pypet Documentation

Release 0.1.0

Robert Meyer

September 03, 2015

Contents

CHAPTER 1

pypet User Manual

1.1 What is *pypet* all about?

Whenever you do numerical simulations in science you come across two major problems: First, you need some way to save your data. Secondly, you extensively explore the parameter space. In order to accomplish both you write some hacky I/O functionality to get it done the quick and dirty way. Storing stuff into text files, as *MATLAB* *m*-files, or whatever comes in handy.

After a while and many simulations later, you want to look back at some of your very first results. But because of unforeseen circumstances, you changed lots of your code. As a consequence, you can no longer use your old data, but you need to write a hacky converter to format your previous results to your new needs. The more complexity you add to your simulations, the worse it gets, and you spend way too much time formatting your data than doing science.

Indeed, this was a situation I was confronted with pretty soon at the beginning of my PhD. So this project was born. I wanted to tackle the I/O problems more generally and produce code that was not specific to my current simulations, but I could also use for future scientific projects right out of the box.

The **python parameter exploration toolkit** (*pypet*) provides a framework to define *parameters* that you need to run your simulations. You can actively explore these by following a *trajectory* through the space spanned by the parameters. And finally, you can get your *results* together and store everything appropriately to disk. The storage format of choice is HDF5 via PyTables.

1.1.1 Main Features

- **Novel tree container** *Trajectory*, for handling and managing of parameters and results of numerical simulations
- **Group** your parameters and results into meaningful categories
- Access data via **natural naming**, e.g. `traj.parameters.traffic.ncars`
- Automatic **storage** of simulation data into HDF5 files via PyTables
- Support for many different **data formats**
  - python native data types: bool, int, long, float, str, complex
  - list, tuple, dict
  - Numpy arrays and matrices
  - Scipy sparse matrices
  - pandas Series, DataFrames, and Panels
  - BRIAN quantities and monitors
- Easily **extendable** to other data formats!

- **Exploration** of the parameter space of your simulations
- **Merging** of *trajectories* residing in the same space
- Support for **multiprocessing**, *pypet* can run your simulations in parallel
- **Analyse** your data on-the-fly during multiprocessing for adaptive exploration of the parameter space
- **Dynamic Loading**, load only the parts of your data you currently need
- **Resume** a crashed or halted simulation
- **Annotate** your parameters, results and groups
- **Git Integration**, let *pypet* make automatic commits of your codebase
- **Sumatra Integration**, let *pypet* add your simulations to the *electronic lab notebook* tool Sumatra

1.2 Getting Started

1.2.1 Requirements

Python 2.6, 2.7, 3.3, or 3.4 [1], and

- numpy >= 1.6.1
- scipy >= 0.9.0
- tables >= 2.3.1
- pandas >= 0.12.0 [2]
- HDF5 >= 1.8.9

If you use Python 2.6 you also need

- ordereddict >= 1.1
- importlib >= 1.0.1
- logutils >= 0.3.3
- unittest2

For git integration you additionally need

- GitPython >= 0.3.1

To utilize the cap feature for *Multiprocessing* you need

- psutil >= 2.0.0

To utilize the continuing of crashed trajectories you need

- dill >= 0.2.1

Automatic sumatra records are supported for

- Sumatra >= 0.7.1

[1] *pypet* might also work under Python 3.0-3.2 but has not been tested.

[2] Preferably use pandas 0.14.1 or higher or 0.12.0 since there are some upcasting issues with version 0.13.x (see https://github.com/pydata/pandas/issues/6526/). *pypet* works under 0.13.x but not all features are fully supported. For instance, these upcasting issues may prevent you from storing Trajectories containing ArrayParameters to disk. These unwanted upcastings did not happen in previous pandas versions and will be, or more precisely, have already been removed in the next pandas version. So please up or downgrade your pandas distribution if your current installation is 0.13.x.

1.2.2 Install

If you don't have all prerequisites (numpy, scipy, tables, pandas) install them first. These are standard python packages, so chances are high that they are already installed. By the way, in case you use the python package manager `pip` you can list all installed packages with `pip freeze`.

Next, simply install *pypet* via `pip install --pre pypet` (`--pre` since the current version is still *beta*)

Or

The package release can also be found on pypi.python.org. Download, unpack and `python setup.py install` it.

Or

In case you use **Windows**, you have to download the tar file from pypi.python.org and unzip it [3]. Next, open a windows terminal [4] and navigate to your unpacked *pypet* files to the folder containing the *setup.py* file. As above, run from the terminal `python setup.py install`.

Support

Checkout the pypet Google Group.

To report bugs please use the issue functionality on **github** (https://github.com/SmokinCaterpillar/pypet).

1.2.3 What to do with *pypet*?

The whole project evolves around a novel container object called *trajectory*. A *trajectory* is a container for *parameters* and *results* of numerical simulations in python. In fact a *trajectory* instantiates a tree and the tree structure will be mapped one to one in the HDF5 file when you store data to disk. But more on that later.

As said before a *trajectory* contains *parameters*, the basic building blocks that completely define the initial conditions of your numerical simulations. Usually, these are very basic data types, like integers, floats or maybe a bit more complex numpy arrays.

For example, you have written a set functions that simulates traffic jam in Rome. Your simulation takes a lot of *parameters*, the amount of cars (integer), their potential destinations (numpy array of strings), number of pedestrians (integer), random number generator seeds (numpy integer array), open parking spots in Rome (your *parameter* value is probably 0 here), and all other sorts of things. These values are added to your *trajectory* container and can be retrieved from there during the runtime of your simulation.

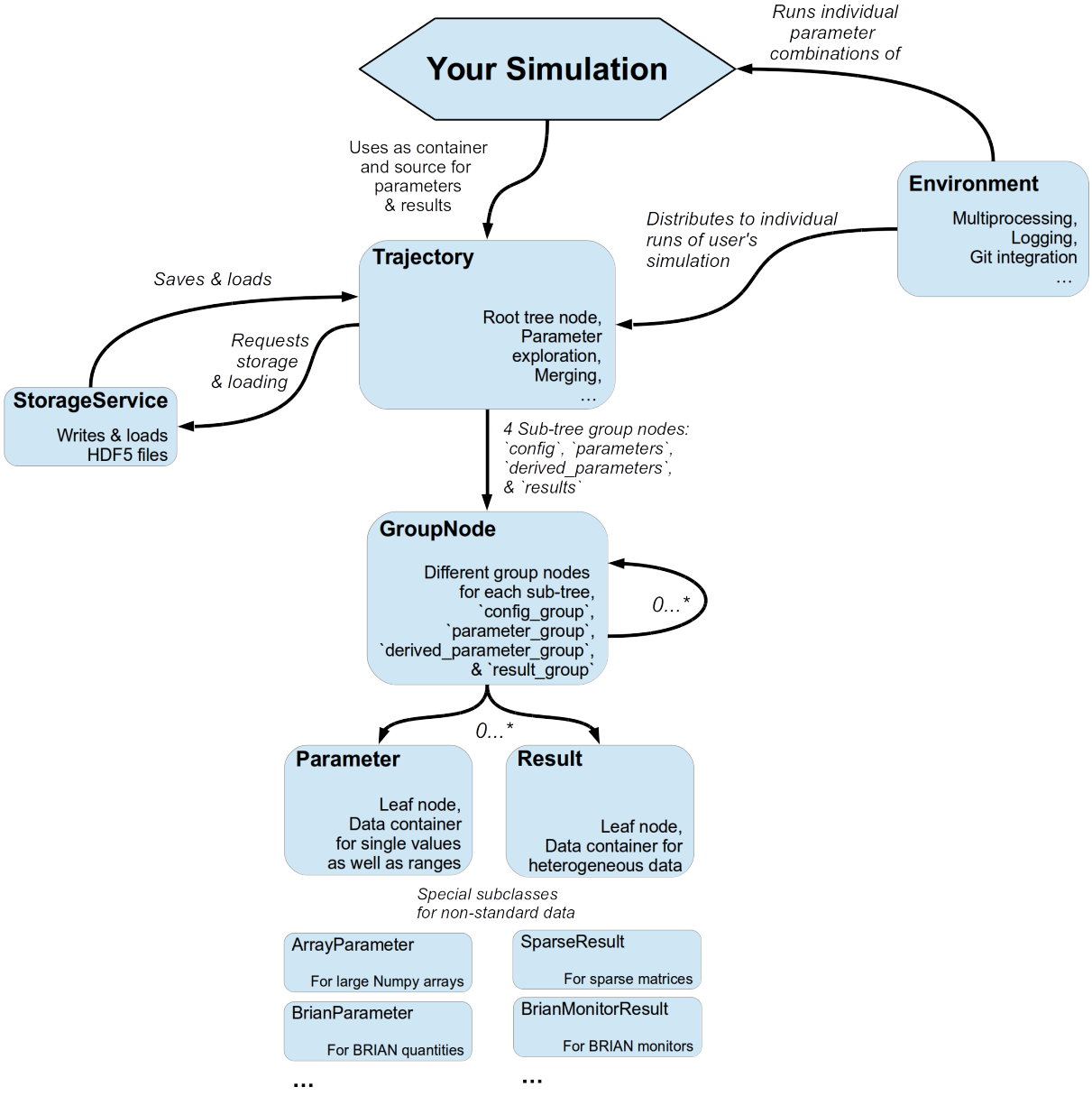
Doing numerical simulations usually means that you cannot find analytical solutions to your problems. Accordingly, you want to evaluate your simulations on very different *parameter* settings and investigate the effect of changing the *parameters*. To phrase that differently, you want to *explore* the parameter space. Coming back to the traffic jam simulations, you could tell your *trajectory* that you want to investigate how different amounts of cars and pedestrians influence traffic problems in Rome. So you define sets of combinations of cars and pedestrians and make individual simulation *runs* for these sets. To phrase that differently, you follow a predefined *trajectory* of points through your *parameter* space and evaluate their outcome. And that's why the container is called *trajectory*.

For each *run* of your simulation, with a particular combination of cars and pedestrians, you record time series data of traffic densities at major sites in Rome. This time series data (let's say they are pandas DataFrames) can also be added to your *trajectory* container. In the end everything will be stored to disk. The storage is handled by an extra service to store the *trajectory* into an HDF5 file on your hard drive. Probably other formats like SQL might be implemented in the future (or maybe **you** want to contribute some code and write an SQL storage service?).

[3] Extract using WinRaR, 7zip, etc. You might need to unpack it twice, first the *tar.gz* file and then the remaining *tar* file in the subfolder.
[4] In case you forgot how, you open a terminal by pressing *Windows Button + R*. Then type *cmd* into the dialog box and press *OK*.

1.2.4 Basic Work Flow

Basic workflow is summarized in the image you can find below. Usually you use an `Environment` for handling the execution and running of your simulation. As in the example code snippet in the next subsection, the environment will provide a `Trajectory` container for you to fill in your parameters. During the execution of your simulation with individual parameter combinations, the *trajectory* can also be used to store results. All data that you hand over to a *trajectory* is automatically stored into an HDF5 file by the `HDF5StorageService`.



1.2.5 Quick Working Example

The best way to show how stuff works is by giving examples. I will start right away with a very simple code snippet (it can also be found here: *First Steps*).

Well, what we have in mind is some sort of numerical simulation. For now we will keep it simple, let's say we need to simulate the multiplication of 2 values, i.e. $z = x * y$. We have two objectives, a) we want to store results of this simulation z and b) we want to *explore* the parameter space and try different values of x and y.

Let's take a look at the snippet at once:

```
from pypet import Environment, cartesian_product


def multiply(traj):
    """Example of a sophisticated simulation that involves multiplying two values.

    :param traj:

        Trajectory containing
        the parameters in a particular combination,
        it also serves as a container for results.

    """
    z = traj.x * traj.y
    traj.f_add_result('z',z, comment='I am the product of two values!')


# Create an environment that handles running our simulation
env = Environment(trajectory='Multiplication',filename='./HDF/example_01.hdf5',
                  file_title='Example_01',
                  comment='I am a simple example!',
                  large_overview_tables=True)

# Get the trajectory from the environment
traj = env.v_trajectory

# Add both parameters
traj.f_add_parameter('x', 1.0, comment='Im the first dimension!')
traj.f_add_parameter('y', 1.0, comment='Im the second dimension!')

# Explore the parameters with a cartesian product
traj.f_explore(cartesian_product({'x':[1.0,2.0,3.0,4.0], 'y':[6.0,7.0,8.0]}))

# Run the simulation with all parameter combinations
env.f_run(multiply)

# Finally disable logging and close all log-files
env.f_disable_logging()
```

And now let's go through it one by one. At first, we have a job to do, that is multiplying two values:

```
def multiply(traj):
    """Example of a sophisticated simulation that involves multiplying two values.

    :param traj:

        Trajectory containing
        the parameters in a particular combination,
        it also serves as a container for results.

    """
    z=traj.x * traj.y
    traj.f_add_result('z',z, comment='I am the product of two values!')
```

This is our simulation function `multiply`. The function makes use of a *`Trajectory`* container which manages our parameters. Here the *trajectory* holds a particular parameter space point, i.e. a particular choice of x and y. In general a *trajectory* contains many parameter settings, i.e. choices of points sampled from the parameter space. Thus, by sampling points from the space one follows a trajectory through the parameter space - therefore the name of the container.

We can access the parameters simply by natural naming, as seen above via `traj.x` and `traj.y`. The value of z is simply added as a result to the `traj` container.

After the definition of the job that we want to simulate, we create an *environment* which will run the simulation.

