Introduction to Coma

Burkhard Ritter

February, 2014

Contents

1 What is Coma?

Coma is the computational condensed matter physics programming toolkit. It is a small Python library aiding with some aspects of conducting “numerical experiments”—running computer simulations. I wrote it for my own needs, but think it might be useful for other people with similar projects and workflows and therefore it is now available under the two-clause BSD license. The first version of Coma was written by Carl Chandler and me as a C++ and Python library. This second iteration of the library is pure Python, but should, in principle, be usable with C++ code as well.

More specifically, Coma tries to help with storing and managing data from numerical simulations. It was largely conceived as an improvement to ad-hoc plain-text data files without a specified format and without any meta data. Such format-less, often obscurely named data files have the tendency to end up scattered all over the file system and become meaningless very, very quickly. Coma improves upon this situation by using standard file formats, rich meta data, a standard naming scheme while at the same time trying to be as flexible as possible. This comes at the cost of an additional dependency and the fact that you have to learn how to use one more library.

In addition to data management, Coma also includes a simple job runner, with parallelization support, a flexible mechanism to extract data from the files, and object serialization. Coma is unit tested, with the notable exception of the parallel code, and works well and reliably for me, but as always, please use it at your own risk.

2 Is Coma for me?

Coma focuses on flexibility and simplicity and makes a couple of basic assumption that should help you decide whether Coma is suitable for your use cases. I specifically wrote it to aid with parameter scans (e.g. run the same program with 100 different sets of parameters) and retrieving and plotting the produced data. It should work well for different, but related use cases. Coma's three basic design decisions are:

1. Plain text. Coma uses plain text data formats and is therefore suitable for small data volumes. It trades big data in favour of flexibilty. Supported formats are JSON and XML. For some use cases, for example for Monte Carlo, it might make sense to store large binary files alongside Coma's plain text files and use Coma only to store preprocessed or filtered data. An advantage of plain text is that you can open it with any text editor. The advantage of standard formats is that there are plenty of tools around to read and process the data. You don't need Coma to make sense of the data files. For long-term storage, if the plain text files are too big, simply compress them with a standard zip program.
2. One file per unique parameter set. While running the simulation, each unique parameter set produces one separate data file. Each such data file stores all meta data, including parameters, program, program version, and so on. Within the file parameters, results and meta data are stored as a tree—think of a file system hierarchy or nested dictionaries. The assumption behind the one-file-per-parameter-set approach is that each data point is expensive. If it takes a day to calculate a data point, then you probably don't mind having to deal with 10,000 files. That said, if you are regularly encountering hundreds of thousands of parameter sets, then Coma probably is not the right tool. Of course, there is some flexibilty in defining what a data point is—you could store a big vector or matrix for each single data point. For long-term storage all data files can be "archived" into one single file. An advantage of the one-file-per-parameter-set paradigm is again its simplicity and robustness. Standard file system tools is all you need to manage the data. In principle, you could do a very simple parallelization just by running programs on different computers and copying the files back and forth.
3. Python-centric workflow. Coma is a Python library and as such assumes a Python based workflow. Coma supports Boost.Serialization compatible XML files and therefore it should be relatively easy to use it with standalone C++ programs which utilize this library. A different approach is to compile the C++ code as a Python extension. (That's what I am personally using.) Still, in all cases, run scripts, data analysis and plotting would very likely be in Python.

3 Getting started with Coma

For the purpose of this tutorial our goal is to "simulate" a pendulum, that is, we will calculate its equation of motion $\theta(t)$ and compare it to the small-angle approximation. As so often, the Wikipedia page has all the details. I'll walk through this very simple example and show how to set up and use Coma. I assume that it is already correctly installed.

Before we get going, I should introduce two key concepts that Coma uses throughout: *Experiments* and *Measurements*. Each unique parameter set corresponds to one simulation run and one data file and this is called a *Measurement*. Consequently, the data file is called a measurement file and it stores all meta data, parameters and results of that particular run. Measurements are grouped together into an *Experiment*. Logically, an experiment might represent one or multiple parameter sweeps that belong together. On the file system level, an experiment is simply a directory that contains all the measurement files, plus a measurement index file that records how many measurements exist and, finally, an experiment file which keeps meta data associated with the experiment.

