An introduction to the Python interface to OrcaFlex

David Heffernan
www.orcina.com/

April 2016

1 Overview

OrcaFlex (www.orcina.com/SoftwareProducts/OrcaFlex/), developed by Orcina Ltd. (www.orcina.com/), is a finite element analysis tool used primarily for offshore engineering. OrcaFlex has a powerful graphical user interface (GUI) that makes tasks such as model development very efficient. The program also offers a number of methods to automate its use. These range from Excel based automation facilities to the low-level programming interface, known as OrcFxAPI.

Interfaces with a number of different programming languages are available. At the time of writing these languages are: C, C++, C#, Delphi, Matlab and Python. This document provides a basic introduction to the Python interface to OrcaFlex.

1.1 OrcFxAPI, the OrcaFlex programming interface

The programming interface, OrcFxAPI, is implemented as a Windows DLL which exposes its capabilities through the functions that it exports. These functions are comprehensively documented, but it can be rather intimidating to call them directly. Typically that would be done using a programming language like C or C++, languages that have a rather steep learning curve. Furthermore, the raw functionality exposed by the OrcFxAPI DLL is quite tedious to use. The programmer must take care of memory allocation and deallocation and write error checking code. These aspects of development are somewhat tiresome and it is very common, especially for non-expert programmers, for programming mistakes (memory leaks, missing error handling, etc.) to be made.

To make life easier for the consumer of OrcFxAPI, high-level interfaces from C#, Matlab and Python are provided. These are built on top of the low-level OrcFxAPI interface but shield the programmer from all the tedious aspects discussed above. Memory allocation is performed by the high-level interface in a manner that is transparent to the programmer. Error handling is taken care of by the high-level interface by converting the low-level error codes into meaningful exceptions.

So, as a general rule, if your needs are met by one of the high-level interfaces, it should be more productive than direct use of the low-level interface. For the rest of this document we consider only the Python interface. However, the overall concepts that are discussed do apply equally to the other high-level interfaces, modulo any differences between language syntax.

1.2 Python

Python (www.python.org/) is a high-level, general-purpose programming language. The language is mature, dating from 1991, and is widely used in many different fields of application. Python is open source and free. A large number of extension packages are available that can be used to perform diverse tasks such as numerical analysis, signal processing, parallel processing, GUI programming, image processing, database programming, etc. Python is commonly used for scientific programming and tends to be very well documented.

The development of the Python interface to OrcaFlex originated as a personal project of mine, motivated by my desire to learn Python. After introducing early versions of the interface to my colleagues in our development team, we very rapidly adopted Python as the language of choice for writing OrcaFlex test code. The OrcaFlex automated test suite is now written entirely in Python, a testament to the versatility of the Python interface.

1.3 Python versions

There are two incompatible flavours of the Python language, Python 2 and Python 3. The main difference between them that is relevant to this document is the `print` statement or function. In Python 2 `print` is a statement.

```
print 'Hello'
```

In Python 3 `print` is a function.

```
print('Hello')
```

This does mean that code written for Python 2 may not execute on Python 3, and vice versa. This document targets Python 3 and so uses `print` as a function. If you attempt to execute any of the code samples that follow on Python 2 then they are likely to behave incorrectly, and so we do recommend the use of Python 3 when executing this document's code.

Note that OrcaFlex itself has full support for both Python 2 and Python 3.

1.4 Presentation of code in this document

Source code in this document is presented inside a box like so:

```
print('Some output')
```

Sometimes we present an entire program in its entirety, and other times we will intersperse the program with explanatory text:

```
print('Some more output')
```

After the entire program has been presented, the output is shown, indicated by a vertical line to the left of the output:

```
print('The end, there is no more')
```

```
Some output
Some more output
The end, there is no more
```

The Python source files presented in this document are available separately.

1.5 A first program

Before getting bogged down in the detail, let us at least write a simple program to demonstrate what can be done with the Python interface to OrcaFlex. The first thing that we shall do is import the `OrcFxAPI` module that provides the interface to OrcaFlex.

```
import OrcFxAPI
```

We create an instance of the `Model` class, which represents the entire OrcaFlex model.

```
model = OrcFxAPI.Model()
```

It would be common at this point to load an existing OrcaFlex base model. But instead we demonstrate that models can be built in code. Whilst you might not wish to build large complex models this way, you can use this capability to load existing models and make modifications. Here we create a vessel object and a line object.

```
vessel = model.CreateObject(OrcFxAPI.otVessel)
line = model.CreateObject(OrcFxAPI.otLine)
```

We can set some data for these objects. We will connect End A of the line to the vessel, offset that connection, and also set the position of End B.

```
line.EndAConnection = vessel.Name
line.EndAX = 45.0
line.EndAZ = -5.0
line.EndBX = 110.0
line.EndBZ = -30.0
# skip setting Y values of position, leave at default value of zero
```

OrcaFlex users with experience of automation using the OrcaFlex batch script language will recognise these names as the line's data names for connection and position.

We can now perform a calculation. We run a simulation with default stage duration and environmental conditions.

```
model.RunSimulation()
```

Finally, let us extract and output some results – the maximum and minimum top tension for the line.

```
tension = line.TimeHistory('Effective Tension',
    objectExtra=OrcFxAPI.oeEndA)
print('Max tension =', max(tension))
print('Min tension =', min(tension))
```

```
Max tension = 91.157585144
Min tension = 9.47248744965
```

2 OrcaFlex object model

OrcaFlex has a structure and hierarchy for its objects and their data that maps over to the Python interface. In order to use the Python interface it is therefore important to understand this structure. To help illustrate, figure 1 shows the OrcaFlex model browser with the model created by the code from section 1.5.

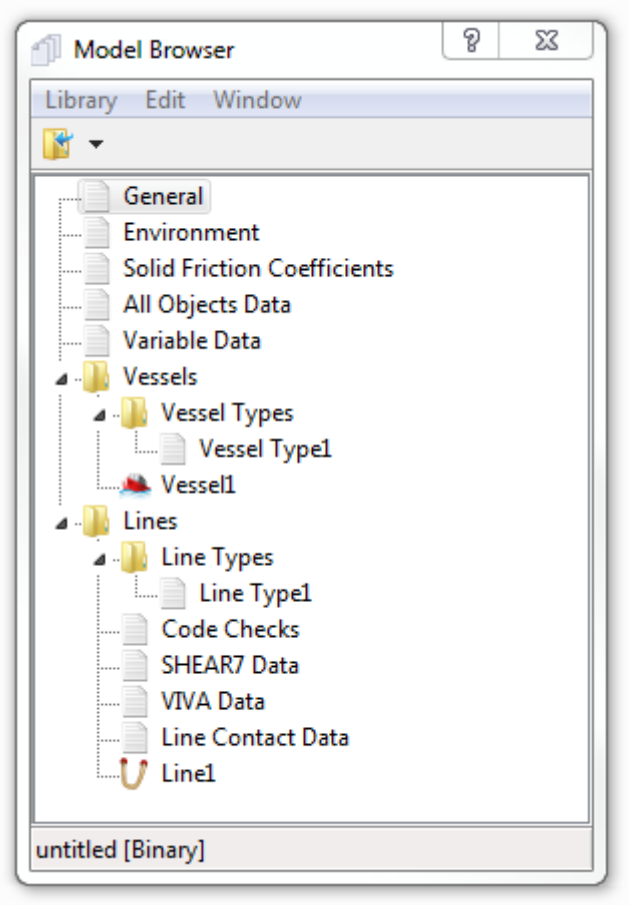


Figure 1: OrcaFlex model browser showing object hierarchy

2.1 Enumerating objects

We can now re-create that object structure, and output it using the `objects` property of the model.

```
import OrcFxAPI

model = OrcFxAPI.Model()
vessel = model.CreateObject(OrcFxAPI.otVessel)
line = model.CreateObject(OrcFxAPI.otLine)
for obj in model.objects:
    print(obj)
```

```
<General Data: 'General'>
<Environment Data: 'Environment'>
<Solid Friction Coefficients: 'Solid Friction Coefficients'>
<Line Contact Data: 'Line Contact Data'>
<Code Checks: 'Code Checks'>
<SHEAR7 Data: 'SHEAR7 Data'>
<VIVA Data: 'VIVA Data'>
```

```
<Vessel: 'Vessel1'>
<Line: 'Line1'>
<Line Type: 'Line Type1'>
<Vessel Type: 'Vessel Type1'>
```

The argument passed to `CreateObject` specifies the *type* of object that is created. Here we create a vessel and a line. Note that OrcaFlex automatically also creates vessel type and line type objects that are needed by the newly created vessel and line objects.

The hierarchy that is shown by the model browser is not reflected by the Python model, which presents a flat Python tuple of objects. These objects can be thought of as falling into two main categories.

- Objects that are created automatically, always exist, and have data that applies to the entire model. The General, Environment, Solid Friction Coefficients, Line Contact, Code Checks, SHEAR7 and VIVA objects are in this category. These objects cannot be destroyed and cannot be renamed.
- Objects that are created by the user, such as Vessel and Line objects. Models may contain any number of such objects which can be both created and destroyed. Their names are chosen by the user.

2.2 Destroying objects

We've already seen how to create objects using the `CreateObject` method. They are destroyed by calling `DestroyObject`.

```
import OrcFxAPI

model = OrcFxAPI.Model()
line = model.CreateObject(OrcFxAPI.otLine)

# destroy object by passing reference to DestroyObject method ...
model.DestroyObject(line)

# ... or by passing the object name if we don't have a reference
# available, for example the automatically created line type
model.DestroyObject('Line Type1')
```

2.3 Referring to objects by name

Objects can also be referred to by name. This is widely used when operating on models that have been built in the GUI and subsequently loaded by the Python interface.

```
import OrcFxAPI

model = OrcFxAPI.Model()
line = model.CreateObject(OrcFxAPI.otLine)

# set the line's name and some section data
line.Name = 'Riser'
line.Length = 87.0, 45.0
line.TargetSegmentLength = 5.0, 2.0

# hard-coded name
print('Riser length =', model['Riser'].Length)
print('Riser segment length =', model['Riser'].TargetSegmentLength)
```

```
# line type name read from section data
lineTypeName = line.LineType[0]
lineType = model[lineTypeName]
print(lineTypeName, 'EA =', lineType.EA)
```

```
Riser length = (87.0, 45.0)
Riser segment length = (5.0, 2.0)
Line Type1 EA = 700000.0
```

The two most commonly used automatically created objects can be accessed directly rather than having to look them up by name.

```
import OrcFxAPI

model = OrcFxAPI.Model()

# we can access the general data and environment data by name
print(model['General'].ImplicitConstantTimeStep)
print(model['Environment'].WaveHeight)

# but it is more convenient to do it this way
print(model.general.ImplicitConstantTimeStep)
print(model.environment.WaveHeight)
```

```
0.1
7.0
0.1
7.0
```

2.4 Selecting objects of a specific type

It is quite common to want to operate on all objects of a specific type, for instance all lines, all vessels, etc. The `type` attribute of an object allows us to determine the OrcaFlex type of a specific object. This attribute is an integer value that identifies the type of the object. This `type` attribute should be compared to pre-defined values like `otVessel`, `otLine`, `otLineType` and so on. These are the same pre-defined values that we use when creating objects.

```
import OrcFxAPI

model = OrcFxAPI.Model()
# create two vessels ...
for i in range(2):
    model.CreateObject(OrcFxAPI.otVessel)
# ... and three lines
for i in range(3):
    model.CreateObject(OrcFxAPI.otLine)

# for loop to perform action on all lines
for obj in model.objects:
    if obj.type == OrcFxAPI.otLine:
        print('(1)', obj.name)

# for loop to create list of vessels
vessels = []
for obj in model.objects:
```

```
    if obj.type == OrcFxAPI.otVessel:
        vessels.append(obj)
print('(2)', vessels)

# list comprehension to do the same more concisely
vessels = [obj for obj in model.objects if obj.type ==
    OrcFxAPI.otVessel]
print('(3)', vessels)

# The type attribute returns an integer value
print('(4)', vessels[0].type, OrcFxAPI.otVessel)

# do not confuse type attribute with built-in type() function
print('(5)', type(vessels[0]))
```

```
(1) Line1
(1) Line2
(1) Line3
(2) [<Vessel: 'Vessel1'>, <Vessel: 'Vessel2'>]
(3) [<Vessel: 'Vessel1'>, <Vessel: 'Vessel2'>]
(4) 5 5
(5) <class 'OrcFxAPI.OrcaFlexVesselObject'>
```

3 OrcaFlex data model

We've already seen some simple examples of accessing OrcaFlex input data. We will now look at this topic in some more depth.