Moreover, the environment will take care that the function `multiply` is called with each choice of parameters once.

```
# Create an environment that handles running our simulation
env = Environment(trajectory='Multiplication',filename='./HDF/example_01.hdf5',
                  file_title='Example_01',
                  comment = 'I am a simple example!',
                  large_overview_tables=True)
```

We pass some arguments here to the constructor. This is the name of the new trajectory, a filename to store the trajectory into, the title of the file, and a descriptive comment that is attached to the trajectory. We also set `large_overview_tables=True` to get a nice summary of all our computed z values in a single table. This is disabled by default to yield smaller and more compact HDF5 files. But for smaller projects with only a few results, you can enable it without wasting much space. You can pass many more (or less) arguments if you like, check out *More about the Environment* and *Environment* for a complete list. The environment will automatically generate a trajectory for us which we can access via the property `v_trajectory`.

```
# Get the trajectory from the environment
traj = env.v_trajectory
```

Now we need to populate our trajectory with our parameters. They are added with the default values of $x = y = 1.0$.

```
# Add both parameters
traj.f_add_parameter('x', 1.0, comment='Im the first dimension!')
traj.f_add_parameter('y', 1.0, comment='Im the second dimension!')
```

Well, calculating $1.0 * 1.0$ is quite boring, we want to figure out more products. Let's find the results of the cartesian product set $\{1.0, 2.0, 3.0, 4.0\} \times \{6.0, 7.0, 8.0\}$. Therefore, we use *f_explore()* in combination with the builder function *cartesian_product()* that yields the cartesian product of both parameter ranges. You don't have to explore a cartesian product all the time. You can explore arbitrary trajectories through your space. You only need to pass a dictionary of lists (or other iterables) of the same length with arbitrary entries to *f_explore()*. In fact, *cartesian_product()* turns the dictionary *{'x':[1.0,2.0,3.0,4.0], 'y':[6.0,7.0,8.0]}* into a new one where the values of 'x' and 'y' are two lists of length 12 containing all pairings of points.

```
# Explore the parameters with a cartesian product:
traj.f_explore(cartesian_product({'x':[1.0,2.0,3.0,4.0], 'y':[6.0,7.0,8.0]}))
```

Finally, we need to tell the environment to run our job *multiply* with all parameter combinations.

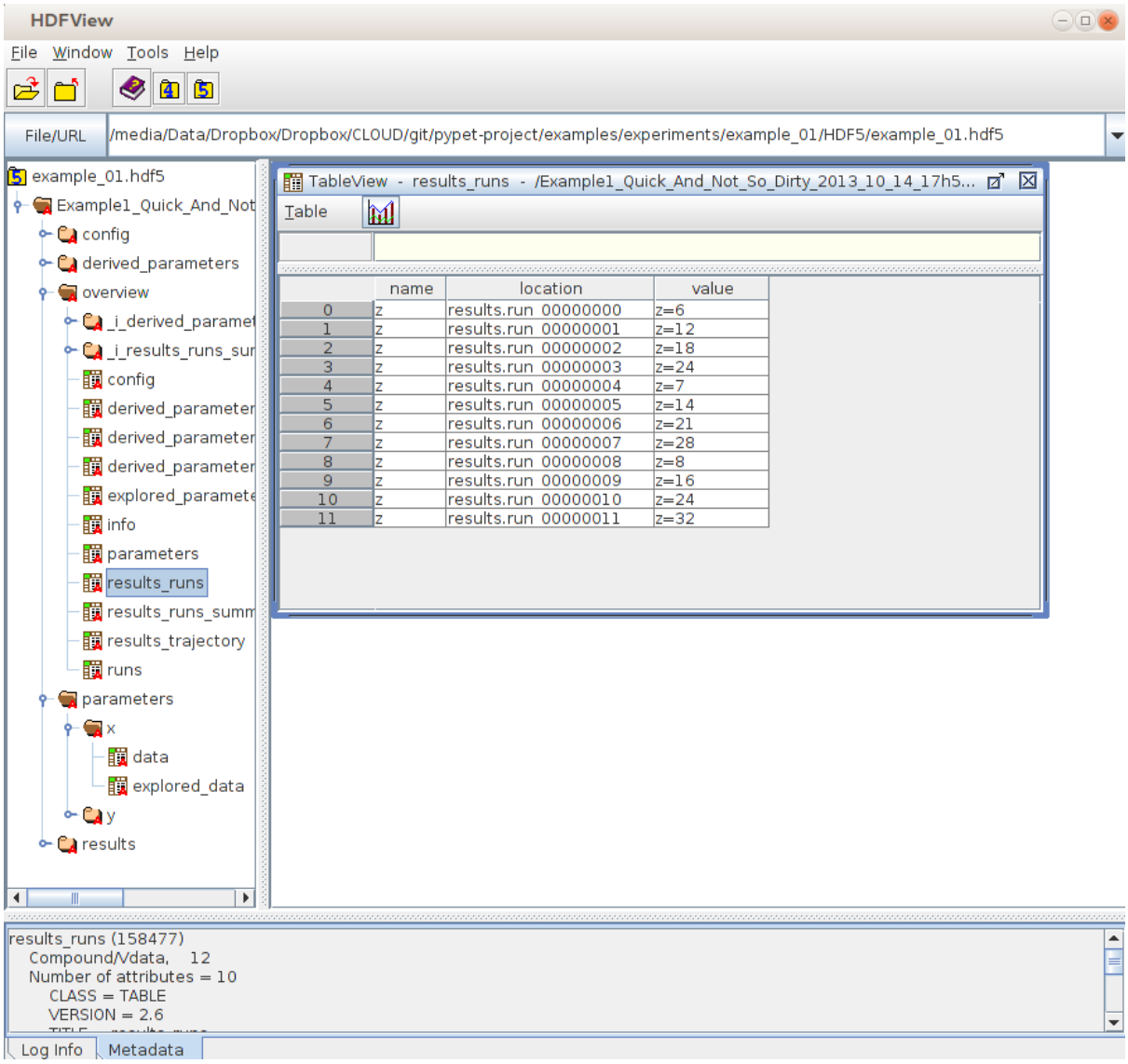
```
# Run the simulation with all parameter combinations
env.f_run(multiply)
```

Usually, if you let *pypet* manage logging for you, it is a good idea in the end to tell the environment to stop logging and close all log files.

```
# Finally disable logging and close all log-files
env.f_disable_logging()
```

And that's it. The environment will evoke the function *multiply* now 12 times with all parameter combinations. Every time it will pass a *Trajectory* container with another one of these 12 combinations of different x and y values to calculate the value of z. And all of this is automatically stored to disk in HDF5 format.

If we now inspect the new HDF5 file in *examples/HDF/example_01.hdf5*, we can find our *trajectory* containing all parameters and results. Here you can see the summarizing overview table discussed above.



Loading Data

We end this example by showing how we can reload the data that we have computed before. Here we want to load all data at once, but as an example just print the result of *run_00000001* where x was 2.0 and y was 6.0. For loading of data we do not need an environment. Instead, we can construct an empty trajectory container and load all data into it by ourselves.

```
from pypet import Trajectory

# So, first let's create a new empty trajectory and pass it the path and name of the HDF5 file.
traj = Trajectory(filename='experiments/example_01/HDF5/example_01.hdf5')

# Now we want to load all stored data.
traj.f_load(index=-1, load_parameters=2, load_results=2)

# Finally we want to print a result of a particular run.
# Let's take the second run named `run_00000001` (Note that counting starts at 0!).
print 'The result of run_00000001 is: '
print traj.run_00000001.z
```

This yields the statement *The result of run_00000001 is: 12* printed to the console.

Some final remarks on the command:

```
# Now we want to load all stored data.
traj.f_load(index=-1, load_parameters=2, load_results=2)
```

Above `index` specifies that we want to load the trajectory with that particular index within the HDF5 file. We could instead also specify a `name`. Counting works also backwards, so `-1` yields the last or newest trajectory in the file.

Next, we need to specify how the data is loaded. Therefore, we have to set the keyword arguments `load_parameters` and `load_results`. Here we chose both to be `2`.

`0` would mean we do not want to load anything at all. `1` would mean we only want to load the empty hulls or skeletons of our parameters or results. Accordingly, we would add parameters or results to our trajectory but they would not contain any data. Instead, `2` means we want to load the parameters and results including the data they contain.

So that's it for the start. If you want to know the nitty-gritty details of *pypet* take a look at the *Cookbook*. If you are not the type of guy who reads manuals but wants hands-on experience, check out the *Tutorial* or the *Examples*. If you consider using *pypet* with an already existing project of yours, I may direct your attention to *Wrapping an Existing Project (Cellular Automata Inside!)*.

Cheers, Robert

1.3 Tutorial

1.3.1 Conceptualization of a Numerical Experiment

I will give a simple but comprehensive tutorial on *pypet* and how to use it for parameter exploration of numerical experiments in python.

pypet is designed to support your numerical simulations in two ways: Allow **a)** easy exploration of the parameter space of your simulations and **b)** easy storage of the results.

We will assume that usually a numerical experiments consist of two to four different stages:

1. **Pre-processing** Parameter definition, preparation of the experiment
2. **The *run phase* of your experiment** Fan-out structure, usually parallel running of different parameter settings, gathering of individual results for each single run
3. **Post-processing (optional)** Cleaning up of the experiment, sorting results, etc.
4. **Analysis of results (optional)** Plotting, doing statistics etc.

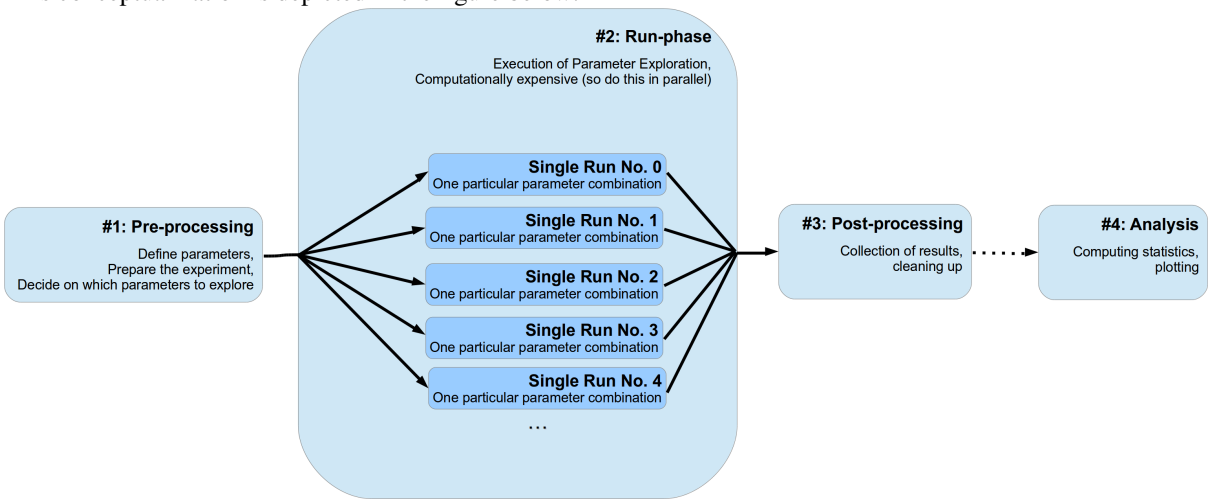
The first stage can be further divided into two sub-stages. In the beginning the definition of parameters (either directly in the source code or by parsing a configuration file) and, next, the appropriate setup of your experiment. This might involve creating particular python objects or pre-computing some expensive functions etc. Moreover, here you also decide if you want to deviate from your default set of parameters and explore the parameter space and try a bunch of different settings. Probably you want to do a sensitivity analysis and determine the effect of changing a critical subset of your parameters.

The second stage, the *run phase*, is the actual execution of your numerical simulation. Here you perform the search or exploration of the parameter space. You try all different parameter settings you have specified before for exploration and obtain the corresponding results. Since this stage is most likely the computational expensive one, you probably want to parallelize your simulations. I will refer to an individual simulation run with one particular parameter combination as a **single run** of your simulation. Since these **single runs** are different individual simulations with different parameter settings, they are completely independent of each other. The results and outcomes of one **single run** should not influence another. Sticking to this assumption makes the parallelization of your experiments much easier. This doesn't mean that non-independent runs cannot be handled by *pypet* (they can!), it rather means you **should not** do this for cleaner and easier portable code and simulations.

Thirdly, after all individual **single runs** are completed you might have a phase of post-processing. This could encompass merging or collecting of results of individual single runs and/or deconstructing some sensitive python objects, etc.

Finally, you do further analysis of the raw results of your numerical simulation, like generating plots and meta statistics, etc. Personally, I would strictly separate this final phase from the previous three. Thus, using a complete different python script than for the phases before.

This conceptualization is depicted in the figure below:



pypet gives is you a tool to make the stages much easier to handle. *pypet* offers a novel tree data container called `Trajectory` that can be used to store all parameters and results of your numerical simulations. Moreover, *pypet* has an `Environment` that allows easy parallel exploration of the parameter space.

We will see how we can use both in our numerical experiment at the different stages. In this tutorial we will simulate a simple neuron model, called leaky integrate-and-fire model. Our neuron model is given by a dynamical variable V that describes the development of the so called *membrane potential* over time. Every time this potential crosses a particular threshold our neuron is *activated* and emits an electrical pulse. These pules, called action potentials or spikes, are the sources of information transmission in the brain. We will stimulate our neuron with an experimental current I and see how this current affects the emission of spikes. For simplicity we assume a system without any physical units except for time in milliseconds.

We will numerically integrate the linear differential equation:

$$\frac{dV}{dt} = -\frac{1}{\tau_V}V + I$$

with a non-linear reset rule $V \leftarrow 0$ if $V \geq 1$ and an additional refractory period of τ_{ref}. If we detect an action potential, i.e. $V \geq 1$, we will keep the voltage V clamped to 0 for the refractory period after the threshold crossing and freeze the differential equation.

