The function **`WPS_Eval`** is called when the result must be evaluated or re-evaluated, depending on the state of the simulation environment. The function definition requires the input arguments **`result`** and **`step`** and the output argument collector. **`Result`** is the result object for the result, and **`stepInfo`** is an object that gives information on the step currently prescribed in the details.

- The function **`WPS_Eval`** queries for the component stresses at each node of elements. The stress tensor computes the three principal stresses (eigenvalues). The signed maximum value over these three is then stored as the final result for each node of elements. **`WPS_Eval`** also deals with the conversion of units. **`DemoResults`** uses a utility library called **`units`**, which is imported at the beginning of the script `demoresults.py`. With this library, **`WPS_Eval`** can derive a conversion factor for the stress units that is consistent with the units used during solution. The result to be returned by the evaluation function must be consistent with the international system of unit. Workbench does the conversion to the current unit system in the product internally. Whenever the result values for **Worst Principal Stress** are needed for display purposes, Mechanical uses the output of the function **`WPS_Eval`**. The two callbacks and their registration to the event infrastructure of Mechanical make possible the full cycle of result definition, creation, evaluation, and display.

To facilitate the development of functions to evaluate simulation results, the third output argument collector is internally initialized based on the content of the scoping property. When defined, this property contains the list of FE entities on which results are evaluated. This information can be used directly in functions without looping over mesh regions. The function **`WPS_Eval`** demonstrates the use of this method.

This extension takes advantage of the fact that the property **`collector.Ids`** is initialized with the element numbers related to the selected geometrical entities. This list can be used to evaluate results for each element. The property **`collector.Ids`** contains nodes or element numbers depending on the type of result defined in the XML file. For results with the location set to **`node`**, **`collector.Ids`** contains a list of node numbers. For results with the location set to **`elem`** or **`elemnode`**, **`collector.Ids`** contains a list of element numbers.

Accessing Results on Shell and Layer Elements

You can access results on shell and layer elements. A shell element is a specific case of a layer element. It is simply a layer element that has only one layer.

For a shell or layer element, the results can have different shapes, depending on the data location. Results can be nodal, element nodal, or elemental.

- For a nodal location, values are localized at each node of the element. No additional specification on the shape or position is necessary.
- For an element nodal location (such as stress), results are available on four positions: **`Top`**, **`Bottom`**, **`Middle`**, and **`Top/Bottom`**. You must specify the position for which to access result values. The default position is **`Top/Bottom`**.
- For an elemental location (such as bending stress), the result for the different components is given by one value by element.

The result reader has a property **`LayeredSolidStressStyle`** that you use to specify the shape for the required result. The default value is **`Top/Bottom`**.

For a layer element, the result reader has the property **`Layer`** that you use to specify the layer number for the required result. The layer number is **`0`** by default, which is equivalent to selecting **Entire Section** in the Mechanical interface.

The following table shows how to specify the properties **`LayeredSolidStressStyle`** and **`ShellPosition`**, depending of the result required.

Result Type Exposed in Worksheet	Location	LayeredSolidStressStyle	ShellPosition
U, R, …	Nodal	(Not Applicable)	(Not Applicable)
S, EPEL, …	Element Nodal	Top/Bottom	Tom, Bottom, Middle, Top/Bottom
BENDING_STRESS, MEMBRANE_STRESS, …	Elemental	Bending, Membrane	(Not Applicable)

The following IronPython functions access a shell element result: stress **`S`** for the Y component.

```
# create a result reader
resultReader = result.Analysis.GetResultsData()

# get the layer number
resultReader.Layer = 0

#specify the shape of the result
resultReader.LayeredSolidStressStyle = StressStyle.TopAndBottom

# select the shell position
resultReader.ShellPosition = ShellPosition.Top

# set the result reader to get the stress on the Y axis (SY) for the current step
resultReader.CurrentResultSet = stepInfo.Set
stress_result = resultReader.GetResult("S")
stress_result.SelectComponents(["Y"])

# get the elemental values SY on "Top" position on the shell
for id in resultCollector.Ids:
    valuesSY = stress_result.GetElementValues(id)
    resultCollector.SetValues(id,valuesSY)
```

For a shell or layer element with the position set to **`Top`**, **`Bottom`**, or **`Middle`**, the method **`SetValues`** must be used:

```
for id in collector.Ids:
    values = stress_result.GetElementValues(id)
    collector.SetValues(id,values)
```

For a shell or layer element with the position set to **`Top/Bottom`**, the method **`SetAllValues`** can be used:

```
values = stress_result.GetElementValues(collector.Ids,False)
lengths = reader.NumberValuesByElement(collector.Ids)
collector.SetAllValues(lengths,values)
```

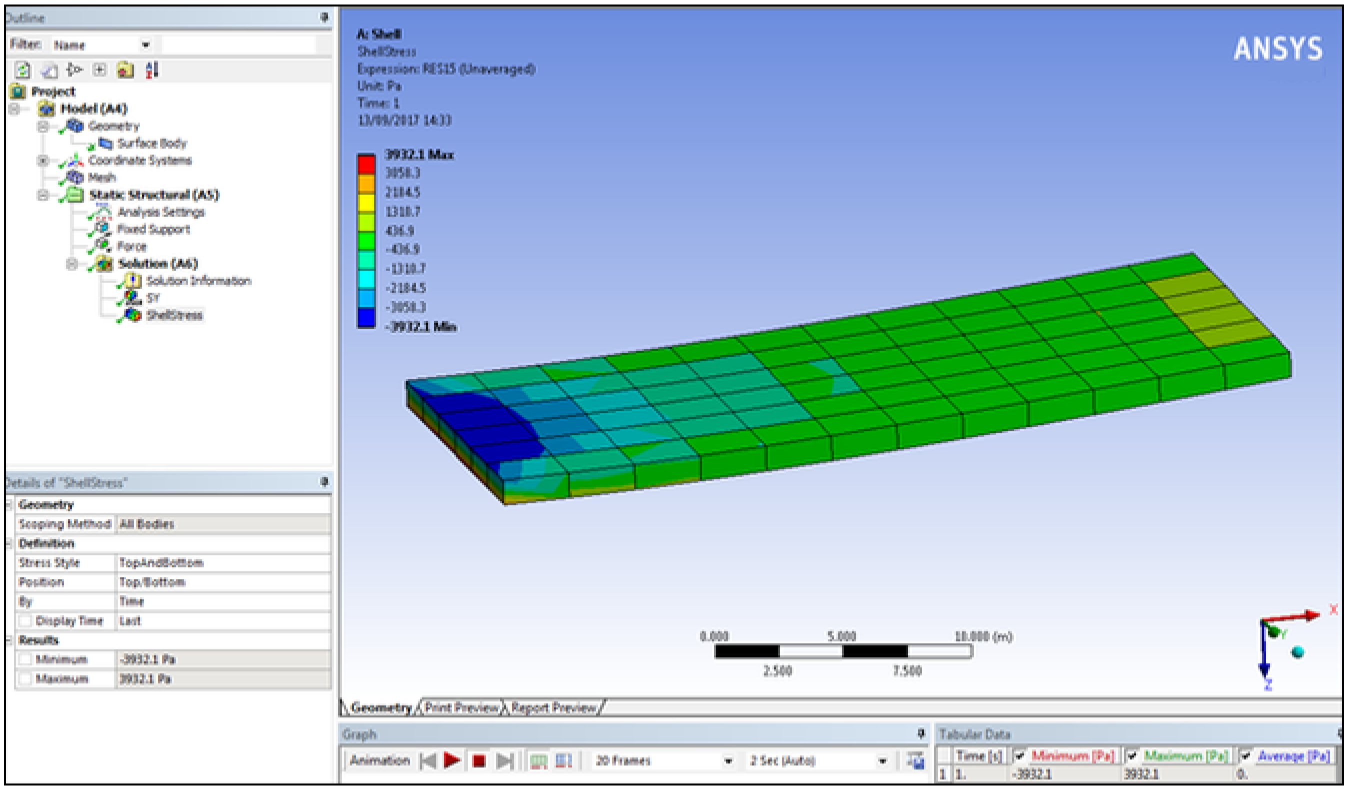
While the code for the method **`SetAllValues`** is bit more complicated, this method does offer better performance when setting a large amount of data.

Note:

The supplied extension **`ACTResults`** defines a toolbar with buttons for calculating custom results for a shell element, a layer element, and a contact. The extension **`ACTResults`** combines the next three extension examples into a single extension. The three-button toolbar for it is shown in the figures.

Creating a Custom Stress Result on a Shell Element

On a shell element in the Mechanical tree, you can create an ACT result object to access the stress in a specified direction and display it as a custom result.



A: Shell
ShellStress
Expression: RES15 (Unaveraged)
Unit: Pa
Time: 1
13/09/2017 14:33

Details of "ShellStress"

Geometry	
Scoping Method	All Bodies
Definition	
Stress Style	TopAndBottom
Position	Top/Bottom
By	Time
Display Time	Last
Results	
Minimum	-3932.1 Pa
Maximum	3932.1 Pa

Tabular Data

	Time [s]	Minimum [Pa]	Maximum [Pa]	Average [Pa]
1	1.	-3932.1	3932.1	0.

Creating the Extension for a Custom Stress Result on a Shell Element

The following XML file adds a toolbar with a single button for creating a custom stress result on a shell element. The custom stress result is in the Y direction.

```
<extension version="1" minorversion="0" name="ACTResult">

   <author>Ansys Inc.</author>
   <description>ACT shell result</description>
   <guid shortid="ACTResults">****</guid>
   <script src="main.py" />

   <interface context="Mechanical">
     <images>images</images>
     <toolbar name="ACTResults" caption="ACTResults">
       <entry name="ShellStress" icon="shellActRes">
         <callbacks>
           <onclick>CreateShellStress</onclick>
         </callbacks>
       </entry>
     </toolbar>
   </interface>

   <simdata context="Mechanical">
     <result name="ShellStress" version="1" caption="ShellStress" unit="Stress" icon="result"
             location="elemnode" type="scalar" class="ActResController">
       <property name="Geometry" caption="Geometry" control="scoping"></property>
       <property name="StressStyle" caption="Stress Style" control="select" default="TopAndBottom">
          <attributes options="Top/Bottom,Top,Bottom,Membrane,Bending"></attributes>
       </property>
       <property name="Position" caption="Position" control="select" default="Top/Bottom">
         <attributes options="Top/Bottom,Top,Middle,Bottom"></attributes>
       </property>
       <callbacks>
         <evaluate>EvaluateShell</evaluate>
       </callbacks>
```

```
      </result>
    </simdata>

  </extension>
```

In this file, the element **`<script>`** references the IronPython script **`main.py`**. The element **`<interface>`** defines the toolbar and a button. In the child element **`<callbacks>`**, the single callback **`<onclick>`** invokes the function **`CreateShellStress`**, which creates and adds the result to the simulation environment.