As mentioned earlier, data are identified by names – the same names used by OrcaFlex batch script. These data names are not listed in the OrcFxAPI Python module that implements the interface and instead are held only internally to the OrcFxAPI DLL. Data access is therefore a *dynamic* process. You ask for the data for a given name, specified as a string, and OrcFxAPI looks for that name in its internal lists of data item names.

3.1 Non-indexed data access

Data can be read by the `GetData` method, and modified by the `SetData` method.

```
import OrcFxAPI

model = OrcFxAPI.Model()
print(model.environment.GetData('WaveHeight', -1))
model.environment.SetData('WaveHeight', -1, 2.5)
print(model.environment.GetData('WaveHeight', -1))
```

```
7.0
2.5
```

The `-1` used above specifies the index. This is needed for data items that form part of a table in OrcaFlex. But the data item we are accessing here, wave height, does not appear in a table. Notice that in figure 2 the data appears in a control that has a single editable row. This is what we refer to as a *non-indexed* data item. It is also worth pointing out that the name used to identify the data appears in the popup hint in figure 2. These data names can also be inspected by right-clicking on the data form and selecting the *data names* menu item, or by pressing `F7`.

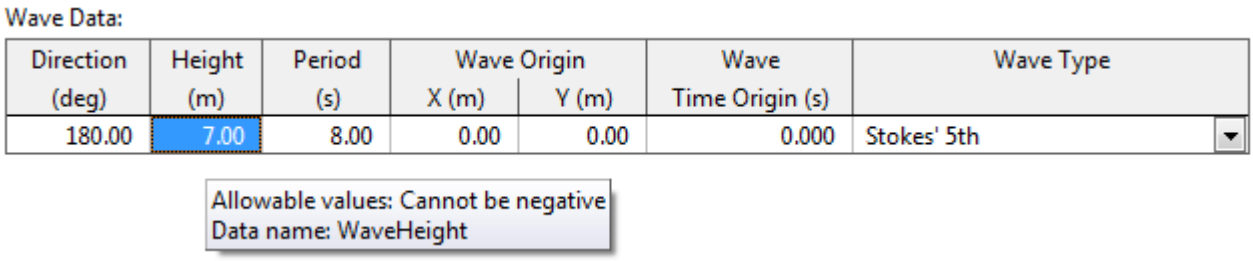


Figure 2: Wave height data item, an example of non-indexed data

The `GetData` and `SetData` methods still require an index to be passed, for both indexed and non-indexed data items. For non-indexed data items that index is ignored and anything can be passed. By convention we use `-1` to indicate to readers of the code that this data item is non-indexed. If for some reason we have made a mistake, and passed the name of an indexed data item, then `-1` cannot be a valid index and so an error will be raised. So the use of `-1` does have the advantage of protecting us from one potential form of programming error.

You will have noticed that this form of data access is more verbose than that which we have seen up until now. We can instead write the previous program using more concise syntax.

```
import OrcFxAPI

model = OrcFxAPI.Model()
print(model.environment.WaveHeight)
model.environment.WaveHeight = 2.5
print(model.environment.WaveHeight)
```

```
7.0
2.5
```

This syntax takes advantage of the *dynamic* nature of the Python language. Normally `environment.WaveHeight` would indicate that `WaveHeight` was a defined attribute of the `environment` object. However, it is possible to override attribute lookup for Python objects and that is how this concise syntax is implemented. When an attribute is not recognised, the object asks the OrcFxAPI DLL if it recognises the attribute's name as a data item. If the name is recognised, then the code is transformed into a call to `GetData` or `SetData`.

By and large this second variant is preferred over directly calling `GetData` or `SetData`. The main exception to that rule is when the data name is only known at runtime, and not when the code was written. This scenario is rather rare and probably occurs most commonly in our own internal OrcaFlex test code.

3.2 Indexed data access

Indexed data items are those that are presented in tables with more than one editable row. We will use the stage duration data (see figure 3) for illustration.

Indexed data can be accessed using `GetData` or `SetData` in the expected manner, by providing an index instead of the dummy value of `-1` that we use for non-indexed data. The concise syntax using the data name as an attribute provides the index via the Python index operator, `[]`. The interface also allows for the row count to be read or modified, for rows to be inserted or deleted, and for entire columns of data to be read or modified.

Indexing is always zero-based which means that the first item has index `0`. This choice was made to match the convention used by Python. The choice of zero-based or one-based indexing provides

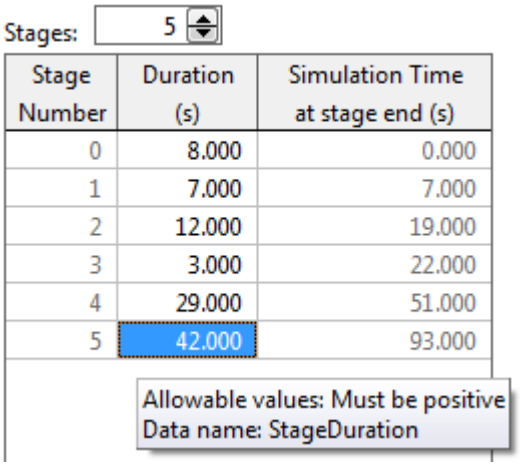


Figure 3: Stage duration data item, an example of indexed data

potential for confusion. Not all OrcaFlex automation methods make the same choice. We have opted to make the choice on a case-by-case basis. So, the Python, C# and high-level C++ interfaces all use zero-based indexing. On the other hand, OrcaFlex batch script and text data files use one-based indexing, as do the Matlab and low-level C++ interfaces.

```python
import OrcFxAPI

model = OrcFxAPI.Model()
general = model.general

# two ways to obtain the count, we prefer the first option
print('(1) Stage count =', len(general.StageDuration))
print('(2) Stage count =', general.GetDataRowCount('StageDuration'))

# modify the count
general.SetDataRowCount('StageDuration', 4)
print('(3) Stage count =', len(general.StageDuration))

# get and set the entire column of data
print('(4) Stage duration =', general.StageDuration)
general.StageDuration = 8, 7, 12, 3, 29, 42 # sets count and values
print('(5) Stage duration =', general.StageDuration)

# get items by index, note Python convention for negative indexing
print('(6) build-up duration =', general.StageDuration[0])
print('(7) final stage duration =', general.StageDuration[-1])

# delete and insert rows
general.StageDuration.DeleteRow(1)
general.StageDuration.InsertRow(3)
general.StageDuration[3] = 25
print('(8) Stage duration =', general.StageDuration)

# iterate over the values
for index, value in enumerate(general.StageDuration):
    print('(9) Stage', index, 'duration =', value)
```

```
(1) Stage count = 2
(2) Stage count = 2
(3) Stage count = 4
(4) Stage duration = (8.0, 16.0, 16.0, 16.0)
(5) Stage duration = (8.0, 7.0, 12.0, 3.0, 29.0, 42.0)
```

```
(6) build-up duration = 8.0
(7) final stage duration = 42.0
(8) Stage duration = (8.0, 12.0, 3.0, 25.0, 29.0, 42.0)
(9) Stage 0 duration = 8.0
(9) Stage 1 duration = 12.0
(9) Stage 2 duration = 3.0
(9) Stage 3 duration = 25.0
(9) Stage 4 duration = 29.0
(9) Stage 5 duration = 42.0
```

It is worth elaborating on the interpretation of negative indices, as seen in `StageDuration[-1]`. Negative indices are interpreted as counting from the end of the array. So index `-1` is the last item, `-2` is the second to last, and so on. Do note that this convention applies when you use the indexing operator `[]` but not when calling `GetData` and `SetData`.

As a rule, data that is presented in OrcaFlex in a control with multiple editable rows is indexed data. However, there are a few notable exceptions. Line type and clump type data are presented in two modes. The data can be viewed either in *all* or *individual* mode. When viewed in *all* mode, the data are shown in a table with one row for each line type or clump type. Whilst these data therefore might appear to be indexed data, they are in fact non-indexed data.

```python
import OrcFxAPI

model = OrcFxAPI.Model()
lineType1 = model.CreateObject(OrcFxAPI.otLineType)
lineType2 = model.CreateObject(OrcFxAPI.otLineType)
print(lineType1.Name)
print(lineType2.Name)
```

```
Line Type1
Line Type2
```

3.3 Selection

Some input data require yet another technique to access them – selection. We will demonstrate this using wave train data (figure 4), but the concepts apply to other forms of data that require selection.

The data in the wave trains group box determine how many wave trains there are, and their names. The rest of the data relating to waves applies to the *selected* wave train. In the figure there are three wave trains, somewhat unimaginitively named *Wave1*, *Wave2* and *Wave3*. The highlighted wave train, *Wave2*, is the selected wave train, and any wave train specific data access refers to the data of that selected wave train.

In order to access wave train specific data programmatically, we must first select a wave train.

```python
import OrcFxAPI

model = OrcFxAPI.Model()
environment = model.environment

# define two wave trains, with informative names
environment.WaveName = 'Swell', 'Wind generated'

# select the swell wave train, and set its data
environment.SelectedWave = 'Swell'
environment.WaveType = 'Single Airy'
environment.WaveDirection = 140.0
```

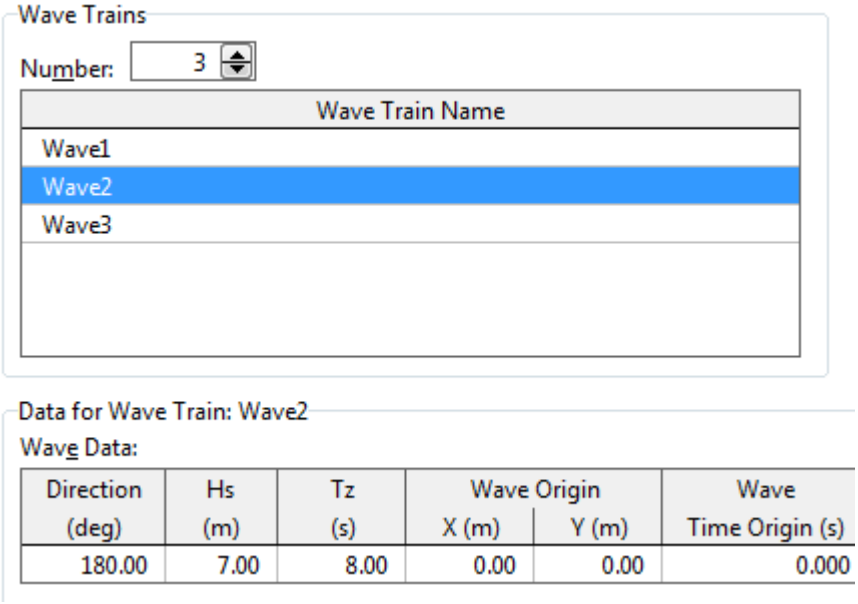


Figure 4: Wave train data

```
environment.WaveHeight = 4.5
environment.WavePeriod = 17.0

# select the wind generated wave train, and set its data
environment.SelectedWave = 'Wind generated'
environment.WaveType = 'JONSWAP'
environment.WaveDirection = 75.0
environment.WaveHs = 3.0
environment.WaveTp = 9.0
```

Again, OrcaFlex users familiar with batch process should be familiar with the concept of selection. In OrcaFlex batch script the selection of wave trains uses the `Select` command.

```
...
Select Environment
  Select Wave 'Wind generated'
    WaveType = JONSWAP
    WaveDirection = 75.0
...
```

The data name `SelectedWave` is not easy to discover from OrcaFlex. There is no visible data item in the program that has that name. So, how are you supposed to discover this name, and other selection data names? The documentation for batch script (`www.orcina.com/SoftwareProducts/OrcaFlex/Documentation/Help/Redirector.htm?BatchProcessing,Examplesofsettingdata.htm`) describes various forms of the `Select` batch script command that are used for data that require selection. For example:

```
...
Select Environment
  Select Current Current1
    // set some data for this current
...
Select 'SHEAR7 data'
  Select SHEAR7SNCurve Curve2
    // set some data for this SHEAR7 S-N curve
```

```
...
Select 'Line Contact Data'
  Select PenetratorLocationsDataSet Locations1
    // set some data for this penetrator locations data set
...
```

These selection commands are all of the form `Select DataType Name`. The batch script interpreter transforms such commands into an assignment of the form `SelectedDataType = Name`. This implies the existence of a data name of the form `SelectedDataType`. Once you understand this rule, you can use it from your Python code.

