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

May 9, 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.

NOTE: This report is in a **very preliminary state**, but should be sufficient for exploring Parampool for creating web GUIs. Please send email to the first author for questions, reporting typos and errors, request of more information, etc.

Contents

2DO.

- More links to additional material: (Read these when returning to this work)
  - Quick Flask++ tutorial
  - Quick overview of Bootstrap 3, another Bootstrap 3 tutorial
  - Flask and Boostrap intro (sample app has sign in functionality and bootstrap layout explained here)
  - Flask, Heroku and Bootstrap (this is a good recipe we can turn into building a science app)
  - Facebook login in Flask

  – Flask-Bootstrap and its doc, read in particular basic usage and see the (only?) example
  – Bootstrap video intro

- A complete modern Python project with virtualenv, testing, setup.py, etc.
- Skulpt: Interactive Python in the browser
- Python for computer games

1 Simulation program

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

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

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, $v = |\boldsymbol{v} - \boldsymbol{w}|$ is the relative velocity between the body, $\boldsymbol{v} = d\boldsymbol{r}/dt$, and a given wind velocity $\boldsymbol{w}$, 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')
```

Figure 1: A simple web interface.

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

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. All figures in this tutorial are made with the Chrome browser.

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 [**?**]. 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 [?] 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 `python controller.py` to start 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.

Django application. The corresponding Django application is generated by the same `generator.py` code as above, except that tword `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,

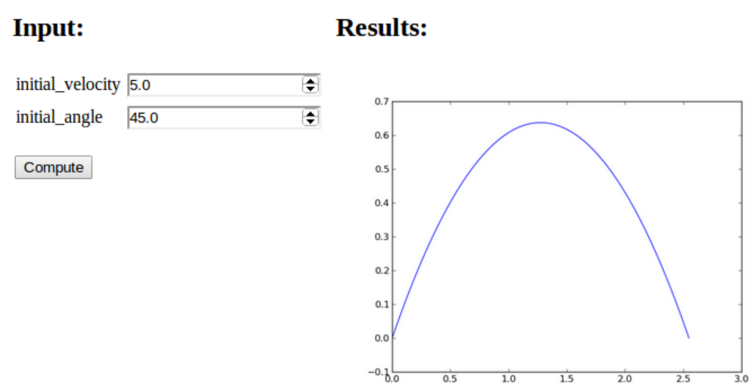


Figure 2: A web interface with graphics.

```
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.

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

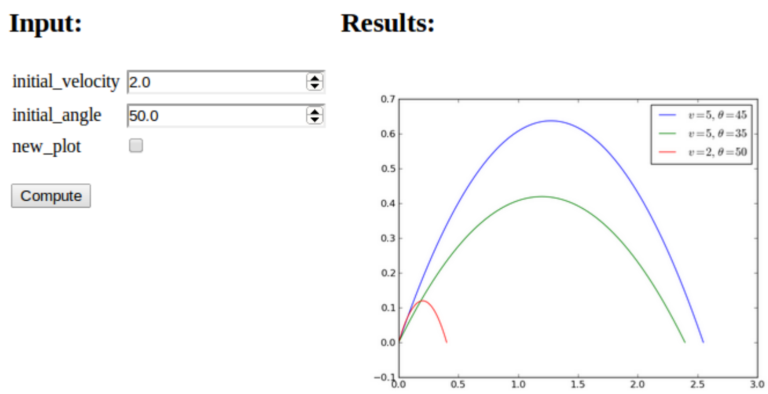


Figure 3: Plot with multiple curves.

```
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 plot
# 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="data:image/png;base64,%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.

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 plot
```

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 as a table. The technique described in the *Avoiding plot files* box at the end of Section pp:plot:output 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="data:image/png;base64,iVBORw0KGgoAAAA..." width="400">
</td>
<td valign="top">
<img src="data:image/png;base64,iVBORw0KGgoAAAA..." 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.