Regarding parameter exploration, we will hold the neuron's time constant $\frac{1}{\tau_V} = 10$ ms fixed and explore the parameter space by varying different input currents I and different lengths of the refractory period τ_{ref}.

During the single runs we will record the development of the variable V over time and count the number of threshold crossings to estimate the so called firing rate of a neuron. In the post processing phase we will collect these firing rates and write them into a pandas DataFrame. Don't worry if you are not familiar with pandas. Basically, a pandas DataFrame instantiates a table. It's like a 2D numpy array, but we can index into the table by more than just integers.

Finally, during the analysis, we will plot the neuron's rate as a function of the input current I and the refractory period τ_{ref}.

The entire source code of this example can be found here: *Post-Processing and Pipelining (from the Tutorial)*.

1.3.2 Naming Convention

To avoid confusion with natural naming scheme (see below) and the functionality provided by the environment, trajectory, parameter containers, and so on, I followed the idea by PyTables to use prefixes: `f_` for functions and `v_` for python variables/attributes/properties.

For instance, given a *pypet* result container `myresult`, `myresult.v_comment` is the object's comment attribute and `myresult.f_set(mydata=42)` is the function for adding data to the result container. Whereas `myresult.mydata` might refer to a data item named `mydata` added by the user.

1.3.3 #1 Pre-Processing

Your experiment usually starts with the creation of an *Environment*. Don't worry about the huge amount of parameters you can pass to the constructor, these are more for tweaking of your experiment and the default settings are usually suitable.

Yet, we will shortly discuss the most important ones here.

- `trajectory`

  Here you can either pass an already existing trajectory container or simply a string specifying the name of a new trajectory. In the latter case the environment will create a trajectory container for you.

- `add_time`

  If `True` and the environment creates a new trajectory container, it will add the current time to the name in the format *_XXXX_XX_XX_XXhXXmXXs*. So for instance, if you set `trajectory='Gigawatts_Experiment'` and `add_time=true`, your trajectory's name will be *Gigawatts_Experiment_2015_10_21_04h23m00s*.

- `comment`

  A nice descriptive comment about what you are going to do in your numerical experiment.

- `log_config`

  The name of a logging `.ini` file specifying the logging set up. See *Logging*, or the logging documentation and how to specify logging config files. If set to *DEFAULT_LOGGING*(`'DEFAULT'`) the default settings are used. Simply set to *None* if you want to disable logging.

- `multiproc`

  If we want to use multiprocessing. We sure do so, so we set this to `True`.

- `ncores`

  The number of cpu cores we want to utilize. More precisely, the number of processes we start at the same time to calculate the single runs. There's usually no benefit in setting this value higher than the actual number of cores your computer has.

- `filename`

  We can specify the name of the resulting HDF5 file where all data will be stored. We don't have to give a filename per se, we can also specify a folder `'./results/'` and the new file will have the name of the trajectory.

- `git_repository`

  If your code base is under git version control (it's not? Stop reading and get git NOW! ;-), you can specify the path to your root git folder here. If you do this, *pypet* will a) trigger a new commit if it detects changes in the working copy of your code and b) write the corresponding commit code into your trajectory so you can immediately see with which version you did your experiments.

- `git_fail`

  If you don't want automatic commits, simply set `git_fail=True`. Given changes in your code base, your program will throw a *GitDiffError* instead of making an automatic commit. Then, you can manually make a commit and restart your program with the committed changes.

- `sumatra_project`

If your experiments are recorded with sumatra you can specify the path to your sumatra root folder here. *pypet* will automatically trigger the recording of your experiments if you use `f_run()`, `f_continue()` or `f_pipeline()` to start your single runs or whole experiment. If you use *pypet* + git + sumatra there's no doubt that you ensure the repeatability of your experiments!

Ok, so let's start with creating an environment:

```
from pypet import Environment
env = Environment(trajectory='FiringRate',
                  comment='Experiment to measure the firing rate '
                        'of a leaky integrate and fire neuron. '
                        'Exploring different input currents, '
                        'as well as refractory periods',
                  add_time=False, # We don't want to add the current time to the name,
                  log_config='DEFAULT',
                  multiproc=True,
                  ncores=2, #My laptop has 2 cores ;-)
                  filename='./hdf5/', # We only pass a folder here, so the name is chosen
                  # automatically to be the same as the Trajectory
                  )
```

The environment provides a new trajectory container for us:

```
traj = env.v_trajectory
```

1.3.4 The Trajectory Container

A `Trajectory` is the container for your parameters and results. It basically instantiates a tree.

This tree has four major branches: *config* (parameters), *parameters*, *derived_parameters* and *results*.

Parameters stored under *config* do not specify the outcome of your simulations but only the way how the simulations are carried out. For instance, this might encompass the number of cpu cores for multiprocessing. In fact, the environment from above has already added the config data we specified before to the trajectory:

```
>>> traj.config.ncores
2
```

Parameters in the *parameters* branch are the fundamental building blocks of your simulations. Changing a parameter usually effects the results you obtain in the end. The set of parameters should be complete and sufficient to characterize a simulation. Running a numerical simulation twice with the very same parameter settings should give also the very same results. So make sure to also add seed values of random number generators to your parameter set.

Derived parameters are specifications of your simulations that, as the name says, depend on your original parameters but are still used to carry out your simulation. They are somewhat too premature to be considered as final results. We won't have any of these in the tutorial so you can ignore this branch for the moment.

Anything found under *results* is, as expected, a result of your numerical simulation.

Adding of Parameters

Ok, for the moment let's fill the trajectory with parameters for our simulation.

Let's fill it using the `f_add_parameter()` function:

```
traj.f_add_parameter('neuron.V_init', 0.0,
                     comment='The initial condition for the '
                             'membrane potential')
traj.f_add_parameter('neuron.I', 0.0,
                     comment='The externally applied current.')
traj.f_add_parameter('neuron.tau_V', 10.0,
```

```
                        comment='The membrane time constant in milliseconds')
traj.f_add_parameter('neuron.tau_ref', 5.0,
                     comment='The refractory period in milliseconds '
                             'where the membrane potnetial '
                             'is clamped.')

traj.f_add_parameter('simulation.duration', 1000.0,
                        comment='The duration of the experiment in '
                                'milliseconds.')
traj.f_add_parameter('simulation.dt', 0.1,
                        comment='The step size of an Euler integration step.')
```

Again we can provide descriptive comments. All these parameters will be added to the branch *parameters*.

As a side remark, if you think there's a bit too much typing involved here, you can also make use of much shorter notations. For example, granted you imported the *Parameter*, you could replace the last addition by:

```
traj.parameters.simulation.dt = Parameter('dt', 0.1, comment='The step size of an Euler integratio
```

Or even shorter:

```
traj.par.simulation.dt = 0.1, 'The step size of an Euler integration step.'
```

Note that we can *group* the parameters. For instance, we have a group `neuron` that contains parameters defining our neuron model and a group `simulation` that defines the details of the simulation, like the euler step size and the whole runtime. If a group does not exist at the time of a parameter creation, *pypet* will automatically create the groups on the fly.

There's no limit to grouping, and it can be nested:

```
>>> traj.f_add_parameter('brian.hippocampus.nneurons', 99999, comment='Number of neurons in my mo
```

There are analogue functions for *config* data, *results* and *derived_parameters*:

- *f_add_config()*
- *f_add_result()*
- *f_add_derived_parameter()*

If you don't want to stick to these four major branches there is the generic addition:

- *f_add_leaf()*

By the way, you can add particular groups directly with:

- *f_add_parameter_group()*
- *f_add_config_group()*
- *f_add_result_group()*
- *f_add_derived_parameter_group()*

and the generic one:

- *f_add_group()*

Your trajectory tree contains two types of nodes, group nodes and leaf nodes. Group nodes can, as you have seen, contain other group or leaf nodes, whereas leaf nodes are terminal and do not contain more groups or leaves.

The leaf nodes are abstract containers for your actual data. Basically, there exist two sub-types of these leaves *Parameter* containers for your config data, parameters, and derived parameters and *Result* containers for your results.

A *Parameter* can only contain a single data item plus potentially a **range** or list of different values describing how the parameter should be explored in different runs.

A *Result* container can manage several results. You can think of it as non-nested dictionary. Actual data can also be accessed via natural naming or squared brackets (as discussed in the next section below).

For instance:

```
>>> traj.f_add_result('deep.thought', answer=42, question='What do you get if you multiply six by
>>> traj.results.deep.thought.question
'What do you get if you multiply six by nine?'
```

Both leaf containers (*Parameter*, *Result*) support a rich variety of data types. There also exist more specialized versions if the standard ones cannot hold your data, just take a look at *More on Parameters and Results*. If you are still missing some functionality for your particular needs you can simply implement your own leaf containers and put them into the *trajectory*.

Accessing Data

Data can be accessed in several ways. You can, for instance, access data via *natural naming*: `traj.parameters.neuron.tau_ref` or square brackets `traj['parameters']['neuron']['tau_ref']` or `traj['parameters.neuron.tau_ref']`, or `traj['parameters','neuron','tau_ref']`, or use the *f_get()* method.

As long as your tree nodes are unique, you can shortcut through the tree. If there's only one parameter `tau_ref`, `traj.tau_ref` is equivalent to `traj.parameters.neuron.tau_ref`.

Moreover, since a *Parameter* only contains a single value (apart from the range), *pypet* will assume that you usually don't care about the actual container but just about the data. Thus, `traj.parameters.neuron.tau_ref` will immediately return the data value for `tau_ref` and not the corresponding *Parameter* container. If you really need the container itself use *f_get()*. To learn more about this concept of *fast access* of data look at *Accessing Data in the Trajectory*.

Exploring the Data

Next, we can tell the trajectory which parameters we want to explore. We simply need need to pass a dictionary of lists (or other iterables) of the **same length** with arbitrary entries to the trajectory function *f_explore()*.

Every single run in the run phase will contain one setting of parameters in the list. For instance, if our exploration dictionary looks like `{'x':[1,2,3], 'y':[1,1,2]}` the first run will be with parameter *x* set to 1 and *y* to 1, the second with *x* set to 2 and *y* set to 1, and the final third one with *x=3* and *y=2*.

If you want to explore the cartesion product of two iterables not having the same length you can use the *cartesian_product()* builder function. This will return a dictionary of lists of the same length and all combinations of the parameters.

Here is our exploration, we try unitless currents I ranging from 0 to 1.01 in steps of 0.01 for three different refractory periods τ_{ref}:

```
from pypet.utils.explore import cartesian_product

explore_dict = {'neuron.I': np.arange(0, 1.01, 0.01).tolist(),
                'neuron.tau_ref': [5.0, 7.5, 10.0]}

explore_dict = cartesian_product(explore_dict, ('neuron.tau_ref', 'neuron.I'))
# The second argument, the tuple, specifies the order of the cartesian product,
# The variable on the right most side changes fastest and defines the
# 'inner for-loop' of the cartesian product

traj.f_explore(explore_dict)
```

Note that in case we explore some parameters, their default values that we passed before via *f_add_parameter()* are no longer used. If you still want to simulate these, make sure they are part of the lists in the exploration dictionary.

1.3.5 #2 The Run Phase

Next, we define a job or top-level simulation run function (that not necessarily has to be a real python function, any callable object will do the job). This function will be called and executed with every parameter combination we specified before with *f_explore()* in the trajectory container.

In our neuron simulation we have 303 different runs of our simulation. Each run has particular index ranging from 0 to 302 and a particular name that follows the structure *run_XXXXXXXX* where *XXXXXXXX* is replaced with the index and some leading zeros. Thus, our run names range from *run_00000000* to *run_00000302*.

Note that we start counting with 0, so the second run is called *run_00000001* and has index 1!

So here is our top-level simulation or run function:

```
def run_neuron(traj):
    """Runs a simulation of a model neuron.

    :param traj:

        Container with all parameters.

    :return:

        An estimate of the firing rate of the neuron

    """

    # Extract all parameters from `traj`
    V_init = traj.par.neuron.V_init
    I = traj.par.neuron.I
    tau_V = traj.par.neuron.tau_V
    tau_ref = traj.par.neuron.tau_ref
    dt = traj.par.simulation.dt
    duration = traj.par.simulation.duration

    steps = int(duration / float(dt))
    # Create some containers for the Euler integration
    V_array = np.zeros(steps)
    V_array[0] = V_init
    spiketimes = []  # List to collect all times of action potentials

    # Do the Euler integration:
    print 'Starting Euler Integration'
    for step in range(1, steps):
        if V_array[step-1] >= 1:
            # The membrane potential crossed the threshold and we mark this as
            # an action potential
            V_array[step] = 0
            spiketimes.append((step-1)*dt)
        elif spiketimes and step * dt - spiketimes[-1] <= tau_ref:
            # We are in the refractory period, so we simply clamp the voltage
            # to 0
            V_array[step] = 0
        else:
            # Euler Integration step:
            dV = -1/tau_V * V_array[step-1] + I
            V_array[step] = V_array[step-1] + dV*dt

    print 'Finished Euler Integration'

    # Add the voltage trace and spike times
    traj.f_add_result('neuron.$', V=V_array, nspikes=len(spiketimes),
                      comment='Contains the development of the membrane potential over time '
                              'as well as the number of spikes.')
```

```
    # This result will be renamed to `traj.results.neuron.run_XXXXXXXX`.


    # And finally we return the estimate of the firing rate
    return len(spiketimes) / float(traj.par.simulation.duration) * 1000
    # *1000 since we have defined duration in terms of milliseconds
```

Our function has to accept at least one argument and this is our `traj` container. During the execution of our simulation function the *trajectory* will contain just one parameter setting out of our 303 different ones from above. The *environment* will make sure that our function is called with each of our parameter choices once.

