Python Automation Examples – Supporting Documentation

As many of you will be aware, there are lots of things you can do with Python / OrcaFlex. Of course there are other options too, but Python seems to be the most popular choice with our users (and with us!) so that's what these examples focus on. We get lots of support questions from users asking how to do all sorts of things with Python, and many requests for us to provide more examples, so here we are trying to cover the techniques behind some of the more commonly asked for Python tasks.

We have deliberately kept these examples relatively short and simple; the aim here is to demonstrate some of the options as clearly as possible, without getting bogged down in the coding details.

There are now numerous ways to use Python scripts in your day to day OrcaFlex work e.g. pre and post processing, external functions, post calculation actions and now the new user defined results. For those of you that haven't ventured into the Python world yet, we hope that this will raise awareness of some of the options that are available.

Example 1: Pre-processing YAML files

This example provides a reminder that text data files, or Yaml files, exist and shows the benefits that they offer over binary data files. It also shows how a simple Python script can be used to generate multiple load cases from a base case model.

You can of course generate your Yaml load cases via the OrcaFlex spreadsheet if you wish, but we'll use Python here to a) show how it's done and b) demonstrate why you might need to do it this way.

The model Basecase.dat is our base case model. If you open this model and run statics you'll see that it's a lazy wave riser, connected to a turret moored FPSO. The turret is modelled using a constraint, and the riser is connected to it, so that when we change the vessel heading later to set it head to waves, we can independently rotate the constraint to maintain the riser orientation.

For this example we are going to generate load case variants for 8 different wave directions and 2 wave periods, so 16 load cases.

Open Example #1.py script.

The wave directions and periods are specified in lists, so that we can loop through them. The ability to do this is one of the reasons why this method offers an improvement over doing this in the OrcaFlex spreadsheet.

Yaml files, or text data files if you prefer, are, as the name suggests, simply files of text. So to create them from a Python script you just write lines of text, in the required format, to a file that is saved as a yaml file.

In the script we first set a casenumber, which we are going to increment each time we create a new load case, and use it to name that file. We then loop through the directions and the periods, using print functions to write the lines of text to each load case file.

Note that outputting data to a file requires it to be opened in writable format ('w') and closed once you've finished with it, however the use of 'with open…' will keep the file open while you're using it and close it automatically once you've finished with it, without having to explicitly close the file.

How do you know how to format the text or what syntax to use? That's all explained in the Help file section on text data files:
https://www.orcina.com/SoftwareProducts/OrcaFlex/Documentation/Help/Redirector.htm?Textdatafiles.htm

Now run the script to generate the Yaml files. Note that we're not using OrcFxAPI at this stage, we're simply writing lines of text to a text file. Open one of the completed Yaml files in OrcaFlex to show the result.

The key different between binary dat files and text data files is that the text file contains only a reference to a base model, and the variations that you want to make to it. Whereas a binary data file includes all the data from the base model, with the changes made to it. This means that one advantage of text data files is the file size, typically less than 1 kB compared to much larger dat files.

The next, and probably most significant advantage is that, because we are referencing the base model, any changes made to the base model are automatically picked up in the load cases without having to re-run the script that generated them.

Now let's do something a bit more sophisticated.

Let's say that we want to also apply a vessel offset, perhaps intact and damaged mooring cases and, because it's a weathervaning FPSO, a change of heading in each load case.

The model Vessel offset.dat attempts to show the type of adjustment that is required for each load case. This model shows the vessel in its nominal position, and one of its rotated and offset position for a 50m offset, 135 degree wave direction. We really want to rotate the vessel about its turret and then offset it from that position, but changing the heading of the vessel rotates it about its origin, which in this case is not at the same position as the turret centre. So calculating the new vessel position requires some thought if you want to code the maths into your script.

However OrcaFlex has the functionality to move and rotate objects about a specified point. And a good reason to use a Python script here is that you can use OrcaFlex's functionality to do the task, rather than having to work it out for yourself.

Take a look at Example #2.py.

This is basically the same script as before, but with a few additions. Firstly, we're going to use OrcaFlex so we need to *import OrcFxAPI*. This is the module that gives us access to the OrcaFlex interface.

We've also added 2 offsets in a list and an additional *for* loop. So we'll now be creating 32 load cases. By the time we reach line 24 we have created a yaml file that calls the base model and makes some changes to it. We can then get OrcaFlex to open this model and do something with it. That something could be running statics to check a result perhaps, or in this case, it's to use the *Move selected object…* functionality to move and rotate the vessel. Lines 27 through to 32 in the script perform the rotation and translation operations.

