Parampool Tutorial

Hans Petter Langtangen[1,2] (hpl@simula.no)

Anders Elstad Johansen[1]

[1]Center for Biomedical Computing, Simula Research Laboratory
[2]Department of Informatics, University of Oslo

Oct 19, 2015

Abstract

Parampool is a Python package for handling a potentially large pool of input parameters in scientific applications. The simplest use is to pass a Python function to Parampool and get back a web interface for setting the arguments to the function. More powerful use consists in defining a *pool* of input parameters, which is a tree structure of input data, where data items (parameters) are organized hierarchically in subpools. Each data item is associated with a name, default value, unit, help text, widget type, validation information, etc. Different types of sophisticated user interfaces can then be automatically generated: a graphical web interface (via Flask or Django), a command-line interface, and a file-based interface. The tutorial describes specific examples on how to to program Parampool to generate user interfaces and how to operate them. With very little efforts, you can take a scientific application and equip it with a fancy GUI.

Contents

1 Simulation program used as sample app

We shall work with a sample application that takes a few variables as input and produces numbers and/or graphs as result. Specifically, our application, later referred to as a *simulation program* or simply a *simulator*, concerns the simulation of a ball thrown through air. Given the initial velocity of the ball, and some other data like mass and radius, we can compute the trajectory of the ball until it hits the ground. The details of the calculations are not of interest here as our focus is on software for assigning input data and for displaying the results. However, the interested reader can consult the box below for the inner workings of the simulation program.

Mathematical model.

The motion of the mass center $\boldsymbol{r}$ of a body through a fluid is given by

$$m\frac{d^2\boldsymbol{r}}{dt^2} = -m\boldsymbol{g} - \frac{1}{2}C_D\varrho A v^2\boldsymbol{i}_t + \frac{1}{2}C_L\varrho A v^2\boldsymbol{i}_n + \frac{1}{2}C_S\varrho A v^2(\boldsymbol{i}_t \times \boldsymbol{i}_n), \quad (1)$$

where

- m is the mass of the body,

- $\boldsymbol{g}$ is the acceleration of gravity vector,
- C_D is a drag coefficient,
- ϱ is the density of air,
- A is the cross-section area of the body perpendicular to the motion,
- $\boldsymbol{v} = d\boldsymbol{r}/dt$ is the velocity of the body,
- $\boldsymbol{w}$ is a given wind velocity,
- $v = |\boldsymbol{v} - \boldsymbol{w}|$ is the relative velocity between the body and the wind,
- C_L is a lift coefficient,
- C_S is a coefficient for the sidewind or lateral aerodynamic force,
- $\boldsymbol{i}_t$ is a unit tangent vector of the body's path, while
- $\boldsymbol{i}_n$ is a unit vector normal to the path tilting upwards.

The drag C_D coefficient for a sphere is taken as 0.45. The lift coefficient C_L depends on the spinrate ω (rad s^{-1}) of the body, and a simple linear relation often suffices: $C_L = 0.2\omega/500$. A negative ω gives a negative lift.

We can simplify the model for a two-dimensional motion in an xy plane with unit vectors $\boldsymbol{i}$ and $\boldsymbol{j}$ in the x and y directions, respectively. Then we skip the sidewind force ($C_S = 0$). We also let gravity point downwards, $\boldsymbol{g} = -g\boldsymbol{j}$, and let the wind velocity be horizontal: $\boldsymbol{w} = -w\boldsymbol{i}$. Furthermore, we have that

$$\boldsymbol{i}_t = \frac{\boldsymbol{v}}{|\boldsymbol{v}|} \equiv a\boldsymbol{i} + b\boldsymbol{j}, \quad \boldsymbol{v} = \frac{d\boldsymbol{r}}{dt}, \tag{2}$$

$$\boldsymbol{i}_n = -b\boldsymbol{i} + a\boldsymbol{j} \text{ if } a > 0 \text{ else } b\boldsymbol{i} - a\boldsymbol{j} \tag{3}$$

The initial conditions associated with (1) express that the body starts at the origin with an initial velocity v_0 making an angle θ with the horizontal. In the two-dimensional case the conditions become

$$\boldsymbol{r}(0) = 0\boldsymbol{i} + 0\boldsymbol{j}, \quad \frac{d\boldsymbol{r}}{dt}(0) = \boldsymbol{v}(0) = v_0\cos\theta\boldsymbol{i} + v_0\sin\theta\boldsymbol{j}.$$

2 User interfaces for Python functions

Parampool can automatically generate user interfaces for communicating with a given function. The usage of this functionality will be explained in problems

of increasing complexity, using the trajectory of a ball as described above as application.

2.1 Real numbers as input and output

Suppose you have some function

```
def compute_drag_free_landing(initial_velocity, initial_angle):
    ...
    return landing_point
```

This function returns the landing point on the ground (`landing_point`) of a ball that is initially thrown with a given velocity in magnitude (`initial_velocity`), making an angle (`initial_angle`) with the ground. There are two real input variables and one real output variable. The function must be available in some module, here the module is simply called `compute.py` (and it also contains a lot of other functions for other examples).

In the following we shall refer to functions like `compute_drag_free_landing`, for which we want to generate a web interface, as a *compute function*.

Flask interface. Flask is a tool that can be used to write a graphical user interface (GUI) to be operated in a web browser. Here we shall use Flask to create a GUI for our compute function, as displayed in Figure 1. To this end, simply create a Python file `generate.py` with the following lines:

```
from parampool.generator.flask import generate
from compute import compute_drag_free_landing

generate(compute_drag_free_landing, default_field='FloatField')
```

The `generate` function grabs the arguments in our compute function and creates the necessary Flask files.

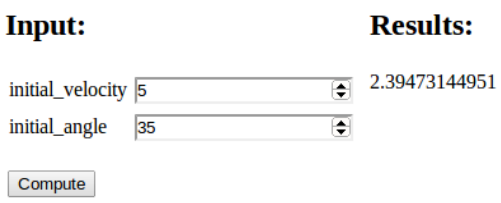


Figure 1: A simple web interface.

Tip.

We recommend to make a new directory for every web application. Since you need access to the `compute` module you must copy `compute.py` to the directory or modify `PYTHONPATH` to contain the path to the directory where `compute.py` resides.

Since the `generate` tool has no idea about the type of variable of the two positional arguments in the compute function, it has to assume some type. By default this will be text, but we can change that behavior to be floats by the setting the `default_field` argument to `FloatField`. This means that the generated interface will (only) accept float values for the input variables, which is sufficient in our case.

A graphical Flask-based web interface is generated by running

```
Terminal> python generate.py
```

Warning.

A message is written in the terminal window, saying that with positional arguments in the compute function, one must normally invoke the generated `controller.py` file and do some explicit conversion of text read from the web interface to the actual variable type accepted by the compute function. This potential manual work can be avoided by using keyword arguments only, so the generator functionalty can see the variable type.

You can now view the generated web interface by running

```
Terminal> python controller.py
```

and open your web browser at the location `http://127.0.0.1:5000/`. Fill in values for the two input variables and press *Compute*. The page in the Chrome browser will now look like Figure 1. Other browsers (Firefox, for instance) may have a slightly different design of the input fields. The figures in this tutorial were made with the Chrome and Opera browsers.

Generated files:

Readers with knowledge of Flask will notice that some files with Flask code have been generated:

- `model.py` with a definition of the forms in the web interface
- `controller.py` which glues the interface with the compute function
- `templates/view.html` which defines the design of the web interface

You are of course free to tailor these files to your needs if you know about Flask programming. An introduction to Flask for scientific applications is provided in [1]. A one-line Bash script, `clean.sh`, is also generated: it will remove all files that were generated by running `generate.py`.

Django interface. Django is a very widespread and popular programming environment for creating web applications. We can easily create our web application in Django too. Just replace `flask` by `django` in `generate.py`:

```
from parampool.generator.django import generate
from compute import compute_drag_free_landing

generate(compute_drag_free_landing, default_field='FloatField')
```

The Django files are now in the directory tree `drag_free_landing` (same name as our compute function, except that any leading `compute_` word is removed). Run the application by

```
Terminal> python drag_free_landing/manage.py runserver
```

and open your browser at the location `http://127.0.0.1:8000/`. The interface looks the same and has the same behavior as in the Flask example above.

Generated files:

Quite some files are needed for a Django application. These are found in the `drag_free_landing` directory tree. The most important ones are

- `models.py` with a definition of the forms in the web interface
- `views.py` which glues the interface with the compute function
- `templates/index.html` which defines the design of the web interface

With some knowledge of basic Django programming you can edit these files to adjust the functionality. Reference [1] provides a basic introduction to Django for creating scientific applications.