3.1 Setup

```
In [1]: import numpy as np
```

```
import matplotlib.pyplot as plt
import coma
print('Coma version: ' + coma.__version__)
```

```
Coma version: 2.0.0-beta.2.7.gec64a74
```

```
In [2]: coma.create_default_config()
```

```
Creating directory /home/burkhard/.config/coma
Creating config file "/home/burkhard/.config/coma/preferences.conf"
Creating experiment index
"/home/burkhard/.config/coma/experiment.index"
```

Coma comes with a convenience function `create_default_config()` to, well, create a default configuration file and a global experiment index. Both files are purely optional. The default config file should be useful to see what config options are available. The global experiment index is used to automatically assign unique ids to new experiments. The experiment id can also be specified as an argument to the Experiment constructor. If there is neither a global index nor a specified id argument, then the id of a newly created experiment is simple `None`, which is perfectly fine as well.

Below we create a new experiment in the directory 'example_experiment'. As we created a global experiment index above and as this is our first experiment, its id will be `1`. This is reflected in the experiment file's filename. The default file format is JSON. The experiment filename as well as the default format are configurable (see the config file). The configuration can also be overwritten per experiment by using the Experiment constructor's `config` argument. If the experiment in the directory `example_experiment` already exists, then it is simply opened and its data loaded.

```
In [3]: e = coma.Experiment('example_experiment')
        !ls example_experiment/
```

```
experiment.000001.json  measurement.index.json
```

3.2 Defining and running measurements

To do something interesting with the experiment we define parameter sets. Our simulation will be run once for each set. First we define the parameters: For the pendulum example there are two parameters, `theta_0` and `theta`. The paths—the second item in the tuples below—are used to retrieve these parameters from the measurement files. Then we add a number of parameter sets by calling `add_parameter_set`.

```
In [4]: e.define_parameter_set(('theta_0','parameters/theta_0'),
                               ('theta','parameters/theta'))
        for theta_0 in [-5.0,-30.0,-60.0]:
            for theta in np.linspace(theta_0,-theta_0,20):
                e.add_parameter_set(theta_0,round(theta,4))
```

So now we have added 60 parameter sets and we would like to run our simulation. To calculate $t(\theta)$ for the pendulum we integrate numerically:

$$t(\theta') = \int_{\theta_0}^{\theta'} \mathrm{d}\theta \left[\frac{2g}{L}(\cos\theta - \cos\theta_0)\right]^{-\frac{1}{2}}$$

We define the function `run_it` to run the simulation. The function takes one argument, a ParameterSet with the parameters θ_0 and θ as defined above, does the integration, calculates the small-angle approximation $t_{approx}(\theta)$ as well, and returns its results and parameters as a dictionary.

In [5]:
```
from collections import OrderedDict
import scipy.constants as const
import scipy.integrate
import math

def run_it(p):
    #print('Computing parameter set {}'.format(p))

    f = lambda theta_,theta_0: \
        1/math.sqrt( math.cos(theta_) - math.cos(theta_0) )

    F = lambda theta_0,theta: \
        scipy.integrate.quad(f, theta_0, theta, (theta_0,)) \
        if theta!=theta_0 else (0,0)

    t = lambda theta_0,theta,g,L: \
        math.sqrt(L/(2*g)) * \
        F(theta_0*math.pi/180,theta*math.pi/180)[0]

    t_ = lambda theta_0,theta,g,L: \
         math.sqrt(L/g) * math.acos(theta/theta_0)

    g = const.g
    L = 1.0
    time = t(p.theta_0, p.theta, g, L) # exact
    time_ = t_(p.theta_0, p.theta, g, L) # small angle approximation

    ps = OrderedDict()
    ps['theta_0'] = p.theta_0
    ps['theta'] = p.theta
    ps['L'] = L

    rs = OrderedDict()
    rs['t'] = time
    rs['t_'] = time_

    o = OrderedDict()
    o['parameters'] = ps
    o['results'] = rs

    return o
```