The element **`<simdata>`** defines the result.

- The attributes in the child element **`<result>`** provide the name, version, caption, unit, and icon that apply to the result. These properties display in the **Details** view of Mechanical when the result is selected in the **Outline** view. The **Details** view shows each property with the corresponding names and result values.
- In the child element **`<callbacks>`**, the single callback **`<evaluate>`** invokes the function **`EvaluateShell`** when Mechanical requests an evaluation. The output of this function is sent directly to Mechanical, which displays the result.

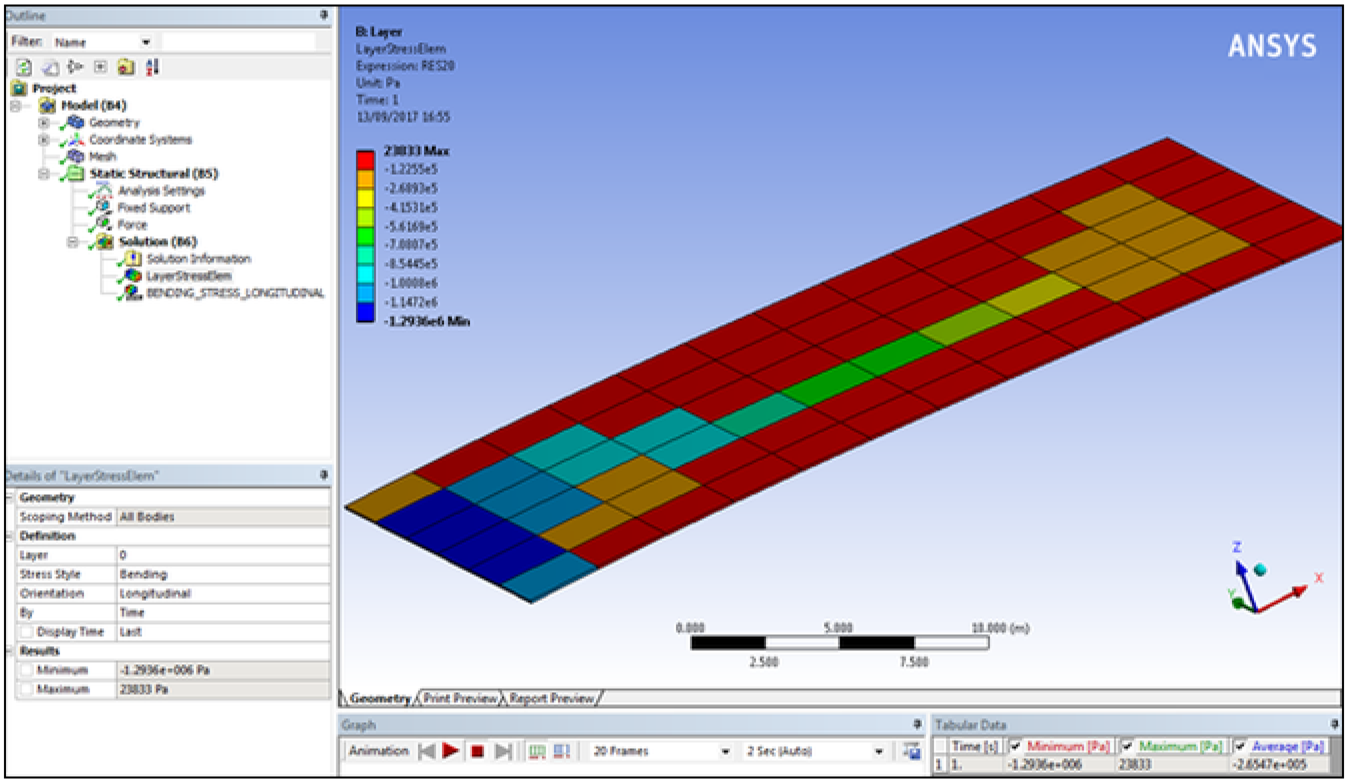
Defining Functions for a Custom Stress Result on a Shell Element

The IronPython script **`main.py`** follows. This sample code shows how to create the result object, calculate the result, and set up the result reader. For more information on the result reader, see Retrieving Mechanical Results More Efficiently (p. 38).

```
def CreateShellStress(analysis):
    analysis.CreateResultObject("ShellStress")

def EvaluateShell(result, stepInfo, collector):
    result.Controller.EvaluateShell(result, stepInfo, collector)

class ActResController:

    def __init__(self, api, result):
        pass

    def EvaluateShell(self, result, stepInfo, collector):
        # create a result reader
        reader = result.Analysis.GetResultsData()

        # get the LayeredSolidStressStyle
        stressStyle_value = result.Properties["StressStyle"].Value
        stress_style = self.stressStyle(stressStyle_value)
        reader.LayeredSolidStressStyle = stress_style

        # get the shell position
        position_value = result.Properties["Position"].Value
        shell_position = self.position(position_value)

        # set up the reader
        reader.ShellPosition = shell_position

        reader.CurrentResultSet = stepInfo.Set
        stress_result = reader.GetResult("S")
        stress_result.SelectComponents(["Y"])

        for id in collector.Ids:
           values = stress_result.GetElementValues(id)
           collector.SetValues(id, values)
    def stressStyle(self,argument):
        switcher = {
            "Top":StressStyle.Top,
            "Bottom":StressStyle.Bottom,
            "TopAndBottom":StressStyle.TopAndBottom,
            "Membrane":StressStyle.Membrane,
            "Bending":StressStyle.Bending
            }
        return switcher.get(argument, StressStyle.TopAndBottom)

    def position(self,argument):
        """equivalent to a switch/case"""
        switcher = {
            "Top":ShellPosition.Top,
            "Bottom":ShellPosition.Bottom,
            "Middle":ShellPosition.Middle,
            "Top/Bottom":ShellPosition.TopAndBottom
            }
        return switcher.get(argument, "Top/Bottom")
```

Creating a Custom Stress Result on a Layer Element

On a layer element in the Mechanical tree, you can create an ACT result object to access the elemental stress (bending stress and membrane stress) and display it as a custom result.



Details of "LayerStressElem"

Geometry	
Scoping Method	All Bodies
Definition	
Layer	0
Stress Style	Bending
Orientation	Longitudinal
By	Time
Display Time	Last
Results	
Minimum	-1.2936e+006 Pa
Maximum	23833 Pa

	Time [s]	Minimum [Pa]	Maximum [Pa]	Average [Pa]
1	1.	-1.2936e+006	23833	-2.6547e+005

Creating an Extension for a Custom Stress Result on a Layer Element

The following XML file adds a toolbar with a single button for creating a custom stress result on a layer element.

```
<extension version="1" minorversion="0" name="ACTResult">

   <author>Ansys Inc.</author>
```

```
  <description>ACT layer result</description>
  <guid shortid="ACTResults">****</guid>
  <script src="main.py" />

  <interface context="Mechanical">
    <images>images</images>
    <toolbar name="ACTResults" caption="ACTResults">
      <entry name="LayerStressElem" icon="LayerActRes">
        <callbacks>
          <onclick>CreateLayerStressElem</onclick>
        </callbacks>
      </entry>
    </toolbar>
  </interface>

  <simdata context="Mechanical">
     <result name="LayerStressElem" version="1" caption="LayerStressElem" unit="Stress" icon="result" locatio
       <property name="Geometry" caption="Geometry" control="scoping"></property>
       <property name="Layer" caption="Layer" control="integer" default="0"></property>
       <property name="StressStyle" caption="Stress Style" control="select" default="TopAndBottom">
             <attributes options="Top,Bottom,TopAndBottom,Membrane,Bending"></attributes>
       </property>
       <property name="Orientation" caption="Orientation" control="select" default="Longitudinal">
         <attributes options="Longitudinal,Transverse,InPlanShear"></attributes>
       </property>
       <callbacks>
         <evaluate>EvaluateLayerStressElem</evaluate>
       </callbacks>
     </result>
  </simdata>

</extension>
```

In this file, the element **`<script>`** references the IronPython script **`main.py`**. The element **`<interface>`** defines the toolbar and a button. In the child element **`<callbacks>`**, the single callback **`<onclick>`** invokes the function **`CreateLayerStressElem`**, which creates and adds the result to the simulation environment.

The element **`<simdata>`** defines the result.

- The attributes in the child element **`<result>`** provide the name, version, caption, unit, and icon that apply to the result. These properties display in the **Details** view of Mechanical when the result is selected in the **Outline** view. The **Details** view shows each property with the corresponding names and result values.
- In the child element **`<callbacks>`**, the single callback **`<evaluate>`** invokes the function **`EvaluateLayerStressElem`** when Mechanical requests an evaluation. The output of this function is sent directly to Mechanical, which displays the result.