For instance, with current data sets the batch script selection command from above is `Select Current Current1`. So that means that `DataType` is `Current` and so the corresponding selection data name is `SelectedCurrent`.

```python
import OrcFxAPI

model = OrcFxAPI.Model()
environment = model.environment
environment.MultipleCurrentDataCanBeDefined = 'Yes'

# define two current data sets, with rather useless names
environment.CurrentName = 'Current1', 'Current2'

# select the first current data set, and set the speed
environment.SelectedCurrent = 'Current1'
environment.RefCurrentSpeed = 0.3

# select the second current data set, and set the speed
environment.SelectedCurrent = 'Current2'
environment.RefCurrentSpeed = 0.5
```

3.4 Special values, default and infinity

Some data items in OrcaFlex can accept what we term *special values*. For example, line end connection stiffness can have the value *Infinity*. Use `OrcinaInfinity()` to obtain the floating point value that represents this special value. The full list of special values is:

- `OrcinaDefaultReal()` for the floating point value displayed as '~'.
- `OrcinaDefaultWord` for the integer value displayed as '~'.
- `OrcinaInfinity()` for the floating point value displayed as '*Infinity*'.
- `OrcinaUndefinedReal()` for the floating point value displayed as '?'.
- `OrcinaNullReal()` for the floating point value displayed as '*N/A*'.

4 Saving and loading files

Whilst it is possible to build complete models, perform analyses, and extract results, it is more usual that existing input files are loaded, modified input files saved, existing simulation files loaded and so on. For instance, one might use the OrcaFlex batch processing functionality, or Distributed OrcaFlex, to produce simulation files, but then use Python code to perform the post-processing on these simulation files.

4.1 Loading

To load OrcaFlex files, use the `LoadData` and `LoadSimulation` methods.

```
import OrcFxAPI

model = OrcFxAPI.Model()
model.LoadData('model.dat') # binary data file
model.LoadData('model.yml') # text data file, YAML
model.LoadData('model.sim') # load the data from a simulation file
model.LoadSimulation('model.sim') # load entire simulation file
```

Because it is very common to create a model and load a file immediately, the constructor for the `Model` class accepts a file name parameter.

```
import OrcFxAPI

model = OrcFxAPI.Model('model.dat') # binary data file
model = OrcFxAPI.Model('model.yml') # text data file, YAML
model = OrcFxAPI.Model('model.sim') # simulation file
```

4.2 Saving

To save files, use the `SaveData` and `SaveSimulation` methods.

```
import OrcFxAPI

model = OrcFxAPI.Model('model.sim')
model.SaveData('model.dat') # binary data file
model.SaveData('model.yml') # text data file, YAML
model.SaveSimulation('model.sim') # save the simulation file
```

When saving data the file extension determines whether to save a binary data file or a text data file.

5 Performing analyses

The primary analyses offered by OrcaFlex are static analysis and dynamic analysis. Whilst OrcFxAPI does support other forms of analysis such as modal analysis, fatigue analysis, etc. we will only cover statics and dynamics here. More details can be found in the documentation.

Static analysis is invoked by the `CalculateStatics` method.

```
import OrcFxAPI

model = OrcFxAPI.Model()
line = model.CreateObject(OrcFxAPI.otLine)

model.CalculateStatics()
print('(1) Model state =', model.state)
```

For dynamic analysis call `RunSimulation`.

```
model.RunSimulation()
print('(2) Model state =', model.state)
```

Call the `Reset` method to bring the model back to reset state.

```
model.Reset()
print('(3) Model state =', model.state)
```

Note that you can call `RunSimulation` from reset state without having to perform the static calculation explicitly. The static calculation is first performed by OrcFxAPI, implicitly, and then the dynamic calculation is performed.

```
model.RunSimulation()
print('(4) Model state =', model.state)
```

Note that modifying data will also result in the model being reset.

```
line.Length = 120.0,
print('(5) Model state =', model.state)
```

```
(1) Model state = InStaticState
(2) Model state = SimulationStopped
(3) Model state = Reset
(4) Model state = SimulationStopped
(5) Model state = Reset
```

These functions to perform analyses can take considerable time to complete for complex models. Because of this these functions provide facilities for reporting progress and cancelling. In order to keep the exposition relatively simple, we omit any discussion of these details here.

6 Extracting results

Now that we know how to load, build and modify models, and how to perform analyses, it remains to learn how to query for results. All of the results that are available in OrcaFlex are available from OrcFxAPI.

We will begin by looking at time history results. These are obtained by calling the `TimeHistory` method specifying the desired result variable by name. The method has the following signature:

```
TimeHistory(varNames, period=None, objectExtra=None)
```

6.1 Specifying simulation period

We will start by looking at the `period` argument that specifies the simulation period.

```
import OrcFxAPI

model = OrcFxAPI.Model()
```

```
vessel = model.CreateObject(OrcFxAPI.otVessel)
model.RunSimulation()

# no period specified, defaults to whole simulation
X = vessel.TimeHistory('X')
print('(1) no period specified', min(X), max(X))

# specify a period, whole simulation, equivalent to previous call
X = vessel.TimeHistory('X', OrcFxAPI.pnWholeSimulation)
print('(2) whole simulation', min(X), max(X))

# build-up, stage 0
X = vessel.TimeHistory('X', 0)
print('(3) build-up', min(X), max(X))

# stage 1
X = vessel.TimeHistory('X', 1)
print('(4) stage 1', min(X), max(X))

# latest wave
X = vessel.TimeHistory('X', OrcFxAPI.pnLatestWave)
print('(5) latest wave', min(X), max(X))

# period specified by time
X = vessel.TimeHistory('X', OrcFxAPI.SpecifiedPeriod(1.0, 2.0))
print('(6) 1.0s to 2.0s', min(X), max(X))
print('(7) 1.0s to 2.0s', X)
```

```
(1) no period specified -0.220513388515 0.235916048288
(2) whole simulation -0.220513388515 0.235916048288
(3) build-up -0.121314510703 0.190882235765
(4) stage 1 -0.220513388515 0.235916048288
(5) latest wave -0.220513388515 0.235916048288
(6) 1.0s to 2.0s -0.220513388515 -0.182141140103
(7) 1.0s to 2.0s [-0.21700118 -0.21944398 -0.22051339 -0.22021785
-0.21857354 -0.21560377
 -0.21133871 -0.20581487 -0.1990746  -0.19116575 -0.18214114]
```

The `TimeHistory` method returns either a tuple, or if the `numpy` (www.numpy.org/) module is available, a `numpy` array object.

6.2 Sample times

It is common to require the sample times corresponding to a time history result. We could request a time history for the *Time* variable of the general object. But it is more idiomatic to use the `SampleTimes` method of the model object.

```
import OrcFxAPI

model = OrcFxAPI.Model()
vessel = model.CreateObject(OrcFxAPI.otVessel)
model.RunSimulation()

period = OrcFxAPI.SpecifiedPeriod(1.0, 2.0)
times = model.SampleTimes(period)
X = vessel.TimeHistory('X', period)
for t, x in zip(times, X):
    print('t={:4.1f}s, X={:7.4f}m'.format(t, x))
```

```
t= 1.0s, X=-0.2170m
t= 1.1s, X=-0.2194m
t= 1.2s, X=-0.2205m
t= 1.3s, X=-0.2202m
t= 1.4s, X=-0.2186m
t= 1.5s, X=-0.2156m
t= 1.6s, X=-0.2113m
t= 1.7s, X=-0.2058m
t= 1.8s, X=-0.1991m
t= 1.9s, X=-0.1912m
t= 2.0s, X=-0.1821m
```

For convenience, objects also provide a `SampleTimes` method so in the program above the following two lines would be equivalent.

```
times = model.SampleTimes(period)
times = vessel.SampleTimes(period)
```

6.3 Extracting results for multiple variables

The `TimeHistory` method allows multiple variable names to be passed, and returns the time histories in column stacked form.

```
import OrcFxAPI

model = OrcFxAPI.Model()
vessel = model.CreateObject(OrcFxAPI.otVessel)
model.RunSimulation()

period = OrcFxAPI.SpecifiedPeriod(1.0, 2.0)
varNames = 'X', 'Y', 'Z'
pos = vessel.TimeHistory(varNames, period)
print(pos)
```

```
[[-0.21700118  0.         -0.14714514]
 [-0.21944398  0.         -0.19857015]
 [-0.22051339  0.         -0.24879079]
 [-0.22021785  0.         -0.29748428]
 [-0.21857354  0.         -0.34433767]
 [-0.21560377  0.         -0.38905022]
 [-0.21133871  0.         -0.43133539]
 [-0.20581487  0.         -0.4709231 ]
 [-0.1990746   0.         -0.50756156]
 [-0.19116575  0.         -0.54101902]
 [-0.18214114  0.         -0.57108533]]
```

The first row of this array contains the X, Y and Z coordinates of the vessel at the first sample time. The second row contains the coordinates at the second sample time, and so on.

This example can be extended to demonstrate the clean and concise code that is possible using `numpy` with OrcFxAPI. We'll add a second vessel, and show how to calculate the distance between the two vessels.

```
import numpy
import OrcFxAPI

```

```
model = OrcFxAPI.Model()
model.environment.WaveType = 'JONSWAP' # use irregular wave
model.environment.WaveDirection = 130.0
vessel1 = model.CreateObject(OrcFxAPI.otVessel)
vessel1.InitialX, vessel1.InitialY = 0.0, 0.0
vessel2 = model.CreateObject(OrcFxAPI.otVessel)
vessel2.InitialX, vessel2.InitialY = 80.0, -25.0
model.RunSimulation()

period = OrcFxAPI.SpecifiedPeriod(1.0, 2.0)
varNames = 'X', 'Y', 'Z'
pos1 = vessel1.TimeHistory(varNames, period)
pos2 = vessel2.TimeHistory(varNames, period)
distance = numpy.linalg.norm(pos1 - pos2, axis=1)
times = model.SampleTimes(period)
for t, d in zip(times, distance):
    print('t={:4.1f}s, d={:7.3f}m'.format(t, d))
```

```
t= 1.0s, d= 82.132m
t= 1.1s, d= 82.074m
t= 1.2s, d= 82.021m
t= 1.3s, d= 81.973m
t= 1.4s, d= 81.930m
t= 1.5s, d= 81.892m
t= 1.6s, d= 81.861m
t= 1.7s, d= 81.835m
t= 1.8s, d= 81.816m
t= 1.9s, d= 81.804m
t= 2.0s, d= 81.798m
```

6.4 Object extra

So far we have considered relatively simple results. But many results require more information to be specified. The most common forms of this additional information are as follows:

- For many environment results you must specify a position in space at which the results are reported.
- For line results you must specify the point along the line where results are reported. This can be *End A*, *End B*, an arc length or a node number. For some results you may also need to specify radial and circumferential position. For clearance results you may specify the name of another line and clearances are reported from that other line.
- For 6D buoys and vessels the translational position, velocity and acceleration results are reported at a specified position on the object.

