Applied Programming Technology, Inc.
Bloomsburg, PA 17815

SNAP

Symbolic Nuclear Analysis Package

Python Directed Job Stream Tutorial

November 2020

Table of Contents

1. Introduction 1
   1.1. Definitions 1
   1.2. Finding More Information 2
   1.3. Setup 2
2. Exercise 1 – Simple Python Directed Job Stream 3
3. Exercise 2 – SNAP Variables and Optimization 7
4. Exercise 3 – Keywords and File Search 11
5. Exercise 4 – Bundled Files 16
6. Exercise 5 – TRACE-PARCS Coupled Job Stream 21
7. Exercise 6 – Uncertainty Quantification 24

1. Introduction

This document introduces the Python Directed Job Streams added in SNAP version 3.1.1. This update allows Python scripts to control analysis codes execution in a Job Stream, and includes the ability to call the Model Editor as a library to work with input models during execution.

1.1.Definitions

This section includes a definition of commonly used terms that will appear throughout this document.

- **Job Stream**
  A SNAP Job Stream is a user defined sequence of external application executions, where the outputs from applications may be passed on as the input to additional applications. Locally accessible files may be included as inputs to job streams, in addition to the
- **Calculation Server**
  The Calculation Server, included with SNAP, acts as a single node cluster for executing analysis codes and managing their inputs and outputs. The Calculation Server communicates with Job Status and the Model Editor to provide feedback to users as to the status of executed tasks.
- **Stream Manager**
  The Stream Manager processes Job Streams and marshals tasks for execution. One Stream Manager is launched to manage each job stream.
- **Python Director**
  The Python Director is the portion of the Stream Manager that executes Python directed streams.
- **Task**
  The task is a single instance of an application or analysis code execution in a Job Stream. One example would be executing TRACE on an input file.
- **Actor**
  An Actor is an object that represent an application or analysis code execution in a Python stream. Once created, adding an actor to the running stream will schedule it for execution.

1.2.Finding More Information

More detailed information about the Python bindings for SNAP and PyPost can be found in their respective User's Manuals and generated API documentation.

- SNAP User's Manual
  - In the Model Editor: **Help →Help Contents**
  - *[SNAP Installation]*/manuals/SNAPUsersManual.pdf
- SNAP API Documentation:
  - *[SNAP Installation]*/doc/python/snap/index.html
- PyPost User's Manual
  - *[AptPlot Installation]*/manuals/PyPostUsersManual.pdf
- PyPost API Documentation:
  - *[AptPlot Installation]*/pypost/doc/pypost/ index.html

1.3.Setup

The following applications definitions are assumed throughout this tutorial.

- **TRACE** – A version 5.0 Patch 5 TRACE executable. Version 5.1350 was used during the development of this tutorial.
- **AptPlot** – Apt Plot Version 7.0+ must be installed and selected in the Plotting Tools entry of the Configuration Tool.
  - **Python** – Either the Jython interpreter included with SNAP or CPython version 3.6+/2.7.

2. Exercise 1 – Simple Python Directed Job Stream

This tutorial will walk you through the process of building a simple Python Directed stream and submitting it to your local Calculation Server.

1. Open the SNAP Model Editor.
2. Open the Exercise1.med model included with this tutorial.

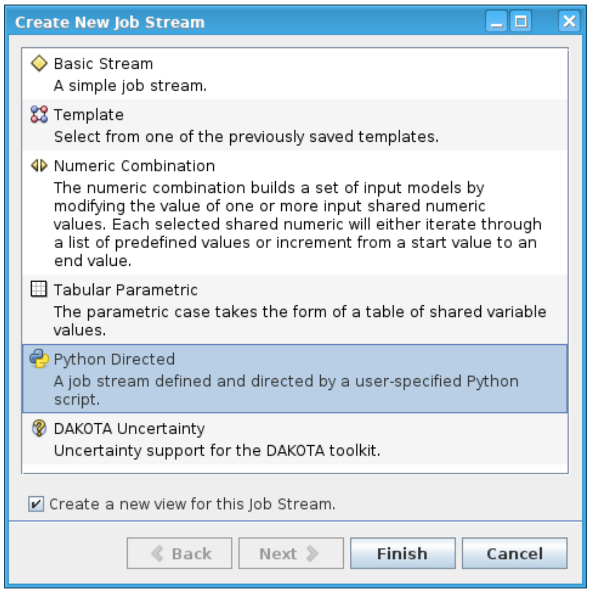
The standpipe model is a simple calculation of a 4m long 6" diameter schedule 80 Pipe, heated from 0.4m to 3.6m. It uses interactive controls to define the boundary conditions for the fill, as well as the heat flux on the outside of the pipe wall heat structure.

The next steps will add a new Python Directed job stream to the model.

3. Right-Click the Job Streams node in the Navigator and select the **New** menu item.
4. Select the **Python Directed** stream type.
5. Press the **Finish** button.

This creates a new stream with the Stream Type set to Python Directed. This will automatically expand the Job Streams node in the Navigator and select the new stream.

6. Set the **Name** property to "Python_Standpipe".
7. Set the **Relative Location** to "DEMO/".



Basic Stream
A simple job stream.

Template
Select from one of the previously saved templates.

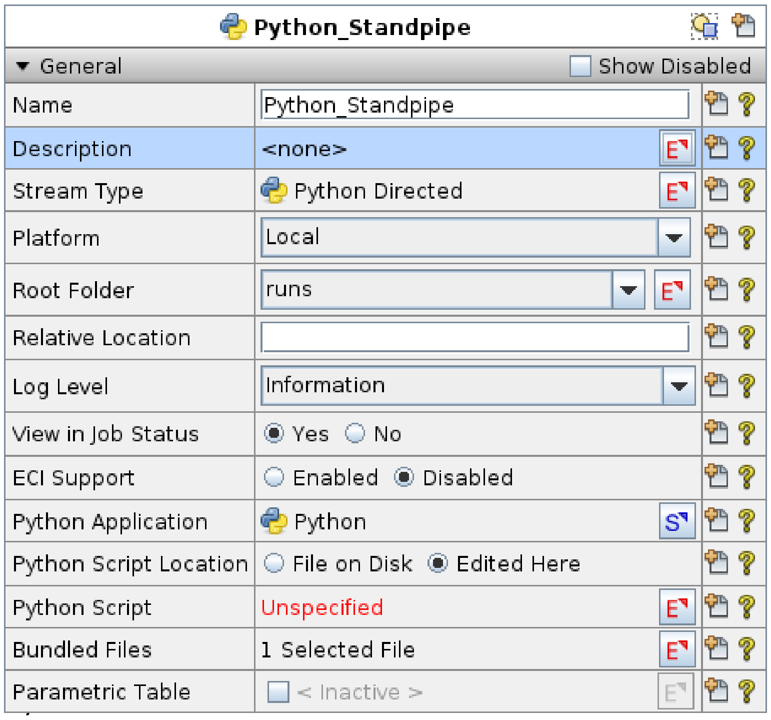
Numeric Combination
The numeric combination builds a set of input models by modifying the value of one or more input shared numeric values. Each selected shared numeric will either iterate through a list of predefined values or increment from a start value to an end value.

Tabular Parametric
The parametric case takes the form of a table of shared variable values.

Python Directed
A job stream defined and directed by a user-specified Python script.

DAKOTA Uncertainty
Uncertainty support for the DAKOTA toolkit.

☑ Create a new view for this Job Stream.



Name	Python_Standpipe
Description	<none>
Stream Type	Python Directed
Platform	Local
Root Folder	runs
Relative Location	
Log Level	Information
View in Job Status	[✓ Yes] No
ECI Support	Enabled [✓ Disabled]
Python Application	Python
Python Script Location	File on Disk [✓ Edited Here]
Python Script	Unspecified
Bundled Files	1 Selected File
Parametric Table	☐ < Inactive >

Python streams include a number of properties not available in other stream types. Refer to the SNAP User's Manual for more detailed information.