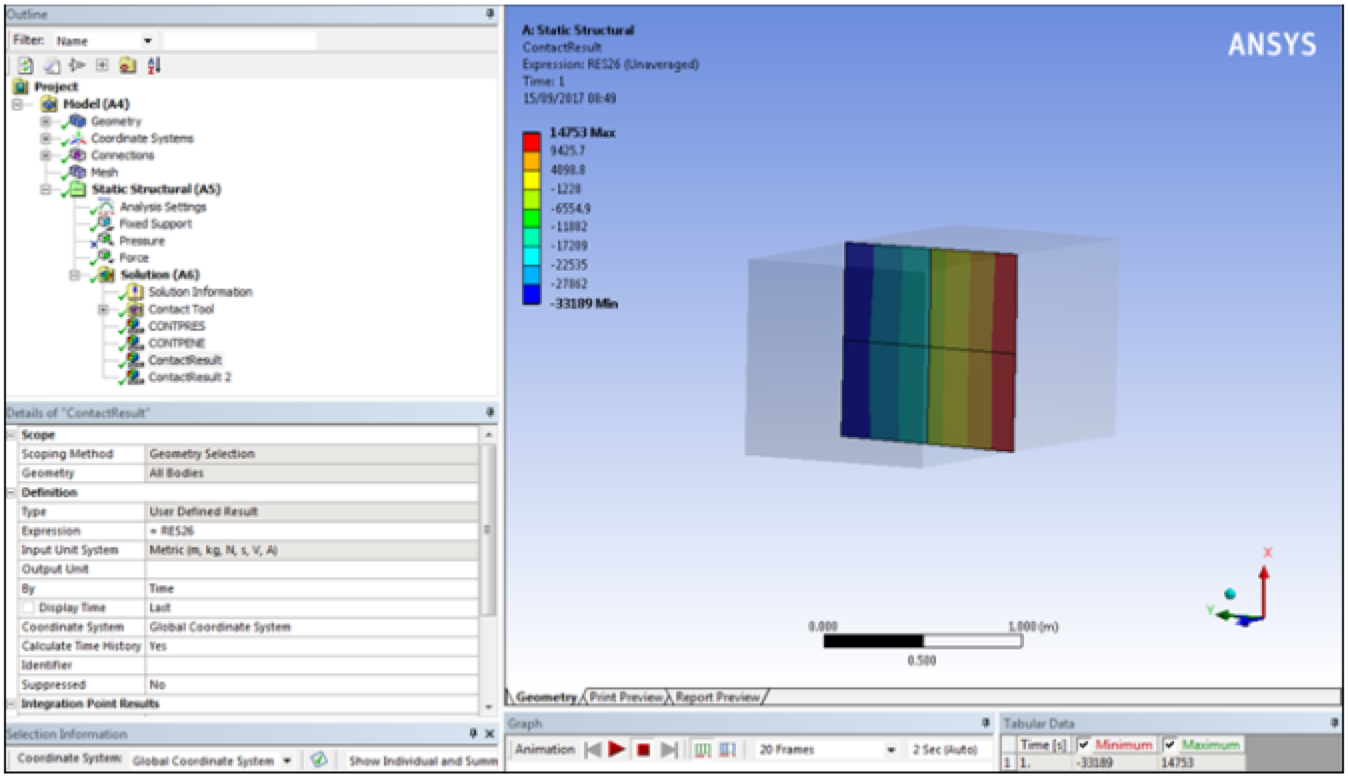
Defining Functions for a Custom Stress Result on a Layer Element

The IronPython script **`main.py`** follows. This sample code shows how to create the result object, calculate the result, and set up the result reader. For more information on the result reader, see Retrieving Mechanical Results More Efficiently (p. 38).

```
def CreateLayerStressElem(analysis):
    analysis.CreateResultObject("LayerStressElem")

def EvaluateLayerStressElem(result, stepInfo, collector):
    result.Controller.EvaluateLayerStressElem(result, stepInfo, collector)

class ActResController:
    def __init__(self, api, result):
        pass

    def EvaluateLayerStressElem(self, result, stepInfo, collector):
        reader = result.Analysis.GetResultsData()

        # get the layer number and set up the reader
        reader.Layer = result.Properties["Layer"].Value

        # get the LayeredSolidStressStyle
        stressStyle_value = result.Properties["StressStyle"].Value
        stress_style = self.stressStyle(stressStyle_value)
        reader.LayeredSolidStressStyle = stress_style

        #get the shell stress orientation
        orientation_value = result.Properties["Orientation"].Value
        shell_orientation = self.shellStressOrientation(orientation_value)

        reader.CurrentResultSet = stepInfo.Set

        if(self.hasPosition(stress_style)):
            ExtAPI.Log.WriteError("No position are available for elemental result.")
        elif (stress_style == StressStyle.Bending):
            stress_result = reader.GetResult("BENDING_STRESS")
            stress_result.SelectComponents([shell_orientation])
        elif (stress_style == StressStyle.Membrane):
            stress_result = reader.GetResult("MEMBRANE_STRESS")
            stress_result.SelectComponents([shell_orientation])

        for id in collector.Ids:
           values = stress_result.GetElementValues(id)
           collector.SetValues(id,values)

    def hasPosition(self, stressStyle):
        return (stressStyle == StressStyle.TopAndBottom or stressStyle == StressStyle.Top or stressStyle ==

    def stressStyle(self,argument):
        switcher = {
            "Top":StressStyle.Top,
            "Bottom":StressStyle.Bottom,
            "TopAndBottom":StressStyle.TopAndBottom,
            "Membrane":StressStyle.Membrane,
            "Bending":StressStyle.Bending
            }
        return switcher.get(argument, StressStyle.TopAndBottom)

    def shellStressOrientation(self,argument):
        switcher = {
            "Longitudinal":"_LONGITUDINAL",
            "Transverse":"_TRANSVERSE",
            "InPlanShear":"_IN_PLANE_SHEAR"
            }
        return switcher.get(argument, "_LONGITUDINAL")
```

Creating a Custom Result on a Contact

On a contact in the Mechanical tree, you can create an ACT result object to access any of the following results: **`Pressure`**, **`Penetration`**, **`Gap`**, **`Frictional Stress`**, **`Sliding Distance`**, **`Status`**, and **`Fluid Pressure`**. In the figure, the result **`Pressure`** is displayed as a custom result.



A: Static Structural
ContactResult
Expression: RES26 (Unaveraged)
Time: 1
15/09/2017 08:49

🟥	14753 Max
🟧	9425.7
🟨	4098.8
🟩	-1228
🟩	-6554.9
🟩	-11882
🟦	-17209
🟦	-22535
🟦	-27862
🟦	-33189 Min

Details of "ContactResult"

Scope	
Scoping Method	Geometry Selection
Geometry	All Bodies
Definition	
Type	User Defined Result
Expression	= RES26
Input Unit System	Metric (m, kg, N, s, V, A)
Output Unit	
By	Time
Display Time	Last
Coordinate System	Global Coordinate System
Calculate Time History	Yes
Identifier	
Suppressed	No
Integration Point Results	

Tabular Data

	Time [s]	✓ Minimum	✓ Maximum
1	1.	-33189	14753

Creating an Extension for a Custom Result on a Contact

The following XML file adds a toolbar with a single button for creating a custom pressure result for a contact.

```
<extension version="1" minorversion="" name="ACTResults">

   <author>Ansys Inc.</author>
   <description>ACT results</description>
   <guid shortid="ACTResults">23EBAB2B-732D-49E3-91B6-818F8BECF010</guid>
   <script src="main.py" />

   <interface context="Mechanical">
     <images>images</images>
     <toolbar name="ACTResults" caption="ACTResults">
       <entry name="ContactResult" icon="contactActRes">
         <callbacks>
           <onclick>CreateContactResult</onclick>
         </callbacks>
       </entry>
     </toolbar>
   </interface>

   <simdata context="Mechanical">
     <result name="ContactResult" version="1" caption="ContactResult" icon="result"
             type="scalar" location="elemnode" IsContactResult="true" class="ActResController">
       <property name="Geometry" caption="Geometry" control="scoping"></property>
       <property name="Type" caption="Type" control="select" default="Pressure">
         <attributes options="Pressure,Penetration,Gap,Frictional Stress,Sliding Distance,Status,Fluid Pressur
       </property>
       <callbacks>
         <evaluate>EvaluateContact</evaluate>
       </callbacks>
     </result>
   </simdata>

 </extension>
```

In this file, the element **`<script>`** references the IronPython script **`main.py`**. The element **`<interface>`** defines the toolbar and a button. In the child element **`<callbacks>`**, the single callback **`<onclick>`** invokes the function **`EvaluateContact`**, which creates and adds the result to the simulation environment.

The element **`<simdata>`** defines the result.

- The attributes in the child element **`<result>`** provide the name, version, caption, unit, and icon that apply to the result. These properties display in the **Details** view of Mechanical when the result is selected in the **Outline** view. The **Details** view shows each property with the corresponding names and result values.
- In the child element **`<callbacks>`**, the single callback **`<evaluate>`** invokes the function **`EvaluateContact`** when Mechanical requests an evaluation. The output of this function is sent directly to Mechanical, which displays the result.

Defining Functions for a Custom Result on a Contact

The IronPython script **`main.py`** follows. This sample code shows how to create the result object, calculate the result, and set up the result reader. For more information on the result reader, see Retrieving Mechanical Results More Efficiently (p. 38).

```
def CreateContactResult(analysis):
    analysis.CreateResultObject("ContactResult")

def EvaluateContact(result, stepInfo, collector):
    result.Controller.EvaluateContact(result, stepInfo, collector)

class ActResController:

    def __init__(self, api, result):
        pass

    def EvaluateContact(self, result, stepInfo, collector):
        #get the contact result type
        type_value = result.Properties["Type"].Value
        [contact_type,contact_comp] = self.contactType(type_value)

        reader = result.Analysis.GetResultsData()

        #set the result collector with the contact elements
        contElemIds = reader.ContactElementIds
        lengths = reader.NumberValuesByElement(contElemIds)
        collector.SetAllIds(contElemIds, lengths)

        reader.CurrentResultSet = stepInfo.Set
        contact_result = reader.GetResult(contact_type)
        contact_result.SelectComponents([contact_comp])

        values = contact_result.GetElementValues(contElemIds,False)
        collector.SetAllValues(values,lengths)

    def contactType(self,argument):
        switcher = {
            "Pressure":["CONT","PRES"],
            "Penetration":["CONT","PENE"],
            "Gap":["CONT","GAP"],
            "Frictional Stress":["CONT","SFRI"],
            "Sliding Distance":["CONT","SLID"],
            "Status":["CONT","STAT"],
            "Fluid Pressure":["CONT","FPRS"],
            }
        return switcher.get(argument, ["CONT","PRES"])
```

Retrieving Mechanical Results More Efficiently

The following topics provide information about efficiently using mechanical results:

Reading Results in a Result (RST) File

The reading of multiple results in the same RST file will perform poorly if the script is not written cleverly. The RST file reader uses buffers to improve its own performance. When using this reader, you must ensure that your code minimizes unnecessary re-population of these buffers.

The following example demonstrates this problem. In an ACT extension for Mechanical, two results (**“`PRIN_S`”** and **“`EPPLEQV`”**) are accessed from the RST file so that a custom acceptance criterion can be calculated. Initially, the script has only one loop on the elements to read multiple result values in the RST file and to compute the custom acceptance criterion.

```
def evaluate_v1(result,stepInfo, collector):
    #mesure elapse time
    time=System.Diagnostics.Stopwatch()
    time.Start()

    # Reader initialization
    reader = getResultsData(ExtAPI, result.Analysis)
    reader.CurrentResultSet = stepInfo.Set

    # Get the principal stress result from the reader
    stress = reader.GetResult("PRIN_S")
    stress.SelectComponents(["1"])

    # Get element nodal equivalent plastic strain result from the reader
    strain = reader.GetResult("EPPLEQV_RST")

    # loop on the elements and reading of the results values
    # to compute the indicator
    for element in collector.Ids:
        stress_values = stress.GetElementValues(element)
        strain_values = strain.GetElementValues(element)
        ind = list(map(lambda x: x[0] + x[1],zip(stress_values,strain_values)))
        collector.SetValues(element,ind)

    #mesure elapse time
    time.Stop()
    tot = time.Elapsed.TotalSeconds
    ExtAPI.Log.WriteMessage("Evaluate callback Total Time: " + str(tot))
```

Using a benchmark testing model, it takes 7.93 seconds to execute the evaluate callback.