2.2 A plot as output

The result of the previous computation was just a number. Let us instead make a plot of the trajectory of a ball without any air resistance. The function

```
def compute_drag_free_motion_plot(
    initial_velocity=5.0,
    initial_angle=45.0):
    ...
    return html_text
```

is now our compute function, in `compute.py`, which takes the same two input arguments as before, but returns some HTML text that will display a plot in the browser window. This HTML text is basically the inclusion of the image file containing the plot,

```
<img src="X">
```

where `X` is the name of the file. However, if you do repeated computations, the name of the image file must change for the browser to update the plot. Inside the compute function we must therefore generate a unique name of each image file. For this purpose, we can use the number of seconds since the Epoch (January 1, 1970) as part of the filename, obtained by calling `time.time()`. In addition, the image file must reside in a subdirectory `static`. The appropriate code is displayed below.

```
import matplotlib.pyplot as plt
...
def compute_drag_free_motion_plot(
    initial_velocity=5.0,
    initial_angle=45.0):
    ...
    plt.plot(x, y)
    import time  # use time to make unique filenames
    filename = 'tmp_%s.png' % time.time()
    if not os.path.isdir('static'):
        os.mkdir('static')
    filename = os.path.join('static', filename)
    plt.savefig(filename)
    html_text = '<img src="%s" width="400">' % filename
    return html_text
```

The string version of the object returned from the compute function is inserted as *Results* in the HTML file, to the right of the input. By returning the appropriate HTML text the compute function can tailor the result part of the page to our needs.

Flask application. The `generate.py` file for this example is similar to what is shown above. Only the name of the compute function has changed:

```
from parampool.generator.flask import generate
from compute import compute_drag_free_motion_plot

generate(compute_drag_free_motion_plot)
```

Tip.

This time we do not need to specify `default_field` because we have used keyword arguments with default values in the compute function. The `generate` function can then from the default values see the type of our

arguments. Remember to use `float` default values (like 5.0) and not simply integers (like 5) if the variable is supposed to be a `float`.

We run `python generator.py` to generate the Flask files and then just writing `python controller.py` starts the web GUI. Now the default values appear in the input fields. These can be altered, or you can just click *Compute*. The computations result in a plot as showed in Figure 2.

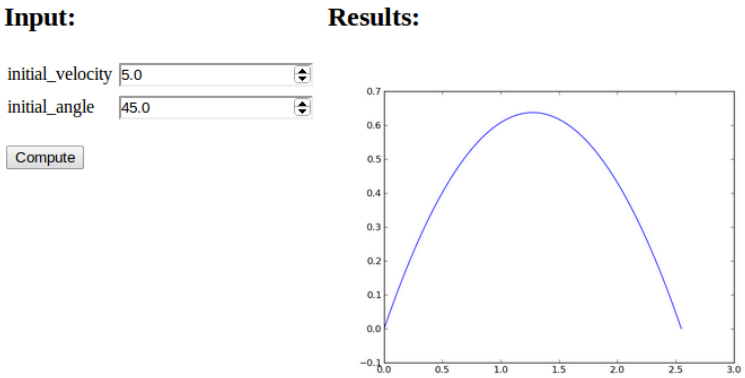


Figure 2: A web interface with graphics.

Django application. The corresponding Django application is generated by the same `generator.py` code as above, except that the word `flask` is replaced by `django`. The Django files are now placed in the `drag_free_motion_plot` subdirectory, and the web GUI is started by running

```
Terminal> python drag_free_motion_plot/manage.py runserver
```

The functionality of the GUI is identical to that of the Flask version.

Comparing graphs in the same plot. With a little trick we can compare several trajectories in the same plot: inserting `plt.figure(X)` makes all `plt.plot` calls draw curves in the same figure (with figure number `X`). We introduce a boolean parameter `new_plot` reflecting whether we want a new fresh plot or not,

```
def compute_drag_free_motion_plot2(
    initial_velocity=5.0,
    initial_angle=45.0,
    new_plot=True):
```

and add the following code before the `plt.plot` call:

```
global fig_no
if new_plot:
    fig_no = plt.figure().number
else:
    plt.figure(fig_no)
plt.plot(x, y, label=r'$v=%g,\ \theta=%g$' %
         (initial_velocity, initial_angle))
plt.legend()
```

The `new_plot` parameter will turn up as a boolean variable in the web interface, and when checked, we create a new figure. Otherwise, we draw curves in the existing figure number `fig_no` which was initialized last time `new_plot` was true (with a global variable we ensure that the value of `fig_no` survives between the calls to the compute function). Figure 3 displays an attempt to not check `new_plot` and compare the curves corresponding to three different parameters (the files are in the `flask3` directory.

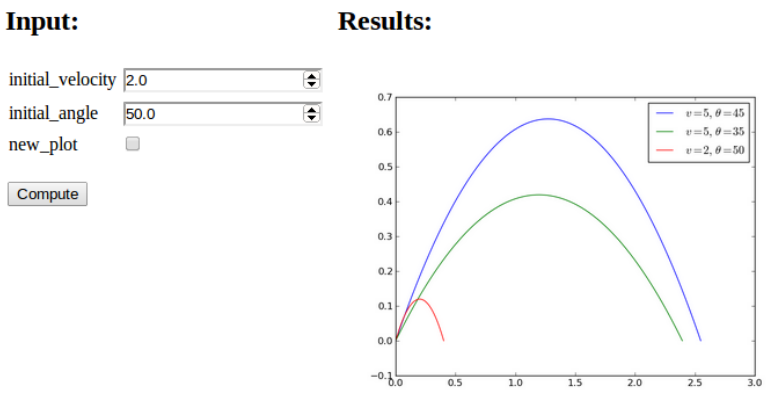


Figure 3: Plot with multiple curves.

Caveat.

If `new_plot` is unchecked before the first computation is carried out, `fig_no` is not defined when we do `plt.figure(fig_no)` and we get a `NameError` exception. A fool-proof solution is

```
if new_plot:
    fig_no = plt.figure().number
else:
    try:
        plt.figure(fig_no)
```

```
    except NameError:
        fig_no = plt.figure().number
```

Avoiding plot files.

The compute function generates plot files with unique names, but we can avoid making files at all and just insert the PNG code of the plot, using base64 encoding, as a long string directly in the HTML image tag. The statements below sketches the idea:

```
import matplotlib.pyplot as plt
# make plot
from StringIO import StringIO
figfile = StringIO()
plt.savefig(figfile, format='png')
figfile.seek(0)  # rewind to beginning of file
figdata_png = figfile.buf  # extract string
import base64
figdata_png = base64.b64encode(figdata_png)
html_text = '<img src="%s" width="400">' % \
            figdata_png
```

There is a convenient function `parampool.utils.save_png_to_str` performing the statements above and returning the `html_text` string:

```
from parampool.utils import save_png_to_str
# make plot in plt (matplotlib.pyplot) object
html_text = save_png_to_str(plt, plotwidth=400)
```

With this construction one can very easily avoid plot files and embed the plot directly in the HTML code `html_text`:

```
<img src="..." width="...">
```

Matplotlib without X server.

Matplotlib is by default configured to work with a graphical user interface which may require an X11 connection. When running applications on a web server there is a possibility that X11 is not enabled, and the user will get an error message. Matplotlib thus needs to be configured for use in such environments. The configuration depends on what kinds of images the user wants to generate, but in most cases it is sufficient to use the `Agg` backend. The `Agg` backend is created to make PNG files, but it also recognizes other formats like PDF, PS, EPS and SVG. The backend needs to be set before importing pyplot or pylab:

```
import matplotlib as mpl
mpl.use('Agg')
import matplotlib.pyplot as plt
```

Bokeh plotting. One can use the Bokeh library for plotting instead of Matplotlib, see [1] for an example. The major problem is that Parampool generates the `view.html` file and the head and body parts of the HTML file generated by Bokeh must be inserted in the `view.html` file at the right places. This can be done manually or by a suitable script.

mpld3 plotting. The mpld3 library can be used to convert Matplotlib plots to a string containing all the HTML code for the plot:

```
# Plot array y vs x
import matplotlib.pyplot as plt, mpld3
fig, ax = plt.subplots()
ax.plot(x, y)
html_text = mpld3.fig_to_html(fig)
```

It is relatively easy to create interactive plots with mpld3.

Pandas highcharts plotting. The pandas-highcharts package is another strong and popular alternative for interative plotting in web pages.