This additional information is provided in the `objectExtra` parameter of the `TimeHistory` method. This parameter accepts an instance of the `ObjectExtra` class. Whilst you can instantiate and populate such an instance directly, helper functions are provided to simplify the task.

```
import OrcFxAPI

model = OrcFxAPI.Model()
environment = model.environment
vessel = model.CreateObject(OrcFxAPI.otVessel)
line = model.CreateObject(OrcFxAPI.otLine)
line.EndAConnection = vessel.Name
```

```
line.EndAX, line.EndAY, line.EndAZ = 40.0, 0.0, 5.0
line.EndBX, line.EndBY, line.EndBZ = 70.0, 0.0, -25.0
model.RunSimulation()

period = OrcFxAPI.pnLatestWave

# sea elevation at position 5, 0, -10
tmp = environment.TimeHistory('Elevation', period,
    OrcFxAPI.oeEnvironment(5.0, 0.0, -10.0))

# vessel velocity at a specified point in local vessel axes
tmp = vessel.TimeHistory('Velocity', period,
    OrcFxAPI.oeVessel(-10.0, 0.0, 3.0))

# line results at End A and End B ...
tmp = line.TimeHistory('Effective Tension', period, OrcFxAPI.oeEndA)
tmp = line.TimeHistory('Effective Tension', period, OrcFxAPI.oeEndB)

# ... at specified arc length or node number
tmp = line.TimeHistory('Effective Tension', period,
    OrcFxAPI.oeArcLength(15.0))
tmp = line.TimeHistory('X', period, OrcFxAPI.oeNodeNum(3))

# if you wish to omit period and default to whole simulation, you
# can use positional parameters, pass None for the period ...
tmp = line.TimeHistory('Effective Tension', None, OrcFxAPI.oeEndA)
# ... or you can use named parameters
tmp = line.TimeHistory('Effective Tension',
    objectExtra=OrcFxAPI.oeEndA)
```

We have only covered a minority of these helper functions. The full list, at the time of writing, is:

- `oeEnvironment`
- `oeBuoy`
- `oeWing`
- `oeVessel`
- `oeConstraint` (from version 10.1)
- `oeSupport`
- `oeWinch`
- `oeLine`
- `oeNodeNum`
- `oeArcLength`
- `oeEndA`
- `oeEndB`
- `oeTouchdown`

More details can be found in the documentation.

6.5 Static state results

Static state results are obtained with the `StaticResult` method. The method does not require a period to be specified for reasons that are hopefully obvious.

```
import OrcFxAPI

model = OrcFxAPI.Model()
line = model.CreateObject(OrcFxAPI.otLine)
model.CalculateStatics()

# static result available when model in static state ...
print('(1)', line.StaticResult('Effective Tension', OrcFxAPI.oeEndA))

# ... and after dynamic simulation
model.RunSimulation()
print('(2)', line.StaticResult('Effective Tension', OrcFxAPI.oeEndA))
```

```
(1) 40.3575448081
(2) 40.3575439453
```

Note that the small difference between the values is because after dynamics has been performed, static state results are calculated from the first log sample. By default the log is sampled to single precision. When in static state, the results are calculated to double precision.

Just as for time history results, if the result variable does not require an object extra parameter, it can be omitted. Again, as for time history results, static state results for multiple variables can be obtained by passing multiple variable names.

As an aside, we note that it is also possible to obtain static state results from the `TimeHistory` method by passing `pnStaticState` as the period. Note that `StaticResult` returns a scalar value, and `TimeHistory` returns an array. When `TimeHistory` is passed the period `pnStaticState` that array has length 1 and so the following two lines of code are equivalent:

```
X = obj.StaticResult(varNames, objectExtra)
Y = obj.TimeHistory(varNames, OrcFxAPI.pnStaticState, objectExtra)[0]
```

6.6 Range graphs

OrcaFlex range graphs present results for a *range* of arc lengths along a line. This means that they are only available for line objects. These results are returned by the `RangeGraph` method which has the following signature:

```
RangeGraph(varName, period=None, objectExtra=None,
    arclengthRange=None, stormDurationHours=None)
```

6.6.1 Dynamic range graphs

The first example produces a dynamic range graph of tension, over the latest wave period.

```
import OrcFxAPI

model = OrcFxAPI.Model()
vessel = model.CreateObject(OrcFxAPI.otVessel)
line = model.CreateObject(OrcFxAPI.otLine)
line.EndAConnection = vessel.Name
line.EndAX, line.EndAY, line.EndAZ = 40.0, 0.0, 5.0
line.EndBX, line.EndBY, line.EndBZ = 70.0, 0.0, -25.0
```

```
model.RunSimulation()

period = OrcFxAPI.pnLatestWave
rg = line.RangeGraph('Effective Tension', period)
print('{:>5} {:>7} {:>7} {:>7}'.format('z', 'min', 'max', 'mean'))
for z, minTe, maxTe, meanTe in zip(rg.X, rg.Min, rg.Max, rg.Mean):
    print('{:5.1f} {:7.3f} {:7.3f} {:7.3f}'.format(
        z, minTe, maxTe, meanTe))
```

```
    z     min     max    mean
  0.0  27.014  84.328  56.673
  5.0  20.501  76.252  48.963
 15.0  14.065  64.224  39.931
 25.0  10.734  51.943  31.981
 35.0   7.636  39.708  24.040
 45.0   4.635  27.689  16.171
 55.0   1.469  16.565   8.618
 65.0  -0.962   9.419   3.733
 75.0   4.609  14.163   8.944
 85.0  12.204  21.799  16.618
 95.0  19.821  29.814  24.466
100.0  23.570  33.900  28.401
```

Here we have specified the period, but if it is omitted then the default period is chosen. That is the whole simulation when dynamics results are available, or the static state if only static results are available. Specification of the period is exactly the same as for time history results.

The `RangeGraph` method returns an object containing multiple arrays that represent the different curves on an OrcaFlex range graph. The output that is produced by the code above mimics that obtained from the OrcaFlex GUI's graph values.

As well as the four attributes listed above, the range graph object also contains `StdDev`, `Upper` and `Lower` curves. Note that the latter two are not available for all result variables. If not available, these attributes will contain `None`. These `Upper` and `Lower` curves, when available, contain limits for the variable. For instance, if an allowable tension is specified then it is reported as the `Upper` curve for tension range graphs. Likewise the compression limit is reported as the `Lower` curve of a tension range graph.

The `X` curve, containing arc length values, is somewhat confusingly named. The name was originally chosen because the values were presented on the X axis of the graph in OrcaFlex. Nowadays, these arc length values can be presented on either X or Y axes. To compound the confusion we customarily use `z` to refer to arc length. This is just one of those historical oddities of the interface that you will have to get used to.

6.6.2 Static state range graphs

Range graphs for static state are similar. They are available after a dynamic analysis using the `pnStaticState` period value, but it is more common to extract such results when the model is in the static state. In that scenario you can, if you wish, omit the period argument and rely on the default.

```
import OrcFxAPI

model = OrcFxAPI.Model()
line = model.CreateObject(OrcFxAPI.otLine)
model.CalculateStatics()

rg = line.RangeGraph('Effective Tension')
print('{:>5} {:>7}'.format('z', 'Te'))
```

```
for z, Te in zip(rg.X, rg.Mean):
    print('{:5.1f} {:7.3f}'.format(z, Te))
```

```
    z      Te
  0.0  40.358
  5.0  36.393
 15.0  28.546
 25.0  20.802
 35.0  13.332
 45.0   7.087
 55.0   7.087
 65.0  13.332
 75.0  20.802
 85.0  28.546
 95.0  36.393
100.0  40.358
```

6.6.3 `objectExtra` for range graphs

One might wonder why we would ever need to pass an `ObjectExtra` object to the `RangeGraph` method. Previously we have seen `ObjectExtra` used to specify a single arc length along the line. However, some results depend upon radial position, circumferential position, clearance line name and so on. For these results we need to specify that additional information.

```
import OrcFxAPI

model = OrcFxAPI.Model()
line = model.CreateObject(OrcFxAPI.otLine)
model.CalculateStatics()

inner = line.RangeGraph('von Mises Stress', objectExtra=
    OrcFxAPI.oeLine(RadialPos=OrcFxAPI.rpInner, Theta=0.0))
outer = line.RangeGraph('von Mises Stress', objectExtra=
    OrcFxAPI.oeLine(RadialPos=OrcFxAPI.rpOuter, Theta=0.0))
print('{:>5} {:>7} {:>7}'.format('z', 'inner', 'outer'))
for z, vmsInner, vmsOuter in zip(inner.X, inner.Mean, outer.Mean):
    print('{:5.1f} {:7.2f} {:7.2f}'.format(z, vmsInner, vmsOuter))
```

```
    z   inner   outer
  0.0  856.81  856.81
  5.0  852.93  864.32
 15.0  943.20  896.01
 25.0 1199.85 1077.63
 35.0 1788.60 1852.51
 45.0 2755.77 3311.63
 55.0 2755.77 3311.63
 65.0 1788.60 1852.51
 75.0 1199.85 1077.63
 85.0  943.20  896.01
 95.0  852.93  864.32
100.0  856.81  856.81
```

6.6.4 `arclengthRange` for range graphs

The `arclengthRange` argument can be used to restrict the range of arc lengths that are reported. Whilst this could be done manually by taking the output for the full line, and removing unwanted

parts, using the `arclengthRange` argument allows the OrcFxAPI engine to optimise the results derivation. So, in addition to allowing simpler code, this can lead to significant performance benefits.

```
import OrcFxAPI

model = OrcFxAPI.Model()
line = model.CreateObject(OrcFxAPI.otLine)
model.CalculateStatics()

rg = line.RangeGraph('Effective Tension',
    arclengthRange=OrcFxAPI.arSpecifiedArclengths(15.0, 45.0))
print('{:>5} {:>7}'.format('z', 'Te'))
for z, Te in zip(rg.X, rg.Mean):
    print('{:5.1f} {:7.3f}'.format(z, Te))
```

```
    z      Te
 15.0  28.546
 25.0  20.802
 35.0  13.332
 45.0   7.087
```

Here we define the arc length range by specifying minimum and maximum arc length values. Pass `OrcinaDefaultReal()` to mean either the beginning or the end of the line.

```
# from the beginning of the line to arc length 45 ...
tmp = OrcFxAPI.arSpecifiedArclengths(
    OrcFxAPI.OrcinaDefaultReal(), 45.0)

# ... but using zero is surely a clearer way to do this
tmp = OrcFxAPI.arSpecifiedArclengths(0.0, 45.0)

# from arc length 45 to the end of the line ...
tmp = OrcFxAPI.arSpecifiedArclengths(
    45.0, OrcFxAPI.OrcinaDefaultReal())

# ... which is arguably simpler than
tmp = OrcFxAPI.arSpecifiedArclengths(45.0, line.CumulativeLength[-1])
```

As an alternative to specifying by arc length you can use `arSpecifiedSections` to specify minimum and maximum section indices.

Finally, `arEntireLine()` can be used to specify explicitly results for the entire length of the line. But this is seldom used because it is simpler to omit the argument altogether.