When the script is rewritten properly, three loops are used:

- One loop reads the values in the RST file for the first result
- One loop reads the values in the RST file for the second result

- One loop computes the custom acceptance criterion

```
def evaluate_v2(result,stepInfo, collector):
    #mesure elapse time
    time=System.Diagnostics.Stopwatch()
    time.Start()

    # Reader initialization
    reader = getResultsData(ExtAPI, result.Analysis)
    reader.CurrentResultSet = stepInfo.Set

    # Get the principal stress result from the reader
    stress = reader.GetResult("PRIN_S")
    stress.SelectComponents(["1"])

    # Get element nodal equivalent plastic strain result from the reader
    strain = reader.GetResult("EPPLEQV_RST")

    # Get result values for stress for all elements
    # the loop on the elements is performed in the method GetElementValues
    stress_values = stress.GetElementValues(collector.Ids, False)

    # Get result values for strain for all elements
    # the loop on the elements is performed in the method GetElementValues
    strain_values = strain.GetElementValues(collector.Ids, False)

    # compute the criterior
    # operation on the results arrays
    values = list(map(lambda x: x[0] + x[1], zip(stress_values,strain_values)))

    # set the values in the result collector
    lengths = reader.NumberValuesByElement(collector.Ids)
    collector.SetAllValues(values, lengths)

    #mesure elapse time
    time.Stop()
    tot = time.Elapsed.TotalSeconds
    ExtAPI.Log.WriteMessage("Evaluate callback Total Time: " + str(tot)
```

Using the same benchmarks, it now takes only 0.01 seconds to execute the evaluate callback.

Explanation

At first glance, a single loop to perform the element iteration might seem most efficient. However, performance metrics show that placing each result's access call into its own loop delivers the best performance.

To limit the access to the disk and improve performance, the reader reads a group of lines, filling a buffer in memory, so that the reader can quickly access the next line. In the first script, for each iteration, the reader must read two different results that are in two different places in the RST file.

It takes time for the reader to access the disk and to move from one location to the other. In addition, the reader does not read only the requested data. At each iteration, the reader reads a full buffer of lines and then clears the buffer. It then goes to the new location on the disk and reads again a full buffer. That is not efficient because the reader must read much more data than is needed.

That is why the second script is much better. In the first loop, the reader reads the first result, with all values in the buffer. In the second loop, the reader reads the second result, with all values in the buffer. In the third loop, the reader computes the custom acceptance criterion.

Using the Standalone Result Reader

A beta method exists for more effectively retrieving Mechanical results. This method uses an external executable in addition to the Mechanical process. By using the postprocessing API via this method, you transparently control the instantiation of the results reader and postprocess results without interfering with Mechanical, especially in the case of analysis over several time steps. This method improves postprocessing performance by avoiding unnecessary actions that could be automatically executed in a standard use of Mechanical.

Note:

This method is available only with the Windows platform.

To enable or disable the use of this method, you set the variable **`ExtAPI.UsesStandaloneActResultReaderImplementation`** to true or false. This variable is set to false by default. The sample code that follows shows how to activate and deactivate the external executable:

```
# Activate postprocessing via external executable
ExtAPI.UsesStandaloneActResultReaderImplementation = True

# Read the Stress from the specified result file
rst = r"C:\\test\\file.rst"
reader = ExtAPI.Tools.GetResultsDataFromFile(rst)
nbSteps = reader.ResultSetCount
rs = reader.GetResult("S")
for i in range(nbSteps):
    reader.CurrentResultSet = i+1
    s = rs.GetElementValues(elemIds,False)
reader.Dispose()

# Deactivate postprocessing via external executable
ExtAPI.UsesStandaloneActResultReaderImplementation = False
```

As the result reader is in a separate process, to ensure good performance, it is necessary to use the two methods that follow to get nodal results (p. 41) and elemental results (p. 41).

```
def EvaluateContact(self, result, stepInfo, collector):
     ExtAPI.ActResultReaderUsingMeshDataFromRst = True
     reader = result.Analysis.GetResultsData()
     mesh= reader.CreateMeshData()
```

Managing Degenerate Elements

You use the method **`Ansys.ACT.Common.Post.CreateMeshData`** to create a **`MeshData`** object. To manage degenerate elements in the ACT postprocessing, the method **`CreateMeshData`** has to be called to build the needed mesh data structure:

```
def EvaluateContact(self, result, stepInfo, collector):
        ExtAPI.ActResultReaderUsingMeshDataFromRst = True
   reader = result.Analysis.GetResultsData()
   mesh= reader.CreateMeshData()
```

Retrieving Nodal Results

To retrieve nodal results efficiently, you must use one call to get the results for a set of nodes. The method **`GetNodeValues`** takes as arguments an array of node indices and returns an array with the values of each selected component of the required result at each node sequentially.

Declaration syntax:

```
public double[] GetNodeValues(int[] nodeIds)
```

Where **`nodeIDs`** is the array of integers containing the list of the node indices for which result values are required.

Example code:

```
ExtAPI.ActResultReaderUsingMeshDataFromRst = False
reader=ExtAPI.DataModel.AnalysisList[0].GetResultsData()
mesh=ExtAPI.DataModel.MeshDataByName("Global")
nodeIds = mesh.NodeIds
ru=reader.GetResult("U")
u = ru.GetNodeValues(nodeIds)
> nodeIds = [ 1, 2, …]
> u = [ U1X, U1Y, U1Z, U2X, U2Y, U2Z, …]
```

Retrieving Elemental Results

To retrieve element results efficiently, you must use one call to get the results for a set of elements. The method **`GetElementValues`** takes as arguments an array of element indices and returns an array with the values of each selected component of the required result for each element sequentially.

Declaration syntax:

```
public double[] GetElementValues(int[] elementIds, boolean computeMidSideNodes)
```

Where:

- **`elementIds`** is the array of integers containing the list of element indices for which result values are required.
- **`computeMidSideNodes`** indicates if the method is to return only values at corner nodes or values at both corner nodes and midside nodes. When set to false, only values at corner nodes are returned. When set to true, values at both corner nodes and midside nodes are returned.

Example code:

```
reader=ExtAPI.DataModel.AnalysisList[0].GetResultsData()
# Retrieve the mesh in Mechanical
mesh=ExtAPI.DataModel.MeshDataByName("Global")
elemIds = mesh.ElementIds
re=reader.GetResult("S")
s = re.GetElementValues(elemIds,False)
>elemIds = [1, 2, …]
>s = [ S11X, S11Y, S12X, S12Y, S13X, S13Y, S21X, S21Y, …]
```

In this example, each element has three nodes. The stress values are located at the nodes of the elements. Thus, **`SijX`** is the value of component X for the stress result at node **`j`** of element **`i`**.

Note:

In the case of a 2D geometry, the result array returned by the method **`GetNodeValues`** or **`GetElementValues`** stays dimensioned as for a 3D geometry. However, the values for the third dimension are dummy values.

For a shell element, the method **`GetElementValues`** returns the values for the three positions in this order:

- Bottom
- Top
- Middle

Example code follows:

```
elemIds = mesh.ElementIds
re=reader.GetResult("S")
re.SelectComponents([“X”])
s = re.GetElementValues(elemIds,False)
>elemIds = [1, 2, …]
>s = [ S11X_bottom, S12X_bottom, S13X_bottom,
       S11X_top, S12X_top, S13X_top,
       S11X_middle, S12X_middle, S13X_middle,
       S21X_bottom, S22X_bottom, S23X_bottom, …]
```

Third-Party Solver Connections in Mechanical

The supplied extension `ExtSolver1` shows how to add a connection to a third-party solver, which allows an external solver to be started from the Mechanical system rather than the ANSYS solver. This example demonstrates the ability to connect to a very simple solver that distributes the values assigned at boundaries inside the structure:

Note:

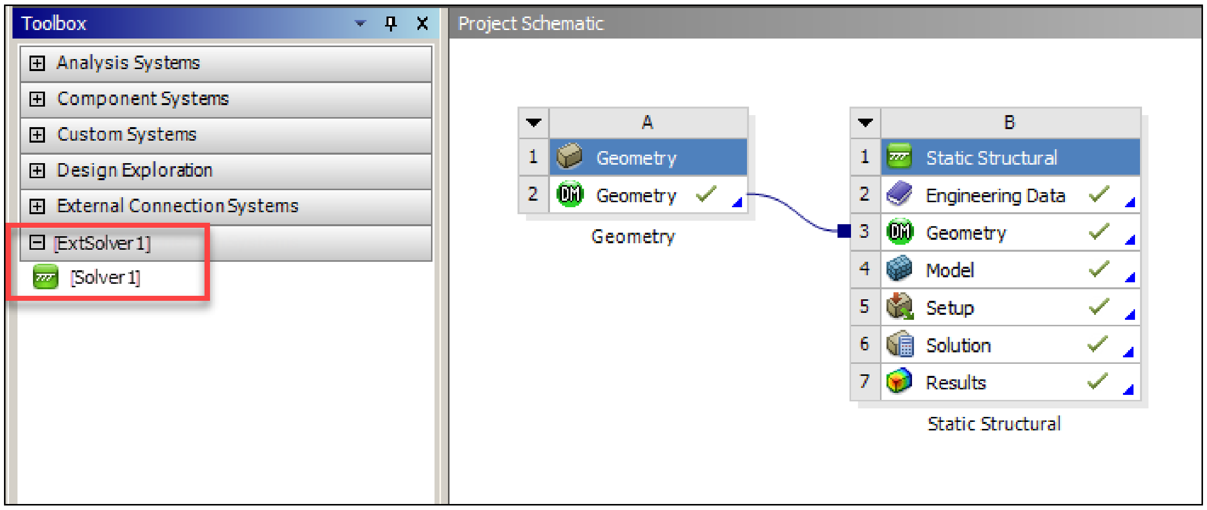
Although the supplied extension **`DemonstratationSolver`** is not described, it also shows how to add a connection to a third-part solver. This newer example is perhaps more reliable.