***Python Application** lets you select the Python application that will be used to execute this Python Directed stream. The application is one of the Python application definitions specified in Configuration Tool*

*The **Python Script Location** property determines whether a Python script will be edited with the model (Edited Here) or referenced on disk (File on*

*Disk). This exercise will use the **Edited Here** mode.*

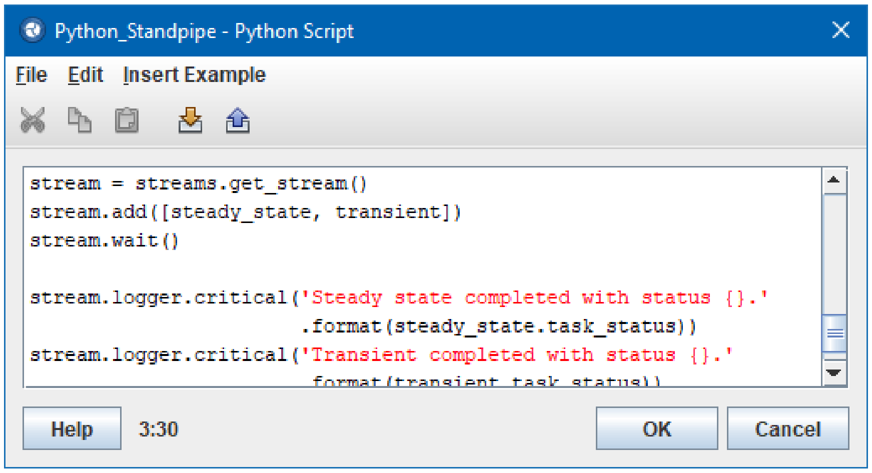
*The **Python Script** property is either, the path to the Python script or the content of the script itself, depending on the value of the Python Script Location property.*

***Bundled Files** allows you to include files with the submission. Typically these are data files or additional input files required by the calculation. This will be covered more in later exercises.*

*The **Parametric Table** property provides many of the features of the Tabular Parametric job stream type to Python Directed streams. Its user interface in the stream is also essentially identical to that used by Tabular Parametric. It can be thought of as a shortcut to simplify the creation of a table that will be made available to the Python script.*

8. **Edit** the **Python Script** property.

This will open the Python source editing window showing an empty script. Menu items and toolbar buttons provide the standard Import/Export and Cut/Copy/Paste operations. At the bottom of the window is a label indicating the cursor location as a line and column. The current number of lines in the script is displayed to the right of the cursor location. The Help button at the bottom left opens the SNAP User's Manual to the Python Directed Job Streams section.



```
stream = streams.get_stream()
stream.add([steady_state, transient])
stream.wait()

stream.logger.critical('Steady state completed with status {}.'
                       .format(steady_state.task_status))
stream.logger.critical('Transient completed with status {}.'
                        .format(transient.task_status))
```

The items in the Insert Example menu will insert one of a variety of code snippets that illustrate everything from opening an MED file to running a job with one of the supported analysis codes.

9. Expand the **Insert Example** menu.
10. Select the **Code Support → Running TRACE** menu item.

This inserts example code to execute a pair of sequential TRACE executions using the stand pipe model that is currently open. The "Examples" in the source editor are intended to give a broad view of what is possible rather showing the simple or most concise way to do something. In this case, the standpipe MED is referenced with an absolute file path. Since we know that it will be bundled with the stream there is a simpler way to locate the MED.

The next steps will convert the portions of the script that open a MED file on disk to use the standpipe.med file that will automatically be bundled with the stream.

11. Delete the following lines from the script.

```
samples_dir = Path('C:/Program Files/SNAP/Samples/TRACE')
med = samples_dir.joinpath('Standpipe', 'standpipe.med')

standpipe = model_editor.open_model(med)
```

12. Insert the following lines in their place:

```
stream = streams.get_stream()
standpipe = model_editor.open_model(stream.get_bundled_file('Exercise1.med'))
```

*The next steps will simplify the creation of the restart case by defining the tracin**,** and trcrst inputs of the Transient step during its creation.*

13. Remove the following lines from the script:

```
transient = TraceActor('Transient')
# Inputs can also be specified later using the >> operator.
# This indicates that the 'Short' restart case should be
# used as the TRACE input for the transient case.
standpipe.case('Short') >> transient.tracin
# The TPR file is passed from the steady state run.
steady_state.trctpr >> transient.trcrst
# But it could be a file location.
Path('/path/to/steady_state.tpr') >> transient.trcrst
```

These lines illustrate how to use the insertion (>>) operator to connect a model's restart case ('Short') and a TRACE TPR file ('trctpr') to the Transient job. This is an important feature in more complex streams or when restarting portions of a completed stream. Since we don't need this complexity, we'll just replace it with something simpler.

14. Enter the following line in their place:

```
transient = TraceActor('Transient',
                       tracin=standpipe.case('Short'),
                       trcrst=steady_state.trctpr)
```

Notice that the parameters to the TraceActor, tracin and trcrst, are TRACE input file names and match the inputs used in the graphical TRACE job step. Passing a file source as either of these parameters will make that file available to the resulting job, automatically exporting, copying, or referencing the file, as necessary

These inputs and all of TRACE's outputs are also available as properties of the TraceActor object. In this example (trcrst=steady_state.trctpr) the trctpr property is being used as the trcrst input, ensuring that the Steady_State TPR output will be copied in as the Transient job's restart file.

15. Press **OK** to close the save the script.
16. Right-click the **Python_Standpipe** job stream in the Navigator and select the **Submit Stream to Local** menu item.
17. Press the **OK** button.

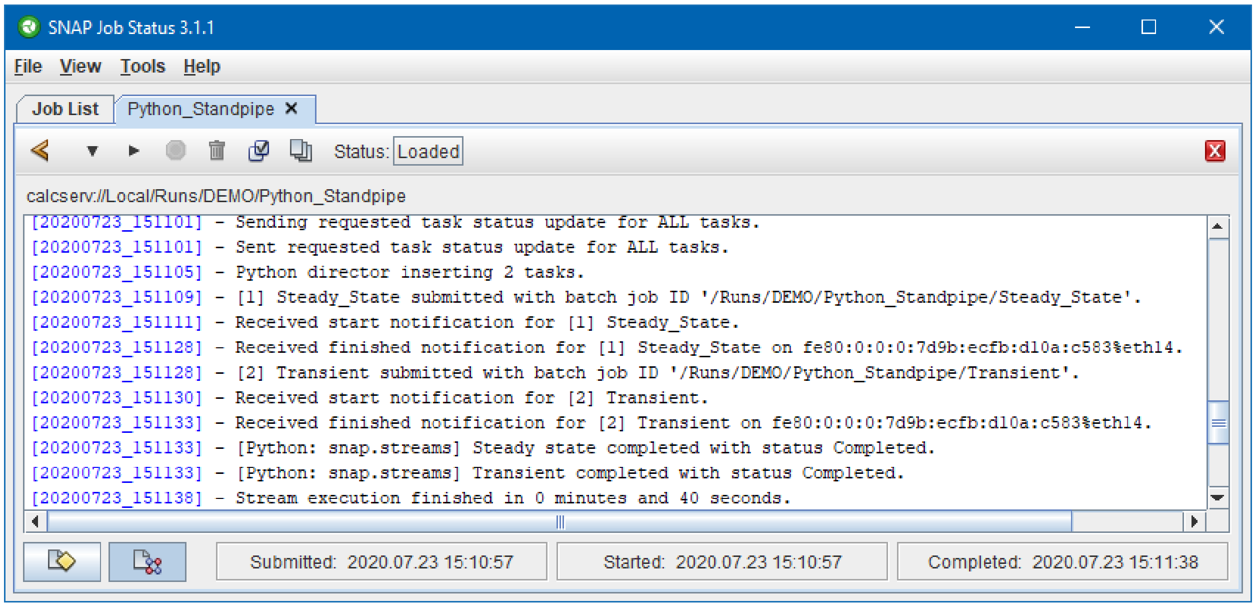
Job Status will automatically open and select the Python_Standpipe job stream.

18. Double-Click **Python Standpipe** in the Job List table in Job Status.

This will open a console to display the stream log output. The log describes the stream as it runs with start notifications, finished notifications, etc. For example, the following lines indicate that the two TRACE jobs (Steady_State, and Transient) finished successfully.

```
[<DATE>] - [Python: snap.streams] Steady state completed with status Completed.
[<DATE>] - [Python: snap.streams] Transient completed with status Completed.
```

Adding your own messages to the stream log will be covered in later exercises.



```
[20200723_151101] - Sending requested task status update for ALL tasks.
[20200723_151101] - Sent requested task status update for ALL tasks.
[20200723_151105] - Python director inserting 2 tasks.
[20200723_151109] - [1] Steady_State submitted with batch job ID '/Runs/DEMO/Python_Standpipe/Steady_State'.
[20200723_151111] - Received start notification for [1] Steady_State.
[20200723_151128] - Received finished notification for [1] Steady_State on fe80:0:0:0:7d9b:ecfb:d10a:c583%eth14.
[20200723_151128] - [2] Transient submitted with batch job ID '/Runs/DEMO/Python_Standpipe/Transient'.
[20200723_151130] - Received start notification for [2] Transient.
[20200723_151133] - Received finished notification for [2] Transient on fe80:0:0:0:7d9b:ecfb:d10a:c583%eth14.
[20200723_151133] - [Python: snap.streams] Steady state completed with status Completed.
[20200723_151133] - [Python: snap.streams] Transient completed with status Completed.
[20200723_151138] - Stream execution finished in 0 minutes and 40 seconds.
```

Submitted: 2020.07.23 15:10:57 Started: 2020.07.23 15:10:57 Completed: 2020.07.23 15:11:38

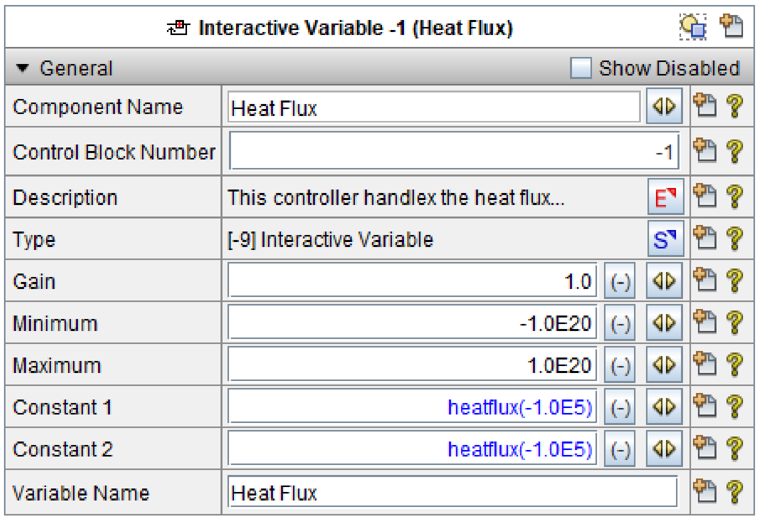
The stream details panel, typically shown when submitting graphical streams, is available by pressing the details button in the bottom left corner of the window.