Here, `p` is a ParameterSet. We can access the parameters via the names we defined in `Experiment.define_parameter_set()`, e.g `p.theta_0` and `p.theta`. Alternatively, `p['parameters/theta_0']` and `p['parameters/theta']` works just as well. The function returns a nested dictionary—a tree. This tree gets written to the measurement file and should contain the parameters used, the results and potentially additional meta data, for example, which computer we are running on, date and time, and so on. While a normal Python dictionary will work, I recommend an OrderedDict so that the file structure is always the same, with a well-defined order of the items in the tree.

With everything in place we can now actually run all the measurements. To do that, we use the method `Experiment.run` (oho!). As its only argument it expects a function which does the actual calculations—our function `run_it` from above.

In [6]:
```
e.run(run_it)
```

Out [6]:
```
(60, 60)
```

In [7]:
```
e.run(run_it)
```

Out [7]:

```
(0, 60)
```

In [8]:

```
e.number_of_measurements()
```

Out [8]:

```
60
```

`Experiment.run` returns a two-tuple, indicating how many measurements out of the total number of defined measurements were run. Thus, the first time we run the experiment, we see that all 60 out of 60 measurements were calculated. The second time no measurements are calculated, because all of them already exist. This makes it easy to resume an experiment where a couple of measurements were not successful, or to gradually expand an experiment by defining successively more parameter sets.

Let's have a look at our experiment directory.

In [9]:

```
!ls -w 72 example_experiment/
```

```
experiment.000001.json   measurement.000031.json
measurement.000001.json  measurement.000032.json
measurement.000002.json  measurement.000033.json
measurement.000003.json  measurement.000034.json
measurement.000004.json  measurement.000035.json
measurement.000005.json  measurement.000036.json
measurement.000006.json  measurement.000037.json
measurement.000007.json  measurement.000038.json
measurement.000008.json  measurement.000039.json
measurement.000009.json  measurement.000040.json
measurement.000010.json  measurement.000041.json
measurement.000011.json  measurement.000042.json
measurement.000012.json  measurement.000043.json
measurement.000013.json  measurement.000044.json
measurement.000014.json  measurement.000045.json
measurement.000015.json  measurement.000046.json
measurement.000016.json  measurement.000047.json
measurement.000017.json  measurement.000048.json
measurement.000018.json  measurement.000049.json
measurement.000019.json  measurement.000050.json
measurement.000020.json  measurement.000051.json
measurement.000021.json  measurement.000052.json
measurement.000022.json  measurement.000053.json
measurement.000023.json  measurement.000054.json
measurement.000024.json  measurement.000055.json
measurement.000025.json  measurement.000056.json
measurement.000026.json  measurement.000057.json
measurement.000027.json  measurement.000058.json
measurement.000028.json  measurement.000059.json
measurement.000029.json  measurement.000060.json
measurement.000030.json  measurement.index.json
```

In [10]:

```
!cat example_experiment/measurement.000001.json
```

```
{
  "measurement": {
    "info": {
      "measurement_id": 1,
      "start_date": "2014-02-25T22:50:11Z",
      "end_date": "2014-02-25T22:50:11Z"
    },
    "parameters": {
      "theta_0": -5.0,
      "theta": -5.0,
      "L": 1.0
    },
    "results": {
      "t": 0.0,
      "t_": 0.0
    }
  }
}
```

Each measurement is stored in a separate file, named `measurement.[id].json`, where `id` is the id of the measurement and unique within each experiment. If you don't like the measurement filename, it can be customized via the config. Each measurement file contains the nested dictionary tree-structure which the `run_it` function returned, plus some general meta data added by Coma. Now we also see how the paths map to items in the tree: "parameters/-theta_0", for example, refers to "theta_0" under "parameters". We used these paths when defining the parameter sets and this is how Coma knows which measurements have already been calculated and which are still outstanding. The same path mechanism is used when extracting results from the data files and we will get to this in a minute.