Creating the Extension for Connecting to a Third-Party Solver

The file `ExtSolver1.xml` follows.

```
 <extension version="1" name="ExtSolver1">

  <script src="[ext.Folder]\main.py" />

  <interface context="Mechanical">

    <images>[ext.Folder]\images</images>

    <callbacks>
      <onload>Load</onload>
      <onsave>Save</onsave>
    </callbacks>

    <toolbar name="ExtSolver1" caption="ExtSolver1">
      <entry name="Values Load" icon="tload">
        <callbacks>
          <onclick>CreateValuesLoad</onclick>
        </callbacks>
      </entry>
    </toolbar>

  </interface>

  <simdata context="Project|Mechanical">

    <solver name="Solver1" version="1" caption="Solver1" icon="result" analysis="Static" physics="Structural">

      <callbacks>
        <onsolve>Solve</onsolve>
        <getsteps>GetSteps</getsteps>
        <getreader>GetReader</getreader>
      </callbacks>

      <property name="MaxIter" caption="Max. Iterations" control="integer" default="10" />

    </solver>

  </simdata>

  <simdata context="Mechanical">

    <load name="Values" version="1" caption="Values" icon="tload" issupport="false" isload="true" color="#0000FF

      <callbacks>
        <getsolvecommands>WriteInitialValues</getsolvecommands>
        <getnodalvaluesfordisplay>NodeValues</getnodalvaluesfordisplay>
      </callbacks>

      <property name="Geometry" control="scoping" />
      <property name="Expression" caption="Expression" control="text" />

    </load>

  </simdata>

</extension>
```

In this file, the element **`<simdata>`** has child elements that define the solver. The attribute **`context`** is **`Project|Mechanical`**. When **`Project`** is specified, the solver appears as a system in the Mechanical **Toolbox**.



Note:

Custom solver-based systems support general data transfer to their **Model**, **Setup**, and **Solution** tasks. For an explanation of general data transfer, see Defining Task-Level Data Transfer in the ACT Customization Guide for Mechanical.

The child element `<solver>` has some mandatory attributes:

- `name`: Internal name of the solver
- `version`: Version of the solver
- `caption`: Display name of the solver
- `analysis`: Analysis type addressed by the solver. For compatibility reasons, this attribute must be set to `Static`, but this does not prevent you from integrating any type of third-party solver.
- `physics`: Physics type addressed by the solver. This attribute must be set to `Structural`, even though it has no real impact on what the solver computes.

You must define the callback `<onsolve>`. This callback is invoked when the product starts the solver, thereby taking the object `solver` as an argument.

You can define a set of properties, which appear in the **Details** view of the analysis. In this example, the property `MaxIter` is created. It is an integer value for defining the maximum number of iterations that the solver is to perform. The default value is set to 10.

As shown in the earlier figure, a new system for the third-party solver `ExtSolver1` is added to the **Toolbox**. Each third-party solver is added into a new category, identified by the caption of the extension. The system is named by the caption of the solver.

The system related to the third-party solver is equivalent to one standard system that can be created with the ANSYS solver. The components that build this new system remain the same. You can add the new system related to the solver to the **Project Schematic** just as you do for any other system.

Defining Functions for Connecting to a Third-Party Solver

The script `main.py` follows.

```
import os
import solver

def CreateValuesLoad(analysis):
    analysis.CreateLoadObject("Values")

initValues = {}
sol = None
solbystep = SerializableDictionary[int,dict]()
values = {}
steps = []
res = [0.]
dScal = [0.]
dVec = [0., 0., 0.]

def WriteInitialValues(load,filename):
    global initValues
    propEx = load.Properties["Expression"]
    exp = propEx.Value
    if exp=="":
        return None
    vexp = compile(exp,'','eval')
    values = []
    propGeo = load.Properties["Geometry"]
    refIds = propGeo.Value.Ids
    mesh = load.Analysis.MeshData
    for refId in refIds:
        meshRegion = mesh.MeshRegionById(refId)
        nodeIds = meshRegion.NodeIds
        for nodeId in nodeIds:
            node = mesh.NodeById(nodeId)
            x = node.X
            y = node.Y
            z = node.Z
            v = 0.
            try:
                v = eval(vexp)
                v = float(v)
            finally:
                initValues.Add(nodeId,v)

def NodeValues(load,nodeIds):
    propEx = load.Properties["Expression"]
    exp = propEx.Value
    if exp=="":
        return None
    try:
        vexp = compile(exp,'','eval')
    except:
        return None
    values = []
    mesh = load.Analysis.MeshData
    for id in nodeIds:
        node = mesh.NodeById(id)
        x = node.X
        y = node.Y
        z = node.Z
        v = 0.
        try:
            v = eval(vexp)
            v = float(v)
        finally:
            values.Add(v)
    return values
```

```
def Solve(s):
    global steps
    global initValues
    global sol
    global solbystep
    global values

    solbystep = SerializableDictionary[int,dict]()
    solbystepTmp = {}

    f = open("solve.out","w")
    f.write("SolverEngine version 1.0\n\n\n")
    try:

        maxIter = int(s.Properties["MaxIter"].Value)
        f.write("Max. iteration : %d\n" % (maxIter))

        mesh = s.Analysis.MeshData

        numEdges = 0
        geo = ExtAPI.DataModel.GeoData
        nodeIds = []
        for asm in geo.Assemblies:
            for part in asm.Parts:
                for body in part.Bodies:
                    for edge in body.Edges:
                        numEdges = numEdges + 1
                        ids = mesh.MeshRegionById(edge.Id).NodeIds
                        nodeIds.extend(ids)
        steps = []
        stepsTmp = []
        f.write("Num. edges : %d\n" % (numEdges))
        sol = solver.SolverEngine(mesh,initValues,nodeIds)
        initValues = sol.Run(maxIter,f,stepsTmp,solbystepTmp)
        nodeIds = mesh.NodeIds
        sol = solver.SolverEngine(mesh,initValues,nodeIds)
        values = sol.Run(maxIter,f,steps,solbystep)

        initValues = {}

    except StandardError, e:
        f.write("Error:\n");
        f.write(e.message+"\n");
        f.close()
        return False

    f.close()

    return True

def GetSteps(solver):
    global steps
    return steps


def Save(folder):
    global solbystep
    fm = System.Runtime.Serialization.Formatters.Binary.BinaryFormatter()
    try:
        stream = System.IO.StreamWriter(os.path.join(folder,"sol.res"),False)
    except:
        return
    try:
        fm.Serialize(stream.BaseStream,solbystep)
    finally:
        stream.Close()

def Load(folder):
    global solbystep
    if folder==None:
        return
    fm = System.Runtime.Serialization.Formatters.Binary.BinaryFormatter()
    try:
        stream = System.IO.StreamReader(os.path.join(folder,"sol.res"))
    except:
        return
    try:
        solbystep = fm.Deserialize(stream.BaseStream)
    finally:
        stream.Close()

class ExtSolver1Reader(ResultReaderBase):
    def __init__(self,infos):
        self.infos = infos
        self.step = 1

    def get_CurrentStep(self):
        return self.step

    def set_CurrentStep(self,step):
        self.step = step

    def StepValues(self):
        global steps
        return steps

    def ResultNames(self):
        return ["U","VALUES"]

    def GetResultLocation(self,resultName):
        return "node"

    def GetResultType(self,resultName):
        if resultName=="U":
            return "vector"
        return "scalar"

    def ComponentNames(self,resultName):
        if resultName=="U":
            return ["X","Y","Z"]
        else:
            return ["VALUES"]

    def ComponentUnit(self,resultName,componentName):
        if resultName=="U":
            return "Length"
        return "Temperature"

    def GetValues(self,resultName,entityId):
        global values
        global solbystep
        global dVec
        global dScal
        if resultName=="U":
            values = solbystep[self.step]
            dVec[0] = 0.
            dVec[1] = 0.
            dVec[2] = 0.
            try:
                dVec[0] = values[entityId]
            finally:
                return dVec
        else:
            values = solbystep[self.step]
            try:
                dScal[0] = values[entityId]
                return dScal
            except:
                return None

def GetReader(solver):
    return ["ExtSolver1Reader"]
```

The second line of the script `main.py` is **`import solver`**. The solver code is located in a separate script, `solver.py`, which is placed in the same folder as `main.py`. This technique of importing another script supports reuse and maintainability. The script in `solver.py` defines the class **`SolverEngine`**.

The function **`Solve`** is associated with the callback **`<onsolve>`**. This function creates a file `solve.out`, which is read interactively by the product and stored in the solution information.

By default, the product starts the resolution directly in the working directory, so it is not necessary to set the folder in which the file `solve.out` must be created.

The callback function must return **`True`** or **`False`** to specify if the solve has succeeded or failed.

Currently, it is not possible to return progress information.

Creating a Result Reader to Expose Results

To create a result reader to expose results, you can create a class that implements the interface **`ICustomResultReader`**. The following methods need to be implemented. For each method, the expected results that must be returned are described.

`GetCurrentStep(self)`
This method must return the current step number.

`SetCurrentStep(self,stepInfo)`
This method is called each time the current step number is changed.

`GetStepValues(self)`
This method must return a lost of double values that represents the time steps or frequencies.

`GetResultNames(self)`
This method must return a list of strings that represents the result names available for the reader.

`GetResultLocation(self,resultName)`
This method must return the location type of the result identified by the name **`resultName`**. The possible values are **`node`**, **`element`**, and **`elemnode`**.

`GetResultType(self,resultName)`
This method must return the type of the result identified by the name **`resultName`**. The possible values are **`scalar`**, **`vector`**, and **`tensor`**.

`GetComponentNames(self,resultName)`
This method must return a list of strings that represents the list of available components available for the result identified by the name **`resultName`**.

`GetComponentUnit(self,resultName,componentName)`
This method must return the unit name related to the result's component identified by the result name **`resultName`** and the component name **`componentName`**.

`GetValues(self,resultName,collector)`
This method must return a list of double values for each component associated with the result identified by the name **`resultName`**.

To specify a dedicated reader, add the callback **`<getreader>`** in the solver definition. This callback returns a list of strings, where the first is the name of the reader class and the remainder represents parameters given to the constructor of the class when the reader is instanced by ACT. Any result exposed in the method **`ResultNames()`** is available in the **Results** worksheet in Mechanical.

If the name of the result matches a standard result name (like **`U`**), Workbench applies the same treatment for this result as it does for a standard one. So, if the result is named **`U`**, Workbench uses this result to draw the deformed model.