For instance, if we currently execute the second run (aka *run_00000001*) all parameters will contain their default values, except `tau_ref` and `I`, they will be set to 5.0 and 0.01, respectively.

Let's take a look at the first few instructions:

```
# Extract all parameters from `traj`
V_init = traj.par.neuron.V_init
I = traj.par.neuron.I
tau_V = traj.par.neuron.tau_V
tau_ref = traj.par.neuron.tau_ref
dt = traj.par.simulation.dt
duration = traj.par.simulation.duration
```

So here we simply extract the parameter values from `traj`. As said before *pypet* is smart to directly return the data value instead of a *Parameter* container. Moreover, remember all parameters will have their default values except `tau_ref` and `I`.

Next, we create a numpy array and a python list and compute the number of steps. This is not specific to *pypet* but simply needed for our neuron simulation:

```
steps = int(duration / float(dt))
# Create some containers for the Euler integration
V_array = np.zeros(steps)
V_array[0] = V_init
spiketimes = []  # List to collect all times of action potentials
```

Also the following steps have nothing to do with *pypet*, so don't worry if you not fully understand what's going on here. This is just the core of our neuron simulation:

```
# Do the Euler integration:
print 'Starting Euler Integration'
for step in range(1, steps):
    if V_array[step-1] >= 1:
        # The membrane potential crossed the threshold and we mark this as
        # an action potential
        V_array[step] = 0
        spiketimes.append((step-1)*dt)
    elif spiketimes and step * dt - spiketimes[-1] <= tau_ref:
        # We are in the refractory period, so we simply clamp the voltage
        # to 0
        V_array[step] = 0
    else:
        # Euler Integration step:
        dV = -1/tau_V * V_array[step-1] + I
        V_array[step] = V_array[step-1] + dV*dt

print 'Finished Euler Integration'
```

This is simply the python description of the following set of equations:

$$\frac{dV}{dt} = -\frac{1}{\tau_V}V + I$$

and $V \leftarrow 0$ **if** $V \geq 1$ **or** $t - t_s \leq \tau_{ref}$ (with t the current time and t_s time of the last spike).

Ok, for now we have finished one particular run ouf our simulation. We computed the development of the membrane potential V over time and put it into `V_array`.

Next, we hand over this data to our trajectory, since we want to keep it and write it into the final HDF5 file:

```
traj.f_add_result('neuron.$', V=V_array, nspikes=len(spiketimes),
                  comment='Contains the development of the membrane potential over time '
                          'as well as the number of spikes.')
```

This statement looks similar to the addition of parameters we have seen before. Yet, there are some subtle differences. As we can see, a result can contain several data items. If we pass them via `NAME=value`, we can later on recall them from the result with `result.NAME`. Secondly, there is this odd `'$'` character in the result's name. Well, recall that we are currently operating in the run phase, accordingly the `run_neuron` function will be executed many times. Thus, we also gather the `V_array` data many times. We need to store this every time under a different name in our trajectory tree. `'$'` is a wildcard character that is replaced by the name of the current run. If we were in the second run, we would store everything under `traj.results.neuron.run_00000001` and in the third run under `traj.results.neuron.run_00000002` and so on and so forth. Consequently, calling `traj.results.neuron.run_00000001.V` will return our membrane voltage array of the second run.

You are not limited to place the `'$'` at the end, for example

```
traj.f_add_result('fundamental.wisdom.$.answer', 42, comment='The answer')
```

would be possible as well.

As a side remark, if you add a result or derived parameter during the run phase but **not** use the `'$'` wildcard, *pypet* will add `runs.'$'` to the beginning of your result's or derived parameter's name.

So executing the following statement during the run phase

```
traj.f_add_result('fundamental.wisdom.answer', 42, comment='The answer')
```

will yield a renaming to `results.runs.run_XXXXXXXXX.fundamental.wisdom.answer`. Where *run_XXXXXXXXX* is the name of the corresponding run, of course.

Moreover, it's worth noticing that you don't have to explicitly write the trajectory to disk. Everything you add during pre-processing, post-processing (see below) is automatically stored at the end of the experiment. Everything you add during the run phase under a group or leaf node called *run_XXXXXXXX* (where this is the name of the current run, which will be automatically chosen if you use the `'$'` wildcard) will be stored at the end of the particular run.

1.3.6 #3 Post-Processing

Each single run of our `run_neuron` function returned an estimate of the firing rate. In the post processing phase we want to collect these estimates and sort them into a table according to the value of I and τ_{ref}. As an appropriate table we choose a pandas DataFrame. Again this is not *pypet* specific but pandas offers neat containers for series, tables and multidimensional panel data. The nice thing about pandas containers is that they except all forms of indices and not only integer indices like python lists or numpy arrays.

So here comes our post processing function. This function will be automatically called when all single runs are completed. The post-processing function has to take at least two arguments. First one is the trajectory, second one is the list of results. This list actually contains two-dimensional tuples. First entry of the tuple is the index of the run as an integer, and second entry is the result returned by our job-function `run_neuron` in the corresponding run. Be aware that since we use multiprocessing, the list is not ordered according to the run indices, but according to the time the single runs did actually finish.

```
def neuron_postproc(traj, result_list):
    """Postprocessing, sorts firing rates into a data frame.

    :param traj:

        Container for results and parameters
```

```
    :param result_list:

        List of tuples, where first entry is the run index and second is the actual
        result of the corresponding run.

    :return:
    """

    # Let's create a pandas DataFrame to sort the computed firing rate according to the
    # parameters. We could have also used a 2D numpy array.
    # But a pandas DataFrame has the advantage that we can index into directly with
    # the parameter values without translating these into integer indices.
    I_range = traj.par.neuron.f_get('I').f_get_range()
    ref_range = traj.par.neuron.f_get('tau_ref').f_get_range()

    I_index = sorted(set(I_range))
    ref_index = sorted(set(ref_range))
    rates_frame = pd.DataFrame(columns=ref_index, index=I_index)
    # This frame is basically a two dimensional table that we can index with our
    # parameters

    # Now iterate over the results. The result list is a list of tuples, with the
    # run index at first position and our result at the second
    for result_tuple in result_list:
        run_idx = result_tuple[0]
        firing_rates = result_tuple[1]
        I_val = I_range[run_idx]
        ref_val = ref_range[run_idx]
        rates_frame.loc[I_val, ref_val] = firing_rates # Put the firing rate into the
        # data frame

    # Finally we going to store our new firing rate table into the trajectory
    traj.f_add_result('summary.firing_rates', rates_frame=rates_frame,
                      comment='Contains a pandas data frame with all firing rates.')
```

Ok, we will go through it one by one. At first we extract the range of parameters we used:

```
I_range = traj.par.neuron.f_get('I').f_get_range()
ref_range = traj.par.neuron.f_get('tau_ref').f_get_range()
```

Note that we use *pypet.naturalnaming.NNGroupNode.f_get()* here since we are interested in the parameter container not the data value. We can directly extract the parameter range from the container via $pypet.parameter.Parameter.f_get_range$.

Next, we create a two dimensional table aka pandas DataFrame with the current as the row indices and the refractory period as column indices.

```
I_index = sorted(set(I_range))
ref_index = sorted(set(ref_range))
rates_frame = pd.DataFrame(columns=ref_index, index=I_index)
```

Now we iterate through the result tuples and write the firing rates into the table according to the parameter settings in this run. As said before, the nice thing about pandas is that we can use the values of I' and τ_{ref}' as indices for our table.

```
for result_tuple in result_list:
        run_idx = result_tuple[0]
        firing_rates = result_tuple[1]
        I_val = I_range[run_idx]
        ref_val = ref_range[run_idx]
        rates_frame.loc[I_val, ref_val] = firing_rates
```

Finally, we add the filled DataFrame to the trajectory.

```
traj.f_add_result('summary.firing_rates', rates_frame=rates_frame,
                  comment='Contains a pandas data frame with all firing rates.')
```

Since we are no longer in the run phase, this result will be found in `traj.results.summary.firing_rate` and **no** name of any single run will be added.

This was our post-processing where we simply collected all firing rates and sorted them into a table. You can, of course, do much more in the post processing phase. You can load all computed data and look at it. You can even expand the trajectory to trigger a new run phase. Accordingly, you can adaptively and iteratively search the parameter space. You may even do this on the fly while there are still single runs being executed, see *Adding Post-Processing*.

1.3.7 Final Steps in the Main Script

Still we actually need to make the environment execute all the stuff, so this is our main script after we generated the environment and added the parameters. First, we add the post-processing function. Secondly, we tell the environment to run our function `run_neuron`. Our postprocessing function will be automatically called after all runs have finished.

```
# Ad the postprocessing function
env.f_add_postprocessing(neuron_postproc)

# Run the experiment
env.f_run(run_neuron)
```

Both function take additional arguments which will be automatically passed to the job and post-processing functions.

For instance,

```
env.f_run(myjob, 42, 'fortytwo', test=33.3)
```

will additionally pass `42, 'fortytwo'` as positional arguments and `test=33.3` as the keyword argument `test` to your run function. So the definition of the run function could look like this:

```
def myjob(traj, number, text, test):
    # do something
```

Remember that the trajectory will always be passed as first argument. This works analogously for the post-processing function as well. Yet, there is the slight difference that your post-processing function needs to accept the result list as second positional argument followed by your positional and keyword arguments.

Finally, if you used *pypet*'s logging feature, it is usually a good idea to tell the environment to stop logging and close all log files:

```
# Finally disable logging and close all log-files
env.f_disable_logging()
```

1.3.8 #4 Analysis

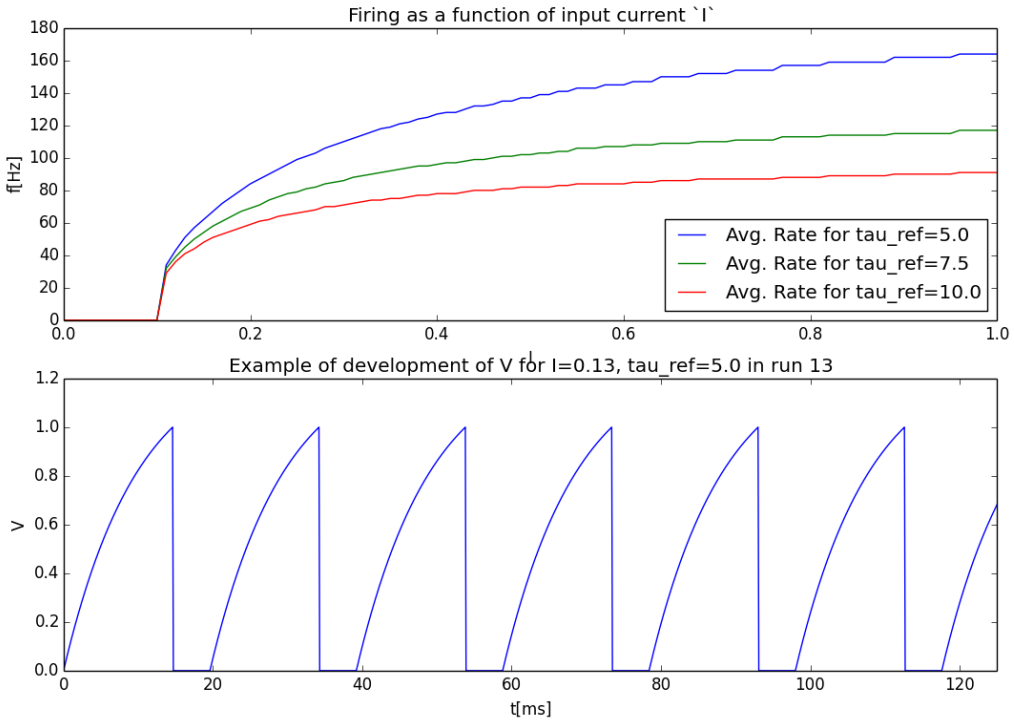
The final stage of our experiment encompasses the analysis of our raw data. We won't do much here, simply plot our firing rate table and show one example voltage trace. All data analysis happens in a completely different script and is executed independently of the previous three steps except that we need the data from them in form of a trajectory.

We will make use of the *Automatic Loading* functionality and load results in the background as we need them. Since we don't want to do any more single runs, we can spare us an environment and only use a trajectory container.

```
from pypet import Trajectory
import matplotlib.pyplot as plt

# This time we don't need an environment since we just going to look
# at data in the trajectory
traj = Trajectory('FiringRate', add_time=False)

# Let's load the trajectory from the file
# Only load the parameters, we will load the results on the fly as we need them
traj.f_load(filename='./hdf5/FiringRate.hdf5', load_parameters=2,
            load_results=0, load_derived_parameters=0)

# We'll simply use auto loading so all data will be loaded when needed.
traj.v_auto_load = True

# Here we load the data automatically on the fly
rates_frame = traj.res.summary.firing_rates.rates_frame

plt.figure()
plt.subplot(2,1,1)
#Let's iterate through the columns and plot the different firing rates :
for tau_ref, I_col in rates_frame.iteritems():
    plt.plot(I_col.index, I_col, label='Avg. Rate for tau_ref=%s' % str(tau_ref))

# Label the plot
plt.xlabel('I')
plt.ylabel('f[Hz]')
plt.title('Firing as a function of input current `I`')
plt.legend()

# Also let's plot an example run, how about run 13?
example_run = 13

traj.v_idx = example_run # We make the trajectory behave as a single run container.
# This short statement has two major effects:
# a) all explored parameters are set to the value of run 13,
# b) if there are tree nodes with names other than the current run aka `run_00000013`
# they are simply ignored, if we use the `$` sign or the `crun` statement,
# these are translated into `run_00000013`.

# Get the example data
example_I = traj.I
example_tau_ref = traj.tau_ref
example_V = traj.results.neuron.crun.V # Here crun stands for run_00000013

# We need the time step...
dt = traj.dt
# ...to create an x-axis for the plot
dt_array = [irun * dt for irun in range(len(example_V))]

# And plot the development of V over time,
# Since this is rather repetitive, we only
# plot the first eighth of it.
plt.subplot(2,1,2)
plt.plot(dt_array, example_V)
plt.xlim((0, dt*len(example_V)/8))

# Label the axis
plt.xlabel('t[ms]')
plt.ylabel('V')
plt.title('Example of development of V for I=%s, tau_ref=%s in run %d' %
          (str(example_I), str(example_tau_ref), traj.v_idx))
```

```
# And let's take a look at it
plt.show()

# Finally revoke the `traj.v_idx=13` statement and set everything back to normal.
# Since our analysis is done here, we could skip that, but it is always a good idea
# to do that.
traj.f_restore_default()
```

The outcome of your little experiment should be the following image:



Finally, I just want to make some final remarks on the analysis script.

```
traj.f_load(filename='./hdf5/FiringRate.hdf5', load_parameters=2,
            load_results=0, load_derived_parameters=0)
```

describes how the different subtrees of the trajectory are loaded (`load_parameters` also includes the `config` branch). 0 means no data at all is loaded, 1 means only the containers are loaded but without any data and 2 means the containers including the data are loaded. So here we load all parameters and all config parameters with data and no results whatsoever.