3.3 Retrieving results and plotting

First, let's have a look how we can iterate over existing measurements. We also demonstrates different ways to access items in the tree of each measurement, via nested dictionary syntax `m['a']['b']`, via path syntax `m['a/b']`, or as object properties `m.a.b`, where the last one is arguably the most convenient.

In [11]:
```
for m in e.measurements():
    if m.id > 3:
        break
    print(m)
    print('  theta_0 = {:.2f}, theta = {:.2f}, t = {:.2f}'
          .format(m['parameters']['theta_0'],
                  m['parameters/theta'],
                  m.results.t))
    print('')
```

```
Measurement 1
  start_date: 2014-02-25T22:50:11Z
  end_date: 2014-02-25T22:50:11Z
  Fields: [u'info', u'parameters', u'results']

  theta_0 = -5.00, theta = -5.00, t = 0.00

Measurement 2
  start_date: 2014-02-25T22:50:11Z
  end_date: 2014-02-25T22:50:11Z
```

```
    Fields: [u'info', u'parameters', u'results']

    theta_0 = -5.00, theta = -4.47, t = 0.15

  Measurement 3
    start_date: 2014-02-25T22:50:11Z
    end_date: 2014-02-25T22:50:11Z
    Fields: [u'info', u'parameters', u'results']

    theta_0 = -5.00, theta = -3.95, t = 0.21
```

Coma provides a more convenient way to extract data from measurements for common use cases: We can specify the columns of a data table as well as a set of parameters which differentiates different results. In both cases we specify a short name to access the column or parameter and a path that maps to the correct item in the measurement tree. For the pendulum example, we would like to plot $\theta(t)$ and $\theta(t_{approx})$ and therefore we put `theta`, `t`, and `t_` in the result table. We would like one graph for each θ_0 and consequently use `theta_0` as the parameter set. We use the method `Experiment.retrieve_results` which returns a list of `Results`.

In [12]:
```
table_def = [('theta','parameters/theta'),
             ('t','results/t'),
             ('t_','results/t_')]

p_def = [('theta_0','parameters/theta_0')]

rs = e.retrieve_results(table_def, p_def)
rs
```

Out [12]:
```
[Result((theta,t,t_),(theta_0=-5.0)),
 Result((theta,t,t_),(theta_0=-30.0)),
 Result((theta,t,t_),(theta_0=-60.0))]
```

In [13]:
```
r = rs[0]
print(r.parameters)
print(r.parameters.theta_0)
print(r.table_columns)
print(r.table.shape)
```

```
(theta_0=-5.0)
-5.0
('theta', 't', 't_')
(20, 3)
```

In [14]:
```
for r in rs['theta_0',-5.0]:
    print(r)
```

```
Result((theta,t,t_),(theta_0=-5.0))
```

As there are three distinct values for θ_0, `retrieve_results` returns a list of three Result objects. Each Result object knows its parameters (`Result.parameters`), its table columns (`Result.table_columns`) and the table itself (`Result.table`), a Numpy array. In the example above, the table has 20 rows and 3 columns, which is, of course, as expected.

One neat feature of result lists is that they can be filtered. If `rs` is a result list then `rs['A',1]` only returns all results where the parameter $A = 1$. These filters can be chained, for example `rs['A',1]['B',2]['C','funky']` would return all results where $A = 1, B = 2$, and $C = \text{funky}$. As result lists are just normal lists (they subclass Python's `list` class) you can still sort them and iterate over them as you are used to. For the pendulum example with only one parameter differentiating the results (θ_0) , filtering is not that impressive. However, it is super useful if you have lots of parameters. For now parameters can only be filtered by their exact value. More complex expressions à la Numpy, e.g. rs[('A' > 3) & ('A' < 5)], are not yet supported.