In the extension **`Extsolver1`**, the reader declares one single scalar nodal result **`VALUES`**. No deformation is considered.

Enabling and Disabling the Reading of Meshing Data Inside the RST File

The following field enables and disables the reading of the meshing data inside the RST file:

`Ansys.ACT.Mechanical.ActResultReaderUsingMeshDataFromRst = True / False`

For some results, meshing data contained inside the RST file is needed for postprocessing. This is the case when the results refer to the entity generated during the solve, such as for contact results.

If the solve is done by a third-party solver rather than the standard ANSYS Mechanical solver, the RST file will not contain meshing data. In this case, the variable **`ActResultReaderUsingMeshDataFromRst`** must be set to false:

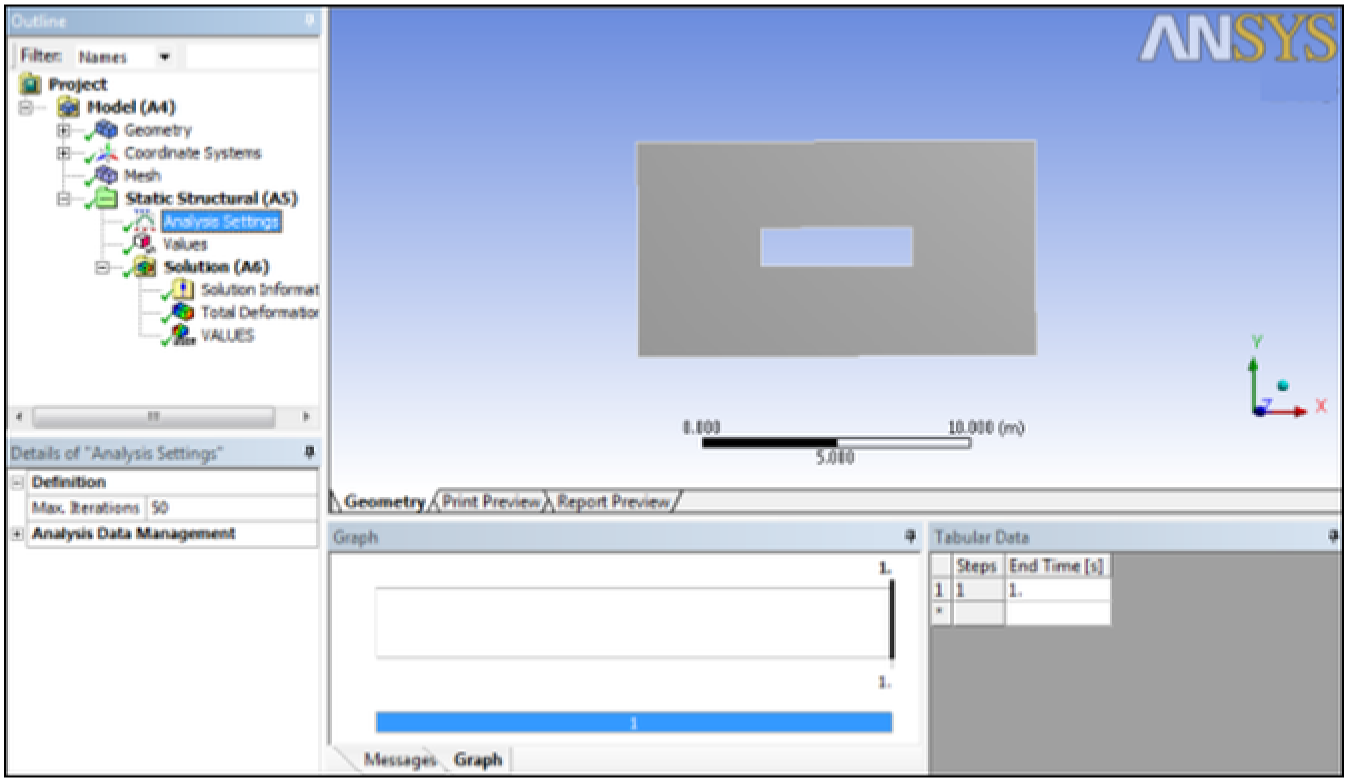
```
def EvaluateResult(self, result, stepInfo, collector):
        ExtAPI.ActResultReaderUsingMeshDataFromRst = False;
      reader = result.Analysis.GetResultsData()
```

Loading and Saving Third-Party Solver Data

You can use the callbacks **`<onload>`** and **`<onsave>`** in the element **`<interface>`** to load and save data associated with the third-party solver. As previously discussed, the extension **`Extsolver1`** uses these callbacks to save and load the computed results. This means that the results are still available if the system **Solver1** is closed and reopened.

The callbacks **`<onload>`** and **`<onsave>`** take in argument the name of the working directory.

The following figure shows the analysis settings object associated with the external solver.



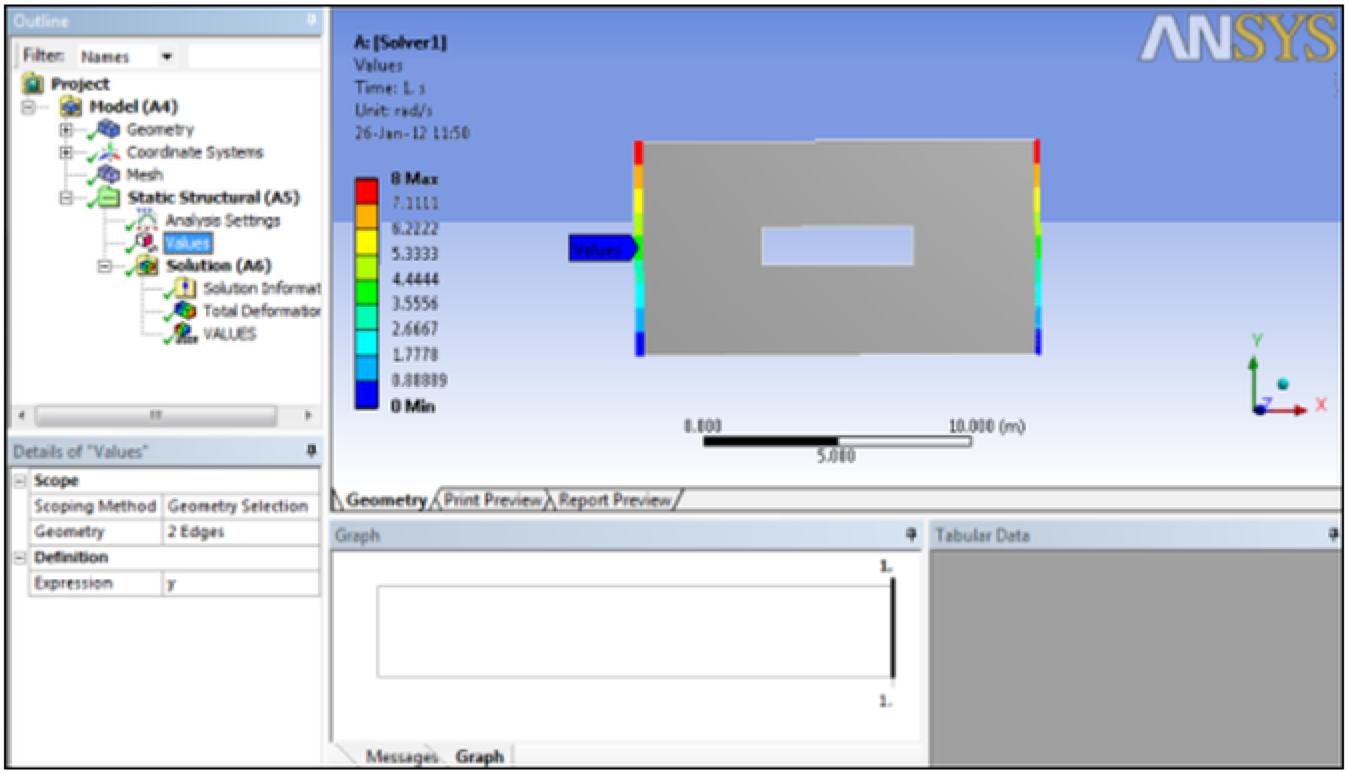
Details of "Analysis Settings"

Definition	
Max. Iterations	50
Analysis Data Management	

Tabular Data

	Steps	End Time [s]
1	1	1.
*		

The next figure shows the boundary condition object associated with the external solver.



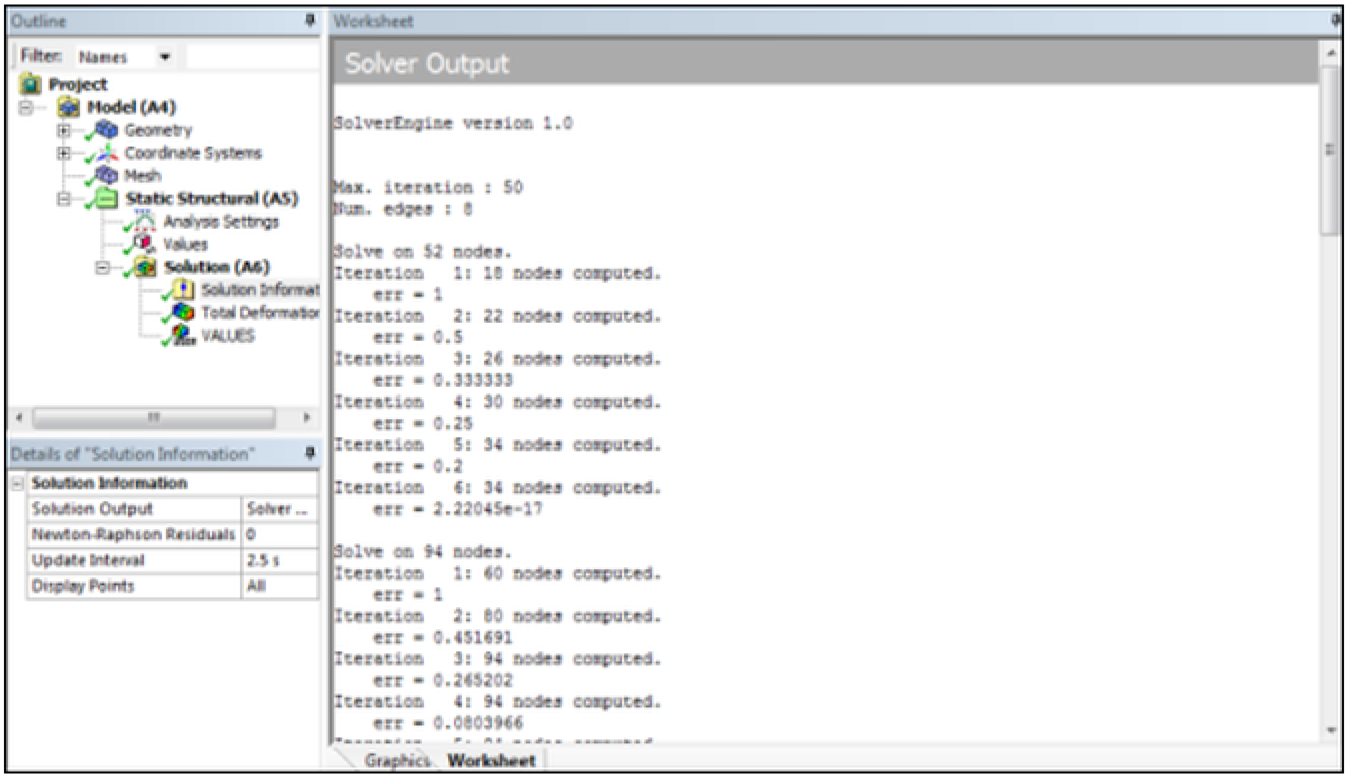
A: [Solver1]
Values
Time: 1. s
Unit: rad/s
26-Jan-12 11:50