Yet, since we say `traj.v_auto_load = True` the statement `rates_frame = traj.res.summary.firing_rates.rates_frame` will return our 2D table of firing rates because the data is loaded in the background while we request it.

Furthermore,

```
traj.v_idx = example_run
```

is an important statement in the code. Setting the properties `v_idx` or `v_crun` or using the function *`f_set_crun()`* are equivalent. These give you a powerful tool in data analysis because they make your trajectory behave like a particular single run. Thus, all explored parameter's values will be set to the corresponding values of one particular run.

To restore everything back to normal simply call *`f_restore_default()`*.

This concludes our small tutorial. If you are interested in more advance concepts look into the cookbook or check out the code snippets in the example section. Notably, if you consider using *pypet* with an already existing project

of yours, you might want to pay attention to *Wrapping an Existing Project (Cellular Automata Inside!)*.

Cheers, Robert

1.4 Cookbook

Here you can find some more detailed explanations of various concepts of *pypet*.

1.4.1 Naming Convention

To avoid confusion with natural naming scheme and the functionality provided by the trajectory, parameters, and so on, I followed the idea by PyTables to use prefixes: `f_` for functions and `v_` for python variables/attributes/properties.

For instance, given a result instance `myresult`, `myresult.v_comment` is the object's comment attribute and `myresult.f_set(mydata=42)` is the function for adding data to the result container. Whereas `myresult.mydata` might refer to a data item named `mydata` added by the user.

Moreover, the following abbreviations are supported by *pypet* for interaction with a *Trajectory*:

- `conf` is directly mapped to `config`
- `par` to `parameters`
- `dpar` to `derived_parameters`
- `res` to `results`
- `crun` or the `$` symbol to the name of the current single run, e.g. `run_00000002`
- `r_X` and `run_X` (e.g. `r_2`) are mapped to the corresponding run name, e.g `run_00000008`.

If you add or request data by using the abbreviations, these are automatically translated into the corresponding long forms.

1.4.2 More on Trajectories

Trajectory

For some example code on on topics discussed here see the *Natural Naming, Storage and Loading* script.

The *Trajectory* is the container for all results and parameters (see *More on Parameters and Results*) of your numerical experiments. Throughout the documentation instantiated objects of the *Trajectory* class are usually labeled `traj`. Probably you as user want to follow this convention, because writing the not abbreviated expression `trajectory` all the time in your code can become a bit annoying after some time.

The *trajectory* container instantiates a tree with *groups* and *leaf* nodes, whereas the trajectory object itself is the root node of the tree. There are two types of objects that can be *leaves*: *parameters* and *results*. Both follow particular APIs (see *Parameter* and *Result* as well as their abstract base classes *BaseParameter*, *BaseResult*). Every parameters contains a single value and optionally a range of values for exploration. In contrast, results can contain several heterogeneous data items (see *More on Parameters and Results*).

Moreover, a trajectory contains 4 major tree branches:

- `config` (in short `conf`)

  Data stored under config does not specify the outcome of your simulations but only the way how the simulations are carried out. For instance, this might encompass the number of CPU cores for multiprocessing. If you use and generate a trajectory with an environment (*More about the Environment*), the environment will add some config data to your trajectory.

  Any leaf added under *config* is a *Parameter* object (or descendant of the corresponding base class *BaseParameter*).

As normal parameters, config parameters can only be specified before the actual single runs.

- `parameters` (in short `par`)

  Parameters are the fundamental building blocks of your simulations. Changing a parameter usually effects the results you obtain in the end. The set of parameters should be complete and sufficient to characterize a simulation. Running a numerical simulation twice with the very same parameter settings should give also the very same results. Therefore, it is recommenced to also incorporate seeds for random number generators in your parameter set.

  Any leaf added under *parameters* is a *Parameter* object (or descendant of the corresponding base class *BaseParameter*).

  Parameters can only be introduced to the trajectory before the actual simulation runs.

- `derived_parameters` (in short `dpar`)

  Derived parameters are specifications of your simulations that, as the name says, depend on your original parameters but are still used to carry out your simulation. They are somewhat too premature to be considered as final results. For example, assume a simulation of a neural network, a derived parameter could be the connection matrix specifying how the neurons are linked to each other. Of course, the matrix is completely determined by some parameters, one could think of some kernel parameters and a random seed, but still you actually need the connection matrix to build the final network.

  Any leaf added under *derived_parameters* is a *Parameter* object (or descendant of the corresponding base class *BaseParameter*).

- `results` (in short `res`)

  I guess results are rather self explanatory. Any leaf added under *results* is a *Result* object (or descendant of the corresponding base class *BaseResult*).

Note that all nodes provide the field 'v_comment', which can be filled manually or on construction via `comment=`. To allow others to understand your simulations it is very helpful to provide such a comment and explain what your parameter is good for.

Addition of Groups and Leaves (aka Results and Parameters)

Addition of *leaves* can be achieved via these functions:

- *f_add_config()*
- *f_add_parameter()*
- *f_add_derived_parameter()*
- *f_add_result()*

Leaves can be added to any group, including the root group, i.e. the trajectory. Note that if you operate in the *parameters* subbranch of the tree, you can only add parameters (i.e. `traj.parameters.f_add_parameter(...)` but `traj.parameters.f_add_result(...)` does not work). For other subbranches this is analogous.

There are two ways to use the above functions, either you already have an instantiation of the object, i.e. you add a given parameter:

```
>>> my_param = Parameter('subgroup1.subgroup2.myparam',42, comment='I am an example')
>>> traj.f_add_parameter(my_param)
```

Or you let the trajectory create the parameter, where the name is the first positional argument:

```
>>> traj.f_add_parameter('subgroup1.subgroup2.myparam', 42, comment='I am an example')
```

There exists a standard constructor that is called in case you let the trajectory create the parameter. The standard constructor can be changed via the `v_standard_parameter` property. Default is the *Parameter* constructor.

If you only want to add a different type of parameter once, but not change the standard constructor in general, you can add the constructor as the first positional argument followed by the name as the second argument:

```
>>> traj.f_add_parameter(PickleParameter, 'subgroup1.subgroup2.myparam', 42, comment='I am an exa
```

Derived parameters, config and results work analogously.

You can sort *parameters/results* into groups by colons in the names. For instance, `traj.f_add_parameter('traffic.mobiles.ncars', data = 42)` creates a parameter that is added to the subbranch `parameters`. This will also automatically create the subgroups `traffic` and inside there the group `mobiles`. If you add the parameter `traj.f_add_parameter('traffic.mobiles.ncycles', data = 11)` afterwards, you will find this parameter also in the group `traj.parameters.traffic.ncycles`.

More Ways to Add Data

Moreover, for each of the adding functions there exists a shorter abbreviation that spares you typing:

- *f_aconf()*
- *f_apar()*
- *f_adpar()*
- *f_ares()*

Besides these functions, *pypet* gives you the possibility to add new leaves via generic attribute setting.

For example, you could also add a parameter (or result) as follows:

```
>>> traj.parameters.myparam = Parameter('myparam', 42, comment='I am a useful comment!')
```

Which creates a novel parameter *myparam* under `traj.parameters`. It is important how you choose the name of your parameter or result. If the names match (`.myparam` and `'myparam'`) as above, or if your parameter has the empty string as a name (`traj.parameters.myparam = Parameter('', 42)`), the parameter will be added and named as the generic attribute, here `myparam`. However, if the names disagree or if the parameter or result name contains groups, the generic attribute will become also a group node. For instance,

```
>>> traj.parameters.mygroup = Parameter('myparam', 42)
```

creates a new parameter at `traj.parameters.mygroup.myparam` and `mygroup` is a new group node, respectively. Likewise

```
>>> traj.parameters.mygroup = Parameter('mysubgroup.myparam', 42)
```

adds a new parameter at `traj.parameters.mygroup.mysubgroup.myparam`.

Finally, there's an even simpler way to add a parameter or result, so called lazy adding. You have to turn it on to via *traj.v_lazy_adding=True*

```
>>> traj.v_lazy_adding=True
>>> traj.parameters.myparam = 42, 'I am a useful comment'
```

Accordingly, this is internally translated into

```
>>> traj.parameters.f_add_leaf('myparam', 42, 'I am a useful comment')
```

Where *f_add_leaf()* is a generic addition function, see *Generic Addition* below. This does work for results as well, but you **cannot** pass comments, because

```
>>> traj.results.myresult = 42, 'I am NOT a comment!'
```

will create a result with two data items, first being the value `42` and the second one a string `'I am NOT a comment'`. Comments can be passed to the standard results only as keyword arguments and all *lazy* values are passed as positional arguments. Yet, you can pass as many items to a result as you want. This, for instance, is legit:

```
>>> traj.results.another_result = 42, 43, 44
>>> traj.results.another_result.v_comment = 'Result containing 3 integer values'
>>> traj.results.another_result[2]
44
```

As long as *lazy adding* is turned on, you cannot change existing values. Thus,

```
>>> traj.parameters.myparam = 43
```

will throw an `AttributeError` because `myparam` already exists, and has the value `42`. Yet, after turning it off, it works again:

```
>>> traj.v_lazy_adding = False
>>> traj.par.myparam = 43
>>> traj.myparam
43
```

The different ways of adding data are also explained in example *Adding Data to the Trajectory*.

Group Nodes

Besides *leaves* you can also add empty *groups* to the trajectory (and to all subgroups, of course) via:

- *f_add_config_group()*
- *f_add_parameter_group()*
- *f_add_derived_parameter_group()*
- *f_add_result_group()*

As before, if you create the group `groupA.groupB.groupC` and if group A and B were non-existent before, they will be created on the way.

Note that *pypet* distinguishes between three different types of name descriptions, the *full name* of a node which would be, for instance, `parameters.groupA.groupB.myparam`, the (short) *name* `myparam` and the *location* within the tree, i.e. `parameters.groupA.groupB`. All these properties are accessible for each group and leaf via:

- `v_full_name`
- `v_location`
- `v_name`

Location and *full name* are relative to the root node. Since a trajectory object is the root of the tree, its *full_name* is `''`, the empty string. Yet, the *name* property is not empty but contains the user chosen name of the trajectory.

Note that if you add a parameter/result/group with `f_add_XXXXXX` the full name will be extended by the *full name* of the group you added it to:

```
>>> traj.parameters.traffic.f_add_parameter('street.nzebras')
```

The *full name* of the new parameter is going to be `parameters.traffic.street.nzebras`. If you add anything directly to the *root* group, i.e. the trajectory, the group names `parameters`, `config`, `derived_parameters` will be automatically added (of course, depending on what you add, config, a parameter etc.).

If you add a result or derived parameter during a single run, the name will be changed to include the current name of the run.

For instance, if you add a result during a single run (let's assume it's the first run) like `traj.f_add_result('mygroup.myresult', 42, comment='An important result')`, the result will be renamed to `results.runs.run_00000000.mygroup.myresult`. Accordingly, all results (and derived parameters) of all runs are stored into different parts of the tree and are kept independent.

If this sorting does not really suit you, and you don't want your results and derived parameters to be put in the sub-branches `runs.run_XXXXXXXXX` (with *XXXXXXXX* the index of the current run), you can make use of the wildcard character `'$'`. If you add this character to the name of your new result or derived parameter, *pypet* will automatically replace this wildcard character with the name of the current run.

For instance, if you add a result during a single run (let's assume again the first one) via `traj.f_add_result('mygroup.$.myresult', 42, comment='An important result')` the result will be renamed to `results.mygroup.run_00000000.myresult`. Thus, the branching of your tree happens on a lower level than before. Even `traj.f_add_result('mygroup.mygroup.$', myresult=42, comment='An important result')` is allowed.

You can also use the wildcard character in the preprocessing stage. Let's assume you add the following derived parameter **before** the actual single runs via `traj.f_add_derived_parameter('mygroup.$.myparam', 42, comment='An important parameter')`. If that happend **during** a single run `'$'` would be renamed to `run_XXXXXXXX` (with *XXXXXXXX* the index of the run). Yet, if you add the parameter BEFORE the single runs, `'$'` will be replaced by the placeholder name `run_ALL`. So your new derived parameter here is now called `mygroup.run_All.myparam`.

Why is this useful?

Well, this is in particular useful if you pre-compute derived parameters before the single runs which depend on parameters that might be explored in the near future.