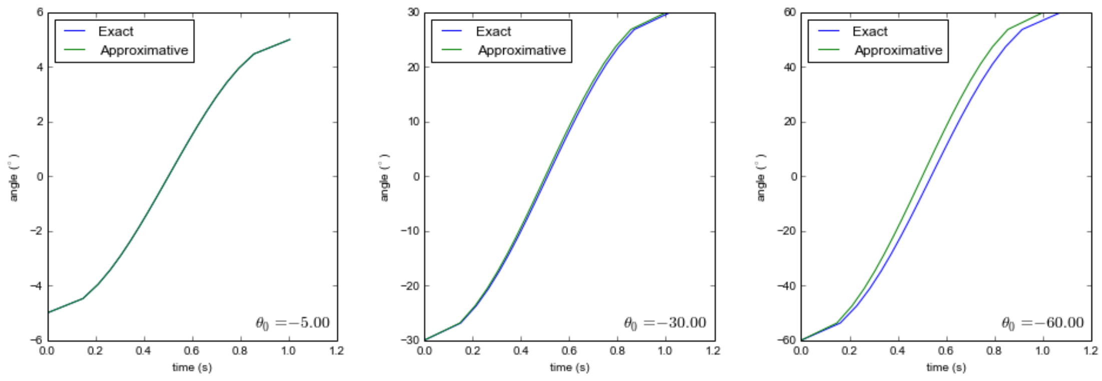
For the data table, `Experiment.retrieve_results` will "unfold" results and parameters which are lists. For example, if you are storing a result `m['results/energies']`, a list of numbers, and are retrieving those energies via a table specification `[ ... ('Es','results/energies') ... ]`, then the data table will have columns `...Es_1, Es_2, Es_3,...`. The number of columns will depend on how many energies there are. This unfolding is very convenient for some use cases, but not always the behaviour you want. If you want to store and retrieve a list as is, without unfolding, store it as a Numpy array—Coma knows how to store and load Numpy arrays. Thus you could use `o['results']['energies'] = np.array([1,2,3])` in the function `run_it` above and then retrieve it the same way as any other result and get your original Numpy array.

We now plot the previously extracted results. The variable `rs` from above contains the three results for the three different θ_0 and `r.table` is a Numpy array with the columns θ, t, and t_{approx}. We iterate over the rs and create one sub-plot for each θ_0.

In [15]:
```
%matplotlib inline

rows,cols = (1,3)
fig = plt.figure(figsize=(5.3*cols,5*rows))
i_p = 0

for r in rs:
    i_p += 1
    p = fig.add_subplot(rows,cols,i_p)
    p.plot(r.table[:,1],r.table[:,0],label='Exact')
    p.plot(r.table[:,2],r.table[:,0],label='Approximative')
    p.set_xlabel('time (s)')
    p.set_ylabel('angle ($^\circ$)')
    p.legend(loc='upper left')
    p.text(0.98,0.02,'$\\theta_0 = {:.2f}$'.format(r.parameters.theta_0),
           fontsize='14', verticalalignment='bottom',
           horizontalalignment='right', transform=p.transAxes)
fig.subplots_adjust(wspace=0.3)
```



We have plotted the angle over time for a pendulum for half a period. We plotted both the exact solution as well as the small-angle approximation. For a small initial angle $\theta_0 = -5°$ the exact and approximative solution agree, for larger initial angles $\theta_0 = -30°, \theta_0 = -60°$ the solutions increasingly diverge. Of course, with the small-angle approximation being called small-angle approximation, that's exactly what we expect.

Granted, this is probably not the most exciting graph in the history of science. (I am open to suggestions for simple, physically meaningful and impressive numerical simulations.) But we have demonstrated how to use Coma to define a set of parameters, run the simulation for these defined parameters, store the data to measurement files, extract results in a flexible and convenient manner and finally produce a graph with the extracted data. In short, we have demonstrated the basic workflow that Coma is designed to support and facilitate.