🟥	8 Max
🟧	7.1111
🟨	6.2222
🟨	5.3333
🟩	4.4444
🟩	3.5556
🟩	2.6667
🟦	1.7778
🟦	0.88889
🟦	0 Min

Details of "Values"

Scope	
Scoping Method	Geometry Selection
Geometry	2 Edges
Definition	
Expression	y

During the resolution, solution information is interactively displayed. Two types of output are derived from the resolution. The following figure shows the solution information associated with the external solver.



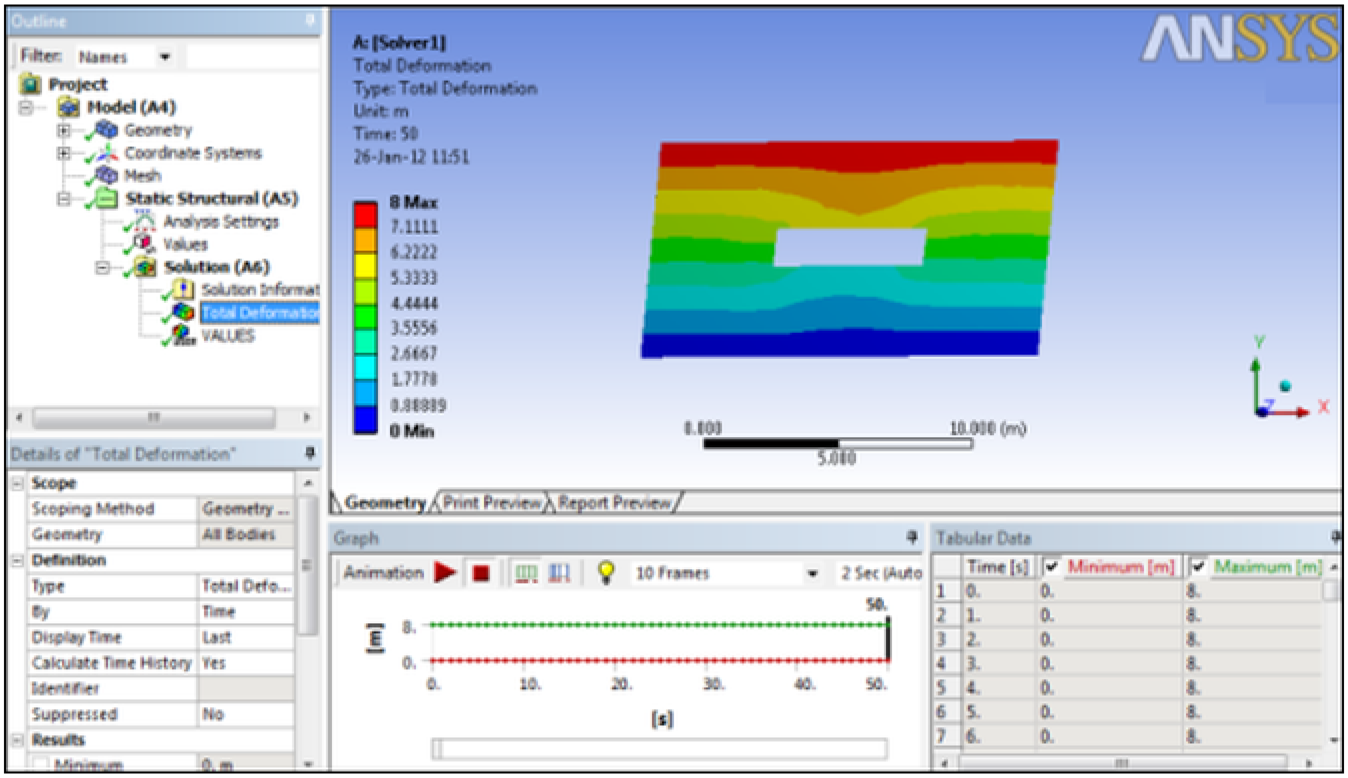
Details of "Solution Information"

Solution Information	
Solution Output	Solver ...
Newton-Raphson Residuals	0
Update Interval	2.5 s
Display Points	All

Solver Output

```
SolverEngine version 1.0


Max. iteration : 50
Num. edges : 8

Solve on 52 nodes.
Iteration   1: 18 nodes computed.
    err = 1
Iteration   2: 22 nodes computed.
    err = 0.5
Iteration   3: 26 nodes computed.
    err = 0.333333
Iteration   4: 30 nodes computed.
    err = 0.25
Iteration   5: 34 nodes computed.
    err = 0.2
Iteration   6: 34 nodes computed.
    err = 2.22045e-17

Solve on 94 nodes.
Iteration   1: 60 nodes computed.
    err = 1
Iteration   2: 80 nodes computed.
    err = 0.451691
Iteration   3: 94 nodes computed.
    err = 0.265202
Iteration   4: 94 nodes computed.
    err = 0.0803966
```

The next figure shows the postprocessing associated with the external solver.



Details of "Total Deformation"

Scope	
Scoping Method	Geometry ...
Geometry	All Bodies
Definition	
Type	Total Defo...
By	Time
Display Time	Last
Calculate Time History	Yes
Identifier	
Suppressed	No
Results	
Minimum	0. m

Tabular Data

	Time [s]	✓ Minimum [m]	✓ Maximum [m]
1	0.	0.	8.
2	1.	0.	8.
3	2.	0.	8.
4	3.	0.	8.
5	4.	0.	8.
6	5.	0.	8.
7	6.	0.	8.

Creating a Custom Solver that Can Bypass the Function Evaluation of Properties on Native Objects

An attribute on the solver tag allows you to define solvers that can bypass the function evaluation of properties on native objects. To define such a solver, you set the attribute **`evaluateFunctionLoads`** to false:

```
<solver name="solverBypass" version="1" caption="Solver ByPass" icon="result" analysis="Static" physics="Structu
```

When the attribute **`evaluateFunctionLoads`** is set to false, no tabular data or graph is shown. Thus, whatever the string content, the function expression will always be valid.

If this attribute is not set, it defaults to true.

Tip:

In **`Template13-SolverBypass`**, you can see an example where this attribute is implemented. To download all Mechanical templates, on the App Developer Resources page, click the **ACT Templates** tab and then click **ACT Templates for Mechanical**.

Additional Methods and Callbacks

The following topics describe additional methods and callbacks of particular interest for Mechanical extensions:

Creating Results with Imaginary Parts

Responding to a Change to the Active Unit System

Localizing Extensions and Wizards

Creating Results with Imaginary Parts

Creating results for analyses that have complex results requires managing both real and imaginary values. You use the method **`SetValues()`** to set values to the real part of the result. You use the method **`SetImaginaryValues()`** to set values to the imaginary part of the result.

IronPython code follows for the creation of a complex result:

```
def Evaluate(result,stepInfo,collector):

    for id in collectors.Ids:
        real_value = 1.
        # Set the real part of the result
        collector.SetValues(id, real_value)

        imaginary_value = 2.
        # Set the imaginary part of the result
        collector.SetImaginaryValues(id, imaginary_value)
```

Responding to a Change to the Active Unit System

When the active units system is changed in Mechanical, values in the extension might need to be converted accordingly. You can register a callback function that is invoked when such a change is made. In the following XML file for the theoretical extension **`UnitsChanged`**, the callback **`<onunitschanged>`** invokes the IronPython function **`unitschanged`**.

```
<extension version="1" name="UnitsChanged">
  <author>ANSYS, Inc.</author>
  <guid shortid="UnitsChanged">D4C1ED2D-5104-4507-B078-5AD95B712DF1</guid>
  <script src="rununitschanged.py" />
  <interface context="Mechanical">
    <images>images</images>
    <callbacks>
      <oninit>init</oninit>
      <onunitschanged>unitschanged</onunitschanged>
    </callbacks>
…
  </interface>
</extension>
```

The code follows for the function **`unitschanged`**.

```
clr.AddReference("Ans.UI.Toolkit.Base")
clr.AddReference("Ans.UI.Toolkit")
from Ansys.UI.Toolkit import *
import System

def init(context):
    ExtAPI.Log.WriteMessage("Init ACT unitschanged example...")
def unitschanged():
     ExtAPI.Log.WriteMessage("***** Units Changed *****")
```

Localizing Extensions and Wizards

For Mechanical-based extensions and wizard, you have two ways of localizing captions for buttons, tabs, groups, object names, and so on:

- callback **`<getlocalstring>`**
- attribute **`localize`**

Callback <getlocalstring>

The callback **`<getlocalstring>`** is exposed on the interface level, providing a way for you to localize captions yourself. This callback has two parameters:

- **`locale`**, which is the language for the caption
- **`caption`**, which is the display text for the caption

The **`locale`** is passed in as a string using these language values:

Language	Value
English	en-us
French	fr

Language	Value
German	de
Japanese	ja
Chinese	zh

An example follows for using the callback **<getlocalstring>**.