2.3 More input parameters and results

It is time to address a more complicated application: we want to compute the trajectory of a ball subject to air drag and lift and compare that trajectory to the one where drag and lift are omitted. We also want to visualize the relative importance between the three forces: gravity, drag, and lift. The lift is caused by spinning the ball.

The function that performs the computations has the following signature:

```
def compute_motion_and_forces0(
    initial_velocity=5.0,
    initial_angle=45.0,
    spinrate=50.0,
    w=0.0,
    m=0.1,
    R=0.11,
    method='RK4',
    dt=None,
    plot_simplified_motion=True,
    new_plot=True
    ):
```

and returns a formatted string `html_text` with two plots organized in an HTML table.

The returned HTML code. The technique described in the *Avoiding plot files* box at the end of Section 2.2 is implemented to embed PNG images directly in the HTML code. Under the plots there is a table of input values and the landing point. Curves can be accumulated in the plots (`new_plot=True`), with the corresponding data added to the table. A rough sketch of the HTML code returned from the compute function goes as follows:

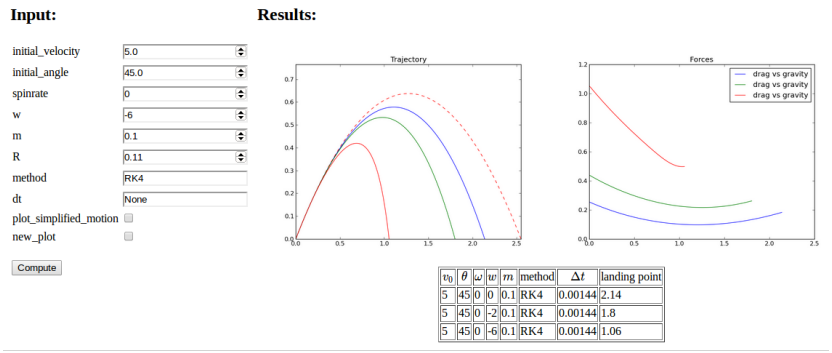
```
<table>
<tr>
<td valign="top">
<img src="..." width="400">
</td>
<td valign="top">
<img src="..." width="400">
</td>
</tr>
</table>

<center>
<table border=1>
<tr>
<td align="center"> \( v_0 \)  </td>
<td align="center"> \( \theta \) </td>
<td align="center"> \( \omega \) </td>
<td align="center"> \( w \)    </td>
<td align="center"> \( m \)    </td>
<td align="center"> \( R \)    </td>
<td align="center"> method     </td>
<td align="center"> \( \Delta t \) </td>
<td align="center"> landing point </td>
</tr>
<tr><td align="right"> 5 </td><td align="right"> 45 </td> ...</tr>
<tr><td align="right"> 5 </td><td align="right"> 45 </td> ...</tr>
<tr><td align="right"> 5 </td><td align="right"> 45 </td> ...</tr>
</table>
</center>
```

Note that we use MathJax syntax for having LaTeX mathematics in the table heading. All details about the computations and the construction of the returned HTML string can be found in the `compute.py` file.

Documentation of the application. The Parampool `generate` function applies the convention that any doc string of the compute function is copied and typeset verbatim at the top of the web interface. However, if the text `# (DocOnce format)` appears somewhere in the doc string, the text is taken as DocOnce source code and translated to HTML, which enables typesetting of LaTeX mathematics and computer code snippets (with nice pygments formatting).

The documentation of the web interface can therefore be included as a doc string in the compute function. Here is descriptive doc string using DocOnce syntax for LaTeX mathematics (equations inside `!bt` and `!et` commands) and monospace font for Python variables (names in backticks). The corresponding view in a browser is shown in Figure 5.

```
"""
This application computes the motion of a ball with radius $R$
```



v_0	θ	ω	w	m	method	Δt	landing point
5	45	0	0	0.1	RK4	0.00144	2.14
5	45	0	-2	0.1	RK4	0.00144	1.8
5	45	0	-6	0.1	RK4	0.00144	1.06

Figure 4: Web interface with two graphs.

```
and mass $m$ under the influence of gravity, air drag and lift
because of a given spinrate $\omega$. The motion starts with a
prescribed initial velocity $v_0$ making an angle initial_angle
$\theta$ with the ground. A wind velocity $w$, positive in
positive $x$ direction, can also be given.

The ordinary differential equation problem governing the
motion reads

!bt
\begin{align*}
m\frac{d^2\bm{r}}{dt^2} &= -mg\bm{j} -
\frac{1}{2}C_D\varrho A v^2\bm{i}_t +
\frac{1}{2}C_L\varrho A v^2\bm{i}_n\\
\bm{r}(0) &= 0\bm{i} + 0\bm{j}\\
\frac{d\bm{r}}{dt}(0) &= v_0\cos\theta\bm{i} + v_0\sin\theta\bm{j},
\end{align*}
!et
where $\bm{i}$ and $\bm{j}$ are unit vectors in the $x$ and $y$
directions, respectively, $g$ is the acceleration of gravity,
$A$ is the cross section area normal to the motion, $\bm{i}_t$
is a unit tangent vector to the trajectory, $\bm{i}_n$ is
a normal vector (pointing upwards) to the trajectory,
$C_D$ and $C_L$ are lift coefficients, and $\varrho$ is the
air density. For a ball, $C_D$ is taken as 0.45, while
$C_L$ depends on the spinrate through $C_L=0.2\omega/500$.

Many numerical methods can be used to solve the problem.
Some legal names are 'ForwardEuler', 'RK2', 'RK4',
and 'Fehlberg' (adaptive Runge-Kutta 4/5 order).  If the
timestep 'dt' is None, approximately 500 steps are used, but
'dt' can also be given a desired 'float' value.

The boolean variable 'plot_simplified_motion' adds the curve
of the motion without drag and lift (the standard parabolic
trajectory). This curve helps illustrate the effect of drag
and lift. When 'new_plot' is 'False' (unchecked), the new
computed curves are added to the previous ones since last
time 'new_plot' was true.
```

```
# (DocOnce format)
"""
```

This application computes the motion of a ball with radius R and mass m under the influence of gravity, air drag and lift because of a given spinrate ω. The motion starts with a prescribed initial velocity v_0 making an angle initial_angle θ with the ground. A wind velocity w, positive in positive x direction, can also be given.

The ordinary differential equation problem governing the motion reads

$$
\begin{aligned}
m\frac{d^2\boldsymbol{r}}{dt^2} &= -mg\boldsymbol{j} - \frac{1}{2}C_D\varrho A v^2\boldsymbol{i}_t + \frac{1}{2}C_L\varrho A v^2\boldsymbol{i}_n \\
\boldsymbol{r}(0) &= 0\boldsymbol{i} + 0\boldsymbol{j} \\
\frac{d\boldsymbol{r}}{dt}(0) &= v_0\cos\theta\boldsymbol{i} + v_0\sin\theta\boldsymbol{j},
\end{aligned}
$$

where $\boldsymbol{i}$ and $\boldsymbol{j}$ are unit vectors in the x and y directions, respectively, g is the acceleration of gravity, A is the cross section area normal to the motion, $\boldsymbol{i}_t$ is a unit tangent vector to the trajectory, $\boldsymbol{i}_n$ is a normal vector (pointing upwards) to the trajectory, C_D and C_L are lift coefficients, and ϱ is the air density. For a ball, C_D is taken as 0.45, while C_L depends on the spinrate through $C_L = 0.2\omega/500$.

Many numerical methods can be used to solve the problem. Some legal names are `ForwardEuler`, `RK2`, `RK4`, and `Fehlberg` (adaptive Runge-Kutta 4/5 order). If the timestep `dt` is None, approximately 500 steps are used, but `dt` can also be given a desired `float` value.

The boolean variable `plot_simplified_motion` adds the curve of the motion without drag and lift (the standard parabolic trajectory). This curve helps illustrate the effect of drag and lift. When `new_plot` is `False` (unchecked), the new computed curves are added to the previous ones since last time `new_plot` was true.

Figure 5: Web interface with documentation.

The `generate.py` code for creating the web GUI goes as in the other examples,

```
from parampool.generator.flask import generate
from compute import compute_motion_and_forces

generate(compute_motion_and_forces, MathJax=True)
```

and we start the application as usual by `python controller.py`. The resulting web interface appears in Figure 4. The table shows the sequence of data we have given; starting with the default values, then turning off the `plot_simplified_motion` curve and `new_plot`, then running two cases with different values for the wind parameter `w`. The plot clearly show the influence of drag and wind against the motion.

MathJax.

The `compute_motion_and_forces` function returns mathematical symbols in the heading line of the table with data. MathJax must be enabled in the HTML code for these symbols to be rendered correctly. This is specified by the `MathJax=True` argument to `generate`. (However, in this particular example MathJax is automatically turned on since we use DocOnce syntax and mathematics in the doc string.)