3.4 Archiving and serialization

We've introduced the basic workflow above, but there are a few more of Coma's features worth knowing about. First, experiments can be archived. We "deactivate" an experiment:

```
In [16]: print('Experiment is active: {}'.format(e.isactive()))
         print('Deactivating...')
         e.deactivate()
         print('Experiment is active: {}'.format(e.isactive()))
         !ls example_experiment/
```

```
Experiment is active: True
Deactivating...
Experiment is active: False
experiment.000001.json
```

For long term storage it is likely not desirable to have thousands of measurement files lying around. Therefore, experiments support the notion of being active of inactive. An inactive experiment absorbs all the measurements into the single experiment file. You can think of deactivating as attaching all of the measurement trees as branches to the one large experiment tree. A deactivated experiment can still be opened, and results can be retrieved and plotted. The only difference is that no new measurements can be added. To do that, the experiment first has to be reactivated:

```
In [17]: e.activate()
         !ls -w 72 example_experiment/
```

```
experiment.000001.json   measurement.000031.json
measurement.000001.json  measurement.000032.json
measurement.000002.json  measurement.000033.json
measurement.000003.json  measurement.000034.json
measurement.000004.json  measurement.000035.json
measurement.000005.json  measurement.000036.json
measurement.000006.json  measurement.000037.json
measurement.000007.json  measurement.000038.json
measurement.000008.json  measurement.000039.json
measurement.000009.json  measurement.000040.json
measurement.000010.json  measurement.000041.json
measurement.000011.json  measurement.000042.json
measurement.000012.json  measurement.000043.json
measurement.000013.json  measurement.000044.json
measurement.000014.json  measurement.000045.json
measurement.000015.json  measurement.000046.json
measurement.000016.json  measurement.000047.json
measurement.000017.json  measurement.000048.json
measurement.000018.json  measurement.000049.json
measurement.000019.json  measurement.000050.json
measurement.000020.json  measurement.000051.json
measurement.000021.json  measurement.000052.json
measurement.000022.json  measurement.000053.json
measurement.000023.json  measurement.000054.json
measurement.000024.json  measurement.000055.json
measurement.000025.json  measurement.000056.json
measurement.000026.json  measurement.000057.json
measurement.000027.json  measurement.000058.json
measurement.000028.json  measurement.000059.json
measurement.000029.json  measurement.000060.json
measurement.000030.json  measurement.index.json
```

Non-trivial numerical simulations are often implemented as a class. An instance of such a simulation might in turn contain other complex objects. Coma can serialize a whole object hierarchy, as long as all objects are either supported primitives (numbers, strings, lists, dictionaries, Numpy arrays, and so on) or implement Coma's serialization protocol. This protocol requires that each object implements a method `coma_getstate()` which returns an (ordered) dictionary, representing its state. The name of this method is configurable, so you could, for example, reuse Python's pickle protocol `__getstate__()`, should that be convenient for your use case. Below is an example implementing the pendulum simulation as a class, which is then serialized via the Coma serialization protocol.

In [18]:
```python
from collections import OrderedDict
import scipy.constants as const

class Pendulum:
    def __init__(self):
        self.program = 'pendulum'
        self.version = '0.0.1'
        self.g = const.g
        self.L = 1.0
        self.theta_0 = None
        self.theta = None
        self.t = None
        self.t_ = None

    def init(self):
        pass

    def run(self):
        f = lambda theta_,theta_0: \
            1/math.sqrt( math.cos(theta_) - math.cos(theta_0) )

        F = lambda theta_0,theta: \
            scipy.integrate.quad(f, theta_0, theta, (theta_0,)) \
            if theta!=theta_0 else (0,0)

        t = lambda theta_0,theta,g,L: \
            math.sqrt(L/(2*g)) * \
            F(theta_0*math.pi/180,theta*math.pi/180)[0]

        t_ = lambda theta_0,theta,g,L: \
             math.sqrt(L/g) * math.acos(theta/theta_0)

        self.t = t(self.theta_0, self.theta, self.g, self.L)
        self.t_ = t_(self.theta_0, self.theta, self.g, self.L)

    def coma_getstate(self):
        ps = OrderedDict()
        ps['theta_0'] = self.theta_0
        ps['theta'] = self.theta
        ps['L'] = self.L

        rs = OrderedDict()
        rs['t'] = self.t
        rs['t_'] = self.t_

        o = OrderedDict()
        o['parameters'] = ps
        o['results'] = rs
        return o

def run_it(p):
    s = Pendulum()
    s.theta_0 = p.theta_0
    s.theta = p.theta
    s.init()
    s.run()
    return s
```