This completes Exercise 1.

3. Exercise 2 – SNAP Variables and Optimization

This exercise demonstrates some advanced capabilities of the Python Directed Job Streams by executing a series of TRACE cases until a predefined output state is achieved. Previously, this would have required multiple job streams and job stream sequences. A simple implementation of Newtons method will be used to iterate the heat flux values of the Stand Pipe model until the last heated cell in the pipe has a void fraction of 0.5.

1. If you are continuing from exercise 1, save the model as Exercise2.med.
   If not, open Exercise2.med included with this tutorial.
2. In the Navigator, expand the **Control Systems → Control Blocks** category node.
3. Select **Interactive Variable -1**.



Interactive Variable -1 (Heat Flux)

General | Show Disabled

Component Name	Heat Flux
Control Block Number	-1
Description	This controller handlex the heat flux...
Type	[-9] Interactive Variable
Gain	1.0
Minimum	-1.0E20
Maximum	1.0E20
Constant 1	heatflux(-1.0E5)
Constant 2	heatflux(-1.0E5)
Variable Name	Heat Flux

The value of interactive variable -1 is set by a SNAP Variable called 'heatflux'. We'll be using this Variable to alter the boundary heat flux on Heat Structure 31 and heat up the pipe.

4. Create a new Python Job Stream as described in the previous exercise.
5. Set the **Name** of the stream to "Opt_Standpipe".
6. **Edit** the **Python Script** property.

We're going to use SNAP streams, TRACE models and jobs, and PyPost, so the script will need to import those modules.

7. Add the following lines to the script.

```
from snap import streams
from snap.codes import trace
from pypost.codes.trace import TRACE
```

To update properties or add new jobs we'll need to get a reference to the currently running stream. This reference is typically cached to make the script more readable.

8. Add the following line to the script.

```
stream = streams.get_stream()
```

To get to the model in Exercise2.med it will first need to be opened.

9. Add the following lines to the script.

```
model_med = stream.get_bundled_file("Exercise2.med")
model = trace.open_model(model_med)

target = 0.5
tolerance = 0.01
```

The first two lines above get the MED file location from the stream's list of "bundled files" and open the file as a TRACE model. The last two lines define the "target" void fraction, and the "tolerance" for convergence.

When using a Job Stream component, the model's MED file is automatically "bundled" with the stream. The stream's bundled files are transmitted with the stream to the server during submission. This can be plot files, table data, input models, and anything else required by the stream. These files are then extracted on the server side just before the script begins. They can be accessed using the stream's get_bundled_file(name) and find_bundled_files(wildcard) methods.

10. Add the following lines to the script.

```
def submit_case( index, heat_flux ):
    model.set_variable( "heatflux", str(heat_flux) )
    actor = trace.TraceActor( name="SS_{:02}".format(index), tracin=model )
    stream.add( actor )
    stream.wait()
```

This is a new python method that sets the 'heatflux' variable in the standpipe model, and submits the model to the calculation server as a new job named SS_<INDEX>. This allows the calculation to be re-run up to 99 times before overriding any data. This method will be used to run each iteration. The next step will extract the void-fraction at the end of the pipe.

11. Add the following lines to the script to the same **submit_case** method from the previous step.

```
    TRACE.openPlotFile( actor.trcxtv_out.shared_location )
    void = TRACE.getData( 0, "alpn-21A20" ).maxYval()
    TRACE.closeAll()
    return void
```

This method extracts the maximum value from the 'alpn-21A20' plot variable. This is the value that will be used to steer the calculation. The next step will add the check_result method.

12. Add the following lines to the script.

```
def check_result( index, heat_flux, void ):
    if abs( void - target ) < tolerance:
        stream.logger.critical(
            "Target Void {:.3g} reached after {} steps with a Heat Flux of {:.3g}".
            format( void, index, heat_flux ))
        return True
    return False
```

The ***check_result*** *method will be used to determine when the optimization has been completed. This will be called before each iteration.*

13. Add the following lines to the script.

```
def calculate_offset( void1, void2, dFlux ):
    error = target - void2
    slope = (( void2 - void1 ) / dFlux)
    return error / slope
```

The ***calculate_offset*** *method calculates the heatflux offset for the next iteration. This is a simple implementation of Newton's Method.*

14. Add the following lines to the script.

```
heat_flux = -2E5
dFlux = -1E5
index = 1
void1 = submit_case( index, heat_flux )
```

These lines initialize the first heat flux, and the initial heat flux step size for the iteration. The first TRACE job is also subumitted using the default index and heat_flux. The next objective is to start writing a text file that will contain all of the iteration values.

15. Add the following lines to the script.

```
result = open("results.csv","w")
result.write( "Iteration,   Heat Flux, Void Fraction,      Error, Delta Heat Flux\n")
result.write( "{:>9}, {:>11.3g}, {:>13.3g}, {:>10.3g}, {:>15.3g}\n"
       .format( index, heat_flux, void1, (target-void1), dFlux) )
```

These lines open a text file in the job stream directory named "results.csv". This file will be updated to contain a log of all of the heat flux, void fraction, error signal, and heat flux offsets for each case. The values are formatted to ensure that they can be read into a spreadsheet if necessary.

16. Add the following lines to the script.

```
while not check_result( index, heat_flux, void1 ):
    index += 1
    heat_flux += dFlux
    stream.logger.critical("Submitting case {} with Heat Flux of {:.3g}".
       format(index, heat_flux))
    void2 = submit_case( index, heat_flux )
    dFlux = calculate_offset( void1, void2, dFlux )
    result.write( "{:>9}, {:>11.3g}, {:>13.3g}, {:>10.3g}, {:>15.3g}\n".
        format( index, heat_flux, void2, (target-void2), dFlux) )
    result.flush()
    void1 = void2
result.close()
```

This portion of the script is the main loop of the optimization. The methods added to the script earlier are used to dial in the heat flux until the desired void fraction is reached.

*Note: The final line, **result.close()**, must not be indented.*

The next few steps will submit the stream and see how it runs.

17. Press **OK** to close the source editor.
18. Right-click on the **Opt_Standpipe** stream and select **Submit Stream to Local**.
19. Press **OK** to submit the stream.

Job Status will automatically open and select the Loop_Standpipe job stream.

20. Double-click **Opt_Standpipe** in the Job List table in Job Status.

Like the previous exercise, this will open a console to display the stream log output. The result.csv file generated by this job stream should be similar to the following. Note that the actual number of iterations and calculated values may vary between code versions.

```
Iteration,   Heat Flux, Void Fraction,      Error, Delta Heat Flux
        1,      -2e+05,      0.000957,      0.499,          -1e+05
        2,      -3e+05,        0.0848,      0.415,       -4.95e+05
        3,   -7.95e+05,         0.665,     -0.165,        1.41e+05
        4,   -6.54e+05,         0.585,    -0.0855,        1.51e+05
        5,   -5.03e+05,         0.403,     0.0974,       -8.03e+04
        6,   -5.84e+05,          0.52,    -0.0201,        1.38e+04
        7,    -5.7e+05,         0.504,    -0.0041,        3.51e+03
```

This completes Exercise 2.

4. Exercise 3 – Keywords and File Search

This exercise explores the process of creating and assigning keywords to jobs and using those keywords to select which output files will be processed based on search criteria. To do this, the logic retrieving the void fraction from each job will need to be moved to the job as a Post-Execute function. Using PyPost to extract the void fraction in the stream script is convenient but, to apply task keywords, the process needs to happen in the task.

To promote better retention and save space, this exercise will begin to gloss over topics explained in previous exercises.

1. If you are continuing from Exercise 2, save the model as Exercise3.med.
   If not, open Exercise3.med included with this tutorial.
2. Create a new Python Job Stream as described in the previous exercises.
3. Set the **Name** of the stream to "Search_Standpipe".
4. **Edit** the **Python Script** property.