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 $R$
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
```

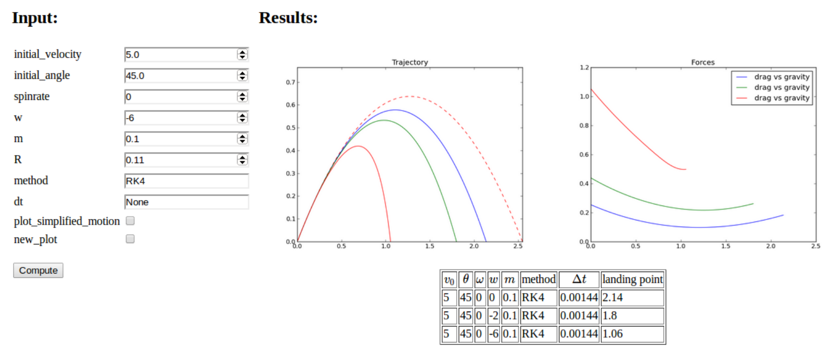
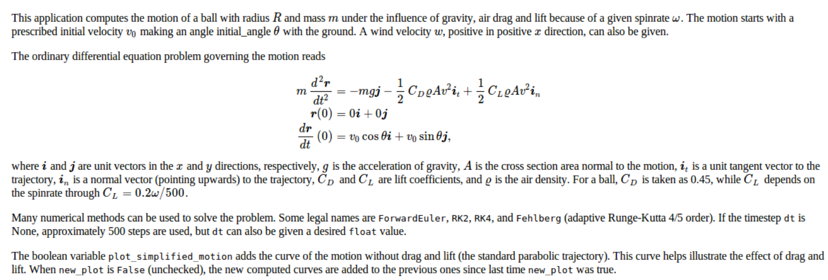


Figure 4: Web interface with two graphs.

```
$\theta$ 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

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

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.

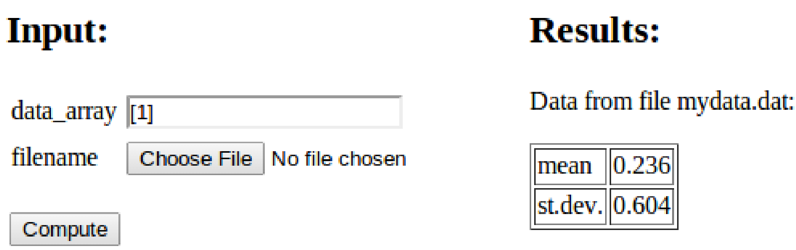


Figure 6: Web interface for uploading a file.

2.4 Other types of function arguments

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 `MySpecialClass`). 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 file to `filename`):

```
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 an 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.

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 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 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 names 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.

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 Specify 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 `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), `integer_range` (requires the `minmax` attribute), `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 `str2type` 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. Here 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`, or `1/s`.
- `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 is automatically assigned if `widget` is given, otherwise one applies the default value to find the right `str2type` function. This means that it is strictly not necessary to assign for the "Wind velocity" data item. 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, here the type of numerical solution methods that can be chosen.
- `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.
- `textline` 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 objects will 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 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, enough of the subpool path must be given, e.g.,

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

Our specific computing 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, 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 perform

```
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 `--Initial_angle` also works since it is a unique name in the pool tree.

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 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

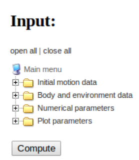


Figure 7: Web interface in closed form.

```
from parampool.generator.flask import generate
from compute import compute_motion_and_forces_with_menu, \
     menu_definition_list

generate(compute_motion_and_forces_with_menu,
         menu_function=menu_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.

3.6 Operating the user interface

The pool tree is mapped onto a visual structure often used for directory trees. 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 typeset in LaTeX and inserted as a picture.

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 (HTML5 number field) is chosen. Choosing `float` or `integer` explicitly as widget may lead to different width of the widget in different browsers.

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

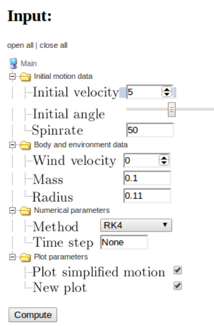


Figure 8: Web interface in fully expanded form.

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 pool 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

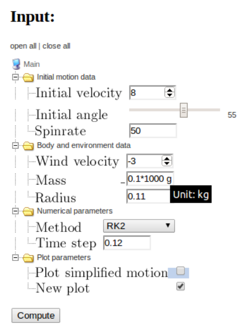


Figure 9: Web interface with input parameters filled out.

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.

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).

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.

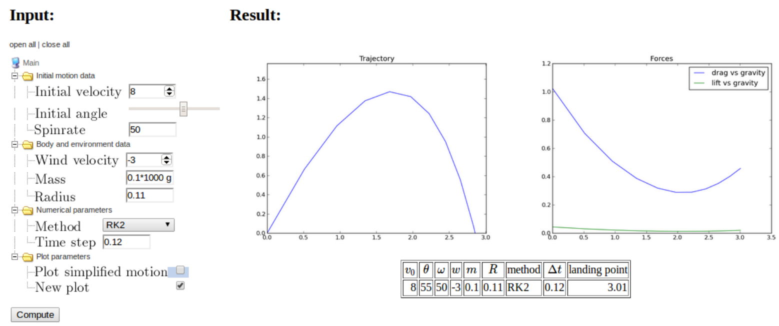


Figure 10: Web interface with input and results.

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 raises a `TypeError` because `float(dt)` fails when `dt` is a list. A remedy is to write a tailored `str2type` function:

```
def convert_time_step(data_item, value):
    # Value must be None or a float
    if value == 'None':
        return None
    else:
        try:
            return float(value)
        except TypeError:
            raise TypeError('%s: could not convert "%s" to float' %
                            (data_item.name, 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 functions 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,
        set_defaults_from_file,
        set_defaults_from_command_line,
        set_defaults_in_model_file,
        write_poolfile,
        )
    # 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

    pool = set_defaults_from_file(pool, command_line_option='--poolfile')
    pool = set_defaults_from_command_line(pool)
    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)

    poolfile = open('pool.dat', 'w')
    poolfile.write(write_poolfile(pool))
    poolfile.close()

    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 is 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_menu_loop, \
     menu_definition_api_with_separate_subpools

generate(compute_motion_and_forces_with_menu_loop,
         menu_function=menu_definition_api_with_separate_subpools,
         MathJax=True, doc=open('doc.html', 'r').read(),
         latex_name='symbol')
```

The values of `latex_name` can be `'symbol'` (symbol only) or `'text, symbol'` (ordinary name 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_motion_and_forces` function that runs the simulation and returns the results as HTML code. Figure 11 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. One realizes 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.

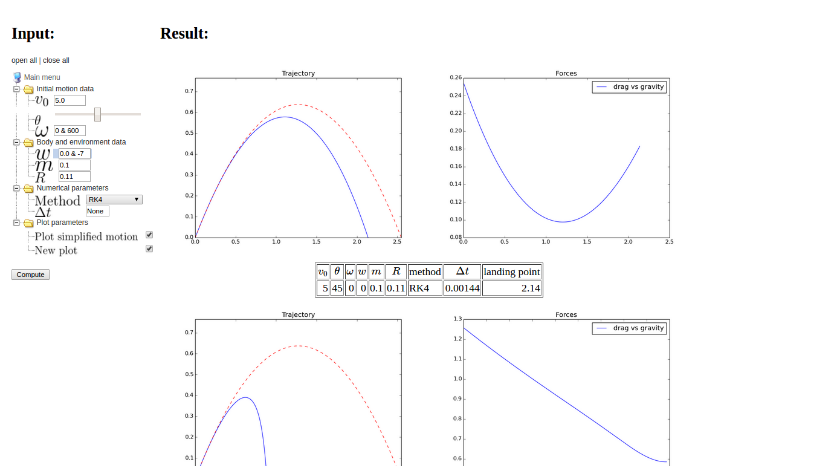


Figure 11: Web interface with documentation, LATEX symbols, and multiple input values.

3.11 2DO

- Lorenz demo as exercise, provide the compute function and graphics, perhaps also good demo in web4sciapps.
- units and text fields: Done
- pool list: no doc
- pool API: add doc in `doc.html`, also show possibility to call
- Check if multiple answers are correctly handled in DataItem, make `enable_multiple_answers(*arg` which turns all widgets into textline (or those given as arguments), make multiple loop parameters, can get them as list of dicts `pool.get_multiple_values(*args)` (all if no argument, otherwise those params listed), let keys be either full path or shortest possible path, can iterate over this list, test in ball example by having a new function with the multiple loop calling up `compute_motion_and_forces_plot` with `new_plot=True` first time and false the next times.
- Let `w` be function, widget=texline, str2type can be StringFunction. Can also invent a special syntax and translate to StringFunction or just make a function.
- Let `w` be filename: upload a Python module with the w(t) function.
- Problem: non-unique data item names cause trouble for the model class since just name is used as static variable...

- Show auto edit of the HTML code, e.g., removing Input: and Result:, this is easier than freezing the files and manually editing them. Reason: if you add new data itmes or subpools to the pool, it is convenient to be able to regenerate the whole setup.

```
from parampool.generator.flask.generate_template import \
     run_doconce_on_text
doc = run_doconce_on_text(compute_function.__doc__)
```

3.12 Reading default values from command line or file

To be programmed: Insert `pool = set_defaults_from_file(pool)` after calling `pool_function`. Then call `set_values_from_command_line(pool)` to override from command line. At the end of the `controller.py` script, dump pool to file again with `write_poolfile(filename)`. (The filename is given on the command line: `--poolfile filename`.)

See the `...separate_subpools` function - it has almost all this functionality!

3.13 Pool with other data structures

Filename and text field with special `str2type` conversion. Allow list syntax, but convert to array in `str2type` after list is eval'ed.

4 Advanced topics

This functionality much in place, but still not documented.

4.1 Login and archival of results

A user can create an account, log in, and save the results. 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:

```python
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.

4.2 Animations

4.3 Python expressions as input

This can only be done in pools unless the value of the parameter is a string (positional argument or default field as text, with explicit conversion). Treat pool only - drop implementation for inspection of compute function.