6.6.5 `stormDurationHours` for range graphs

The `stormDurationHours` argument is only used with frequency domain dynamics, and specifies the storm duration in hours used for the calculation of the MPM.

6.7 Linked statistics

Linked statistics results report statistics for multiple results variables. Call `LinkedStatistics` to obtain a results object that can subsequently be queried.

```
LinkedStatistics(varNames, period=None, objectExtra=None)
```

The `period` and `objectExtra` arguments should, by now, need no further explanation. The novel aspect of this result type is the object returned by `LinkedStatistics` and how it is queried.

```
import OrcFxAPI

model = OrcFxAPI.Model()
model.general.StageDuration = 10.0, 200.0
model.environment.WaveType = 'JONSWAP' # use irregular wave
vessel = model.CreateObject(OrcFxAPI.otVessel)
line = model.CreateObject(OrcFxAPI.otLine)
line.EndAConnection = vessel.Name
line.EndAX, line.EndAY, line.EndAZ = 40.0, 0.0, 5.0
line.EndBX, line.EndBY, line.EndBZ = 70.0, 0.0, -25.0
model.RunSimulation()

varNames = 'Effective Tension', 'Bend Moment', 'Curvature'
period = 1
objectExtra = OrcFxAPI.oeArcLength(25.0)
stats = line.LinkedStatistics(varNames, period, objectExtra)

# query tension and bend moment
query = stats.Query('Effective Tension', 'Bend Moment')
print(' (1) max tension =', query.ValueAtMax)
print(' (2) time of max tension =', query.TimeOfMax)
print(' (3) bend moment at this time =', query.LinkedValueAtMax)
print(' (4) min tension =', query.ValueAtMin)
print(' (5) time of min tension =', query.TimeOfMin)
print(' (6) bend moment at this time =', query.LinkedValueAtMin)

# query tension and curvature
query = stats.Query('Effective Tension', 'Curvature')
print(' (7) time of max tension =', query.TimeOfMax)
print(' (8) curvature at this time =', query.LinkedValueAtMax)
print(' (9) time of min tension =', query.TimeOfMin)
print('(10) curvature at this time =', query.LinkedValueAtMin)

# can also extract time series statistics
tss = stats.TimeSeriesStatistics('Effective Tension')
print('(11) mean =', tss.Mean)
print('(12) stddev =', tss.StdDev)
print('(13) m0 =', tss.m0)
print('(14) m2 =', tss.m2)
print('(15) m4 =', tss.m4)
print('(16) Tz =', tss.Tz)
print('(17) Tc =', tss.Tc)
print('(18) Bandwidth =', tss.Bandwidth)
```

```
 (1) max tension = 50.07413101196289
 (2) time of max tension = 164.5
 (3) bend moment at this time = 0.2841897608585272
 (4) min tension = 10.75954818725586
 (5) time of min tension = 167.8
 (6) bend moment at this time = 0.6568732168296456
 (7) time of max tension = 164.5
 (8) curvature at this time = 0.0023682480071543933
 (9) time of min tension = 167.8
(10) curvature at this time = 0.00547394347358038
(11) mean = 32.163606738043335
(12) stddev = 5.348383715390616
(13) m0 = 28.605208367055532
```

```
(14) m2 = 0.4578602459846008
(15) m4 = 0.012873914746648361
(16) Tz = 7.904166666666668
(17) Tc = 5.963636363636364
(18) Bandwidth = 0.6563083922142144
```

6.8 Time series statistics

We saw in the previous section how time series statistics can be returned from a linked statistics object. The `TimeSeriesStatistics` method returns the same information directly.

```
TimeSeriesStatistics(varNames, period=None, objectExtra=None)
```

6.9 Rainflow half cycles

Post-processing automation is commonly used for bespoke fatigue analysis, crack propagation analysis, etc. These forms of analysis depend upon the Rainflow cycle counting algorithm. OrcaFlex provides an cycle counting implementation through the `RainflowHalfCycles` method.

```
RainflowHalfCycles(varNames, period=None, objectExtra=None)
```

For the sake of a shorter example, we will cycle count some vessel results, but in real world use we'd expect to see line structural results being cycle counted.

```
import OrcFxAPI

model = OrcFxAPI.Model()
model.general.StageDuration = 10.0, 50.0
model.environment.WaveType = 'JONSWAP' # use irregular wave
vessel = model.CreateObject(OrcFxAPI.otVessel)
model.RunSimulation()

period = 1
halfCycleRanges = vessel.RainflowHalfCycles('X', period)
for halfCycleRange in halfCycleRanges:
    print(halfCycleRange)
```

```
0.159379020333
0.159379020333
0.916084051132
0.916084051132
1.54313793778
2.05583393574
2.63803243637
2.86304700375
```

When designing the automation interfaces we try to make it possible for the functionality to be made available as flexibly as possible. For instance, the function being considered at the moment performs cycle counting for a specific OrcaFlex results variable. But what if instead you wish to extract one or more time histories, perform some post-processing on these time histories to create a derived result, and then cycle count that derived result?

We anticipated this possibility and provide an alternative way to access the functionality. Again this is through a function named `RainflowHalfCycles` but this function is at the module scope.

```
RainflowHalfCycles(values)
```

We illustrate the use of this module scoped function by cycle counting a damped harmonic oscillator.

```
import numpy
import matplotlib.pyplot as pyplot
import OrcFxAPI

def f(x, a, omega, gamma):
    # damped harmonic oscillator
    return numpy.exp(-gamma*x) * a * numpy.cos(omega*x)

# 1000 equally spaced values between 0 and 100
X = numpy.linspace(0.0, 100.0, num=1000)
# evaluate f on these values
values = [f(x, 10.0, 0.4, 0.02) for x in X]

# plot the damped oscillator
pyplot.plot(values)
pyplot.show()

# calculate and output rainflow half cycles
halfCycleRanges = OrcFxAPI.RainflowHalfCycles(values)
for halfCycleRange in halfCycleRanges:
    print(halfCycleRange)
```

```
3.29894708985
3.85996361396
4.51657039367
5.28460374211
6.18355215086
7.2351081279
8.46583018246
9.9059245794
11.590851258
13.5625335167
15.8692692175
```

6.10 Other results types

We have only covered the most commonly used result types here. There are more types of result available – refer to the documentation for details of the other available results.

7 Interacting with other Python modules

As mentioned at the very beginning, one of the benefits of OrcaFlex automation through Python is the availability of a great many extension packages that can simplify coding for certain tasks. We've seen and used a few of these packages incidentally in the code above, but now we shall highlight some of the packages that are most commonly used with OrcaFlex automation. This list is not meant to be even remotely comprehensive, but is intended to give a flavour of what is available.

Few of the packages used in this section are part of a basic Python installation. The `pip` package manager can be used to install any extension packages that may be missing from your installation.

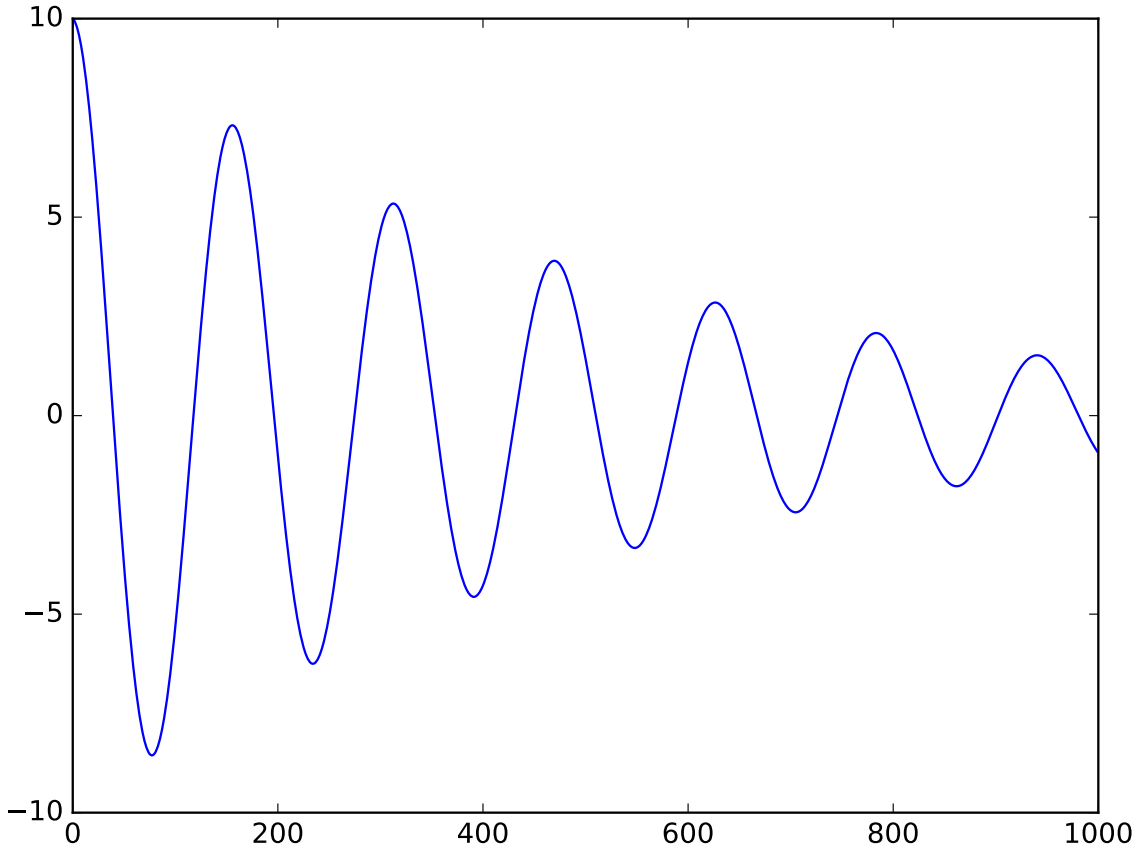


Figure 5: Damped harmonic oscillator

7.1 Reading and writing Excel files with `xlrd`, `xlsxwriter` and `openpyxl`

Whilst there are some OrcaFlex users who shun Excel with fanatical zeal, it is often very useful to be able to read and write Excel workbooks as part of your OrcaFlex workflow. There are many extension packages that can be used for interop with Excel, and notable mentions go to `xlrd`, `xlsxwriter` and `openpyxl`, all of which we have used and can be recommended. None of these packages require Excel to be installed, and instead operate directly on Excel files.

We will start with a simple example using `openpyxl`.

```
import openpyxl

waveHeights = [0.5, 1.0, 2.0, 3.0, 4.0, 5.5]
wavePeriods = [6.0, 7.0, 8.0, 10.0]

book = openpyxl.Workbook()
sheet = book.active
sheet.title = 'Data'
boldFont = openpyxl.styles.Font(bold=True)
centredAlignment = openpyxl.styles.Alignment(horizontal='center')
sheet['A1'] = 'Height'
sheet['A1'].font = boldFont
sheet['A1'].alignment = centredAlignment
sheet['B1'] = 'Period'
sheet['B1'].font = boldFont
sheet['B1'].alignment = centredAlignment

row = 2
for H in waveHeights:
    for T in wavePeriods:
        sheet.cell(row=row, column=1).value = H
        sheet.cell(row=row, column=2).value = T
        row += 1

book.save('scratch/LoadCaseData.xlsx')
```

Here we have created a simple table of data. Now we show how to read this data and use it to create OrcaFlex models.

```
import OrcFxAPI
import openpyxl

book = openpyxl.load_workbook('scratch/LoadCaseData.xlsx')
sheet = book['Data']
model = OrcFxAPI.Model()

row = 2
while sheet.cell(row=row, column=1).value:
    H = sheet.cell(row=row, column=1).value
    T = sheet.cell(row=row, column=2).value
    model.environment.WaveHeight = H
    model.environment.WavePeriod = T

    id = row - 1
    fileName = 'scratch/case{0:03d},H={1:.1f},T={2:.1f}.dat'.format(
        id, H, T)
    model.SaveData(fileName)

    row += 1
```

This isn't a particularly realistic example, but it does hopefully make obvious the possibilities that are afforded by this package.