Django interface. As before, the Django interface is generated by importing the function `generate` from `parampool.generator.django`. A subdirectory `motion_and_forces` contains the files, and the Django application is started as shown in previous examples and has the same functionality as the Flask application.

2.4 Other types of input data

The `generate` function will recognize the following different types of keyword arguments in the compute function: `float`, `int`, `bool`, `str`, `list`, `tuple`, `numpy.ndarray`, name of a file, as well as user-defined class types (a la `MyClass`).

Uploading a file. Here is a minimalistic example on computing the mean and standard deviation of data either in an array or in a file (we use the file if the operator of the web interface assigns a name in the "filename" entry):

```
def compute_average(data_array=np.array([1]), filename=None):
    if filename is not None:
        data = np.loadtxt(os.path.join('uploads', filename))
        what = 'file %s' % filename
    else:
        data = data_array
        what = 'data_array'
    return """
Data from %s:
<p>
<table border=1>
<tr><td> mean    </td><td> %.3g </td></tr>
<tr><td> st.dev. </td><td> %.3g </td></tr>
</table></p>
""" % (what, np.mean(data), np.std(data))
```

The output is simple, basically two numbers in a table and an intro line.

We write a `generate.py` file as shown before, but with `compute_average` as the name of the compute function. For any argument containing the string `filename` it is assumed that the argument represents the name of a file. The web interface will then feature a button for uploading the file.

When the application runs, we have two data fields: one for setting an array with list syntax and one for uploading a file. Clicking on the latter and uploading a file `mydata.dat` containing just columns of numbers, results in the web page displayed in Figure 6. In this case, when a filename was assigned, we use the data in the file. Alternatively, we can instead fill in the data array and click *Compute*, which then will compute the basic statistics of the assigned data array.

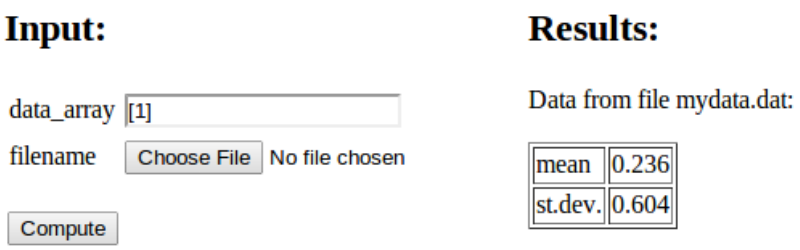


Figure 6: Web interface for uploading a file.

3 Working with a pool of input parameters

Parampool's main focus is on scientific applications with lots of input data of different type, organized in a hierarchical tree fashion. The various input parameters are defined in terms of a *pool*. The pool can be defined as a nested list or through a function application programming interface (known as an API, and here consisting of calls functionality in the `parampool.pool` package).

To exemplify the use of pools, we apply the `compute_motion_and_forces` function (from Section 2.3):

```
def compute_motion_and_forces0(
    initial_velocity=5.0,
    initial_angle=45.0,
    spinrate=50.0,
    w=0.0,
    m=0.1,
    R=0.11,
    method='RK4',
    dt=None,
    plot_simplified_motion=True,
    new_plot=True
    ):
```

Let us organize the 10 input parameters into four subpools. At the top level we need a subpool, usually called "Main pool" or named after the application. Each subpool is here specified with a logical name of each parameter and the corresponding variable name in the compute function:

- Main pool
  - Initial motion data
    - Initial velocity: `initial_velocity`
    - Initial angle: `initial_angle`
    - Spinrate: `spinrate`
  - Body and environment data

    - Wind velocity: `w`
    - Mass: `m`
    - Radius: `R`
  - Numerical parameters
    - Method: `method`
    - Time step: `dt`
  - Plot parameters
    - Plot simplified motion: `plot_simplified_motion`
    - New plot: `new_plot`

With a pool we can give the parameters more readable logical names (not restricted to a valid variable name in Python), but we can also specify a lot other properties too, as will be explained.

: Parameter names must be unique!

The generated Flask or Django code has a class (in `model.py` or `models.py` for Flask and Django, respectively) where each parameter name is transformed to a static class variable. Such code requires each parameter to have a unique name. (Using variable names that merge the parameter name with its subpool path would solve this problem.)

A pool is a hierarchical *tree structure* with *subpools* and *data items*, where each data item describes an input parameter in the problem. The task now is to make a Python specification of the of subpools and data items in the pool tree.

3.1 Specifying a pool as a list

The pool tree can be specified as a list of lists, strings, and dictionaries. Each list represents a subpool, each string the name of the subpool, and each dict is a data item. The pool must be return from some function, hereafter called the *pool function*. In our case, the pool function goes as follows:

```
def pool_definition_list():
    """Create and return pool defined through a nested list."""
    pool = [
        'Main', [
            'Initial motion data', [
                dict(name='Initial velocity', default=5.0),
                dict(name='Initial angle', default=45,
                     widget='range', minmax=[0,90], range_step=1),
                dict(name=r'Spinrate', default=50, widget='float',
                     unit='1/s'),
                ],
            'Body and environment data', [
                dict(name='Wind velocity', default=0.0,
```

```
                    help='Wind velocity in positive x direction.',
                    minmax=[-50, 50], number_step=0.5,
                    widget='float', str2type=float),
                dict(name='Mass', default=0.1, unit='kg',
                    validate=lambda data_item, value: value > 0,
                    help='Mass of body.'),
                dict(name='Radius', default=0.11, unit='m',
                    help='Radius of spherical body.'),
                ],
            'Numerical parameters', [
                dict(name='Method', default='RK4',
                    widget='select',
                    options=['RK4', 'RK2', 'ForwardEuler'],
                    help='Numerical solution method.'),
                dict(name='Time step', default=None,
                    widget='textline', unit='s'),
                ],
            'Plot parameters', [
                dict(name='Plot simplified motion', default=True,
                    help='Plot motion without drag+lift forces.'),
                dict(name='New plot', default=True,
                    help='Erase all old curves.'),
                ],
            ],
        ]
    from parampool.pool.UI import listtree2Pool
    pool = listtree2Pool(pool)
    return pool
```

Actually, the pool function must return a `parampool.pool.Pool` object, so after the definition of the pool tree as a list we must make the shown conversion from a list to a `Pool` object via the `listtree2Pool` function.

3.2 Attributes in data items

Each data item has a name and preferably a default value, as in the case of "Initial velocity". More attributes can be added:

- `widget` specifies the type of widget used in a graphical user interface. Legal values are `integer`, `float`, `range` (requires the `minmax` attribute too), `integer_range` (requires the `minmax` attribute too), `textline`, `textarea` (for larger multi-line texts), `checkbox` (for boolen variables), `select` (list of options), `email`, `password`, `file` (for a filename of a file to be uploaded, `url`, `hidden` (for an invisible field), and `tel` (for a phone number). If not given, `widget` is based on the value of the `str2type` attribute or the type of the default value.

- `minmax` is a 2-list or 2-tuple with lower and upper bound in the interval of legal values of a number.

- The `range_steps` attribute, valid when `widget` is `range`, specifies the steps in the slider used to select the number. In our example, we can select the "Initial angle" in unit steps between 0 and 90 degrees.

- `unit` specifies a unit, e.g., `1/s` or `kg/m**3`. If the input contains another unit, e.g., `4 1/h`, the value will be automatically converted to the registered unit (4/3600 1/s if 1/s is the registered unit). (Parampool applies the the `PhysicalQuantity` object from the ScientificPython package to perform computations with units. A copy of this object is bundled with Parampool.)
- `help` adds a help string to explain more about the parameter and how it can be set.
- `number_step` specifies the precision of `float` or `integer` widgets if `minmax` is also specified (default 0.001), otherwise the precision is arbitrary.
- `str2type` is a conversion function from a string (text given in a user interface) to the right type for the parameter. A value of `str2typ2` is automatically assigned if `widget` is given, otherwise Parampool applies the default value to find the right `str2type` function. This means that it is strictly not necessary to assign `str2type` for the "Wind velocity" data item since the default value `0.0` implies `str2type=float`. With more complicated objects one can assign a user-given conversion function to `str2type` (shown later).
- `option` is a list of options for a `select` widget. The "Method" data item (for the name of the numerical solution method) provides an example of this widget type.
- `validate` holds a function that takes the value of the data item as argument and returns `True` or `False` depending on whether the value can be accepted or not.
- `symbol` contains a mathematical LaTeX symbol that will be used in Flask or Django interfaces instead of the name of data item.
- `widget_size` specifies the size (width) of fields in graphical user interfaces.
- The `textline` widget must be used for default values that are `None`, because another value or the text `None` can be written in the field. In this case, `str2type` is automatically set to `eval` and any valid Python expression is then essentially allowable, but wrong object types may give errors in the compute function.