As before, this script will use streams, TRACE models and jobs, AptPlot, and PyPost, so it will need to import those modules.

5. Add the following lines to the script.

```
from snap import streams
from snap.codes import trace
from pypost.codes.trace import TRACE
from pypost.codes.aptplot import APTPLOT
```

This stream will be executing the TRACE code using the model in Exercise3.med.

6. Add the following lines to the script.

```
stream = streams.get_stream()
model_med = stream.get_bundled_file("Exercise3.med")
model = trace.open_model(model_med)
```

This section is nearly unchanged from the previous exercise. It opens the bundled MED file for the current exercise so it can be used by actors later on.

7. Add the following lines to the script.

```
def update_task_keywords():
    from snap import tasks
    from pypost.codes.trace import TRACE
```

This will define the post-execute function used to find the void fraction for each TRACE job. You'll notice that the function starts with import statements. Since pre- and post-execute functions are executed inside the job rather than in the stream, any modules required by the function must be included. In this case, the SNAP Tasks module (snap.tasks) and the TRACE support module for PyPost are both imported.

8. Add the following lines to the script.

```
    xtv_file = tasks.get_output_file("trcxtv")
```

This line retrieves the XTV output file definition (snap.tasks.OutputFile) for this job from the snap.tasks module. For more detailed information about the snap.tasks module, refer to the SNAP User's manual and API documentation listed in section 1.2 on page 2.

9. Add the following lines to the script.

```
    file_index = TRACE.openPlotFile(xtv_file.location)
    alpn_vector = TRACE.getData(file_index, "alpn-21A20")
```

The output file's "location" is a Path from the built-in Python module 'pathlib'. Nearly all path related properties in the SNAP package are handled using pathlib.

The openPlotFile method returns the "file index" of the newly opened file. In this case, there's only one open file so the index will always be zero. In more complex streams it may be a good idea to use the index to avoid inadvertently reading data from the wrong file with getData(..).

10. Add the following lines to the script.

```
    tasks.set_keyword("void_fraction", alpn_vector.maxYval())
```

This will set the value of the 'void_fraction' keyword to the largest void fraction value in the cell at the top the standpipe. Note that, for this to work, the keyword must be added to the Actor before it is added to the stream. Adding keywords to an Actor will be shown later in this exercise.

11. Add the following lines to the script.

```
    TRACE.closeAll()
```

It's not strictly necessary to close the open files at this point because the post-execution function is nearly the last thing the job does before shutting down. It is, however, a good habit to get into.

12. Add the following lines to the script.

```
    tasks.get_logger().critical("Heat flux {} resulted in a void fraction of {:.3f}."
                                .format(tasks.get_keyword("heatflux"),
                                        alpn_vector.maxYval()))
```

*Log messages like this one can be a great help when diagnosing problems with a new stream. Note that this message will appear in the Task log rather than the stream log. To send a message to the stream log, use the **get_stream_logger** method instead of **get_logger**.*

13. Add the following lines to the script.

```
starting_flux = -1.0E5
for index in range(1, 10):
    current_flux = starting_flux * index
    model.set_variable("heatflux", str(current_flux))

    actor = trace.TraceActor(name="SS_{:02}".format(index), tracin=model)
```

This section is essentially identical to the loop in the last exercise. It creates a loop using heat flux values between -1.0e5 and -1.0e6, applies the value to the model, and passes it to a new TRACE actor.

14. Add the following lines to the script.

```
    actor.add_custom_keyword("heatflux", current_flux)
    actor.add_custom_keyword("void_fraction", "Unknown")
```

As mentioned above, the task can't set keywords that haven't already been added by the stream. The first line creates a keyword to hold the heatflux value set on the model for reference. The second adds the 'void_fraction' keyword that will be assigned by the post-execute function and used find the task's outputs later.

15. Add the following lines to the script.

```
    actor.post_execute = update_task_keywords
```

This line illustrate simplicity of pre and post execute scripting in python streams. It indicates to the actor that it should pass the contents of the update_task_keywords method to the task on the server side where it will be executed. This process uses the built-in Python 'inspect' module to retrieve the source code for the function and insert it into the task.

16. Add the following lines to the script.

```
    stream.add(actor)
```

Add the TRACE job to the stream's queue.

17. Add the following lines to the script.

```
stream.wait()
```

Pause the script until the stream has no jobs left to execute. Make sure that that this line is not indented. In this case, the script should add all of the actors to the stream before waiting.

18. Add the following lines to the script.

```
xtv_files = (stream.search().label_eq("trcxtv")
                            .task_kw_gt("void_fraction", 0.5)
                            .results())
```

The stream file search module (snap.stream.search) is used to find files in either the running stream or a completed locally accessible stream. It can search for files by file type, actor name, task or file keywords, file names, etc.. This feature is intended to be used as file and keyword based conditional logic. It could be to perform additional post-processing on only those tasks that meet a specific set of criteria. It could also be used to eliminate failures from a series of jobs.

In this case, the search is being used to get the plot file (trcxtv) of every job with a void fraction over 0.5. The files are returned as a list of search results (snap.stream.search.SearchResult). A similar search could be used to find restart files and create a restart job for each result.

For more detailed information about the snap.streams.search module, or anything in the SNAP package, refer to the SNAP User's manual and API documentation listed on page 2.

19. Add the following lines to the script.

```
for file in xtv_files:
    file_index = TRACE.openPlotFile(file.location)
```

This loops over the plot files, opening each one. The 'location' property of a search result, like the task output file, is a Path object from the pathlib module.

20. Add the following line to the script.

```
    alpn_vector = TRACE.getData(file_index, "alpn-21A20")
```

Get the void fraction values for the top (cell 20) of the stand pipe (pipe 21).

21. Add the following line to the script.

```
    APTPLOT.plotChannels(alpn_vector)
```

This line adds the time dependent void fraction data to the AptPlot virtual graph. This includes the engineering units and channel labels. For more detailed information about the plotChannels method, the pypost.codes.aptplot, or anything in the PyPost package, refer to the User's manual and API documentation listed on page 2.

22. Add the following lines to the script.

```
    TRACE.closeAll()
```

As always, closing any files you're no longer using saves memory and is good practice.

23. Add the following line to the script.

```
APTPLOT.printFile("PDF", "VoidFraction.pdf")
```

This method writes all of the time dependent void fraction data to VoidFraction.pdf in PDF format. It currently supports PDF, SVG, JPEG, TIFF, PNG, Postscript, EPS, and EMF.

Make sure that that this line is not indented or the file will be written several times rather than once.

24. Add the following line to the script.

```
APTPLOT.clearData()
```

Following good practice, this removes all data from the virtual graph.

25. Press **OK** to close the source editor.
26. Right-click on the **Search_Standpipe** stream and select **Submit Stream to Local**.
27. Press **OK** to submit the stream.

Job Status will automatically open and select the Search_Standpipe job stream.

28. Wait for the stream to complete.

If you would like to follow the stream's execution, double-click on the stream name in the Job List. This will open the stream console to display the stream's log messages as they appear.

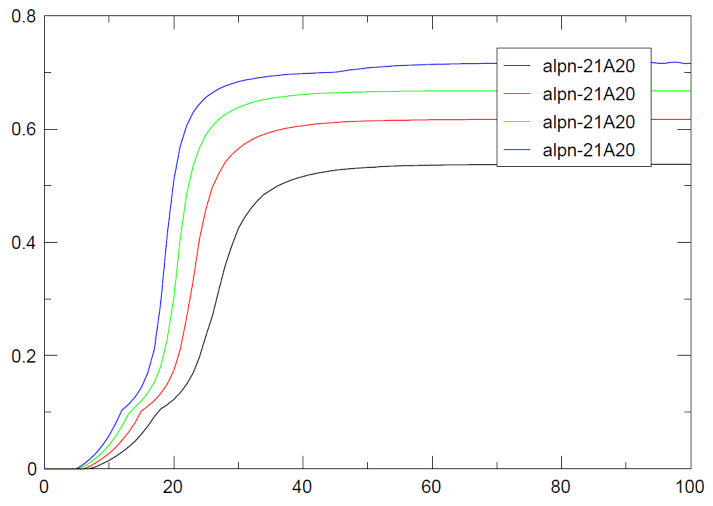
29. Expand the Job Status Navigator to **Runs/DEMO/Search_Standpipe/**
30. Right-click on **Search_Standpipe/** in the Navigator.
31. Select the **Open Folder** item from the pop-up menu.

This will open a local file browser for the directory that contains the stream.

32. Open **VoidFraction.pdf** in your local PDF viewer.

This simple plot contains the void fraction values at the top of the standpipe for each of the jobs with a void fraction over 0.5.

This completes Exercise 3.