Now `run_it` simply instantiates an object of the Pendulum class, runs it and finally returns the Pendulum instance—which is automatically serialized by Coma and saved to the measurement file. If the class defines instance variables `program` and `version` then these will be saved to the file as well. Let's run the experiment again, this time using the Pendulum class. `Experiment.reset` resets the experiment and deletes all existing measurements.

```
In [19]: e.reset()
         e.number_of_measurements()
```

```
Out [19]:
         0
```

```
In [20]: e.run(run_it)
```

```
Out [20]:
         (60, 60)
```

```
In [21]: !cat example_experiment/measurement.000001.json
```

```
{
  "measurement": {
    "info": {
      "measurement_id": 1,
      "start_date": "2014-02-25T22:50:16Z",
      "end_date": "2014-02-25T22:50:16Z",
      "program": "pendulum",
      "version": "0.0.1"
    },
    "parameters": {
      "theta_0": -5.0,
      "theta": -5.0,
      "L": 1.0
    },
    "results": {
      "t": 0.0,
      "t_": 0.0
    },
    "__class__": "__main__.Pendulum"
  }
}
```

3.5 Running measurements in parallel

As mentioned before, Coma comes with support for simple job-level parallelization via IPython.parallel, which is implemented by ParallelExperiment, a subclass of Experiment. In addition to the arguments of the constructor of Experiment, ParallelExperiment's constructor accepts an optional keyword argument `profile` to specify which IPython profile to use. Apart from that ParallelExperiment can be used exactly the same way as Experiment. It will simply run all measurements in parallel on the active engines in the given IPython profile. Have a look at the IPython.parallel documentation for details on how to set up the controller and engines.

IPython.parallel uses Python's pickle module to serialize an object and send it from the controller to an engine (either on the same or a different computer). This is something worth keeping in mind and for more complex objects you'll likely have to implement the Pickle protocol, for example the methods `__getstate__` and `__setstate__`. Local class and function definitions can be made known to the engines via IPython's `@require` decorator. In my experience, this sometimes needs some tweaks to the class definitions. For example, I've found that extensive use of closures (i.e. lambdas which reference local and class variables) does not always work well with `@require`.

In the future I would like to provide a more thorough example, extending the Pendulum class above, to demonstrate how IPython.parallel can be used with Coma for simple parallelization of parameter sweeps.

4 Conclusion

Coma is a small Python library that aids with conducting "numerical experiments"—running numerical simulations. Simulation runs—called Measurements—are grouped into Experiments. Coma helps with defining measurements, running these measurements and storing the results, parameters, and meta data for each measurement. An experiment allows to easily and flexibly retrieve results from all its measurements. For long term storage each experiment with all its measurements can be archived into a single file. Coma uses plain text, standard file formats and is aimed at handling small data volumes. It favours simplicity, flexibility and robustness. The produced data files can be opened and processed with standard tools and libraries, and therefore remain meaningful even when Coma is not available.

In this document I introduced Coma's basic assumptions and key concepts and then demonstrated a typical workflow. A pendulum was used as a very simple example of a numerical simulation. For the pendulum, a numerical experiment was conducted, where we defined three parameter sweeps, ran the experiment, retrieved and finally plotted the results.

Coma is concerned with improving data management. However, its largest conceptual unit is the experiment and in general how those experiments are organized on the file system will vary a lot depending on the particular project and personal preference. I tend to simply give numerical names to my experiment directories (e.g. `experiment.000012`, where 12 is the id) and then group experiments together in logical units, each unit usually with an IPython notebook that contains analysis and graphs. I have a Python run script for each experiment. These run scripts simply define all parameter sets and then run the experiment.

I wrote Coma primarily for my own needs, but made it publicly available in the hope that it might be useful for others. If you think it is useful, if you think it is not useful, or if you are even using it, any thoughts, comments or questions, I would love to hear from you!

Burkhard Ritter (burkhard@seite9.de), February 2014.