Another use case would be to gather results output from OrcaFlex to present in Excel. Just for the fun of it, we will show how to do that and produce Excel charts.

```
import OrcFxAPI
import openpyxl

model = OrcFxAPI.Model()
line = model.CreateObject(OrcFxAPI.otLine)
line.TargetSegmentLength = 1.0,
model.CalculateStatics()
Te = line.RangeGraph('Effective Tension')

book = openpyxl.Workbook()
sheet = book.active
sheet.append(['Arclength', 'Te'])
for data in zip(Te.X, Te.Mean):
    sheet.append(data)

chart = openpyxl.chart.ScatterChart()
chart.title = 'Effective tension in static state'
chart.x_axis.title = 'Arclength (m)'
chart.y_axis.title = 'Te (kN)'

N = len(Te.X)
x = openpyxl.chart.Reference(sheet, min_col=1, min_row=2, max_row=N+1)
y = openpyxl.chart.Reference(sheet, min_col=2, min_row=2, max_row=N+1)
series = openpyxl.chart.Series(y, x)
chart.series.append(series)
sheet.add_chart(chart, 'D2')

book.save('scratch/chart_embedded.xlsx')
```

A limitation of chart support in `openpyxl` is that charts must be located in worksheets. The `xlsxwriter` package is capable of producing charts in chart sheets. Let us re-create the previous example using this other package. Do note that the syntax for `xlsxwriter` differs from that for `openpyxl`. Both packages have good documentation online and you will need to make use of it should you ever have to write such code yourself.

```
import OrcFxAPI
import xlsxwriter

model = OrcFxAPI.Model()
line = model.CreateObject(OrcFxAPI.otLine)
line.TargetSegmentLength = 1.0,
model.CalculateStatics()
Te = line.RangeGraph('Effective Tension')

book = xlsxwriter.Workbook('scratch/chart_sheet.xlsx')
sheet = book.add_worksheet('ChartData')
sheet.write_row('A1', ['Arclength', 'Te'])
sheet.write_column('A2', Te.X)
sheet.write_column('B2', Te.Mean)

chart = book.add_chart({'type': 'scatter',
                        'subtype': 'straight'})
chart.add_series({
```

```
    'name': 'Effective Tension',
    'categories': '=ChartData!$A$2:$A${0}'.format(len(Te.X) + 1),
    'values': '=ChartData!$B$2:$B${0}'.format(len(Te.X) + 1),
})
chart.set_title({'name': 'Effective Tension in static state'})
chart.set_x_axis({'name': 'Arclength (m)',
                  'min': Te.X[0],
                  'max': Te.X[-1]})
chart.set_y_axis({'name': 'Te (kn)'})
chart.set_legend({'none': True})
chartsheet = book.add_chartsheet('Effective Tension chart')
chartsheet.set_chart(chart)

book.close()
```

7.2 Serialization with `json`, `yaml` or `h5py`

Sometimes it is necessary to save data structures to file. For instance, you may have a flow of work where one program extracts results, and the next program performs post-processing of those results. Indeed you may have multiple post-processing steps and by breaking the task up into smaller pieces you can choose to do some, but not necessarily all, of the post-processing. OrcaFlex post-calculation actions would be an effective way to perform the results extraction.

Splitting the overall post-processing task this way allows for greater flexibility. Such an approach does require the ability to save and load the data on which you operate. You might consider using text files or Excel files for this purpose, and that might be a reasonable approach. However, when the data are more complex and have more structure, such flat formats can be difficult to work with. Structured file formats like JSON, YAML, HDF5 etc. can be very effective.

We will start by looking at JSON (http://www.json.org/) which is made available by the standard Python package `json`. We are going to extract a range graph for a line, and save the information to a text file in JSON format.

```
import json
import OrcFxAPI

model = OrcFxAPI.Model()
line = model.CreateObject(OrcFxAPI.otLine)
line.TargetSegmentLength = 1.0,
model.CalculateStatics()
Te = line.RangeGraph('Effective Tension')

data = {
    'title': 'Effective Tension in static state',
    'x_axis_title': 'Arclength (m)',
    'y_axis_title': 'Te (kN)',
    'x_axis_values': Te.X.tolist(),
    'y_axis_values': Te.Mean.tolist()
}
with open('scratch/serialization.json', 'w') as f:
    json.dump(data, f)
```

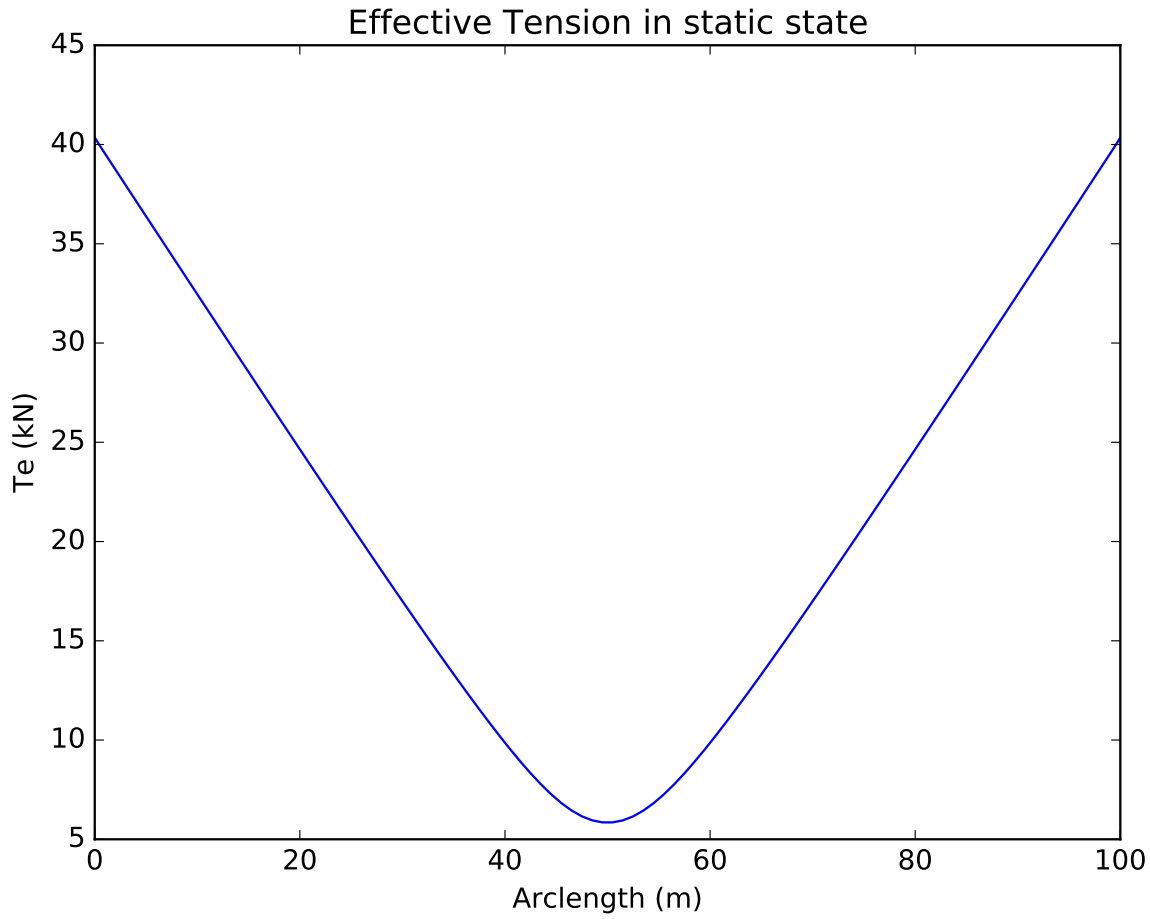
Note that we used the `tolist` method of the `numpy` array objects to convert them to plain Python list objects. This is because `numpy` array objects are not JSON serializable.

We can follow this up with a program that reads the file, and plots the data contained within. This second program makes no use of `OrcFxAPI` and is able to get all the information it needs from the file created in the previous step.

```
import json
import matplotlib.pyplot as pyplot

with open('scratch/serialization.json', 'r') as f:
    data = json.load(f)

pyplot.plot(data['x_axis_values'], data['y_axis_values'])
pyplot.title(data['title'])
pyplot.xlabel(data['x_axis_title'])
pyplot.ylabel(data['y_axis_title'])
pyplot.show()
```



We can perform the same task with close on identical code, but this time using the YAML (http://yaml.org/) file format. YAML will be familiar to any OrcaFlex users who have dabbled in OrcaFlex text data files. Although YAML and JSON are very similar (indeed JSON files can often be valid YAML files) the YAML format tends to be more readable to human eyes.

```
import yaml
import OrcFxAPI

model = OrcFxAPI.Model()
line = model.CreateObject(OrcFxAPI.otLine)
line.TargetSegmentLength = 1.0,
model.CalculateStatics()
Te = line.RangeGraph('Effective Tension')

data = {
```

```
    'title': 'Effective Tension in static state',
    'x_axis_title': 'Arclength (m)',
    'y_axis_title': 'Te (kN)',
    'x_axis_values': Te.X.tolist(),
    'y_axis_values': Te.Mean.tolist()
}
with open('scratch/serialization.yaml', 'w') as f:
    yaml.dump(data, f)
```

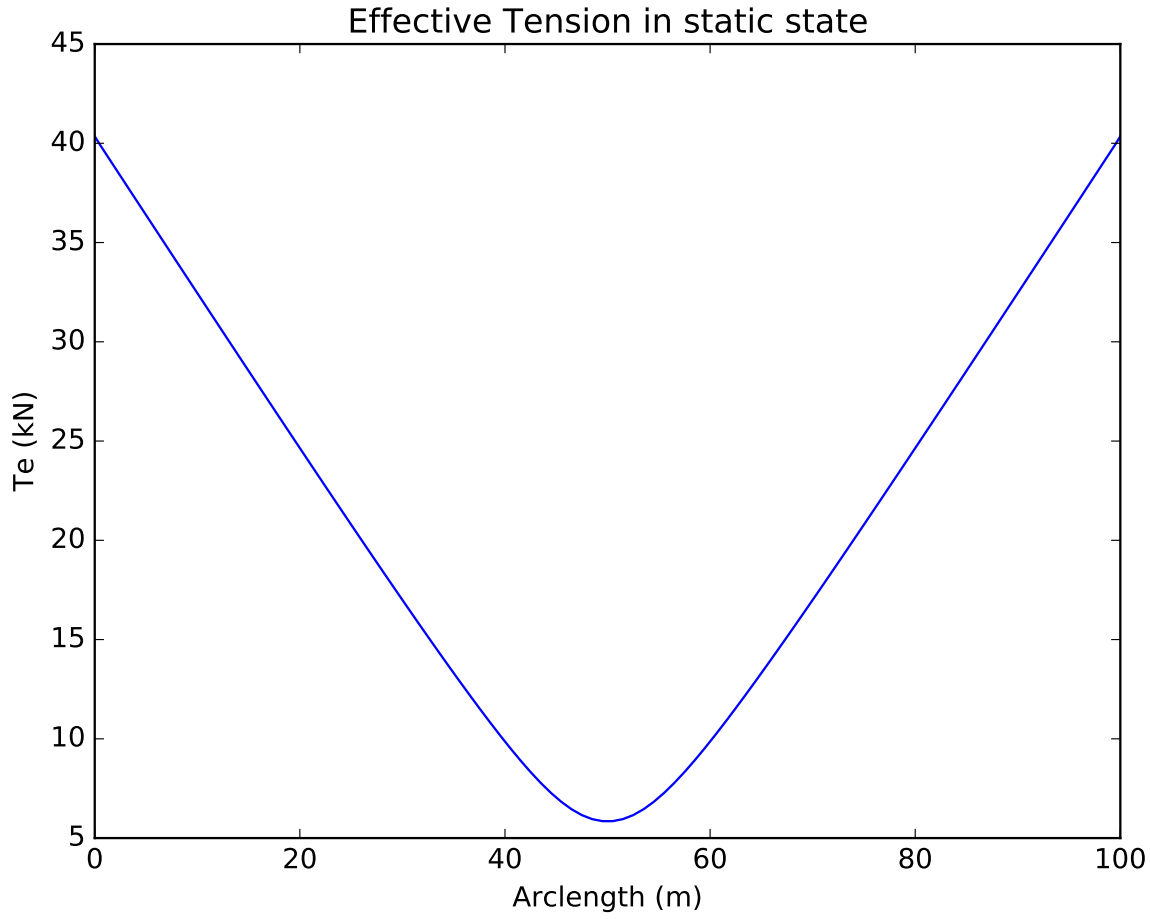
Again we use the `tolist` on the `numpy` array objects. Although `numpy` array objects are YAML serializable the output is much less readable than that produced for Python list objects.