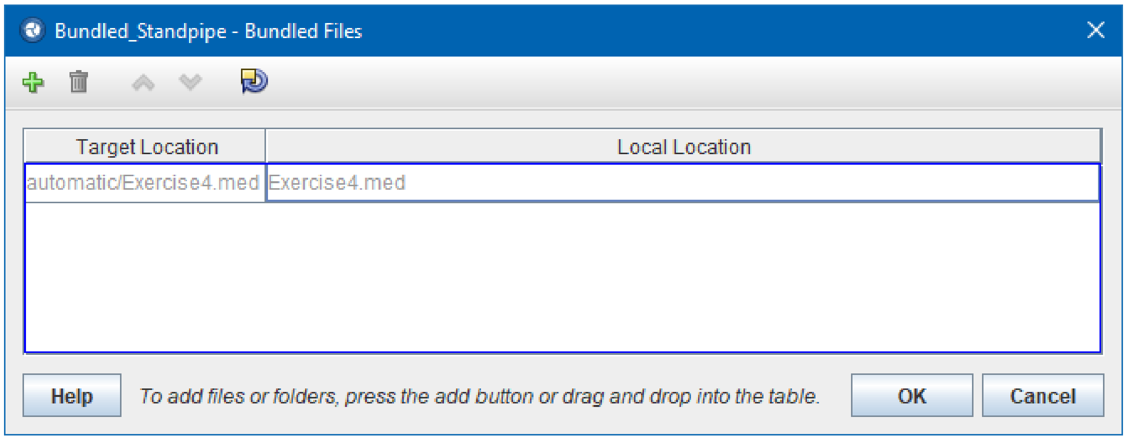


5. Exercise 4 – Bundled Files

This exercise focuses on the Bundled Files feature of the SNAP streams module and its user interface in the Model Editor. These features will be illustrated by creating a set of TRACE restart jobs using bundled input files.

Additional information about these features can be found in the SNAP User's manual and API documentation listed on page 2.

1. Open **Exercise4.med** included with this tutorial.



Target Location	Local Location
automatic/Exercise4.med	Exercise4.med

To add files or folders, press the add button or drag and drop into the table.

2. Create a new Python Job Stream as described in previous exercises.
3. Set the **Name** of the stream to "Bundled_Standpipe".
4. **Edit** the **Bundled Files** property.

*The center piece of the Bundled Files window is the table of bundled files. The toolbar at the top of the dialog contains buttons to **Add** (), **Remove** (), **Move Up** (), **Move Down** (), and **Sort** (), the table. The **Help** button, at the bottom left, will open the SNAP User's manual to the section describing this window.*

*Each bundled file in the table has a **Target Location** and a **Local Location**. The target location is used to determine where to place the file under the bundled files directory. The local location is the path to the file on the local computer. The current model is always bundled with the stream and thus is always included as the first row in the table. When adding files to this table, the dialog will attempt use a relative path to the selected file, where possible.*

5. Press the **Add** button on the toolbar.
6. Select the file **standpipe_short.inp** included with this tutorial.

This will include standpipe_short.inp with the stream, accessible with stream.get_bundled_file(…). Note that the target location of a file can be changed to suit the stream. For example, if you were including a *plot file named simply, “trcxtv”, you could decide to include this with a target location of “peak_void_fraction.xtv”.*

7. Press the **Add** button again.
8. Select the **RestartInputs/** directory included with this tutorial.

*The **RestartInputs/** directory contains four TRACE inputs: short1.inp, short2.inp, long1.inp, and long.inp.*

9. Press the **Select** button to complete the addition.

*This will add every file in the **RestartInputs/** directory to the bundled files table. Adding files to the table automatically selects the newly added files. This makes it easy to see exactly which files were in the directory. As a side effect, it also makes it easy to remove the newly added files.*

A target location has been automatically assigned for each of the selected files relative to the directory that was added. This can be useful for situations that require a large number of similarly named files in a directory structure.

10. Select the newly added **RestartInput/** files, if they aren’t already selected.
11. Press the **Remove** button on the toolbar.

This will remove the restart inputs that were just added.

12. Open a file local filesystem browser such as Windows File Explorer.
13. Navigate to the directory containing this tutorial.
14. Drag the **RestartInputs/** directory into the bundled files table.

*Files and directories dragged into the table are treated exactly the same as those added using the **Add** button.*

15. Select the newly added **RestartInput/** files, if they aren’t already selected.
16. Press the **Remove** button on the toolbar.

This will, again, remove the restart inputs that were just added.

17. In the local file browser, open the **RestartInputs/** directory.
18. Drag the **Long/** and **Short/** directories into the bundled files table.

*This adds the files in both directories to the bundled files table. Notice that the target locations do not include the **RestartInputs/** directory. This time they have been generated relative to the **Long/** and **Short/** directories that were selected.*

19. Press the **OK** button to close the Bundled Files window.

Now that the files are bundled, the remainder of the exercise will look at how these files should be accessed.

20. **Edit** the **Python Script** property.
21. Add the following line to the script.

```
from snap import streams
from snap.codes import trace
from snap.codes.trace import TraceActor
```

This simple stream only needs the streams and TRACE modules.

22. Add the following line to the script.

```
stream = streams.get_stream()
model = trace.open_model(stream.get_bundled_file("Exercise4.med"))
ss = TraceActor("SS", tracin=model)
stream.add(ss)
```

The first actor is a steady state calculation using the automatically bundled Exercise4.med.

23. Add the following line to the script.

```
stream.add(TraceActor("Long1",
                      tracin=stream.get_bundled_file("Long/long1.inp"),
                      trcrst=ss.trctpr))
```

The second actor, “Long1”, is a restart of the steady state. In this case, both the input file (tracin) and the restart input file (trcrst) are being passed to the constructor. The input file (long1.inp) is accessed using the target location shown in the Bundled Files window. The restart input file (trcrst) will be copied from the restart output (trctpr) file of the steady state job.

24. Add the following line to the script.

```
short1 = TraceActor("Short1")
ss.trctpr >> short1.trcrst
```

Rather than passing everything in the constructor, the second actor is using the “right shift” or “insertion” (>>) operator to request that the restart input file be copied from the restart output of the steady state job.

The most commonly used input files (tracin, trcrst, etc.) for each actor type can usually be passed to the constructor. This is a convenience to reduce the amount of code you have to write. Not all of an actor’s inputs are available this way. Refer to the documentation listed on page 2 for more detailed information about the actors for each supported code.

25. Add the following line to the script.

```
stream.get_bundled_file("Short/short1.inp") >> short1.tracin
stream.add(short1)
```

The insertion (>>) operator can also be used with path-like objects such as those returned by get_bundled_file(). In this case, using the bundled short1.inp as the input file for Short1. Once it's connected, it's added to the stream.

*The **get_bundled_file** method is a convenient way to access bundled files individually. A second method is available to access multiple files: **find_bundled_files**.*

26. Add the following line to the script.

```
for file in stream.find_bundled_files("**/*2.inp"):
```

*The **find_bundled_files** method returns a list of paths to the bundled files with names that match the given pattern. The pattern is assumed to be a wildcard or "glob" expression such as '*.inp'. Wildcard patterns use "special characters" such as '*' to match a series of any character, ? to match any single character, '**' to match any files and zero or more directories and subdirectories, and bracketed expressions such as [abc] to match a single character included in the brackets (a, b, or c).*

*In this case, the pattern matches two of the four bundled restart input: **Long/long2.inp** and **Short/short2.inp**. These files will be used to create two additional restart jobs.*

27. Add the following line to the script.

```
    actor_name = file.stem
```

*Because **find_bundled_files** returns a list of Path objects, we can use a bit of Python magic to get the name of the file without its suffix. So, for the two files found, long2.inp and short2.inp, the actor_name will be "long2" and "short2" respectively.*

28. Add the following line to the script.

```
    stream.add(TraceActor(actor_name, tracin=file, trcrst=ss.trctpr))
```

This last line adds a TRACE actor to the stream using an actor_name based on the file name. It's using the bundled file as its input model and the restart output of the steady state as its restart input.

29. Press **OK** to close the source editor.
30. Right-click on the Bundled_Standpipe stream and select **Submit Stream to Local**.
31. Press **OK** to submit the stream.

Job Status will automatically open and select the Bundled_Standpipe job stream.

32. Double-click **Bundled_Standpipe** in the Job List table in Job Status.

Like the previous exercise, this will open a console to display the stream log output.

33. Note the log messages that appear after each job finishes.

```
[<DATE>] - Received finished notification for [1] SS on localhost
```

```
[<DATE>] - Received finished notification for [2] Long1 on localhost.
[<DATE>] - Received finished notification for [5] short2 on localhost.
[<DATE>] - Received finished notification for [3] Short1 on localhost.
[<DATE>] - Received finished notification for [4] long2 on localhost.
```

There we can see that all five jobs were submitted and executed successfully.

This completes Exercise 4.