Check that default values are real numbers.

If a default value is set to `5`, Parampool will interpret this as an integer and let `string2type` be `int` and force all input to be converted to integers. Normally, you want input to be real, so check that the default value is `5.0` unless the pool item is really meant to be an integer.

3.3 The compute function

When working with pools, the compute function is allowed to take *only one argument* called `pool`. This object is used to extract input data. Basically, the value of any data item `my parameter` in the pool is extracted by

```
variable = pool.get_value('my parameter')
```

In case multiple data items have the same name, a sufficient part of the subpool path must be given, e.g.,

```
variable = pool.get_value('My Subpool1/my parameter')
```

Our specific compute function is a wrapper for `compute_motion_and_forces`:

```
def compute_motion_and_forces_with_pool(pool):
    initial_velocity = pool.get_value('Initial velocity')
    initial_angle = pool.get_value('Initial angle')
    spinrate = pool.get_value('Spinrate')
    w = pool.get_value('Wind velocity')
    m = pool.get_value('Mass')
    R = pool.get_value('Radius')
    method = pool.get_value('Method')
    dt = pool.get_value('Time step')
    plot_simplified_motion = pool.get_value('Plot simplified motion')
    new_plot = pool.get_value('New plot')
    return compute_motion_and_forces(
        initial_velocity, initial_angle, spinrate, w,
        m, R, method, dt, plot_simplified_motion,
        new_plot)
```

The assumption is that the pool object provides enough input data for the compute function. If this assumption does not hold and extra information is needed, one can simply make a class, store extra data as attributes in the class, and let the compute function be a method in the class.

3.4 A command-line and file interface

Having defined a pool, it is trivial to get a command-line interface in the application. Just write

```
from parampool.pool.UI import set_values_from_command_line
pool = set_values_from_command_line(pool)
```

Now `pool` has values loaded from the command line. The name of the command-line options follow the names in the pool, but with underscore replacing whitespace: `--Initial_motion_data/Initial_angle`. However, in this case just writing `--Initial_angle` also works since it is a unique name in the pool tree, and then we do not need the complete path with the subpool name.

One can also read data from a file with syntax

```
subpool Initial motion data
  Initial angle = 45.5        # small perturbation
  Spinrate = 20
end
subpool Body and environment data
  Wind velocity = -10  ! m/s  # units appear after ! (before #)
end
```

Data from the file is loaded into the pool by

```
from parampool.pool.UI import set_defaults_from_file
pool = set_defaults_from_file(pool)
```

To activate reading from file `mydat.dat`, one must supply the command-line arguments `-poolfile mydat.dat`.

Tip: autogenerate the file with default data.

The function `write_poolfile(pool)` in `parampool.pool.UI` writes the current pool data to a file with the right syntax. This is a simple way of getting the complete pool in the file.

Often, an application will first load default values from file, then from the command line, and finally launch the graphical web interface for enabling interactive setting of values in the pool system. Automatic generation of such interactive web interfaces constitutes the next topic. The lines above for loading parameters from file and command line are automatically generated when a web interface is requested (see also comments in Section 3.8).

3.5 Generating a web-based user interface

With a pool function and a compute function at hand, it remains to make a new directory, copy the module(s) containing the pool function and compute function to this directory, and write a `generate.py` file with the content

```
from parampool.generator.flask import generate
from compute import compute_motion_and_forces_with_pool, \
     pool_definition_list

generate(compute_motion_and_forces_with_pool,
         pool_function=pool_definition_list,
         MathJax=True)
```

The `generate` function will now use the information in the pool (and not the arguments in the compute function!) to generate a flexible user interface. Note that an Internet connection is required. After running

```
Terminal> python generate.py
```

several Flask files and directories are generated (`model.py`, `controller.py`, `templates`, `static`, and a simple clean-up script `clean.sh`). The user interface is started by

```
Terminal> python controller.py
```

Open the URL `http://127.0.0.1:5000` in a web broswer to see an interface as the one in Figure 7.

Figure 7: Web interface in closed form.

3.6 Operating the user interface

The pool tree is mapped onto a visual structure often used for directory trees. The look and feel resemble that of the Windows Explorer application in the Windows operating system.

Clicking on *open all* at the top of the user interface expands all subpools so that all parameters (data items) become visible. Figure 8 displays the result in the Opera browser. Note that in this type of user interface, the name of each data item is automatically typeset in LaTeX and inserted as a picture (the `http://latex.codecogs.com` utility is used).

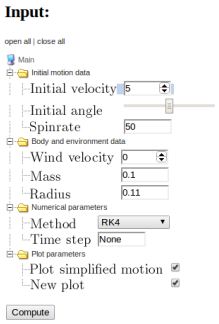


Figure 8: Web interface in fully expanded form.

The following technical points must be mentioned.

1. A plain float or integer value gives a `textline` widget, while if a `minmax` range is specified, a `float` or `integer` widget (so-called HTML5 number field) is chosen.

2. Data items whose widgets are specified as `float` or `integer`, or where this is implied because `str2type` is `float` or `int`, or the default value is

a `float` or `int` *and* the `minmax` attribute is assign, are shown using the HTML5 input field called number. This is recogned by the small (and not so useful) arrows that can be used to adjust the number, but usually typing in the number manually is faster and more precise. An extra attribute, `number_step` controls the stepping when clicking on the arrows and also the allowed precision of a typed number (same as `number_step`, which by default is 0.001).

3. When the widget is `range`, an HTML5 range field is used, which is usually rendered as a slider in browsers. The slider gets by default 100 steps (can be changed or specified individually for any data item).

4. With the `select` widget we get a pull-down menu with the different options.

5. Any data item whose default value is `True` or `False` maps directly to a checkbox for boolean parameters.

6. Any data item with `unit` specified maps to an ordinary text field, since input consists of a number with an optional text for the unit. That is, if we choose to set `unit='m/s'` for the "Initial velocity" data item, the input field will not the an HTML5 number field, but a standard HTML text field.

7. The names of the data items are typeset in LaTeX and shown as PNG images. This means that data item names may contain mathematical expressions: `Spinrate $\omega$` for instance.

Warning.

The HTML5 number field is rendered differently in different browsers. This can lead to strange layout of the input fields. In such cases it is recommended to avoid the HTML5 number field. This is easiest accomplished by explicitly specifying `widget` to be `textline`. This is also the default widget type if you equip the number with a unit or do not specify any widget, just a float or integer default value.

We can try out the interface:

1. Set "Initial velocity" to 8.

2. Move the slider for "Initial angle" to 55.

3. Add a positive "Wind velocity" of -3.

4. Specify "Mass"as the text `0.1*1000 g` (i.e., we use g rather than the default kg as unit, but the value is still 0.1 kg).

5. Choose `RK2` for "Method".

6. Set “Time step” to 0.12.
7. Uncheck the “Plot simplified motion” boolean value.
8. Hold the mouse pointer over the “Wind velocity” field to see the help string. Then point the mouse to “Mass” input field and the specified unit pops up. A combination of help and unit information is showed if both are given in the data item definition.

You should see something like Figure 9.

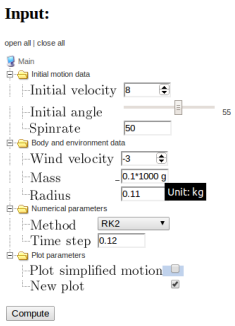


Figure 9: Web interface with input parameters filled out.

Now, press the *Compute* button. Figure 10 shows the resulting response. You can now play around and click the checkbox for *Plot simplified motion* and the recompute to see the effects of wind against the motion, drag, and lift (which are substantial in this example).

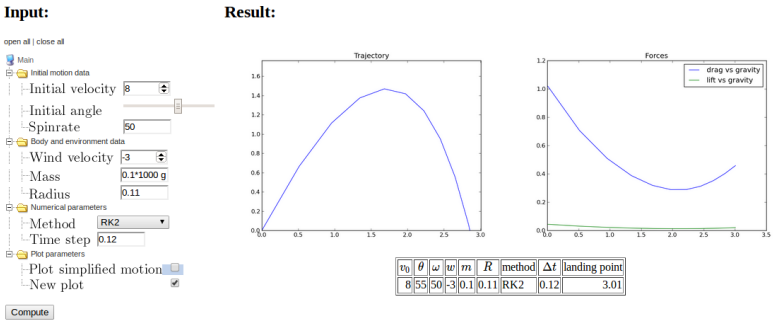


Figure 10: Web interface with input and results.