Once the vessel has been moved and rotated we could save the model, but if we do then that would then save all the model data, rather than only the variations to the base model, and we'd lose the advantages that text data files give you.

So instead we re-access the yaml file and append the vessel's X, Y and Heading data, as calculated for us by OrcaFlex.

Example 2: Post-processing: code check results

In this example we show how to automate the post processing of code check results.

We're using DNV OS F101 (Submarine pipeline systems), which is probably the most commonly used code check in OrcaFlex terms.

The model *Code Checks.sim* contains a riser and some code check data (on the Code checks data form) and is therefore ready for code check results to be calculated.

The most important thing to note when it comes to post-processing code check results is that none of the data on the Code Check data form contributes to the simulation in any way. The code check calculation is purely a result which is post processed from the simulation results. This means that it is not necessary to re-run a simulation if you change the code check data; you only need to recalculate the code check results.

A commonly asked question refers to the ultimate limit state (ULS) design check, where two different load effect combinations are required: a system check and a local check.

Table 4-4 Load effect factor combinations

Limit state/load combination	*Load effect combination*		*Functional loads* [1]	*Environmental load*	*Interference loads*	*Accidental loads*
			γ_F	γ_E	γ_F	γ_A
ULS	*a*	System check[2]	1.2	0.7		
	b	Local check	1.1	1.3	1.1	
FLS	*c*		1.0	1.0	1.0	
ALS	*d*		1.0	1.0	1.0	1.0

1) If the functional load effect reduces the combined load effects, γ_F shall be taken as 1/1.1.
2) This load effect factor combination shall only be checked when system effects are present, i.e. when the major part of the pipeline is exposed to the same functional load. This will typically only apply to pipeline installation.

This means that the code check results need to be calculated twice, with two different combinations of γF and γE parameters, but the code checks data form only lets you input one set.

This 'double check' is of course best automated, particularly if you have a number of load cases to consider. It can be done using the OrcaFlex spreadsheet, but a Python script offers a much better option.

In this model we've got a long steel catenary riser, and the code check data form has been populated with appropriate data for this pipe. The default workspace opens the range graph results for the pipe's load controlled code check, with load effect combination 'a'.

To automate this process, and therefore make it easy to extract these results for multiple simulations, we need a simple Python script: *CodeChecks_a_b.py.*

Most post-processing scripts will begin in the same way i.e.:

import OrcFxAPI	*to import the API*
model = OrcFxAPI.Model(<model name>)	*to identify the model we want to process*
object1 = model['<object name>']	*to identify one or more objects that we need*
object2 = model['<object name>']	*either for data or results*

and that's the case here.

In this case we've used Python dictionaries to hold the condition a and b parameters, but there are various other options.

We use a *for* loop to set the Gamma F and Gamma E values, find the range graph results for the F101 load controlled unity check, and then print the overall maximum of the maximum set. Run the script to see the results output.

But that example handled just one load case, what if you want to do the same task for multiple files? Well you could do that in this script, by looping through all the sim files in a folder for example, but that will process the models one at a time. A better option is.....

Example 3: Post calculation actions

If you use the OrcaFlex spreadsheet for your post processing, then multi-processing happens automatically, but if you use a Python script to extract results from multiple simulations, then you will need to do some extra work to enable multi-processing. The simplest way to do this is to use a post calculation action.

Post calculation actions are a way to include your post processing actions as part of the model processing. Any models that are run either through the batch processor or through distributed OrcaFlex can trigger a script to run when each simulation completes. The big advantage of working this way is that you make use of the multiprocessing capabilities of OrcaFlex i.e. the results extraction is also multi-processed. If the sim files are large, and therefore opening and extracting results from the files is time consuming, then there are significant speed ups available.

In this example we show how to automate the same code check results, but this time we'll do it in a post calc. action.

Note that you can use post calc. actions with tools other than Python, that's all explained in the Help file, but we're focussing on the Python method here.

Open *PCA code check.py*

Note the differences from our previous example:

- No need to import OrcFxAPI explicitly - This import is, and must be, performed automatically by the host OrcaFlex process.
- But in this example we do import the module *os.path*, which provides us with some operating system dependent functionality, which we use to manipulate file paths and names.

- The post-processing action is contained within a function called *Execute*(), which receives a single parameter. In other words a bunch of information is passed to the function, wrapped up in this parameter. General convention is to call this parameter 'info'. The *info* object contains attributes including *model* which you use to do your post-processing in the same way as with a standalone Python script, and *modelFileName*, which contains the absolute path of the simulation file.
- This time we write the results to text files, one per simulation, so that we can collate the results for all the simulations later. The text file name is created by splitting the '.dat' extension from the model name and replacing it with '.txt'.