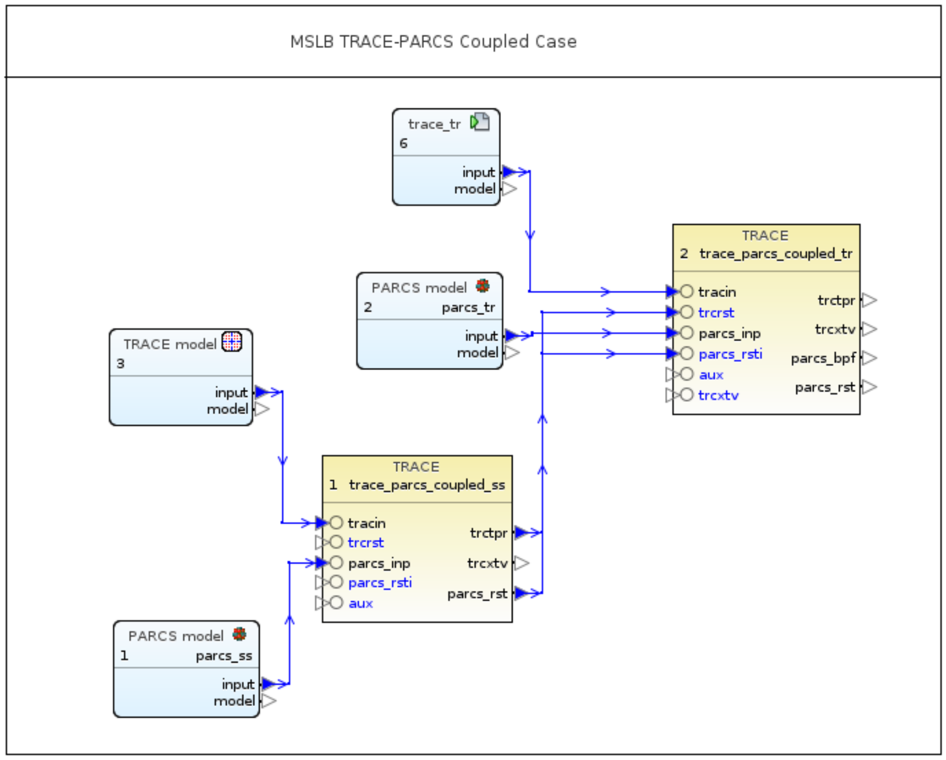
6. Exercise 5 – TRACE-PARCS Coupled Job Stream

This tutorial will walk you through the process of building a coupled TRACE-PARCS job stream and submitting it to your local Calculation Server.

1. Open the SNAP Model Editor.
2. Open the model **Samples/TRACE/Coupled_TraceParcs_Python_Streams/trace_parcs_coupled_mslb.med** included in the distribution.

This is an engineering template model representation of the TRACE-PARCS coupled MSLB model. It consists of a coupled steady state task which is then fed into a coupled transient task. In this example, the steady state is represented by two model nodes, one which is the TRACE steady state model, and the other is a steady state PARCS model. The TRACE model includes a PARCS Mapping and TRACE restart case. The transient portion of this model consists of a model node which references the restart case contained in the TRACE model in conjunction with a PARCS transient model.

This figure below contains a job step representation of the job stream being setup in this exercise.



3. **Right-Click** on the **Job Streams** category in the Navigator and create a new **Python Directed job stream**.
4. **Set** the **Name** of the new job stream to “Coupled_Demo”.
5. Set the **Relative Location** to “DEMO”.
6. **Edit** the **Python Script** property.

This Python stream is going to use SNAP, streams, TRACE and PARCS models, and Model Editor utilities so it will need to import those modules.

7. Add the following lines to the script:

```
import snap
from snap import streams
from snap.codes import trace, parcs
from snap.codes.trace import TraceActor
import snap.model_editor
```

This example uses models that are included with SNAP so we'll use the SNAP installation directory to locate and open these models.

8. Add the following lines to the script

```
trace_path = snap.get_install_dir() / "Samples" / "TRACE"
models_path = trace_path / "Coupled_TraceParcs_Python_Streams" / "models_py"

tm = trace.open_model(models_path / "trace_ss_and_tr.med")
pm_ss = parcs.open_model(models_path / "parcs_ss.med")
pm_tr = parcs.open_model(models_path / "parcs_tr.med")
```

Now that the models are open, we can create a TraceActor to execute them. Because the TRACE model has the MAPTAP file active, it will be automatically included.

```
ss = TraceActor("mslb_ss_coupled")
tm >> ss.tracin
pm_ss >> ss.parcs_inp
```

That's all we need to setup the coupled TRACE-PARCS steady state job. The next steps will submit the job and wait for its completion.

```
stream = streams.get_stream()
stream.add(ss)
stream.wait()
stream.logger.critical("Coupled Steady state completed with status {}.".format(ss.task_status))
```

Once the steady state has completed we then can submit the coupled transient. In order to do this the TRACE and PARCS restart files must first be connected to a new TraceActor.

9. Add the following lines to the script

```
tr = TraceActor("mslb_tr_coupled")
ss.parcs_restart_out >> tr.parcs_restart_in
ss.trctpr >> tr.trcrst
```

Lastly, the transient TRACE model for this run is included in the TraceModel as a case. That model and the transient PARCS model will then need to be connected to the TraceActor.

10. Add the following to the script

```
tm.case("trace_tr") >> tr.tracin
pm_tr >> tr.parcs_inp
```

Now that the transient is configured we just need to submit the job.

11. Add the following to the script

```
stream.add(tr)
stream.wait()
stream.logger.critical("Coupled Transient completed with status {}.".format(tr.task_status))
```

The next few steps will submit the stream and monitor the results

12. Press **OK** to close the source editor.
13. Right-click on the **Coupled_Demo** stream and select **Submit Stream to Local**.

    Job Status will automatically open and select the Coupled_Demo job stream.

14. Double-click **Coupled_Demo** in the Job List table in Job Status.
15. Note the log messages that appear after the steady-state and transient jobs finish.

```
[Python: snap.streams] Coupled Transient completed with status Completed.
```

16. Close the model

This completes Exercise 5.

7. Exercise 6 – Uncertainty Quantification

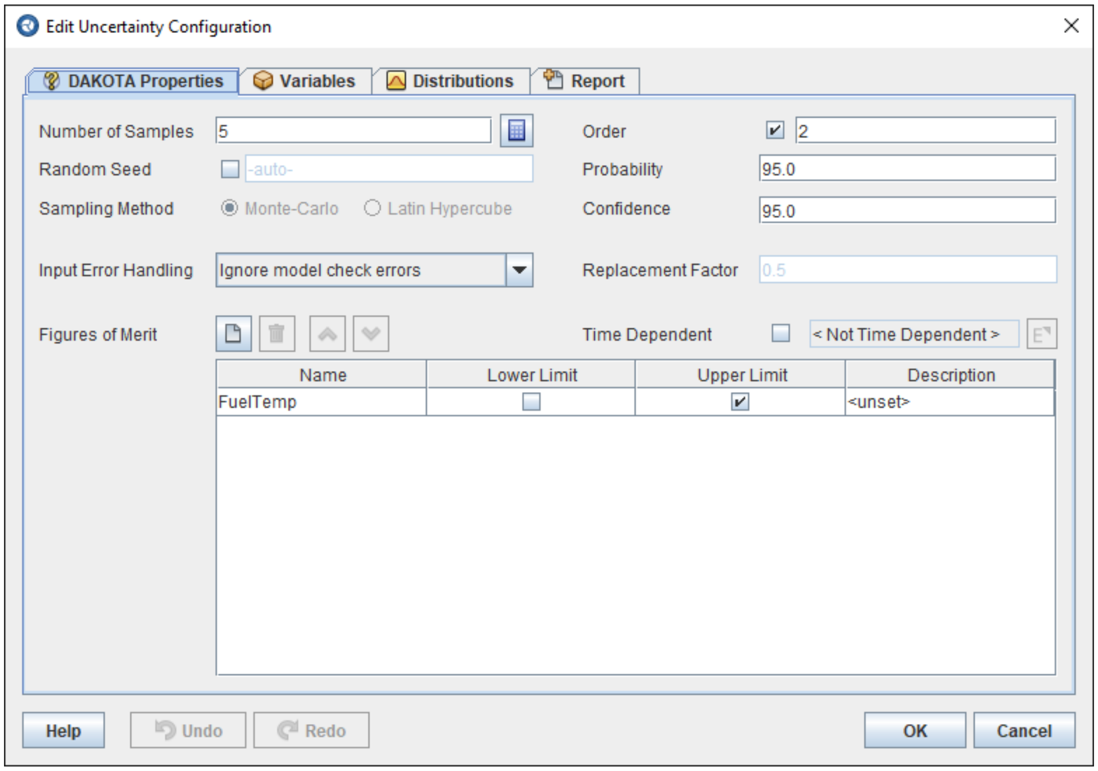
This exercise introduces the DAKOTA Uncertainty Quantification (UQ) support in Python Directed streams. After this exercise is complete the analyst will be familiar with defining the parameters of a UQ calculation in a TRACE model using sensitivity coefficients, extracting figures of merit from resulting plot files, and the DAKOTA report generation.

This exercise will take a null-transient W4Loop model and perform an uncertainty quantification on the fuel gap gas conductivity factor.