3.7 Detection of wrong input

Text in a number field. Write `abc` in the “Initial velocity” field and press the *Compute* button. The error message “Please enter a number” pops up.

Failure of user-provided validate function. Give a negative value for "Mass". The "Mass"data item has a validation function provided by us. A `False` value returned from this function gives rise to a `DataItemValueError` shown in the browser. It reads here

```
Mass = -0.1: validate function <lambda> claims invalid value.
```

Failure of converting string to right type. Write `abc` for "Radius". This is a text field so any text is in principle valid, but Parampool raises a `TypeError` with the message

```
could not apply str2type=<type 'float'> to value abc <type 'str'>
```

Failure in the compute function. Give a list `[0.1, 0.2]` for "Time step". Since the default is `None`, which causes `str2type=eval`, any Python expression is accepted in the interface, but the compute function used in our example in this tutorial will raise a `TypeError` because `float(dt)` fails when `dt` is a list. One could think of providing a tailored `str2type` function in this case:

```
def convert_time_step(value):
    # Value must be None or a float
    if value == 'None':
        return None
    else:
        try:
            return float(value)
        except TypeError:
            raise TypeError(
            'Time step: could not convert "%s" to float, '
            'must be None or float' % value)
```

Setting `str2type=convert_time_step` for the "Time step" data item gives an informative error message if the answer is not as expected: `None` or a floating-point number.

3.8 Loading parameters from file and the command line

Parameters can be assigned default values in a file and then other values on the command line, see Section 3.4, before the web GUI is offered to the user. When autogenerating the web interface, the magic lines from Section 3.4 are automatically inserted in the `controller.py` file (for Flask or `views.py` for Django). This means that when starting `python controller.py` we may add `-poolfile name` and any set of command-line options for setting individual parameters. This makes it easy to control which default values that will appear in the web GUI.

3.9 Specifying a pool using an API

Instead of listing all the entries in the pool tree as strings, lists, and dicts in a nested data structure, you can use the Application Programming Interface

(API) of the `parampool.pool` package. The pool defined above is alternatively programmed like this using the API:

```
def pool_definition_api():
    """Create and return pool using the parampool.pool API."""
    from parampool.pool.Pool import Pool
    pool = Pool()
    # Go to a subpool, but create it if it does not exist
    pool.subpool('Main pool')
    pool.subpool('Initial motion data')
    # Define data items for the current subpool
    pool.add_data_item(
        name='Initial velocity', default=5.0)
    pool.add_data_item(
        name='Initial angle', default=45,
        widget='range', minmax=[0,90])
    pool.add_data_item(
        name='Spinrate', default=50, widget='float', unit='1/s')

    # Move to (and create) another subpool, as in a file tree
    pool.subpool('../Body and environment data')
    # Add data items for the current subpool
    pool.add_data_item(
        name='Wind velocity', default=0.0,
        help='Wind velocity in positive x direction.',
        minmax=[-50, 50], number_step=0.5,
        widget='float', str2type=float)
    pool.add_data_item(
        name='Mass', default=0.1, unit='kg',
        validate=lambda data_item, value: value > 0,
        help='Mass of body.')
    pool.add_data_item(
        name='Radius', default=0.11, unit='m',
        help='Radius of spherical body.')

    pool.subpool('../Numerical parameters')
    pool.add_data_item(
        name='Method', default='RK4',
        widget='select',
        options=['RK4', 'RK2', 'ForwardEuler'],
        help='Numerical solution method.')
    pool.add_data_item(
        name='Time step', default=None,
        widget='textline', unit='s', str2type=convert_time_step)

    pool.subpool('../Plot parameters')
    pool.add_data_item(
        name='Plot simplified motion', default=True,
        help='Plot motion without drag+lift forces.')
    pool.add_data_item(
        name='New plot', default=True,
        help='Erase all old curves.')
    pool.update()
    return pool
```

The API is in many ways easier to use than the nested data structure with lists, strings, and dicts. The API resembles moving around in a file tree. The rules are simple:

- `pool.subpool(path)` moves us to a subpool `path`, and creates it first if it does not exist. This is similar to `cd path` in a file tree, or `mkdir path; cd path`, if `path` does not exist.
- The name of a subpool, `path`, follows the rule of file and directory names in a file tree: a slash is used as delimiter between subpools and data items. For example:
  - `/Main pool/Initial motion data/Initial velocity` is the full path to the "Initial velocity" data item.
  - Standing in the "Initial motion data" subpool, `..` is the parent subpool ("Main pool"), while `../Numerical parameters` is the correct path to the "Numerical parameters" subpool. That is, we can use relative and absolute paths as in a file tree.
- A data item is appended to the current subpool by calling `pool.add_data_item`.

The look and functionality of this GUI (found in the `flask_pool2` directory) are the same as in the previous one (found in the `flask_pool1` directory).

3.10 Specifying a pool using an alternative API

There is an another way of defining subpools as well: make a function for defining each subpool.

```
def pool_definition_api_with_separate_subpools():
    """
    Create and return a pool by calling up other functions
    for defining the subpools. Also demonstrate customization
    of pool properties and inserting default values from file
    or the command line.
    """
    from parampool.pool.Pool import Pool
    pool = Pool()
    pool.subpool('Main pool')
    pool = motion_pool(pool)
    pool.change_subpool('..')
    pool = body_and_envir_pool(pool)
    pool.change_subpool('..')
    pool = numerics_pool(pool)
    pool.change_subpool('..')
    pool = plot_pool(pool)
    pool.update()  # finalize pool construction

    from parampool.pool.UI import set_data_item_attribute
    # Change default values in the web GUI
    import parampool.pool.DataItem
    parampool.pool.DataItem.DataItem.defaults['minmax'] = [0, 100]
    parampool.pool.DataItem.DataItem.defaults['range_steps'] = 500
    # Can also change 'number_step' for the step in float fields
    # and 'widget_size' for the width of widgets

    # Let all widget sizes be 6, except for Time step
    pool = set_data_item_attribute(pool, 'widget_size', 6)
```

```
    pool.get('Time step').data['widget_size'] = 4

    # Example on editing hardcoded defaults in the model files
    # (not necessary, but a possible technique along with
    # setting defaults in the pool, in a file, or on the command line)
    from parampool.pool.UI import set_defaults_in_model_file
    flask_modelfile = 'model.py'
    django_modelfile = os.path.join(
        'motion_and_forces_with_pool', 'app', 'models.py')
    if os.path.isfile(flask_modelfile):
        set_defaults_in_model_file(flask_modelfile, pool)
    elif os.path.isfile(django_modelfile):
        set_defaults_in_model_file(django_modelfile, pool)

    return pool

def motion_pool(pool, name='Initial motion data'):
    pool.subpool(name)
    pool.add_data_item(
        name='Initial velocity', default=5.0, symbol='v_0',
        unit='m/s', help='Initial velocity',
        str2type=float, widget='float',
        validate=lambda data_item, value: value > 0)
    pool.add_data_item(
        name='Initial angle', default=45, symbol=r'\theta',
        widget='range', minmax=[0,90], str2type=float,
        help='Initial angle',
        validate=lambda data_item, value: 0 < value <= 90)
    pool.add_data_item(
        name='Spinrate', default=50, symbol=r'\omega',
        widget='float', str2type=float, unit='1/s',
        help='Spinrate')
    return pool

def body_and_envir_pool(pool, name='Body and environment data'):
    pool.subpool(name)
    pool.add_data_item(
        name='Wind velocity', default=0.0, symbol='w',
        help='Wind velocity in positive x direction.', unit='m/s',
        minmax=[-50, 50], number_step=0.5,
        widget='float', str2type=float)
    pool.add_data_item(
        name='Mass', default=0.1, symbol='m',
        help='Mass of body.', unit='kg',
        widget='float', str2type=float,
        validate=lambda data_item, value: value > 0)
    pool.add_data_item(
        name='Radius', default=0.11, symbol='R',
        help='Radius of spherical body.', unit='m',
        widget='float', str2type=float,
        validate=lambda data_item, value: value > 0)
    return pool

def numerics_pool(pool, name='Numerical parameters'):
    pool.subpool(name)
    pool.add_data_item(
        name='Method', default='RK4',
        widget='select',
        options=['RK4', 'RK2', 'ForwardEuler'],
        help='Numerical solution method.')
    pool.add_data_item(
```

```
        name='Time step', default=None, symbol=r'\Delta t',
        widget='textline', unit='s', str2type=eval,
        help='None: ca 500 steps, otherwise specify float.')
    return pool

def plot_pool(pool, name='Plot parameters'):
    pool.subpool(name)
    pool.add_data_item(
        name='Plot simplified motion', default=True,
        help='Plot motion without drag and lift forces.')
    pool.add_data_item(
        name='New plot', default=True,
        help='Erase all old curves.')
    return pool
```

This application, found in the directory `flask_pool3` (with a corresponding Django counterpart), will now be used to illustrate three important additional features of Parampool:

1. documentation of the application is in an external file `doc.html`
2. the name of a parameter can be a mathematical symbol
3. parameters can have multiple values for investigating many parameter sets at once

File with documentation of the application. We have in Section 2.3 seen that the doc string of the compute function may contain a mathematical description of the problem with rich typesetting (using DocOnce syntax). It is also possible to make such a description in a separate file. Any HTML file will work, and the filename is specified by the `doc` argument to `generate`.