For example you have parameter `seed` and `n` and which you use to draw a vector of random numbers. You keep this vector as a derived parameter. As long as you do not explore different seeds or values of `n` you can compute the random numbers before the single runs to save time. Now, if you use the `'$'` statement right from the beginning it would not make a difference if the following statement was executed during the pre-processing stage or during the single runs:

```
np.random.seed(traj.parameters.seed)
traj.f_add_derived_parameter('random_vector.$', np.random(traj.paramaters.n))
```

In both cases **during** the single run, you can access your data via `traj.dpar.random_vector.crun` and *pypet* will return the data regardless when you added the derived parameter. Internally *pypet* tries to resolve `traj.dpar.random_vector.run_XXXXXXXX` (with `run_XXXXXXXXX` referring to the current run, like `run_00000002`) first. If this fails, it will fall back to `traj.dpar.random_vector.run_ALL` (if this fails, too, *pypet* will throw an error).

Accordingly, you have to write less code and post-processing and data analysis become easier.

More on Wildcards

So far we have seen that the *'$'* wildcard translates into the current run name. Similarly does *crun*. So, `traj.res.runs['$'].myresult` is equivalent to `traj.res.runs.crun.myresult`. By default, there exists another wildcard called *'$set'* or *crunset*. Both translate to grouping of results into buckets of 1000 runs. More precisely, they are translated to *run_set_XXXXX* where *XXXXX* is just the set number. So the first 1000 runs are translated into `run_set_00000`, the next 1000 into `run_set_00001` and so on.

Why is this useful? Well, if you perform many runs, more than 10,000, HDF5 becomes rather slow, because it cannot handle nodes with so many children. Grouping your results into buckets simply overcomes this problem. Accordingly, you could add a result as:

```
>>> traj.f_add_result('$set.$.myresult', 42)
```

And all results will be sorted into groups of 1000 runs, like `traj.results.run_set_00002.run_00002022` for run 2022.

This is also shown in *Large Explorations with Many Runs*.

Moreover, you can actually define your own wildcards or even replace the existing ones. When creating a trajectory you can pass particular wildcard functions via `wildcard_functions`. This has to be a dictionary containing tuples of wildcards like `('$', 'crun')` as keys and translation functions as values. The function needs to take

a single argument, that is the current run index and resolve it into a name. So it must handle all integers of 0 and larger. Moreover, it must also handle *-1* to create a *dummy* name. For instance, you could define your own naming scheme via:

```
from pypet import Trajectory

def my_run_names(idx):
    return 'this_is_run_%d' % d

my_wildcards = {('$', 'crun'): my_run_names}
traj = Trajectory(wildcard_functions=my_wildcards)
```

Now calling `traj.f_add_result('mygroup.$.myresult', 42)` during a run, translates into `traj.mygroup.this_is_run_7` for index 7.

There's basically no constrain on the wildcard functions, except for the one defining *('$', 'crun')* because it has to return a unique name for every integer from -1 to infinity. However, other wildcards can be more open and group many runs together:

```
from pypet import Trajectory

def my_run_names(idx):
    return 'this_is_run_%d' % d

def my_group_names(idx):
    if idx == -1:
        return 'dummy_group'
    elif idx < 9000:
        return 'smaller_than_9000'
    else:
        return 'over_9000'

  my_wildcards = {('$', 'crun'): my_run_names,
                  ('$mygrouping', 'mygrouping'): my_group_names}
  traj = Trajectory(wildcard_functions=my_wildcards)
```

Thus, *traj.f_add_result('mygroup.$mygrouping.$.myresult', 42)*' would translate into `traj.results.mygroup.over_9000.this_is_run_9009` for run 9009.''

Generic Addition

You do not have to stick to the given trajectory structure with its four subtrees: `config`, `parameters`, `derived_parameters`, `results`. If you just want to use a trajectory as a simple tree container and store groups and leaves wherever you like, you can use the generic functions *`f_add_group()`* and *`f_add_leaf()`*. Note however, that the four subtrees are reserved. Thus, if you add anything below one of the four, the corresponding speciality functions from above are called instead of the generic ones.

Accessing Data in the Trajectory

To access data that you have put into your trajectory you can use

- *`f_get()`* method. You might want to take a look at the function definition to check out the other arguments you can pass to `f_get`. `f_get` not only works for the trajectory object, but for any group node in your tree.
- Use natural naming dot notation like `traj.nzebras`. This natural naming scheme supports some special features see below.
- Use the square brackets - as you do with dictionaries - like `traj['nzebras']` which is equivalent to calling `traj.nzebras`.

Natural Naming

As said before *trajectories* instantiate trees and the tree can be browsed via natural naming.

For instance, if you add a parameter via `traj.f_add_parameter('traffic.street.nzebras', data=4)`, you can access it via

```
>>> traj.parameters.street.nzebras
4
```

Here comes also the concept of *fast access*. Instead of the parameter object you directly access the *data* value 4. Whether or not you want fast access is determined by the value of *v_fast_access* (default is True):

```
>>> traj.v_fast_access = False
>>> traj.parameters.street.nzebras
<Parameter object>
```

Note that fast access works for parameter objects (i.e. for everything you store under *parameters*, *derived_parameters*, and *config*) that are non empty. If you say for instance `traj.x` and `x` is an empty parameter, you will get in return the parameter object. Fast access works in one particular case also for results, and that is, if the result contains exactly one item with the name of the result. For instance, if you add the result `traj.f_add_result('z', 42)`, you can fast access it, since the first positional argument is mapped to the name 'z' (See also *Results*). If the result container is empty or contains more than one item, you will always get in return the result object.

```
>>> traj.f_add_result('z', 42)
>>> traj.z
42
>>> traj.f_add_result('k', kay=42)
>>> traj.k
<Result object>
>>> traj.k.kay
42
>>> traj.f_add_result('two_data_values', 11, 12.0)
>>> traj.two_data_values
<Result object>
>>> traj.two_data_values[0]
11
```

Shortcuts

As a user you are encouraged to nicely group and structure your results as fine grain as possible. Yet, you might think that you will inevitably have to type a lot of names and colons to access your values and always state the *full name* of an item. This is, however, not true. There are two ways to work around that. First, you can request the group above the parameters, and then access the variables one by one:

```
>>> mobiles = traj.parameters.traffic.mobiles
>>> mobiles.ncars
42
>>> mobiles.ncycles
11
```

Or you can make use of shortcuts. If you leave out intermediate groups in your natural naming request, a breadth first search is applied to find the corresponding group/leaf.

```
>>> traj.mobiles
42
>>> traj.traffic.mobiles
42
>>> traj.parameters.ncycles
11
```

Search is established with very fast look up and usually needs much less then $O(N)$ [most often $O(1)$ or $O(d)$, where d is the depth of the tree and N the total number of nodes, i.e. *groups* + *leaves*].

However, sometimes your shortcuts are not unique and you might find several solutions for your natural naming search in the tree. *pypet* will return the first item it finds via breadth first search within the tree. If there are several items with the same name but in different depths within the tree, the one with the lowest depth is returned. For performance reasons *pypet* actually stops the search if an item was found and there is no other item within the tree with the same name and same depth. If there happen to be two or more items with the same name and with the same depth in the tree, *pypet* will raise a `NotUniqueNodeError` since *pypet* cannot know which of the two items you want.

The method that performs the natural naming search in the tree can be called directly, it is *f_get()*.

```
>>> traj.parameters.f_get('mobiles.ncars')
<Parameter object ncars>
>>> traj.parameters.f_get('mobiles.ncars', fast_access=True)
42
```

If you don't want to allow this shortcutting through the tree use `f_get(target, shortcuts=False)` or set the trajectory attribute `v_shortcuts=False` to forbid the shortcuts for natural naming and *getitem* access.

As a remainder, there also exist nice naming shortcuts for already present groups (these are always active and cannot be switched off):

- *par* is mapped to *parameters*, i.e. `traj.parameters` is the same group as `traj.par`
- *dpar* is mapped to *derived_parameters*
- *res* is mapped to *results*
- *conf* is mapped to *config*
- *crun* is mapped to the name of the current run (for example *run_00000002*)
- *r_X* and *run_X* are mapped to the corresponding run name, e.g. *r_3* is mapped to *run_00000003*

For instance, `traj.par.traffic.street.nzebras` is equivalent to `traj.parameters.traffic.street.nzebras`.

Links

Although each node in the trajectory tree is identified by a unique *full name*, there can potentially many paths to a particular node established via links.

One can add a link to every group node simply via *f_add_link()*.

For instance:

```
>>> traj.parameters.f_add_link('mylink', traj.f_get('x'))
```

Thus, `traj.mylink` now points to the same data as `traj.x`. Colon separated names are not allowed for links, i.e. `traj.parameters.f_add_link('mygroup.mylink', traj.f_get('x'))` does not work.

Links can also be created via generic attribute setting:

```
>>> traj.mylink2 = traj.f_get('x')
```

See also the example *Using Links*.

Links will be handled as normal children during interaction with the trajectory. For example, using *f_iter_nodes()* with `recursive=True` will also recursively iterate all linked groups and leaves. Moreover, *pypet* takes care that all nodes are only visited once. To skip linked nodes simply set `with_links=False`. However, for storage and loading (see below) links are **never** evaluated recursively. Even setting `recursive=True` linked nodes are, of course, stored or loaded but not their children.

Parameter Exploration

Exploration can be prepared with the function *f_explore()*. This function takes a dictionary with parameter names (not necessarily the full names, they are searched) as keys and iterables specifying how the parameter values for each single run. Note that all iterables need to be of the same length. For example:

```
>>> traj.f_explore({'ncars':[42,44,45,46], 'ncycles' :[1,4,6,6]})
```

This would create a trajectory of length 4 and explore the four parameter space points $(42, 1), (44, 4), (45, 6), (46, 6)$. If you want to explore the cartesian product of parameter ranges, you can take a look at the *cartesian_product()* function.

You can extend or expand an already explored trajectory to explore the parameter space further with the function *f_expand()*.

Using Numpy Iterables

Since parameters are very conservative regarding the data they accept (see *Values supported by Parameters*), you sometimes won't be able to use Numpy arrays for exploration as iterables.

For instance, the following code snippet won't work:

```
import numpy a np
from pypet.trajectory import Trajectory
traj = Trajectory()
traj.f_add_parameter('my_float_parameter', 42.4, comment='My value is a standard python float')

traj.f_explore( { 'my_float_parameter': np.arange(42.0, 44.876, 0.23) } )
```

This will result in a `TypeError` because your exploration iterable `np.arange(42.0, 44.876, 0.23)` contains `numpy.float64` values whereas you parameter is supposed to use standard python floats.

Yet, you can use numpy's `tolist()` function to overcome this problem:

```
traj.f_explore( { 'my_float_parameter': np.arange(42.0, 44.876, 0.23).tolist() } )
```

Or you could specify your parameter directly as a numpy float:

```
traj.f_add_parameter('my_float_parameter', np.float64(42.4),
                     comment='My value is a numpy 64 bit float')
```

Presetting of Parameters

I suggest that before you calculate any results or derived parameters, you should define all parameters used during your simulations. Usually you could do this by parsing a config file, or simply by executing some sort of a config python script that simply adds the parameters to your trajectory (see also *Tutorial*).

If you have some complex simulations where you might use only parts of your parameters or you want to exclude a set of parameters and include some others, you can make use of the **presetting** of parameters (see *f_preset_parameter()*). This allows you to add control flow on the setting or parameters. Let's consider an example:

```
traj.f_add_parameter('traffic.mobiles.add_cars', True , comment='Whether to add some cars or '
                                                        'bicycles in the traffic simulation')
if traj.add_cars:
    traj.f_add_parameter('traffic.mobiles.ncars', 42, comment='Number of cars in Rome')
else:
    traj.f_add_parameter('traffic.mobiles.ncycles', 13, comment'Number of bikes, in case '
                                                            'there are no cars')
```

There you have some control flow. If the variable `add_cars` is True, you will add 42 cars otherwise 13 bikes. Yet, by your definition one line before `add_cars` will always be `True`. To switch between the use cases you can

rely on **presetting** of parameters. If you have the following statement somewhere before in your main function, you can make the trajectory change the value of `add_cars` right after the parameter was added:

```
traj.f_preset_parameter('traffic.mobiles.add_cars', False)
```

So when it comes to the execution of the first line in example above, i.e. `traj.f_add_parameter('traffic.mobiles.add_cars', True , comment='Whether to add some cars or bicycles in the traffic simulation')`, the parameter will be added with the default value `add_cars=True` but immediately afterwards the *f_set()* function will be called with the value `False`. Accordingly, `if traj.add_cars:` will evaluate to `False` and the bicycles will be added.

In order to preset a parameter you need to state its *full name* (except the prefix *parameters*) and you cannot shortcut through the tree. Don't worry about typos, before the running of your simulations it will be checked if all parameters marked for presetting were reached, if not a *PresettingError* will be thrown.

Storing

Storage of the trajectory container and all it's content is not carried out by the trajectory itself but by a service. The service is known to the trajectory and can be changed via the *v_storage_service* property. The standard storage service (and the only one so far, you don't bother write an SQL one? :-) is the *HDF5StorageService*. As a side remark, if you create a trajectory on your own (for loading) with the *Trajectory* class constructor and you pass it a `filename`, the trajectory will create an *HDF5StorageService* operating on that file for you.

You don't have to interact with the service directly, storage can be initiated by several methods of the trajectory and it's groups and subbranches (they format and hand over the request to the service).

The most straightforward way to store everything is to say:

```
>>> traj.f_store()
```

and that's it. In fact, if you use the trajectory in combination with the environment (see *More about the Environment*) you do not need to do this call by yourself at all, this is done by the environment.