Describe input as

- `sin(2.5)*exp(-1)` (need to specify py code to be included in the controller/views or add such code manually? or do from numpy import * always? an argument with pycode is probably smart, the pool needs something similar for running str2type=eval in the right namespaces, could just be an additional data item attr namespace that, if present, is used if str2type is eval - yes, that's the solution)
- `MySpecialObject(2,3,4)` (works)

5 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 returns the integral of $\int_a^b f(x)dx$ computed by some numerical integration rule (e.g., the Trapezoidal rule). Also try to integrate $f(x)$ symbolically with the aid `sympy` and write out the anti-derivative of $f(x)$ if the integration is successful.

Hint. The relevant `sympy` code needs to turn the string expression for $f(x)$ into a valid symbolic expression via `eval`. This process requires import of the mathematical functions in `sympy`. Thereafter, the `integrate` functionality in `sympy` can be used to compute the integral.

```
def symbolic_integration(f_str):
    """
    Return the LaTeX code of the anti-derivative of f(x),
    where f_str holdes the formula for f(x) in a string.
    Return None if sympy cannot compute the anti-derivative.
    """
    from sympy import Symbol, integrate, \
      sin, cos, tan, log, asin, acos, atan, \
      sinh, cosh, tanh, asinh, acosh, atanh, erf, erfc,
    x = Symbol('x')
    f = eval(f_str)
    I = integrate(f, x)
    if isinstance(I, Integral):
        return None
    else:
        return latex(I)
```

6 Exercise: Make a web app for studying vibrations

Download `bumpy`. Set up prms.

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

Make an interface to it such that

- s=k*u, have k as parameter, linear damping, F=0, V=0
- F=A*sin(w*t), A and w are parameters
- damping with two values and radio buttons, b
- s is a text expression
- F is a filename (0 is default)
- there is a subpool for F with different models: filename, A*sin(w*t) (subpool), white noise with intensity, formula of t
- another subpool for s models

On the main pool: damping can have radio buttons for linear b*u and quadratic b*u**2, F has list of different type of forces

Problem 2: Make a coin flipper

Make a web application where we can set the number of coins to be flipped. 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.

Hint. See `random.org/coins` for inspiration and images of coins.
Filename: `coin_flipper`.

7 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.

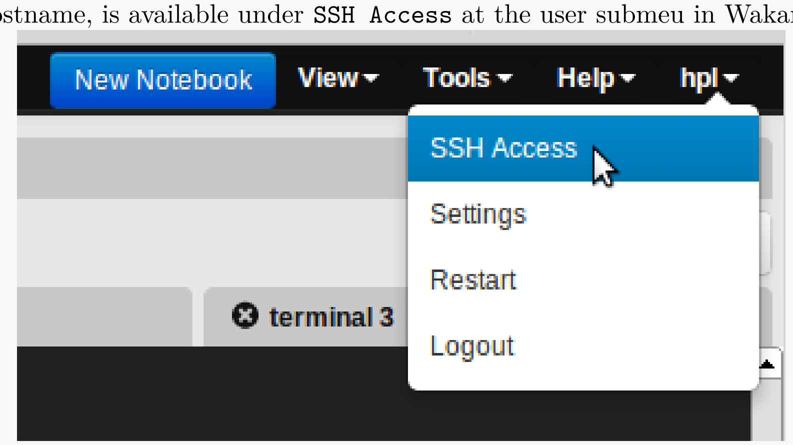
7.1 Wakari

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 Django source and install it locally to 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.

7.2 Uni Oslo

The biggest downside with the Uni Oslo server is that you need to be a student or employee to access it. Nevertheless, the procedure described below for running web applications through a CGI script can be applied to any server.

If only little traffic has to be handled, 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 CGI script running Flask follows:

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

CGIHandler().run(app)
```

This code assumes that the Python executive in `/usr/bin/` is available and up-to-date, and that all required Python modules are on the `PYTHONPATH` environment and are available to be run by others. As the Python version on the Uni Oslo server, version 2.5, is no longer supported by the Python web frameworks, we need to install a newer version locally and use the absolute path to its executive in the header of the CGI script.

With a new Python installation we also need to (re)install all required modules and add them to `sys.path`. The script `install_on_uio.sh` in the top directory of cbc-websolver installs all required packages locally. Then we can add the location of these modules to `sys.path` in the CGI script:

```
import sys
sys.path += ['/path/to/local/installation/lib/python2.7/site-packages/']

CGIHandler().run(app)
```

One also needs to assure that the location of controller.py is in sys.path, or the import of app will fail.

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']` to settings. 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)
```

Make CGI script accessible for others.

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.,

```
Terminal> chmod 755 run_django.cgi
```

A Installation

We need

- Flask: `sudo pip install Flask`
- Django (optional): `sudo pip install django`
- 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`