PHIMECA
Solutions for robust engineering

Creating a distributed python wrapper with otwrapy

HPC and Uncertainty Treatment

PRACE Advanced Training Center, May 10-12 2021
EDF – Phimeca – Airbus Group – IMACS – CEA

Gaëtan Blondet, Phimeca Engineering

PRACE

MAISON DE LA SIMULATION

afaq ISO 9001 Qualité AFNOR CERTIFICATION

Outline

Introduction

- Presentation goal: Show you how to carry on distributed uncertainty studies with an external code.
- Based on the module otwrapy available at GitHub. Initially developed at Phimeca engineering.
- A good working example can be found on the otwrapy repository example.

What makes a good wrapper ?

- Wrapper : a python interface with your external code.
- Distributed, without conflict between runs.
- Compatible with different environment (Workstation, HPC clusters, cloud-computing).
- You can use it as a script (argsparse module):

```
>> python wrapper.py -X 170 3 0.05
```

- It catches and logs errors for easy debugging.
- It can either run or simply prepare runs –> useful when running on clusters.

All of this might seem complex, but wrappers are repetitive and otwrapy is here for you !

Basic skeleton of a wrapper

- Assumption: you want to wrap an external code not written in Python.
- An OpenTURNS wrapper is a subclass of ot.OpenTURNSPythonFunction() for which at least the method `_exec(X)` should be overloaded. ot.PythonFunction() is a simpler alternative to , but you loose the ability to parameterize your wrapper when instantiating it.
- If possible and required, you can also overload `_gradient(X)` and `_hessian(X)`.

```
class Wrapper(ot.OpenTURNSPythonFunction):
    """Wrapper of my external code.
    """
    def __init__(self):
        """Initialize the wrapper with 4 and 1 as input and output dimension.
        """
        super(Wrapper, self).__init__(4, 1)
        # Do other stuff if necessary
    def _exec(self, X):
        """Run the model in the shell for the input vector X
        """
        pass
```

Overloading the exec function

- _exec is the default OpenTURNS method that executes the function on a given point, in 3 steps:
  1. Prepare the input parameters (e.g. input file).
  2. Run the external code.
  3. Get the output by parsing the result given by the external code.
- With otwrapy.TempWorkDir,these steps are executed on a temporary working directory.

```
def _exec(self, X):
    """Run the model in the shell for the input vector X
    """

    # Move to temp work dir. Cleanup at the end
    with otw.TempWorkDir(cleanup=True):
        # Prepare the input
        self._prepare_input(X)
        # Run the external code
        self._run_code(X)
        # Parse the output parameters
        Y = self._parse_output()

    return Y
```

Temporary working directory

- Efficiently and safely work on temporary directories with otwrapy.TempWorkDir
  - Avoid conflict between simulations running in parallel.
  - If an exception is raised during execution, the Python interpreter come back to the preceding current directories.
  - Cleanup upon exit, or don't if you want a full backup of the simulations.
  - Transfer files required by the external code.
- Example:

```
import otwrapy as otw
# I'm on a given dir, e.g. ~/beam-wrapper
with otw.TempWorkDir(base_temp_work_dir='/tmp', prefix='run-', cleanup=True, transfer=None):
    """
    ...
    Do stuff safely on an exclusive temporary directory and erase it afterwards
    ...
    """
    # The current working directory is something like /tmp/run-pZYpzQ

# Back on ~/beam-wrapper and /tmp/run-pZYpzQ does not exist anymore
```

Prepare the input parameters

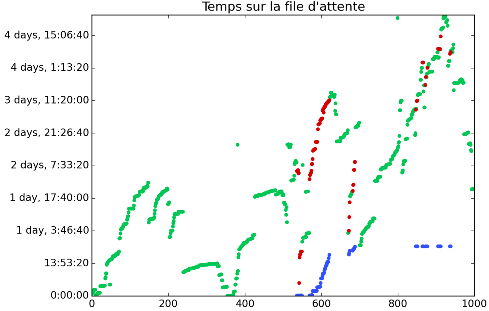
- For each simulation, your wrapper must communicate the input parameters to the external code.
- Most scientific codes use input files that describe, among other thing, the parameters of your model/simulation.
- With OpenTURNS coupling tools, the values of the vector X are placed on an input template file that have tokens/placeholders for where the expected parameters should be.