LaTeX symbol as parameter name. One can add a mathematical LaTeX symbol for the parameter names (the `symbol` keyword argument). This symbol can either be displayed as the parameter's complete name, or the symbol can be added to the standard name of the parameter. The choice is set by the `latex_name` keyword argument in the `generate` call in `generate.py`:

```
from parampool.generator.flask import generate
from compute import compute_motion_and_forces_with_pool_loop, \
     pool_definition_api_with_separate_subpools

generate(compute_motion_and_forces_with_pool_loop,
         pool_function=pool_definition_api_with_separate_subpools,
         MathJax=True, doc=open('doc.html', 'r').read(),
         latex_name='symbol')
```

The values of `latex_name` can be `'symbol'`, meaning symbol only, or `'text, symbol'`, meaning that the ordinary name is followed by a comma and the symbol.

Multiple input values for parameters. We can specify multiple values for parameters whose input fields are pure text fields. For example, for the wind velocity (w) parameter we can assign two values separated by the `&` character: `0 & -8`. Calling `pool.get_values('Wind velocity')` will then return a list `[0, -8]` rather than one number. We can hence easily make a loop over the multiple values for each parameter where we use pure text as input. Our compute function looks in this case like this:

```
def compute_motion_and_forces_with_pool_loop(pool):
    html = ''
    initial_angle = pool.get_value('Initial angle')
    method = pool.get_value('Method')
    new_plot = pool.get_value('New plot')  # should be True here
    plot_simplified_motion = pool.get_value('Plot simplified motion')
    for initial_velocity in pool.get_values('Initial velocity'):
        for spinrate in pool.get_values('Spinrate'):
            for m in pool.get_values('Mass'):
                for R in pool.get_values('Radius'):
                    for dt in pool.get_values('Time step'):
                        for w in pool.get_values('Wind velocity'):
                            html += compute_motion_and_forces(
                                initial_velocity, initial_angle,
                                spinrate, w, m, R, method, dt,
                                plot_simplified_motion, new_plot)
    return html
```

Note that we accumulate the HTML code returned from the compute function `compute_motion_and_forces` that runs the simulation and returns the results as HTML code. Figure 11 features a mathematical description of the application, LaTeX symbols as parameter names, and two input values for two parameters, leading to $2 \times 2 = 4$ runs, and hence four lines of plots. The example demonstrates how easy it is to quickly perform parameter studies by simply 1) writing the compute function with loops and `pool.get_values`, 2) separating input values by `&` in the GUI.

However, writing nested loops for a lot of parameters is unnecessary tedious. We can use the `itertools.product` function to compute all possible combinations and simplify the loop. This function takes a set of lists or tuples and returns an iterator over all combinations of all the elements in the lists/tuples. As an example,

```
>>> import itertools
>>> values1 = [1, -1]
>>> values2 = [2, 4, 6]
>>> combinations = itertools.product(values1, values2)
>>> for combination in combinations:
...     print combination
(1, 2)
(1, 4)
(1, 6)
(-1, 2)
(-1, 4)
(-1, 6)
```

The single loop is equivalent to

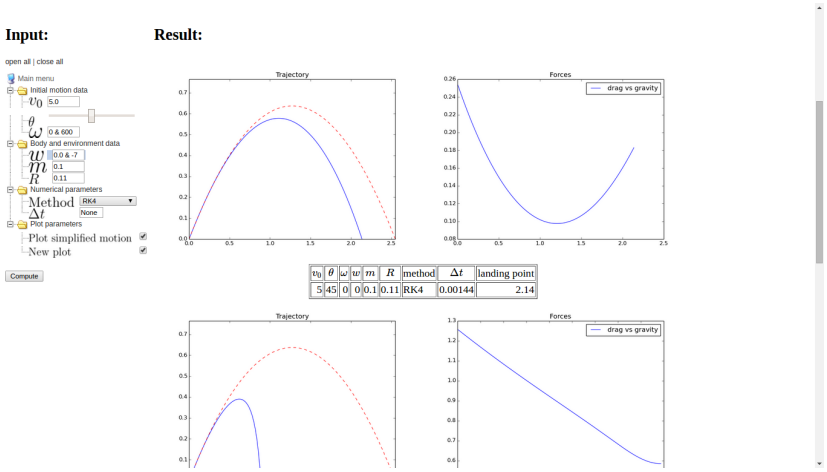


Figure 11: Web interface with documentation, LaTeX symbols, and multiple input values.

```
for value1 in values1:
    for value2 in values2:
        combination = (value1, value2)
        print combination
```

With an unknown number of lists as arguments to `itertools.prodct` one can use the construction

```
values = [values1, values2]
combinations = itertools.product(*values)
```

With these ideas we can generalize our implementation of nested loops over multiple values of the parameters:

```
def compute_motion_and_forces_with_pool_loop_product(pool):
    import itertools
    initial_angle = pool.get_value('Initial angle')
    method = pool.get_value('Method')
    new_plot = pool.get_value('New plot')  # should be True here
    plot_simplified_motion = pool.get_value('Plot simplified motion')
    # Make list of all names with potentially multiple values
    names = """
    Initial velocity
    Spinrate
    Mass
    Radius
    Time step
    Wind velocity
    """
    names = [line.strip() for line in names.splitlines()]
    # Get all values of all parameters
    values = [pool.get_values(name) for name in names]
    combinations = itertools.product(*values)
    html = ''
```

```
    for combination in combinations:
        initial_velocity, spinrate, m, R, dt, w = combination
        html += compute_motion_and_forces(
            initial_velocity, initial_angle,
            spinrate, w, m, R, method, dt,
            plot_simplified_motion, new_plot)
    return html
```

This is the type of implementation recommended in applications.

3.11 Login and archival of results

Parampool has a feature that allows a user to create an account, log in, and save the results from simulations. The results of a run consists of the HTML code and plots, which are stored in a database. Later, old results can be retrieved in the GUI. Enabling login is just a matter of writing `enable_login=True` in the call to the `generate` function:

```
from parampool.generator.flask import generate
from compute import compute_motion_and_forces

generate(compute_motion_and_forces, MathJax=True,
         enable_login=True)
```

The generated `controller.py` file is now much more complicated in the Flask case, but the details are not important for the plain user of Parampool. The usage should be explanatory: first click on `Register` to register a new user, later one can just click on `Login`. After having run a simlation, a *Comments* field arises where one can add comments about this run, and then click on *Add* if the results are sufficiently interesting to be stored in the database. The upper right corner has a link *Previous simulations* which gives access to previous results in the database. At the bottom of the page with previous runs, there is a *Delete all* button that clears the database.

Problems with sending email. In case of error messages regarding sending email through Google's server, it may help to add the following line to `controller.py`, e.g., right before the `send_email` function:

```
app.config['MAIL_SERVER'] = 'smtp.gmail.com'
```

3.12 Python expressions as input

One can use Python expressions like `sin(0.5)*exp(-1)` in the input fields. Parampool will recognize arithmetic expressions or use of mathematical functions, and in such cases run `eval` on the input to interpret it. If special functions are needed, e.g., the input is like `myfunc(0.1)`, one can supply a namespace to the data item in question through the `namespace` parameter. Here we create a custom namespace for a parameter, where the namespace includes `myfunc` from `mymodule` and all functions (or more precisely, all names) in `yourmodule` that do not start with an underscore:

```
import mymodule, yourmodule
namespace = {'myfunc': mymodule.myfunc}
namespace.update({key: value for key, value in
                  yourmodule.__dict__.iteritems()
                  if not key.startswith('_')}
```

4 Exercises

Exercise 1: Make a web app for integration

The purpose of this exercise is to use Parampool to generate a simple web application for integrating functions: $\int_a^b f(x)dx$. Provide a symbolic expression for $f(x)$ and the limits a and b as input. The application first attempts to compute the integral $\int f(x)dx$ symbolically using `sympy`, without limits. If that does not succeed, it computes $\int_a^b f(x)dx$ numerically by the, e.g., Trapezoidal rule. The compute function should return nicely typeset formula for the integrand (use `sympy.latex`) and also the result in case of symbolic integration.