Once again, we use near identical code to read the file.

```
import yaml
import matplotlib.pyplot as pyplot

with open('scratch/serialization.yaml', 'r') as f:
    data = yaml.load(f)

pyplot.plot(data['x_axis_values'], data['y_axis_values'])
pyplot.title(data['title'])
pyplot.xlabel(data['x_axis_title'])
pyplot.ylabel(data['y_axis_title'])
pyplot.show()
```



Yet another possibility is the HDF5 (https://en.wikipedia.org/wiki/Hierarchical_Data_Format) format. This is a binary

format widely used in scientific data applications. When dealing with very large amounts of data, HDF5 may offer more efficient storage and access than JSON or YAML. However, it is undoubtedly more complicated to use than JSON or YAML. We shall implement our now familiar example using HDF5 with the aid of the h5py package.

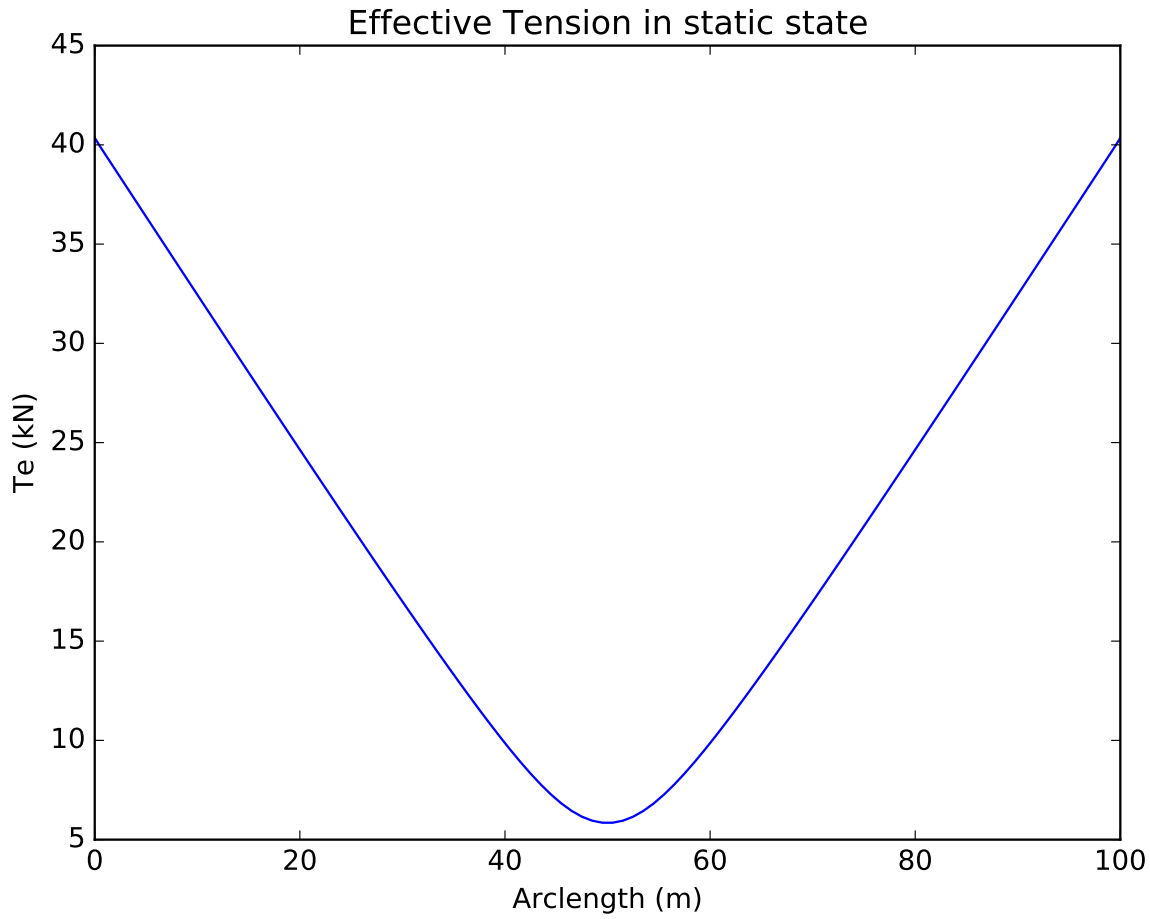
```
import h5py
import numpy
import OrcFxAPI

model = OrcFxAPI.Model()
line = model.CreateObject(OrcFxAPI.otLine)
line.TargetSegmentLength = 1.0,
model.CalculateStatics()
Te = line.RangeGraph('Effective Tension')

f = h5py.File('scratch/serialization.hdf5', 'w')
dataset = f.create_dataset('values', data=numpy.column_stack([Te.X,
Te.Mean]))
dataset.attrs['title'] = 'Effective Tension in static state'
dataset.attrs['x_axis_title'] = 'Arclength (m)'
dataset.attrs['y_axis_title'] = 'Te (kN)'
f.close()
```

```
import h5py
import matplotlib.pyplot as pyplot

f = h5py.File('scratch/serialization.hdf5', 'r')
dataset = f['values']
pyplot.plot(dataset[...,0], dataset[...,1])
pyplot.title(dataset.attrs['title'])
pyplot.xlabel(dataset.attrs['x_axis_title'])
pyplot.ylabel(dataset.attrs['y_axis_title'])
f.close()
pyplot.show()
```



Now we are going to try a slightly more complicated example, and make use of the hierarchical nature of HDF5. Let's try extracting multiple range graphs.

```
import h5py
import numpy
import OrcFxAPI

model = OrcFxAPI.Model()
line = model.CreateObject(OrcFxAPI.otLine)
line.TargetSegmentLength = 1.0,
model.CalculateStatics()

varNames = [
    'Effective Tension',
    'Bend Moment',
    'Declination',
    'Depth'
]
varUnits = {}
for details in line.varDetails(OrcFxAPI.rtRangeGraph):
    varUnits[details.VarName] = details.VarUnits

f = h5py.File('scratch/serialization_advanced.hdf5', 'w')
for varName in varNames:
    rg = line.RangeGraph(varName)
    dataset = f.create_dataset(varName, data=numpy.column_stack([rg.X,
rg.Mean]))
    dataset.attrs['x_axis_title'] = \
        'Arclength ({0})'.format(varUnits['X'])
    dataset.attrs['y_axis_title'] = \
```

```
        '{0} ({1})'.format(varName, varUnits[varName])

f.close()
```

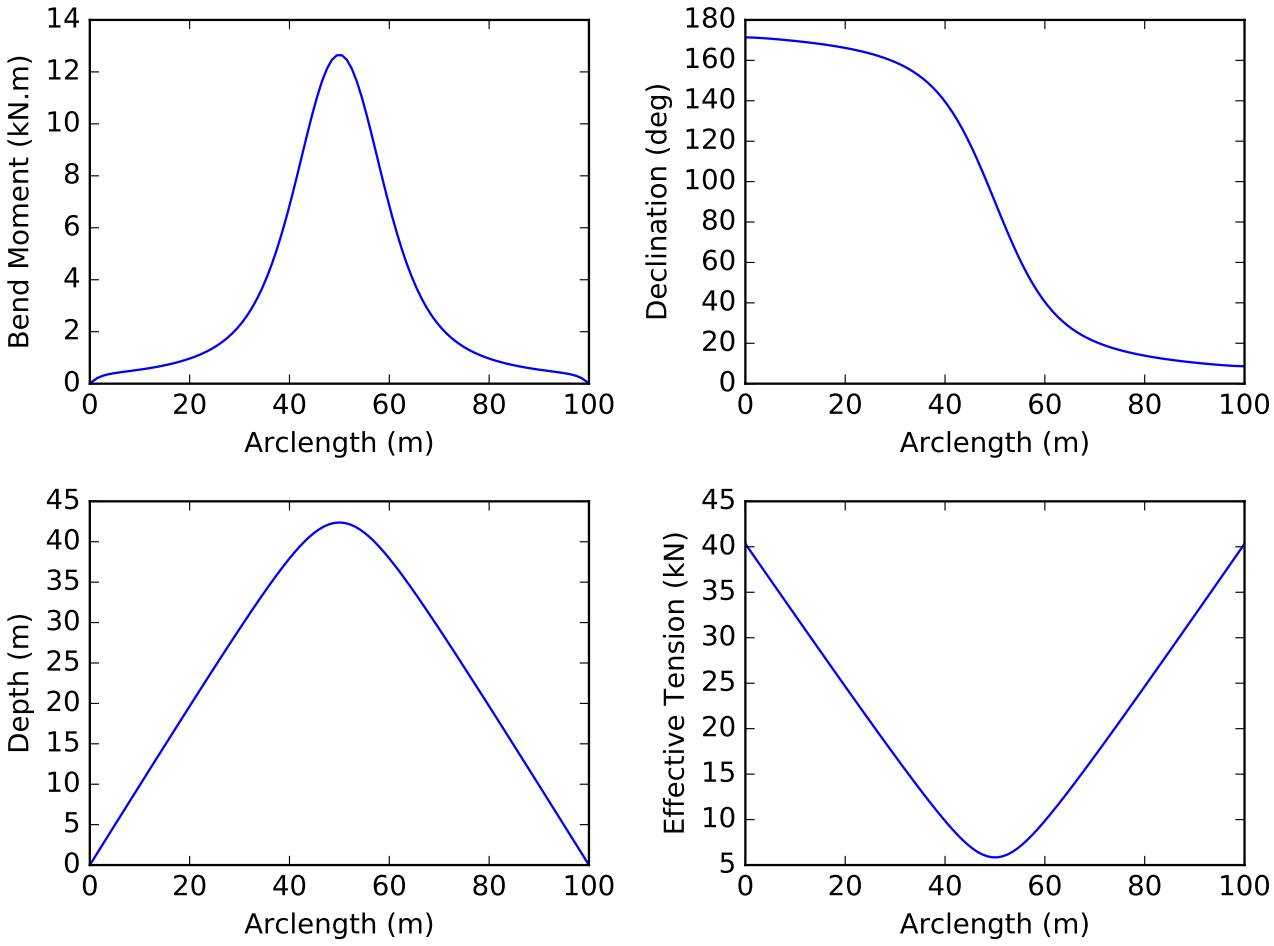
Next we load the HDF5 file, and enumerate its datasets and then plot each dataset.

```
import h5py
import math
import matplotlib.pyplot as pyplot

f = h5py.File('scratch/serialization_advanced.hdf5', 'r')
count = len(f)
ncols = int(math.sqrt(count))
nrows = math.ceil(count / ncols)
i = 1
for varName, dataset in f.items():
    pyplot.subplot(nrows, ncols, i)
    pyplot.plot(dataset[...,0], dataset[...,1])
    pyplot.xlabel(dataset.attrs['x_axis_title'])
    pyplot.ylabel(dataset.attrs['y_axis_title'])
    i += 1

f.close()

pyplot.tight_layout() # avoid label overlapping
pyplot.show()
```



This example could equally have been implemented using `json` or `yaml`, but do notice that the `h5py` package is able to accept `numpy` array objects directly. Because HDF5 has a focus on scientifc applications, it is geared up to work with numeric data.