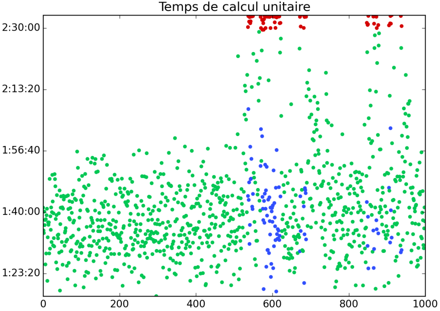
```
def _prepare_input(self, X):
    """Create the input file required by the code.
    """
    ot.coupling_tools.replace(
        infile='input_templatefile.xml',
        outfile='input.xml',
        tokens=['@X1','@X2','@X3','@X4'],
        values=X)
```

Run the external code

- Most of the time this is a fairly straightforward call to an executable with an input file.
- Sometimes, it is useful to time your runtime.

```
def _run_code(self):
    time_start = time.time()
    ot.coupling_tools.execute('/path/to/executable -x input.xml'))
    return time.time() - time_start
```





Parse output parameters 1/2

- Common practice among scientific code is to create output files with the results of the simulation.
- The output should then be parsed in order to get the output parameters of interest.
- If it is a `.csv` file
  - pandas.read_csv is the fastest option, but it introduces pandas as a dependency.
  - if speed is not an issue, try ot.coupling_tools.get_value,
  - or numpy.loadttxt.

Parse output parameters 2/2

- For `.xml` files, minidom package from the python standard library does the trick.
- If the external code returns the output parameters of interest to STDOUT, set `get_stdout=True` when calling `ot.coupling_tools.execute(...)`. (or use subprocess.check_output)
- For standard binary formats, there are python interfaces to netcdf and HDF5.
- Otherwise, be creative and pythonic !

```
def _parse_output(self):
    # Retrieve output (see also )
    xmldoc = minidom.parse('outputs.xml')
    itemlist = xmldoc.getElementsByTagName('outputs')
    Y = float(itemlist[0].attributes['Y1'].value)

    return [Y]
```

Managing data backups

- 2 useful functions: otwrapy.dump_array and otwrapy.load_array
  - Fast solution.
  - Data can be compressed (gzip library). If the extension is ‘pklz‘, compression is automatic.
  - Tips: Convert your `ot.Sample` to a `np.array` before, it is lighter !
- Dump and compress

```
import otwrapy as otw
otw.dump_array(np.array(X), 'InputSample.pklz', compress=True)
```

- ... and load

```
import otwrapy as otw
import openturns as ot
X = otw.load_array('InputSample.pklz')
X = ot.Sample(X)
```

Catch exceptions when your code fails

- In order to catch exceptions: otwrapy.Debug() !
  - Avoids crashes, raises exceptions and saves logs.
  - Useful when the wrapper is not used on an interactive environment (IPython, Jupyter notebook).

```
import otwrapy as otw
class Wrapper(ot.OpenTURNSPythonFunction):
    @otw.Debug('wrapper.log')
    def _exec(self, X):
        #Do stuff
        return Y
```

Creating a CLI for your wrapper

- Command Line Interface (CLI): to run your wrapper in detached mode, e.g., through submission scripts on HPC clusters.
- The argparse library might be useful to link the python code with the external code.
- Take a look at the beam wrapper for an example of a CLI interface

```
if __name__ == '__main__':
    import argparse
    parser = argparse.ArgumentParser(description="Python wrapper example.")
    parser.add_argument('-X', nargs=3, metavar=('X1', 'X2', 'X3'),
        help='Vector on which the model will be evaluated')
    args = parser.parse_args()

    model = Wrapper(3, 1)
    X = ot.Point([float(x) for x in args.X])
    Y = model(X)
    dump_array(X, 'InputSample.pkl')
    dump_array(Y, 'OutputSample.pkl')
```

- You can then execute your code from the command line :

```
python wrapper.py -X 170 3 0.05
```

Parallelizing the wrapper

- Uncertainty studies imply independant and repetitive tasks: very simple to parallelize.
- One function: otwrapy.Parallelizer() !!

```
import otwrapy as otw
from otwrapy.examples.beam import Wrapper
parallelized_beam_wrapper = otw.Parallelizer(Wrapper())
```

Distributing calls on clusters or the cloud

- otw.Parallelizer is no longer the way to go...
- You can manage to make an heterogeneous office cluster with IPyparallel or dispy
- For clusters and the cloud, rely on a good CLI interface of your wrapper and distribute your calls through submission scripts or cloud APIs (e.g., Simulagora or Domino)

Conclusion

- Main steps:
  - build your simulation code, and define inputs and outputs;
  - identify ways to communicate inputs and outputs;
  - make a wrapper to drive your simulation with python (» otwrappy makes it easier);
  - distribute your computations (» otwrappy makes it easier).
- By creating a CLI of your wrapper, you can easily distribute your calls on a cluster or on cloud platforms.
- It is important to protect your wrapper with otw.Debug() so that you can have a traceback of raised Exceptions.
- otwrapy is here for you ! Use it to avoid code boilerplate or as a simple cookbook.

Thank you for your attention

Gaëtan Blondet

blondet@phimeca.com

Github : otwrapy