The current limitation of PCAs is that you can't gather together the results from all the simulations in order to present them as a summary (but we're considering ways to do this for the future). You could output the results from each file to a common spreadsheet, but because we are multi-processing the simulations, there's a chance that 2 or more simulations might try to access the sheet at the same time, which Excel won't allow. So this won't work. Instead we output the results to separate files (text files in this instance, but it could be spreadsheets) and then run a separate script to collate them. This still offers significant efficiency advantages over the previous method (where we used a standalone script to extract the results) because the results extraction is also multi-processed with the PCA version.

Open one of the dat files to see how the PCA is set up (General data form). Note that the script is set to run at the end of dynamics, and that we've ticked the Skip simulation file save option.

Skipping the file save is useful if you're short of disc space, but don't forget that viewing completed simulations is an essential part of checking your work. If you can't see what's happened in the model, how do you know the results are any good? What if you forgot to connect the riser to the vessel and therefore the results are meaningless? So use this option carefully!

Now take a look at the script that collates the results together: *PCA collate results.py*

This is nothing to do with OrcaFlex, it's simply reading some text files and transferring their contents to a single spreadsheet. We're using a module called xlsxwriter to do this, and a further module called glob, which is useful for looking for a list of files with names that match a certain pattern, in this case all the text files in the working folder.

The majority of the script is taken up with adding some headings and formatting the cells. The real work is done in the for loop, where we read in the text from the file, and write it back out to the results.xlsx sheet.

So this script needs to be run once the OrcaFlex models have finished and output their results to text files. As mentioned earlier today, a new feature in 10.3 is the ability to add Python scripts to the batch processor, so we can set up all of this to run while we go away and do something else. The batch processor knows that the models need to run first, before the Python script is run.

Open the batch processor and load up the 4 dat files and the *PCA collate results.py* (note: NOT the PCA code check.py script), and process them. 4 txt files are generated and an Excel sheet that contains the results from all the load cases.

So that has extracted our results as part of the batch processing, making full use of as many processors as the machine that we've run it on has available, and then collated them into a single summary sheet. If we wanted to take this further, we could also delete the text files as part of the collation script, once we've got the results from them written to the spreadsheet.

Take a look at the spreadsheet: *Results.xlsx* to see the final output.

Example 4: Stinger tip clearance optimisation

Something a bit more involved now… in this example we show how you can use a Python script to set up your model to meet a particular criterion. In this case we're using an S-lay model and setting the clearance between the pipe and the stinger tip in the static condition.

In pipelay analysis, the position of the pipe or A&R wire to the last roller on the stinger is one of the critical parameters, so setting this initial clearance is often something that's needed.

The model is one of our standard examples: *E01 Explicit Geometry Stinger.sim*. The default workspace will show the static state result for pipe – stinger tip clearance. We want to set the pipe length so that this value is 0.2m. The result is currently negative, showing that the pipe is in contact with the last roller.

The script operates in a similar way to OrcaFlex's line setup wizard, in that it runs multiple static analyses and adjusts a specified parameter, until it meets the target condition. There are many other instances where this type of optimisation might be required, and the method will be very similar to what we're showing here, so it certainly isn't limited to pipelay models.

The parameter that we're adjusting here is the pipe length, but this could equally be the roller position or the stinger radius etc.

The script is called *FinalRollerClearance.py.*

As usual, we first import the modules that we're going to use. In this case we're using a 3rd party module called SciPy to access a root finding function called fsolve. More about fsolve in a moment, but SciPy is a very commonly used Python library for scientific computing.

We then identify the model we're working with, and the names of the vessel and line objects.

Then we set our target clearance and the support number that we need to extract the clearance results.

The SciPy documentation explains the use of fsolve; suffice it to say here that fsolve will iterate through a specified calculation until a solution is found to an equation in the form:

$$f(x) = 0$$

when given a starting estimate.

The calculation needs to be contained in a function. In this case the function is called calcClearance, and the name of this function is the first argument passed to fsolve(). Fsolve also needs to know the starting estimate for the parameter that is going to vary, which in this case is simply the current (initial) pipe length, which we get from the model.

The calculation that we are iterating over is performed by OrcaFlex i.e. the static calculation. And the equation we're solving is:

$$\text{clearance} - \text{target clearance} = 0.$$

So each iteration through the calculation means running statics, checking the clearance result, and finding out how close that is to the target clearance. The fsolve function decides how much to adjust the line length in order to get closer to the solution, and the process repeats until the desired solution is found.

Note that in this script we have included some very basic error handling, in that we stop the script if the OrcaFlex static calculation fails.

Run the script, and open the *Results.sim* model to show the result (default workspace shows the static clearance).