XML Definition:

```
<interface>
    <toolbar name="Toolbar 1" caption="Toolbar">
            <entry name="custom_result_1" icon="hand" caption="Result">
                <callbacks>
                    <onclick>add_result1</onclick>
                </callbacks>
            </entry>
    <toolbar>

    <callbacks>
        <getlocalstring>GetLocalString</getlocalstring>
    </callbacks>
</interface>
```

IronPython Script:

```
localized_strings_french = {
    "Toolbar" : "Barre d'outil",
    "Result" : "Résultat"
}

def GetLocalString(caption, locale):
    if locale == "fr":
        return localized_string_french[caption]
```

Note:

Because the callback **<getlocalstring>** can be called multiple times, you should define it in a way similar to the example.

Attribute localize

The attribute **localize** provides a way of passing the work of localizing to Mechanical. When **localize** is set to true, Mechanical's string table is used to localize the caption. This means that for Mechanical to be able to localize the caption, the caption must be a string ID retrieved from the file `dsstringtable.xml`.

The file `dsstringtable.xml` is a mapping from string IDs to localized strings. Because five languages are supported, there are five different versions of this file in the installation. For a given language, the file contains all strings that are used as captions in Mechanical, thereby providing an easy way of localizing extension captions if these same captions appear in Mechanical's user interface. If you use a string that is not present in `dsstringtable.xml`, the string is shown without any localization.

The attribute **localize** is available on the following tags: **<action>**, **<entry>**, **<toolbar>**, **<property>**, **<propertygroup>**, **<object>**, **<load>**, **<solver>**, **<result>**, and **<step>**.

The string table file is located in `*ANSYS_INSTALL*/aisol/DesignSpace/DSPages/Language/*LOCALE*/xml/dsstringtable.xml`.

An example follows for using **`localize="true"`**.

```
<property name="Mag" caption="ID_MagnitudeProperty" localize="true">
</property>
```

Mechanical Wizards

You can use ACT to create target product wizards for Mechanical. The supplied extension **`WizardDemos`** contains a project wizard, multiple target product wizards, and a mixed wizard.

This section describes the target product wizard for Mechanical. Named **`SimpleAnalysis`**, this wizard performs a simple analysis on the bridge built by either the DesignModeler or SpaceClaim wizard **`CreateBridge`**. These wizards are described in the ACT customization guides for DesignModeler and SpaceClaim.

The following topics describe the wizard **`SimpleAnalysis`**:

Mechanical Wizard Definition

Mechanical Wizard Function Definition

Note:

You use the Extension Manager to install and load extensions and the Wizards launcher to start a target product wizard.

Tip:

Included in the supplied package ACT Wizard Templates is the extension **`Template-MechanicalWizard`**. It contains a target product wizard for Mechanical that creates the mesh for the geometry, adds boundary conditions, and then solves and displays a result. For download information, see Extension and Template Examples.

Mechanical Wizard Definition

The file `WizardDemo.xml` follows.

An excerpt from the file `WizardDemos.xml` follows. Code is omitted for the element **`<uidefinition>`** and all wizards other than the Mechanical wizard **`SimpleAnalysis`**.

```
<extension version="2" minorversion="1" name="WizardDemos">
 <guid shortid="WizardDemos">7fdb141e-3383-433a-a5af-32cb19971771</guid>
 <author>ANSYS Inc.</author>
 <description>Simple extension to test wizards in different contexts.</description>

 <script src="main.py" />
 <script src="ds.py" />
 <script src="dm.py" />
 <script src="sc.py" />

 <interface context="Project|Mechanical|SpaceClaim">
  <images>images</images>
 </interface>
```

```
<interface context="DesignModeler">
 <images>images</images>

 <toolbar name="Deck" caption="Deck">
  <entry name="Deck" icon="deck">
   <callbacks>
    <onclick>CreateDeck</onclick>
   </callbacks>
  </entry>
  <entry name="Support" icon="Support">
   <callbacks>
    <onclick>CreateSupport</onclick>
   </callbacks>
  </entry>
 </toolbar>

</interface>

...

<wizard name="SimpleAnalysis" version="1" context="Mechanical" icon="wizard_icon">
 <description>Simple wizard to illustrate how to setup, solve and analyse results of a simulation process.</descr

 <step name="Mesh" caption="Mesh" version="1" HelpFile="help/ds1.html">
  <description>Setup some mesh controls.</description>

  <callbacks>
   <onreset>RemoveControls</onreset>
   <onupdate>CreateMeshControls</onupdate>
  </callbacks>

  <propertygroup display="caption" name="Sizing" caption="Mesh Sizing" >
   <property name="Location" caption="Edge Location" control="geometry_selection">
    <attributes selection_filter="edge" />
    <callbacks>
     <isvalid>IsLocationValid</isvalid>
    </callbacks>
   </property>
   <property name="Ndiv" caption="Divisions" control="integer" />
  </propertygroup>

 </step>

 <step name="Solution" caption="Solution" version="1" HelpFile="help/ds2.html">
  <description>Setup loads.</description>

  <callbacks>
   <onrefresh>RefreshLoads</onrefresh>
   <onreset>RemoveLoads</onreset>
   <onupdate>CreateLoads</onupdate>
  </callbacks>

  <propertygroup display="caption" name="Mesh" caption="Mesh Statistics" >
   <property name="Nodes" caption="Nodes" control="text" readonly="true" />
   <property name="Elements" caption="Elements" control="text" readonly="true" />
  </propertygroup>
  <propertygroup display="caption" name="FixedSupport" caption="Fixed Support" >
   <property name="Location" caption="Face Location" control="geometry_selection">
    <attributes selection_filter="face" />
    <callbacks>
     <isvalid>IsLocationFSValid</isvalid>
    </callbacks>
   </property>
  </propertygroup>

 </step>

 <step name="Results" caption="Results" version="1" HelpFile="help/ds3.html">
  <description>View Results.</description>

  <callbacks>
```

```
     <onrefresh>RefreshResults</onrefresh>
    </callbacks>

    <property name="Res" caption="Deformation" control="text" readonly="true" />

   </step>

  </wizard>

  ...

</extension>
```

Wizard Interface Definition
The element **`<interface>`** defines two user interfaces for the extension **`WizardDemos`**. The first element **`<interface>`** is used by the Mechanical wizard **`SimpleAnalsyis`**.

Wizard Definition
The element **`<wizard>`** named **`SimpleAnalsyis`** has the attribute **`context`** set to **`Mechanical`** to indicate that this is the product in which the wizard executes.

Step Definition
The element **`<step>`** defines a step in the wizard. This extension has three steps: **`Mesh`**, **`Solution`**, and **`Results`**.

- For the step **`Mesh`**, the callback **`<onreset>`** executes the function **`RemoveControls`** to clear existing mesh controls. The callback **`<onupdate>`** executes the function **`CreateMeshControls`** to create new mesh controls.
- For the step **`Solution`**:
  - The callback **`<onrefresh>`** executes the function **`RefreshLoads`** to initialize various properties, including the number of nodes, number of elements computed in the previous step, and so on.
  - The callback **`<onreset>`** executes the function **`RemoveLoads`** to clear loads.
  - The callback **`<onupdate>`** executes the function **`CreateLoads`** to create new loads, create new results, and perform the solve.
- For the step **`Results`**, the callback **`<onrefresh>`** executes the function **`RefreshResults`** to fill the property value associated with the result of the computation (Maximum of Total Deformation).

Mechanical Wizard Function Definition

The IronPython script `ds.py` follows. This script defines all functions executed by the callbacks in the steps for the Mechanical wizard **`SimpleAnalsysis`**.

```
ef IsLocationValid(step, prop):
    if prop.Value==None:
        return False
    if prop.Value.Ids.Count!=1:
        prop.StateMessage = "Select only one edge."
        return False
    return True

def CreateMeshControls(step):
    model = ExtAPI.DataModel.Project.Model
    mesh = model.Mesh
    sizing = mesh.AddSizing()
    sel = step.Properties["Sizing/Location"].Value
    entity = ExtAPI.DataModel.GeoData.GeoEntityById(sel.Ids[0])
    len = entity.Length
    ids = []
    for part in ExtAPI.DataModel.GeoData.Assemblies[0].Parts:
        for body in part.Bodies:
            for edge in  body.Edges:
                if abs(edge.Length-len)/len<1.e-6:
                    ids.Add(edge.Id)
    sel = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
    sel.Ids = ids
    sizing.Location = sel
    sizing.Type = SizingType.NumberOfDivisions
    sizing.NumberOfDivisions = step.Properties["Sizing/Ndiv"].Value
    step.Attributes.SetValue("sizing", sizing)
    mesh.GenerateMesh()

def RemoveControls(step):
    sizing = step.Attributes["sizing"]
    sizing.Delete()

def IsLocationFSValid(step, prop):
    if prop.Value==None:
        return False
    if prop.Value.Ids.Count!=1:
        prop.StateMessage = "Select only one face."
        return False
    return True

def RefreshLoads(step):
    model = ExtAPI.DataModel.Project.Model
    step.Properties["Mesh/Nodes"].Value = model.Mesh.Nodes.ToString()
    step.Properties["Mesh/Elements"].Value = model.Mesh.Elements.ToString()
    panel = step.UserInterface.GetComponent("Properties")
    panel.UpdateData()
    panel.Refresh()

def CreateLoads(step):
    model = ExtAPI.DataModel.Project.Model
    analysis = model.Analyses[0]
    support = analysis.AddFixedSupport()
    sel = step.Properties["FixedSupport/Location"].Value
    entity = ExtAPI.DataModel.GeoData.GeoEntityById(sel.Ids[0])
    area = entity.Area
    ids = []
    for part in ExtAPI.DataModel.GeoData.Assemblies[0].Parts:
        for body in part.Bodies:
            for face in  body.Faces:
                if abs(face.Area-area)/area<1.e-6:
                    ids.Add(face.Id)
    sel = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
    sel.Ids = ids
    support.Location = sel
    loads = []
    loads.Add(support)
    step.Attributes.SetValue("loads", loads)

    loads.Add(analysis.AddEarthGravity())
    res = analysis.Solution.AddTotalDeformation()
    step.Attributes.SetValue("res", res)
    loads.Add(res)
    analysis.Solve(True)
    ExtAPI.Extension.SetAttributeValueWithSync("result", res.Maximum.ToString())

def RemoveLoads(step):
    loads = step.Attributes["loads"]
    for load in loads:
        load.Delete()

def RefreshResults(step):
    model = ExtAPI.DataModel.Project.Model
    res = step.PreviousStep.Attributes["res"]
    step.Properties["Res"].Value = res.Maximum.ToString()
    panel = step.UserInterface.GetComponent("Properties")
    panel.UpdateData()
    panel.Refresh()
```