Hint. The relevant `sympy` code needs to turn the string expression for $f(x)$ into a valid Python function. This can be done with the `sympify` and `lamdify` utilities. Given some string `s` containing an expression, e.g., s = 'x*sin(x)', the following code makes a `sympy` expression and a valid Python function `f(x): return x*sin(x)`:

```
import sympy as sym
expr = sym.sympify(s)
x = sym.Symbol('x')
f = sym.lambdify([x], expr)
```

Using these constructions, the following function tries to integrate symbolically and then numerically if necessary:

```
def integrate(string_expression, a, b):
    assert isinstance(string_expression, str)
    import sympy as sym
    expr = sym.sympify(string_expression)
    x = sym.Symbol('x')
    I = sym.integrate(expr, x)
    if isinstance(I, sym.Integral):
        # Did not succeed to integrate symbolically
        f = sym.lambdify([x], expr)  # Python function
        I = trapezoidal(f, a, b, n=100)
    else:
        I = sym.latex(I)  # make LaTeX expression
    return expr, I  # str if symbolic, float if numerical integr.
```

Filename: `integrate`.

Exercise 2: Make a web app for plotting data

Make a web app with the `parampool.pool.Pool` functionality for uploading a file with columns of data and plotting columns 2, 3, and so forth against column 1.

Hint. Use the `file` widget to provide the name of the data file in the input. Use `numpy.loadtxt` to easily load the data from file.
Filename: `upload_and_plot`.

Exercise 3: Make a web app for studying vibrations

Download the bumpy for simulating vibrating mechanical systems governed by the differential equation

$$mu'' + f(u') + s(u) = F(t). \tag{4}$$

a) Make an interface to the bumpy application where the user can set the following items.

- Subpool "Main":
  - m
  - damping type: linear $f(u') = bu'$ or quadratic $f(u') = b|u'|u'$ (option list)
  - damping parameter b
  - subpool "spring"
  - subpool "excitation"
- Subpool "spring":
  - an option list for selected formulas for the spring $s(u)$:
    - $s(u) = ku$ (linear spring)
    - $s(u) = ku(+\frac{1}{6}x^3)$ (first two Taylor-series terms for $k \sin u$)
    - $s(u) = k \sin u$ (pendulum "spring" caused by gravity)
  - k
- Subpool "excitation":
  - an option list for selected formulas for the excitation $F(t)$:
    - $F(t) = A * sin(w * t)$
    - $F(t) = A * cos(w * t)$
    - $F = A$ for $t \leq w$, $F = 0$ for $t > w$
    - $F(t)$ read from file
  - A
  - w
  - filename

b) Can you simplify the interface and accept general mathematical expressions for $F(t)$ and $s(u)$?

Problem 4: Make a coin flipper

Make a web application where we can set the number of coins, m, to be flipped. Guess the maximum number of heads, n. If the number of heads is less than or equal to n, you earn $m - n$ points. Clicking on the *Submit* button (whose name should rather be *Flip*) shows images of coins with heads and tails according to a random flip of each coin as well as the total number of earned points. Make a button to reset the game.

Hint. See `random.org/coins` for inspiration and images of coins. Use a global variable in the compute module to hold the number of earned points.
Filename: `coin_flipper`.

Exercise 5: Make a web app for the Lorenz system

The famous Lorez system for illustrating chaotic motion is given by

$$x'(t) = s(y(t) - x(t)), \tag{5}$$
$$y'(t) = x(t)(r - z(t)) - y(t), \tag{6}$$
$$z'(t) = x(t)y(t) - bz(t), \tag{7}$$

with initial conditions $x(0) = x_0$, $y(0) = y_0$, and $z(0) = z_0$. Make a web app that can solve this system and visualize its solution. The web interface must allow for setting x_0, y_0, z_0, r, s, b, the number of time steps (or the time step), and the final time T for the simulation. Filename: `lorenz`.

5 Deployment

The most obvious servers to deploy web applications on, like Google App Engine, only support very light weight Python. For heavier scientific applications we may need more tools; SSH access, a Fortran compiler, etc. Therefore we introduce two servers we recommend for the scientific computing usage.

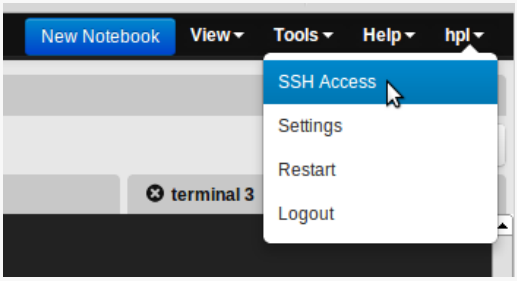
5.1 Wakari

Wakari is originally meant to be a Python data analysis environment for internet-accessible services and sharing of computing environments. It does not allow users to deploy webservers that can be accessed by others. However, accessing a Flask server process running in Wakari is possible using SSH tunneling:

```
wakari-terminal> python controller.py
laptop-terminal> ssh -p [port] -f -N -L 5000:localhost:5000 \
                 [username]@[wakari-hostname].wakari.io
```

Port - username - hostname.

Information about which port to forward, as well as username and wakari-hostname, is available under `SSH Access` at the user submeu in Wakari.



It is also necessary to add your public SSH key to Wakari Settings.

Now the application is available as usual at `http://127.0.0.1:5000/` on your laptop.

Even though only Flask and not Django is pre-installed in Wakari, it is relatively straight-forward to download the Django source and install it locally on your user. (Also, if Gnuplot is to be installed and compiled with PNG support, the library `pnglib` needs to be installed before Gnuplot is compiled. The Parampool repo features a script that demonstrates how to install various scientific computing packages on Wakari.)

5.2 Local server

If only little traffic is expected for a web application, it is possible to run Flask and Django through a CGI script. The script imports and starts the application's wsgi handler and works as a gateway between the Internet and the Flask or Django server.

The simplest example of a Python CGI script running Flask goes like

```
#!/usr/bin/python
from wsgiref.handlers import CGIHandler
from controller import app

CGIHandler().run(app)
```

This code assumes that the Python executable is located in `/usr/bin/` and up-to-date version wise, and that all required Python modules are in directories listed in the `PYTHONPATH` environment variable. These modules must be accessible for any user. Sometimes `/usr/bin/python` is too old so you need to compiler a newer version and use its path in the header of the CGI script. In case you run such a "private" Python installation, all necessary modules must also be installed. Also make sure that `controller.py` is in a directory listed in `PYTHONPATH`, or add the directory to `sys.path`:

```
import sys
# controller.py is in /usr/local/my/software
```

```
sys.path.insert(0, '/usr/local/my/software')
from controller import app
```

The only difference between the CGI script for Flask and Django is that for Django one needs to add the directory containing the `settings.py` file to `sys.path` and set `os.environ['DJANGO_SETTINGS_MODULE']`. Also, the import of the app is a bit different than before:

```
sys.path += ['path/to/myproject'] # The folder containing settings.py
os.environ['DJANGO_SETTINGS_MODULE'] = 'myproject.settings'

app = django.core.handlers.wsgi.WSGIHandler()
CGIHandler().run(app)
```

Remember that all scripts and modules to be accessed from the web need to have permissions for everyone to read and execute. This can be done by, e.g., `chmod 755 filename`.

References

[1] H. P. Langtangen and A. E. Johansen. Using web frameworks for scientific web applications. Web document, Simula Research Laboratory and University of Oslo, 2013.

A Installation of Parampool

Parampool needs

- Python version 2.7
- Numerical Python: `sudo pip install numpy`
- Parampool itself from https://github.com/hplgit/parampool
- Flask: `sudo pip install Flask`
- progressbar: `sudo pip install progressbar`
- Flask-WTF: `pip install Flask-WTF`
- Flask-SQLAlchemy: `sudo pip install Flask-SQLAlchemy`
- Flask-Login: `sudo pip install Flask-Login`
- Flask-Mail: `sudo pip install Flask-Mail`
- Django (optional): `sudo pip install django`
- Odespy (optional, but used in most of the tutorial examples): `pip install -e git+https://github.com/hplgit/odespy.git#egg=odespy`
- DocOnce (optional, for documentation, used in some tutorial examples) from https://github.com/hplgit/doconce

Index