1. Open **UQ_W4Loop.med** included with this tutorial.
2. Create a new **Python Directed** stream as described in previous exercises.
3. Set the **Name** of the stream to "UQ_Stream".
4. Activate the **Uncertainty Quantification** property by checking the check-box next to its editor.

The Uncertainty Quantification property is used to configure the uncertainty quantification data structures that will be used in the stream. These structures can be created and configured directly by Python code in the stream, as we'll see later. This property simplifies the process by generating much of that code for you.

5. **Edit** the **Uncertainty Quantification** property.



Number of Samples: 5
Random Seed: [ ] -auto-
Sampling Method: [✓ Monte-Carlo] Latin Hypercube
Input Error Handling: Ignore model check errors
Order: [✓] 2
Probability: 95.0
Confidence: 95.0
Replacement Factor: 0.5
Time Dependent: [ ] < Not Time Dependent >
Figures of Merit

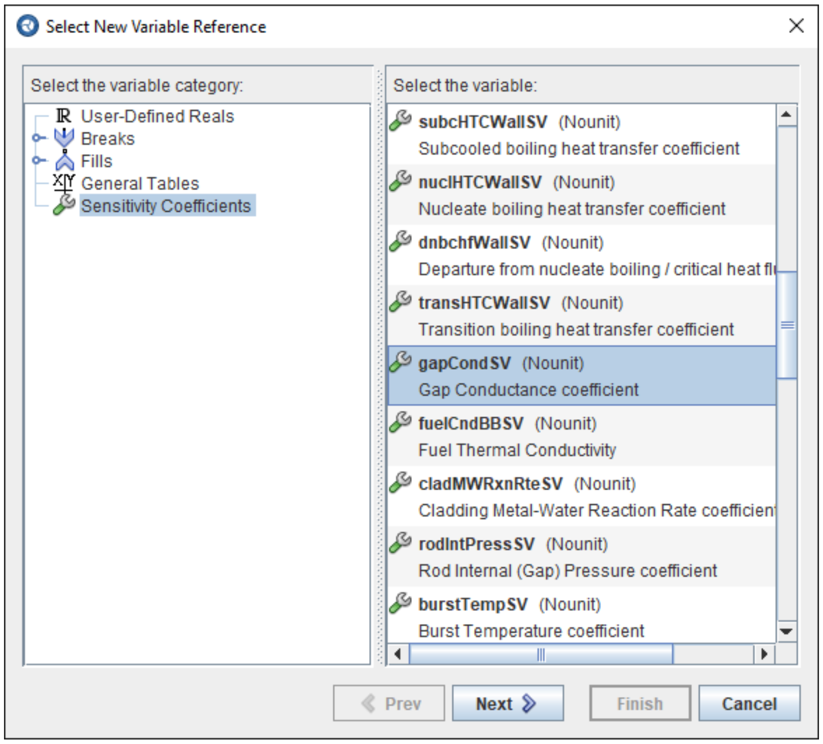
Name	Lower Limit	Upper Limit	Description
FuelTemp	☐	✓	<unset>

This dialog is used to define the labels for the Figures of Merit, select the input variables, and define the distribution used to generate the input variable values. Finally, the Report tab allows the contents of the generated DAKTOA report to be configured.

6. Add a new Figure of Merit (FoM) by pressing the new (🗋) button next to the **Figures of Merit** label.
7. Set the **Name** of the FoM to “FuelTemp”.
8. Check the **Upper Limit** check box for the “FuelTemp” FoM.
9. Select the **Variables** tab.

This tab contains the model input values that are modified as part of the Uncertainty Quantification. The TRACE plug-in allows UQ stream types to modify sensitivity coefficients, boundary component table data, and variable values. For this exercise, we will be modifying the sensitivity coefficient that controls the gap conductance of the fuel rods.

10. Add a new variable by pressing the new (🗋) button at the top of the dialog.
11. Ensure that the **Sensitivity Coefficients** entry is selected in the variable category list.
12. Select “gapCondSV” and press the **Next** button.
13. Press the **Next** button to apply the coefficient to all heat structures.


Select New Variable Reference

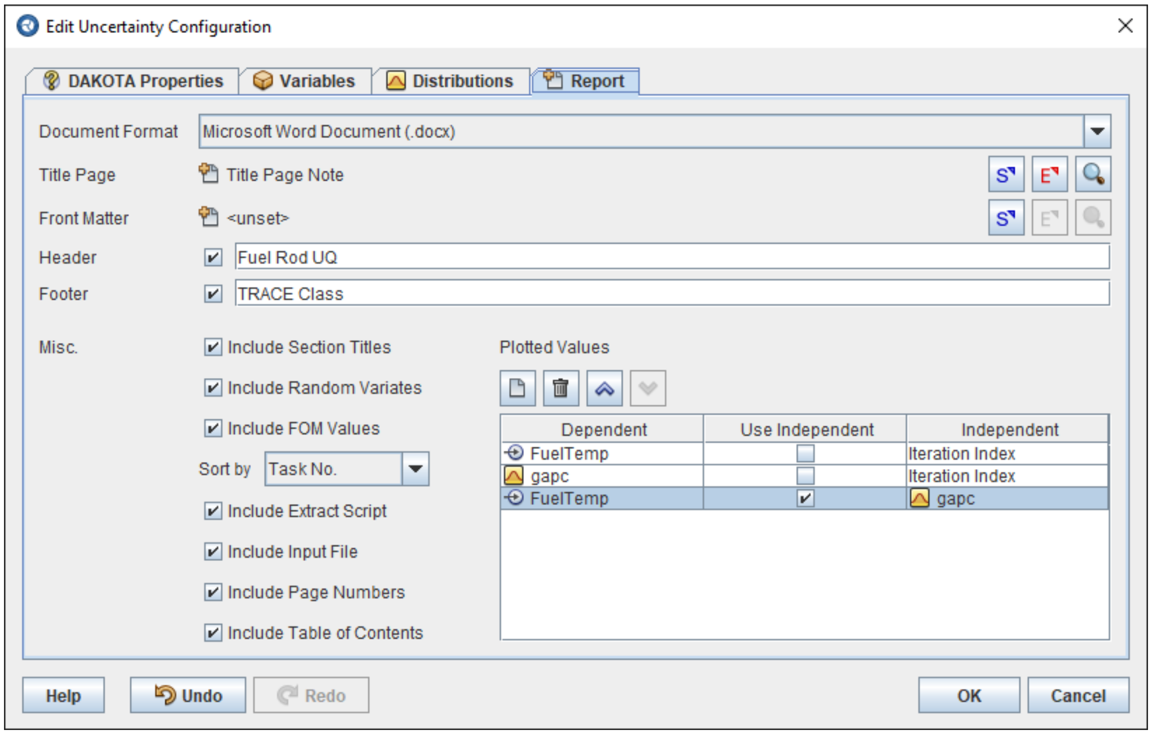
Select the variable category:
- User-Defined Reals
- Breaks
- Fills
- General Tables
- Sensitivity Coefficients

Select the variable:
- **subcHTCWallSV** (Nounit) Subcooled boiling heat transfer coefficient
- **nuclHTCWallSV** (Nounit) Nucleate boiling heat transfer coefficient
- **dnbchfWallSV** (Nounit) Departure from nucleate boiling / critical heat fl
- **transHTCWallSV** (Nounit) Transition boiling heat transfer coefficient
- **gapCondSV** (Nounit) Gap Conductance coefficient
- **fuelCndBBSV** (Nounit) Fuel Thermal Conductivity
- **cladMWRxnRteSV** (Nounit) Cladding Metal-Water Reaction Rate coefficien
- **rodIntPressSV** (Nounit) Rod Internal (Gap) Pressure coefficient
- **burstTempSV** (Nounit) Burst Temperature coefficient

14. Select the **Factor** radio button and press the **Finish** button.
15. Select the **Distributions** tab.
16. Set the **Name** of distribution d1 to “gapc”.
17. Set the **Distribution** type to Normal.
18. Set the **μ (Mean)** value to 1.0.
19. Set the **σ (STDV)** value to 0.1.

This defines Normal distribution about 1.0 with a standard deviation of 0.1. This will result in values that range between 0.7 and 1.3, as shown in the Probability Distribution graph. The distribution could be further modified with the Minimum and Maximum fields.