If you store a trajectory to disk it's tree structure is also found in the structure of the HDF5 file! In addition, there will be some overview tables summarizing what you stored into the HDF5 file. They can be found under the top-group *overview*, the different tables are listed in the *HDF5 Overview Tables* section. By the way, you can switch the creation of these tables off passing the appropriate arguments to the *Environment* constructor to reduce the size of the final HDF5 file.

There are four different storage modes that can be chosen for `f_store(store_data=2)` and the `store_data` keyword argument (default is `2`).

- *pypet.pypetconstants.STORE_NOTHING*: (0)

  Nothing is stored, basically a no-op.

- *pypet.pypetconstants.STORE_DATA_SKIPPING*: (1)

  A speedy version of the choice below. Data of nodes that have not been stored before are written to disk. Thus, skips all nodes (groups and leaves) that have been stored before, even if they contain new data that has not been stored before.

- *pypet.pypetconstants.STORE_DATA*: (2)

  Stores data of groups and leaves to disk. Note that individual data already found on disk is not overwritten. If leaves or groups contain new data that is not found on disk, the new data is added. Here addition only means creation of new data items like tables and arrays, but data is **not** appended to existing data arrays or tables.

- *pypet.pypetconstants.OVERWRITE_DATA*: (3)

  Stores data of groups and leaves to disk. All data on disk is overwritten with data found in RAM. Be aware that this may yield fragmented HDF5 files. Therefore, use with care. Overwriting data is not recommended as explained below.

Although you can delete or overwrite data you should try to stick to this general scheme: **Whatever is stored to disk is the ground truth and, therefore, should not be changed**.

Why being so strict? Well, first of all, if you do simulations, they are like numerical *scientific experiments*, so you run them, collect your data and keep these results. There is usually no need to modify the first raw data after collecting it. You may analyse it and create novel results from the raw data, but you usually should have no incentive to modify your original raw data. Second of all, HDF5 is bad for modifying data which usually leads to fragmented HDF5 files and does not free memory on your hard drive. So there are already constraints by the file system used (but trust me this is minor compared to the awesome advantages of using HDF5, and as I said, why the heck do you wanna change your results, anyway?).

Again, in case you use your trajectory with or via an *Environment* there is no need to call *f_store()* for data storage, this will always be called at the end of the simulation and at the end of a single run automatically (unless you set `automatic_storing` to `False`). Yet, be aware that if you add any custom data during a single run not under a group or leaf with *run_XXXXXXXX* in their *full name* this data will not be immediately saved after the completion of the run. In fact, in case of multiprocessing this data will be lost if not manually stored.

Storing data individually

Assume you computed a result that is extremely large. So you want to store it to disk, than free the result and forget about it for the rest of your simulation or single run:

```
>>> large_result = traj.results.f_get('large_result')
>>> traj.f_store_item(large_result)
>>> large_result.f_empty()
```

Note that in order to allow storage of single items, you need to have stored the trajectory at least once. If you operate during a single run, this has been done before, if not, simply call `traj.f_store()` once before. If you do not want to store anything but initialise the storage, you can pass the argument `only_init=True`, i.e. `traj.f_store(only_init=True)`.

Moreover, if you call *f_empty()* on a large result, only the reference to the giant data block within the result is deleted. So in order to make the python garbage collector free the memory, you must ensure that you do not have any external reference of your own in your code to the giant data.

To avoid re-opening an closing of the HDF5 file over and over again there is also the possibility to store a list of items via *f_store_items()* or whole subtrees via *f_store_child()* or *f_store()*. Keep in mind that *Links* are always stored non-recursively despite the setting of `recursive` in these functions.

Loading

Sometimes you start your session not running an experiment, but loading an old trajectory. You can use the *load_trajectory()* function or create a new empty trajectory and use the trajectory's *f_load()* function. In both cases you should to pass a `filename` referring to your HDF5 file. Moreover, pass a `name` or an `index` of the trajectory you want to select within the HDF5 file. For the index you can also count backwards, so `-1` would yield the last or newest trajectory in an HDF5 file.

There are two different loading schemes depending on the argument `as_new`

- `as_new=True`

  You load an old trajectory into your current one, and only load everything stored under *parameters* in order to rerun an old experiment. You could hand this loaded trajectory over to an *Environment* and carry out another the simulation again.

- `as_new=False`

  You want to load and old trajectory and analyse results you have obtained. If using the trajectory's *f_load()* method, the current name of the trajectory will be changed to the name of the loaded one.

If you choose the latter load mode, you can specify how the individual subtrees *config*, *parameters*, *derived_parameters*, and *results* are loaded:

- *pypet.pypetconstants.LOAD_NOTHING*: (0)

  Nothing is loaded, just a no-op.

- *pypet.pypetconstants.LOAD_SKELETON*: (1)

  The skeleton is loaded including annotations (See *Annotations*). This means that only empty *parameter* and *result* objects will be created and you can manually load the data into them afterwards. Note that *pypet.annotations.Annotations* do not count as data and they will be loaded because they are assumed to be small.

- *pypet.pypetconstants.LOAD_DATA*: (2)

  The whole data is loaded. Note in case you have non-empty leaves already in your trajectory, these are left untouched.

- *pypet.pypetconstants.OVERWRITE_DATA*: (3)

  As before, but non-empty nodes are emptied and reloaded.

Compared to manual storage, you can also load single items manually via *f_load_item()*. If you load a large result with many entries you might consider loading only parts of it (see *f_load_items()*) In order to load a parameter, result, or group, with *f_load_item()* it must exist in the current trajectory in RAM, if it does not you can always bring your skeleton of your trajectory tree up to date with `f_update_skeleton()`. This will load all items stored to disk and create empty instances. After a simulation is completed, you need to call this function to get the whole trajectory tree containing all new results and derived parameters.

And last but not least, there are also *f_load_child()* or *f_load()* methods in order to load whole subtrees. Keep in mind that links (*Links*) are always loaded non-recursively despite the setting of `recursive` in these functions.

Automatic Loading

The trajectory supports the nice feature to automatically loading data while you access it. Set `traj.v_auto_load=True` and you don't have to care about loading at all during data analysis.

Enabling automatic loading will make *pypet* do two things. If you try to access group nodes or leaf nodes that are currently not in your trajectory on RAM but stored to disk, it will load these with data. Note that in order to automatically load data you cannot use shortcuts! Secondly, if your trajectory comes across an empty leaf node, it will load the data from disk (here shortcuts work again, since only data and not the skeleton has to be loaded).

For instance:

```
# Create the trajectory independent of the environment
traj = Trajectory(filename='./myfile.hdf5')

# We add a result
traj.f_add_result('mygropA.mygroupB.myresult', 42, comment='The answer')

# Now we store our trajectory
traj.f_store()

# We remove all results
traj.f_remove_child('results', recursive=True)

# We turn auto loading on
traj.v_auto_loading = True

# Now we can happily recall the result, since it is loaded while we access it.
# Stating `results` here is important. We removed the results node above, so
# we have to explicitly name it here to reload it, too. There are no shortcuts allowed
# for nodes that have to be loaded on the fly and that did not exist in memory before.
```

```
answer= traj.results.mygroupA,mygroupB.myresult
# And answer will be 42


# Ok next example, now we only remove the data. Since everything is loaded we can shortcut
# through the tree.
traj.f_get('myresult').f_empty()
# Btw we have to use `f_get` here to get the result itself and not the data `42` via fast
# access

# If we now access `myresult` again through the trajectory, it will be automatically loaded.
# Since the result itself is still in RAM but empty, we can shortcut through the tree:
answer = traj.myresult
# And again the answer will be 42
```

Logging and Git Commits during Data Analysis

Automated logging and git commits are often very handy features. Probably you do not want to miss these while you do your data analysis. To enable these in case you simply want to load an old trajectory for data analysis without doing any more single runs, you can again use an *Environment*.

First, load the trajectory with *f_load()*, and pass the loaded trajectory to a new environment. Accordingly, the environment will trigger a git commit (in case you have specified a path to your repository root) and enable logging. You can additionally pass the argument `do_single_runs=False` to your environment if you only load your trajectory for data analysis. Accordingly, no config information like whether you want to use multiprocessing or resume a broken experiment is added to your trajectory. For example:

```
# Create the trajectory independent of the environment
traj = Trajectory(filename='./myfile.hdf5',
                  dynamic_imports=[BrianParameter,
                                                BrianMonitorResult,
                                                BrianResult])

# Load the first trajectory in the file
traj.f_load(index=0, load_parameters=2,
            load_derived_parameters=2, load_results=1,
            load_other_data=1)

# Just pass the trajectory as the first argument to a new environment.
# You can pass the usual arguments for logging and git integration.
env = Environment(traj
                  log_folder='./logs/',
                  git_repository='../gitroot/',
                  do_single_runs=False)

# Here comes your data analysis...
```

Removal of items

If you only want to remove items from RAM (after storing them to disk), you can get rid of whole subbranches via *f_remove_child()*. *f_remove()*.

But usually it is enough to simply free the data and keep empty results by using the *f_empty()* function of a result or parameter. This will leave the actual skeleton of the trajectory untouched.

Although I made it pretty clear that in general what is stored to disk should be set in stone, there are a functions to delete items not only from RAM but also from disk: *f_delete_item()* and *f_delete_items()*. Note that you cannot delete explored parameters.

Merging and Backup

You can backup a trajectory with the function *f_backup()*.

If you have two trajectories that live in the same space you can merge them into one via *f_merge()*. There are a variety of options how to merge them. You can even discard parameter space points that are equal in both trajectories. You can simply add more trials to a given trajectory if both contain a *trial parameter*. This is an integer parameter that simply runs from 0 to N1-1 and 0 to N2-1 with N1 trials in your current and N2 trials in the other trajectory, respectively. After merging the trial parameter in your merged trajectory runs from 0 to N1+N2-1.

Also checkout the example in *Merging of Trajectories*.

Moreover, if you need to merge several trajectories take a look at the faster *f_merge_many()* function.

Single Runs

A single run of your simulation function is identified by it's index and position in your trajectory, you can access this via *v_idx* of your trajectory. As a proper informatics nerd, if you have N runs, than your first run's index is 0 and the last is indexed as N-1! Also each run has a name `run_XXXXXXXX` where *XXXXXXXX* is the index of the run with some leading zeros, like `run_00000007`. You can access the name via the *v_crun* property.

During the execution of individual runs the functionality of your trajectory is reduced:

- You can no longer add data to *config* and *parameters* branch
- You can usually not access the full exploration range of parameters but only the current value that corresponds to the index of the run.
- Some functions like *f_explore()* are no longer supported.

Conceptually one should regard all single runs to be *independent*. As a consequence, you should **not** load data during a particular run that was computed by a previous one. You should **not** manipulate data in the trajectory that was not added during the particular single run. This is **very important**! When it comes to multiprocessing, manipulating data put into the trajectory before the single runs is useless. Because the trajectory is either pickled or the whole memory space of the trajectory is forked by the OS, changing stuff within the trajectory will not be noticed by any other process or even the main script!

1.4.3 Interaction with Trajectories after an Experiment

Iterating over Loaded Data in a Trajectory

The trajectory offers a way to iteratively look into the data you have obtained from several runs. Assume you have computed the value z with *z=traj.x*traj.x* and added z to the trajectory in each run via `traj.f_add_result('z', z)`. Accordingly, you can find a couple of `traj.results.runs.run_XXXXXXXX.z` in your trajectory (where *XXXXXXXX* is the index of a particular run like *00000003*). To access these one after the other it is quite tedious to write `run_XXXXXXXX` each time.

There is a way to tell the trajectory to only consider the subbranches that are associated with a single run and blind out everything else. You can use the function *f_set_crun()* to make the trajectory only consider a particular run (it accepts run indices as well as names). Alternatively, you can set the run idx via changing *v_idx* of your trajectory object.

In order to set everything back to normal call *f_restore_default()* or set `v_idx` to `-1`.

For example, consider your trajectory contains the parameters x and y and both have been explored with $x \in \{1.0, 2.0, 3.0, 4.0\}$ and $y \in \{3.0, 3.0, 4.0, 4.0\}$ and their product is stored as z. The following code snippet will iterate over all four runs and print the result of each run:

```
for run_name in traj.f_get_run_names():
    traj.f_as_run(run_name)
    x=traj.x
```

```
    y=traj.y
    z=traj.z
    print '%s: x=%f, y=%f, z=%f' % (run_name,x,y,z)

# Don't forget to reset your trajectory to the default settings, to release its belief to
# be the last run:
traj.f_restore_default()
```

This will print the following statement:

run_00000000: x=1.000000, y=3.000000, z=3.000000

run_00000001: x=2.000000, y=3.000000, z=6.000000

run_00000002: x=3.000000, y=4.000000, z=12.000000

run_00000003: x=4.000000, y=4.000000, z=16.000000

To see this in action you might want to check out *Merging of Trajectories*.

Looking for Subsets of Parameter Combinations (f_find_idx)

Let's say you already explored the parameter space and gathered some results. The next step would be to post-process and analyse the results. Yet, you are not interested in all results at the moment but only for subsets where the parameters have certain values. You can find the corresponding run indices with the *f_find_idx()* function.

In order to filter for particular settings you need a *lambda* filter function and a list specifying the names of the parameters that you want to filter. You don't know what *lambda* functions are? You might wanna read about it in Dive Into Python.

For instance, let's assume we explored the parameters *'x'* and *'y'* and the cartesian product of $x \in \{1, 2, 3, 4\}$ and $y \in \{6, 7, 8\}$. We want to know the run indices for `x==2` or `y==8`. First we need to formulate a lambda filter function:

```
>>> my_filter_function = lambda x,y: x==2 or y==8
```

Next we can ask the trajectory to return an iterator (in fact it's a generator) over all run indices that fulfil the above named condition:

```
>>> idx_iterator = traj.f_find_idx(['parameters.x', 'parameters.y'], my_filter_function)
```

Note the list `['parameters.x', 'parameters.y']` to tell the trajectory which parameters are associated with the variables in the lambda function. Make sure they are in the same order as in your lambda function.