This barely scratches at the surface of what can be performed with HDF5. If you wish to learn more, there are many resources that can be found online. In fact, `h5py` is not even the only option for working with HDF5 – PyTables (`http://www.pytables.org/`) and pandas (`http://pandas.pydata.org/`) are widely used alternatives.

7.3 Plotting with `matplotlib`

We have already seen a number of examples of plotting using the `matplotlib` package. Rather than create some more examples, we refer to those that appeared earlier, and note that `matplotlib` is the de facto standard for Python plotting.

7.4 Linear algebra with `numpy`

The `numpy` package is used extensively in scientific programming with Python. Although it is not a built-in package, it is incredibly widely used, and we would recommend installing it whenever you install Python. Indeed, many Python distributions that target scientific applications will include `numpy`.

As already mentioned, if `numpy` is available then the Python interface to OrcaFlex will make use of it and return `numpy` array objects for non-scalar data. In section 6.3 we saw an example of a program that performed post-processing on these `numpy` array objects using the facilities provided by the `numpy` package.

7.5 Interpolation with `scipy`

Formally, SciPy is stack of scientific computing tools. The website (`http://www.scipy.org/about.html`) lists the following core packages, some of which we have already encountered:

- Python, a general purpose programming language. It is interpreted and dynamically typed and is very suited for interactive work and quick prototyping, while being powerful enough to write large applications in.
- NumPy, the fundamental package for numerical computation. It defines the numerical array and matrix types and basic operations on them.
- The SciPy library, a collection of numerical algorithms and domain-specific toolboxes, including signal processing, optimization, statistics and much more.
- Matplotlib, a mature and popular plotting package, that provides publication-quality 2D plotting as well as rudimentary 3D plotting.
- pandas, providing high-performance, easy to use data structures.
- SymPy, for symbolic mathematics and computer algebra.
- IPython, a rich interactive interface, letting you quickly process data and test ideas. The IPython notebook works in your web browser, allowing you to document your computation in an easily reproducible form.
- nose, a framework for testing Python code.

By `scipy` we are referring to just a single package from this stack, the SciPy library which is a very rich source of numerical algorithms. For the sake of picking an example, we shall demonstrate how to

perform interpolation using `scipy`. We will create an OrcaFlex variable bending stiffness data source which represents a table of bend moment against curvature.

```
import numpy
import scipy.interpolate
import OrcFxAPI

model = OrcFxAPI.Model()
stiffness = model.CreateObject(OrcFxAPI.otBendingStiffness)
stiffness.IndependentValue = 0.0, 0.012, 0.021, 0.074, 0.28
stiffness.DependentValue = 0.0, 1200.0, 1450.0, 1700.0, 1850.0
```

Now we will create a `scipy` interpolation object using the data from the variable data source. We don't specify an interpolation method and so the default linear interpolation method is used. This matches the interpolation method used for variable bend stiffness data in OrcaFlex.

```
interp = scipy.interpolate.interp1d(
    stiffness.IndependentValue,
    stiffness.DependentValue
)
```

Finally, we can perform interpolation to calculate bend moment, which we do at 10 equally spaced values of curvature.

```
for C in numpy.linspace(interp.x[0], interp.x[-1], num=10):
    M = interp(C)
    print('C =', C, 'M =', M)
```

```
C = 0.0 M = 0.0
C = 0.0311111111111 M = 1497.6939203354298
C = 0.0622222222222 M = 1644.4444444444446
C = 0.0933333333333 M = 1714.0776699029127
C = 0.124444444444 M = 1736.7313915857605
C = 0.155555555556 M = 1759.3851132686084
C = 0.186666666667 M = 1782.0388349514562
C = 0.217777777778 M = 1804.6925566343043
C = 0.248888888889 M = 1827.3462783171522
C = 0.28 M = 1850.0
```

This is a pattern that we use very frequently in OrcaFlex test code.

7.6 Root finding with `scipy`

Another possible application for `scipy` is root finding. That is to find solutions to equations of the form $f(x) = 0$ where x is unknown. To give this very abstract mathematical definition some context, let us consider an OrcaFlex model of a riser with a buoyant section.

```
import scipy.optimize
import OrcFxAPI

model = OrcFxAPI.Model()

riserType = model.CreateObject(OrcFxAPI.otLineType)
riserType.Name = 'Riser type'
```

```
buoyedType = model.CreateObject(OrcFxAPI.otLineType)
buoyedType.Name = 'Buoyed type'
riser = model.CreateObject(OrcFxAPI.otLine)
riser.Name = 'Riser'

riser.LineType = riserType.Name, buoyedType.Name, riserType.Name
riser.Length = 90.0, 80.0, 100.0
riser.TargetSegmentLength = 2.0, 2.0, 2.0
riser.EndBX = riser.EndAX + 160.0
riser.EndBConnection = 'Anchored'
riser.EndBHeightAboveSeabed = 0.0

buoyedType.WizardCalculation = 'Line with Floats'
buoyedType.FloatBaseLineType = riserType.Name
buoyedType.FloatDiameter = 1.1
buoyedType.FloatLength = 2.0
buoyedType.FloatPitch = 8.0
buoyedType.InvokeWizard()
```

The model that we have built is shown in figure 6.

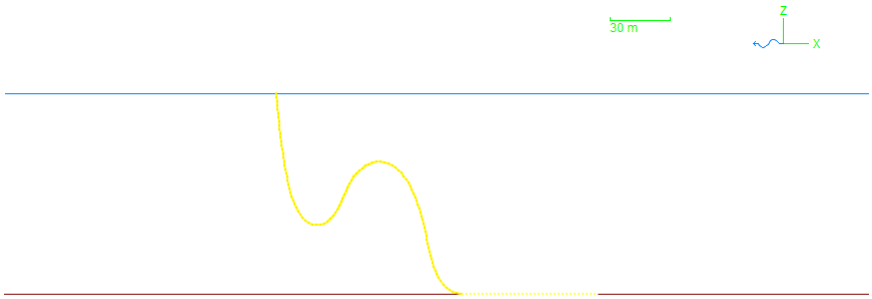


Figure 6: Riser with buoyant section, before root finding

Now suppose that we wish to modify the float length to achieve a goal of the maximum hog bend Z coordinate being -50m. We can use the `fsolve` method from `scipy.optimize` to achieve this.

```
targetZ = -50.0
initialFloatLength = buoyedType.FloatLength
arclengthRange = OrcFxAPI.arSpecifiedSections(2, 2)

def calcHogBendZ(floatLength):
    buoyedType.FloatLength = floatLength[0]
    buoyedType.InvokeWizard()
    model.CalculateStatics()
    Z = riser.RangeGraph('Z', arclengthRange=arclengthRange).Mean
    hogBendZ = max(Z) # Z value at high point of hog bend
    print('Float length = {0}, hog bend Z =
{1}'.format(floatLength[0], hogBendZ))
    return hogBendZ - targetZ

scipy.optimize.fsolve(calcHogBendZ, initialFloatLength)
```

```
Float length = 2.0, hog bend Z = -34.116378169350945
Float length = 2.0, hog bend Z = -34.116378169350945
Float length = 2.0, hog bend Z = -34.116378169350945
Float length = 2.0000000298023224, hog bend Z = -34.116376901460505
```

```
Float length = 1.6266484834636032, hog bend Z = -49.88879328629302
Float length = 1.6240160904595078, hog bend Z = -50.003984076185006
Float length = 1.6241071364041273, hog bend Z = -49.99998031148309
Float length = 1.624106688685604, hog bend Z = -49.9999999965303
Float length = 1.6241066886066888, hog bend Z = -49.999999999999915
```

Mapping this back to the original abstract equation, $f(x) = 0$, the unknown variable x represents the float length. The function f is encapsulated in `calcHogBendZ` and does the following:

- Set the line type wizard `FloatLength` data item to x.
- Invoke the line type wizard with a call to `InvokeWizard`.
- Perform the static calculation.
- Extract the Z coordinate for section 2 of the line (the buoyant section representing the hog bend), using `arclengthRange`.
- Find the maximum Z coordinate for the hog bend.
- Subtract the target Z value.

This is quite a complex function, but `fsolve` does not care about that, and quite happily finds the root. The resulting model is shown in figure 7.

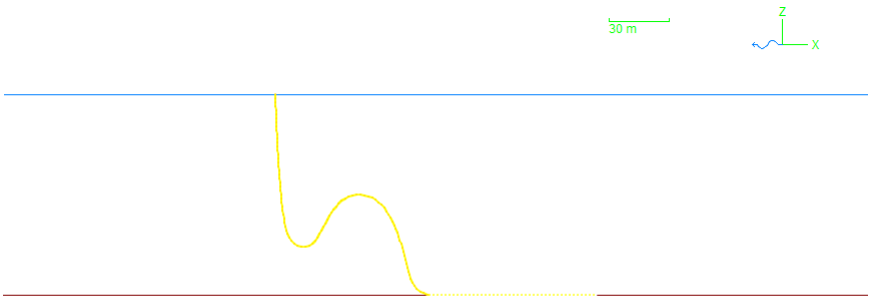


Figure 7: Riser with buoyant section, after root finding

This example demonstrates the power and flexibility of combining OrcaFlex with Python and its extension packages.

8 Exploring further

This document has attempted to cover a broad range of topics without going into every last detail. So what if you do need to learn in more detail? The Python interface to OrcaFlex documentation (www.orcina.com/SoftwareProducts/OrcaFlex/Documentation/OrcFxAPIHelp/) has comprehensive details of every class and function.

For braver individuals, you can read the Python source code of the `OrcFxAPI` module. This is found in the `OrcFxAPI.py` file in the `OrcFxAPI\Python` sub-directory under your OrcaFlex installation directory. Usually the reference documentation should provide enough information, but there are times where reading the source code can help in gaining understanding. Do note though that you should never modify this file.

There are a great many general resources on Python. The official documentation (docs.python.org/) is very good. We also recommend the book Learning Python by Mark Lutz (http://learning-python.com/books/about-lp.html) as an excellent introduction to Python.

Stack Overflow (http://stackoverflow.com/) is a great resource for answers to very specific questions. Usually a web search containing the keywords for your question will turn up any number of excellent answers on Stack Overflow.

For recent versions of Python, the `pip` package manager is usually the best way to install extension packages. For instance, the following command, executed at a command prompt, will install `numpy`.

```
> pip install numpy
Collecting numpy
  Downloading numpy-1.10.4-cp35-none-win_amd64.whl (7.5MB)
    100% |################################| 7.5MB 114kB/s
Installing collected packages: numpy
Successfully installed numpy-1.10.4
```

However, not all packages will install correctly through pip. When preparing this document I attempted to follow up the previous command with `pip install scipy`. This failed for reasons that I did not attempt to understand. What to do in such circumstances? Christophe Gohlke maintains an excellent selection of Python extension packages on his website at www.lfd.uci.edu/~gohlke/pythonlibs/. For Python packages that are otherwise hard to obtain or install, Christophe's site is invaluable.

Finally, if all else fails, you can always contact us at Orcina by e-mail (orcina@orcina.com).