20. Select the **Report** tab.



Document Format	Microsoft Word Document (.docx)
Title Page	Title Page Note
Front Matter	<unset>
Header	✓ Fuel Rod UQ
Footer	✓ TRACE Class

Misc.
- ✓ Include Section Titles
- ✓ Include Random Variates
- ✓ Include FOM Values
- Sort by: Task No.
- ✓ Include Extract Script
- ✓ Include Input File
- ✓ Include Page Numbers
- ✓ Include Table of Contents

Plotted Values

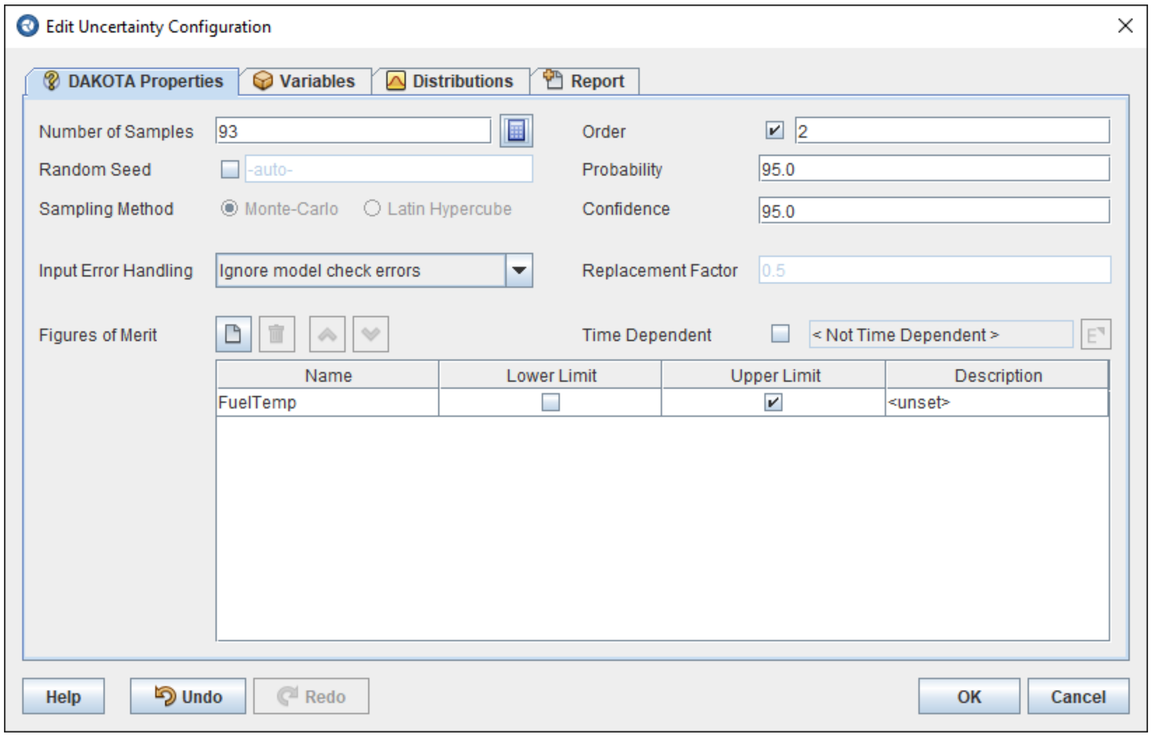
Dependent	Use Independent	Independent
FuelTemp	☐	Iteration Index
gapc	☐	Iteration Index
FuelTemp	✓	gapc

21. Set the **Document Format** to “Microsoft Word Document (.docx)”.
22. Set the **Title Page** to “Title Page Note” by pressing the **S** button to the right of **Title Page**.
23. Activate the **Header** and set the text to “Fuel Rod UQ”.
24. Activate the **Footer** and set the text to “TRACE Class”.

The report will include three plots. The first is a plot that shows the figure of merit value vs the sensitivity coefficient, the second shows the figure of merit value vs the iteration index, and the third shows the sensitivity coefficient values vs the iteration index.

25. Add a new plot by pressing the (🗋) button below the **Plotted Values** label.
26. Select “FuelTemp” in the list and press the **OK** button.
27. Check the **Use Independent** check-box for the new plot.
28. Click in the **Independent** column in the **FuelTemp** row and press the **S** button that appears.
29. Select the “gapc” entry and press the **OK** button.
30. Add a second new plot and select “FuelTemp” in the list.
31. Add a third new plot and select “gapc” in the list.

The final step in setting up the initial uncertainty properties is to define the number of executions that will be performed. This can be entered manually or calculated to ensure the desired confidence level.



Number of Samples: 93
Random Seed: ☐ -auto-
Sampling Method: ● Monte-Carlo ○ Latin Hypercube
Input Error Handling: Ignore model check errors
Figures of Merit
Order: ✓ 2
Probability: 95.0
Confidence: 95.0
Replacement Factor: 0.5
Time Dependent: ☐ < Not Time Dependent >

Name	Lower Limit	Upper Limit	Description
FuelTemp	☐	✓	<unset>

32. Select the **DAKOTA Properties** tab.
33. Activate the **Order** property by checking the check box.
34. Set the value of the **Order** property to “2”.
35. Press the Calculate ( ) button next to the **Number of Samples** field.

*This uses the selected **Sampling Method**, **Probability** and **Confidence** levels to determine the number of samples required for the targeted number of **Figures of Merit**. The number of samples indicates the number of input models that will be generated.*

36. Press **Yes** to proceed.
37. Press the **OK** button to close the dialog.

Now that the Uncertainty Quantification structures been configured, the next section will create a script to execute the analysis.

38. **Edit** the **Python Script** property.
39. Add the following line to the script.

```
import parametric
```

The Python script will need the uncertainty quantification configuration we just entered. SNAP creates a small Python module called 'parametric' that contains this configuration. The first line imports this module.

40. Add the following lines to the script.

```
from snap import streams
from snap.codes.trace import *
from pypost.codes.trace import *
from pypost.codes.aptplot import *
from snap import streams
```

We're going to use SNAP streams, TRACE models and jobs, and PyPost, so the script will need to import those modules.

41. Add the following line to the script.

```
stream = streams.get_stream()
```

To update properties or add new jobs we'll need to get a reference to the currently running stream. This reference is typically cached to make the script more readable.

42. Add the following line to the script.

```
w4loop = open_model(stream.get_bundled_file("UQ_W4Loop.med"))
```

The TRACE model and restart case we're working with will have to be opened to apply the generated variates, which we'll do next.

43. Add the following lines to the script.

```
for row in parametric.get_table():
    row.apply_values(w4loop)
```

This two lines of code loops through every row in the Uncertainty Quantification "Table" that we configured using the user interface earlier. Each "row" contains a set of variates that can be applied to one or more models. In this case, we're applying them to the base model.

44. Add the following lines to the script.

```
    base = TraceActor(row.new_task_name("Base_Job"), tracin=w4loop)

    restart = TraceActor(row.new_task_name("Restart_Job"),
                         tracin=w4loop.case('Null-Transient'),
                         trcrst=base.trctpr)

    stream.add([base, restart])

stream.wait()
```

This portion of the script creates the base and restart TRACE jobs and adds them to the stream in much the same fashion as previous exercises. It then waits for the entire set of jobs to finish before continuing.

45. Add the following line to the script.

```
for row in parametric.get_table():
```

The script loops through the table rows again, this time to populate the figure of merit values.

46. Add the following lines to the script.

```
    file = (row.search().label_eq("trcxtv")
                        .task_name_contains("Restart_Job")
                        .result())
```

These lines use the Uncertainty Table's search feature to find the plot output of the restart job corresponding to this row. The search is initialized to find files created by models that this row has been applied to. Adding the file label and task name criteria narrows the results to the plot output we're looking for.

47. Add the following lines to the script.

```
    if file is None or not file.location.exists():
        get_logger().error("Row {} did not produce a plot file."
                           .format(row.row_index))
        continue
```

Better safe than sorry: This section adds a readable error message if TRACE fails to execute for some reason.

48. Add the following lines to the script.

```
    file_index = TRACE.openPlotFile(file.location)

    tsurfi_01 = TRACE.getData(file_index, 'tsurfi-140A01')
    tsurfi_02 = TRACE.getData(file_index, 'tsurfi-140A02')
    tsurfi_03 = TRACE.getData(file_index, 'tsurfi-140A03')

    end_time = tsurfi_01.maxXval()

    max_temperature = max(tsurfi_01.yvalAt(end_time),
                          tsurfi_02.yvalAt(end_time),
                          tsurfi_03.yvalAt(end_time))

    row.set_fom_value("FuelTemp", max_temperature)
    TRACE.closeAll()
```

*This section uses PyPost to parse the plot file generated by each TRACE execution and extract the necessary values. For this exercise, the maximum center-line temperature of heat structure 140 at the end of the calculation will be stored as the figure of merit "FuelTemp" using the row's **set_fom_value** method.*

49. Add the following line to the script.

```
parametric.get_table().generate_report()
```

This line creates an uncertainty quantification report job and submits it to be executed by the stream then waits for it to finish before returning.

50. Press **OK** to save the script.

The job is now ready to submit. The next steps will submit the job to the Calculation Server, and examine the generated report.

51. Right-click the **UQ_Stream** node in the Navigator and select the **Submit Stream to Local** menu item.
52. Press **OK** in the confirmation dialog and wait for the stream to complete.
53. Once the stream is complete, expand the **UQ_Stream** node in the Job Status Navigator.
54. Select the **Uncertainty_Report** job in the table.
55. Press the View Files (🗐) button on the toolbar and select:
    Documents → report – uq_report.docx

This completes Exercise 6.