Now if we print the indexes found by the lambda filter, we get:

```
>>> print [idx for idx in idx_iterator]
[1, 5, 8, 9, 10, 11]
```

To see this in action check out *Using the f_find_idx Function*.

1.4.4 Annotations

Annotations are a small extra feature. Every group node (including your trajectory root node) and every leaf has a property called `v_annotations`. These are other container objects (accessible via natural naming of course), where you can put whatever you want! So you can mark your items in a specific way beyond simple comments:

```
>>> ncars_obj = traj.f_get('ncars')
>>> ncars_obj.v_annotations.my_special_annotation = ['peter','paul','mary']
>>> print ncars_obj.v_annotations.my_special_annotation
['peter','paul','mary']
```

So here you added a list of strings as an annotation called *my_special_annotation*. These annotations map one to one to the attributes of your HDF5 nodes in your final hdf5 file. The high flexibility of annotating your items comes with the downside that storage and retrieval of annotations from the HDF5 file is very slow. Hence, only use short and small annotations. Consider annotations as a neat additional feature, but I don't recommend using the annotations for large machine written stuff or storing large result like data (use the regular result objects to do that).

1.4.5 More on Parameters and Results

Parameters

The parameter container (Base API is found in `BaseParameter`) is used to keep data that is explicitly required as parameters for your simulations. They are the containers of choice for everything in the trajectory stored under *parameters*, *config*, and *derived_parameters*.

Parameter containers follow these two principles:

- A key concept in numerical simulations is **exploration** of the parameter space. Therefore, the parameter containers not only keep a single value but can hold a **range** of values. These values typically reside in the same dimension, i.e. only integers, only strings, only numpy arrays, etc.

  Exploration is initiated via the trajectory, see *Parameter Exploration*. The individual values in the exploration range can be accessed one after the other for distinct simulations. How the exploration range is implemented depends on the parameter.

- The parameter can be **locked**, meaning as soon as the parameter is assigned to hold a specific value and the value has already been used somewhere, it cannot be changed any longer (except after being explicitly unlocked). This prevents the nasty error of having a particular parameter value at the beginning of a simulation but changing it during runtime for whatever reason. This can make your simulations really buggy and impossible to understand by other people. In fact, I ran into this problem during my PhD using someone else's simulations. Thus, debugging took ages. As a consequence, this project was born.

  By definition parameters are fixed values that once used never change. An exception to this rule is solely the *exploration* of the parameter space (see *Parameter Exploration*), but this requires to run a number of distinct simulations anyway.

Values supported by Parameters

Parameters are very restrictive in terms of the data they except. For example, the `Parameter` excepts only:

- python natives (int, str, bool, float, complex),
- numpy natives, arrays and matrices of type np.int8-64, np.uint8-64, np.float32-64, np.complex, np.str
- python homogeneous non-nested tuples

And by *only* I mean they handle exactly these types and nothing else, not even objects that are derived from these data types.

Why so very restrictive? Well, the reason is that we store these values to disk into HDF5 later on. We want to recall them occasionally, and maybe even rerun our experiments. However, as soon as you store data into an HDF5 files, most often information about the exact type is lost. So if you store, for instance, a numpy matrix via PyTables and recall it, you will get a numpy array instead.

The storage service that comes with this package will take care that the exact type of an instance is **NOT** lost. However, this guarantee of type conservations comes with the cost that types are restricted.

However, that does not mean that data which is not supported cannot be used as a parameter at all. You have two possibilities if your data is not supported: First, write your own parameter that converts your data to the basic types supported by the storage service. This is rather easy, the API `BaseParameter` is really small. Or second of all, simply put your data into the `PickleParameter` and it can be stored later on to HDF5 as the pickle string.

As soon as you add data or explore data it will immediately be checked if the data is supported and if not a TypeError is thrown.

Types of Parameters

So far, the following parameters exist:

- *Parameter*

  Container for native python data: int, long, float, str, bool, complex; and Numpy data: np.int8-64, np.uint8-64, np.float32-64, np.complex, np.str. Numpy arrays and matrices are allowed as well.

  However, for larger numpy arrays, the ArrayParameter is recommended, see below.

- *ArrayParameter*

  Container for native python data as well as tuples and numpy arrays and matrices. The array parameter is the method of choice for large numpy arrays or python tuples. Individual arrays are kept only once (and by the HDF5 storage service stored only once to disk). In the exploration range you can find references to these arrays. This is particularly useful if you reuse an array many times in distinct simulation, for example, by exploring the parameter space in form of a cartesian product.

  For instance, assume you explore a numpy array with default value `numpy.array([1,2,3])`. A potential exploration range could be: `[numpy.array([1,2,3]), numpy.array([3,4,3]), numpy.array([1,2,3]), numpy.array([3,4,3])]` So you reuse `numpy.array([1,2,3])` and `numpy.array([3,4,3])` twice. If you would put this data into the standard Parameter, the full list `[numpy.array([1,2,3]), numpy.array([3,4,3]), numpy.array([1,2,3]), numpy.array([3,4,3])` would be stored to disk. The ArrayParameter is smarter. It will ask the storage service only to store `numpy.array([1,2,3])` and `numpy.array([3,4,3])` once and in addition a list of references `[ref_to_array_1, ref_to_array_2, ref_to_array_1, ref_to_array_2]`.

  Subclasses the standard Parameter and, therefore, supports also native python data.

- *SparseParameter*

  Container for Scipy sparse matrices. Supported formats are csr, csc, bsr, and dia. Subclasses the ArrayParameter, and handles memory management similarly.

- *PickleParameter*

  Container for all the data that can be pickled. Like the array parameter, distinct objects are kept only once and are referred to in the exploration range.

Parameters can be changed and values can be requested with the getter and setter methods: *f_get()* and *f_set()*. For convenience `param.data` works as well instead of *f_get()*. Note that `param.v_data` is not valid syntax. The idea is that `.data` works as an extension to the natural naming scheme.

For people using BRIAN quantities, there also exists a *BrianParameter*.

Results

Results are less restrictive in their acceptance of values and they can handle more than a single data item.

They support a constructor and a getter and setter that have positional and keyword arguments. And, of course, results support natural naming as well.

For example:

```
>>> res = Result('supergroup.subgroup.myresult', comment='I am a neat example!')
>>> res.f_set(333, mystring = 'String!', test = 42)
>>> res.f_get('myresult')
333
>>> res.f_get('mystring')
'String!'
>>> res.mystring
'String!'
>>> res.myresult
333
>>> res.test
42
```

If you use `f_set(*args)` the first positional argument is added to the result having the name of the result, here 'myresult'. Subsequent positional arguments are added with 'name_X' where *X* is the position of the argument. Positions are counted starting from zero so `f_set('a','b','c')` will add the entries `'myresult, myresult_1, myresult_2'` to your result.

Using *f_get()* you can request several items at once. If you ask for `f_get(itemname)` you will get in return the item with that name. If you request `f_get(itemname1, itemname2, ....)` you will get a list in return containing the items. To refer to items stored with 'name_X' providing the index value is sufficient:

```
>>> res.f_get(0)
333
```

If your result contains only a single item you can simply call `f_get()` without any arguments. But if you call `f_get()` without any arguments and the result contains more than one item a ValueError is thrown.

```
>>> res = Result('myres', 42, comment='I only contain a single value')
>>> res.f_get()
42
```

Other more pythonic methods of data manipulation are also supported:

```
>>> res.myval = 42
>>> res.myval
42
>>> res['myval'] = 43
>>> res['myval']
43
```

Types of Results

The following results exist:

- *Result*

  Light Container that stores python native data and numpy arrays.

  Note that no sanity checks on individual data is made in case your data is a container. For instance, if you hand over a python list to the result it is not checked if the individual elements of the list are valid data items supported by the storage service. You have to take care that your data is understood by the storage service. It is assumed that results tend to be large and therefore sanity checks would be too expensive.

  Data that can safely be stored into a *Result* are:

  - python natives (int, long, str, bool, float, complex),
  - numpy natives, arrays and matrices of type np.int8-64, np.uint8-64, np.float32-64, np.complex, np.str
  - python lists and tuples

    Non nested with homogeneous data of the previous types.

  - python dictionaries

    Non-nested with strings as keys; values must be of the previously listed types (including numpy arrays and matrices) and can be heterogeneous.

  - pandas DataFrames, Series, Panels

  - *ObjectTable*

    Object tables are special pandas DataFrames with `dtype=object`, i.e. everything you keep in object tables will keep its type and won't be auto-converted py pandas.

- *SparseResult*

  Can handle sparse matrices of type csc, csr, bsr and dia and all data that is handled by the *Result*.

- *PickleResult*

  Result that digest everything and simply pickles it!

  Note that it is not checked whether data can be pickled, so take care that it works!

For those of you using BRIAN, there exists also the *BrianMonitorResult* for monitor data and the *BrianResult* to handle brian quantities.

1.4.6 More about the Environment

Creating an Environment

In most use cases you will interact with the *Environment* to do your numerical simulations. The environment is your handyman for your numerical experiments, it sets up new trajectories, keeps log files and can be used to distribute your simulations onto several CPUs.

You start your simulations by creating an environment object:

```
>>> env = Environment(trajectory='trajectory', comment='A useful comment')
```

You can pass the following arguments. Note usually you only have to change very few of these because most of the time the default settings are sufficient.

- `trajectory`

  The first argument `trajectory` can either be a string or a given trajectory object. In case of a string, a new trajectory with that name is created. You can access the new trajectory via `v_trajectory` property. If a new trajectory is created, the comment and dynamically imported classes are added to the trajectory.

- `add_time`

  Whether the current time in format XXXX_XX_XX_XXhXXmXXs is added to the trajectory name if the trajectory is newly created.

- `comment`

  The comment that will be added to a newly created trajectory.

- `dynamic_imports`

  Only considered if a new trajectory is created.

  The argument `dynamic_imports` is important if you have written your own *parameter* or *result* classes, you can pass these either as class variables `MyCustomParameterClass` or as strings leading to the classes in your package: `'mysim.myparameters.MyCustomParameterClass'`. If you have several classes, just put them in a list `dynamic_imports=[MyCustomParameterClass, MyCustomResultClass]`. In case you want to load a custom class from disk and the trajectory needs to know how they are built.

It is **VERY important**, that every class name is **UNIQUE**. So you should not have two classes named `'MyCustomParameterClass'` in two different python modules! The identification of the class is based only on its name and not its path in your packages.

- `wildcard_functions`

  Dictionary of wildcards like *$* and corresponding functions that are called upon finding such a wildcard. For example, to replace the *$* aka *crun* wildcard, you can pass the following: `wildcard_functions = {('$', 'crun'): myfunc}`.

  Your wildcard function *myfunc* must return a unique run name as a function of a given integer run index. Moreover, your function must also return a unique *dummy* name for the run index being *-1*.

  Of course, you can define your own wildcards like *wildcard_functions = {('$mycard', 'mycard'): myfunc)}. These are not required to return a unique name for each run index, but can be used to group runs into buckets by returning the same name for several run indices. Yet, all wildcard functions need to return a dummy name for the index '-1*.

  You may also want to take a look at *More on Wildcards*.

- `automatic_storing`

  If `True` the trajectory will be stored at the end of the simulation and single runs will be stored after their completion. Be aware of data loss if you set this to `False` and not manually store everything.

- `log_config`

  Can be path to a logging *.ini* file specifying the logging configuration. For an example of such a file see *Logging*. Can also be a dictionary that is accepted by the built-in logging module. Set to *None* if you don't want *pypet* to configure logging.

  If not specified, the default settings are used. Moreover, you can manually tweak the default settings without creating a new *ini* file. Instead of the *log_config* parameter, pass a `log_folder`, a list of *logger_names* and corresponding *log_levels* to fine grain the loggers to which the default settings apply.

  For example:

  ```
  log_folder='logs', logger_names='('pypet', 'MyCustomLogger'),
  log_levels=(logging.ERROR, logging.INFO)
  ```

- `log_stdout`

  Whether the output of `stdout` and `stderr` should be recorded into the log files. Disable if only logging statement should be recorded. Note if you work with an interactive console like *IPython*, it is a good idea to set `log_stdout=False` to avoid messing up the console output.

- `report_progress`

  If progress of runs and an estimate of the remaining time should be shown. Can be *True* or *False* or a triple `(10, 'pypet', logging.Info)` where the first number is the percentage and update step of the resulting progressbar and the second one is a corresponding logger name with which the progress should be logged. If you use *'print'*, the *print* statement is used instead. The third value specifies the logging level (level of logging statement *not* a filter) with which the progress should be logged.

  Note that the progress is based on finished runs. If you use the *QUEUE* wrapping in case of multiprocessing and if storing takes long, the estimate of the remaining time might not be very accurate.

- `multiproc`

  `multiproc` specifies whether or not to use multiprocessing (take a look at *Multiprocessing*). Default is `False`.

- `ncores`

If `multiproc` is `True`, this specifies the number of processes that will be spawned to run your experiment. Note if you use `'QUEUE'` mode (see below) the queue process is not included in this number and will add another extra process for storing. If you have psutil installed, you can set *ncores=0* to let psutil determine the number of CPUs available.

- `use_pool`

  If you choose multiprocessing you can specify whether you want to spawn a new process for every run or if you want a fixed pool of processes to carry out your computation.

  When to use a fixed pool of processes or when to spawn a new process for every run? Use the former if you perform many runs (50k and more) which are inexpensive in terms of memory and runtime. Be aware that everything you use must be picklable. Use the latter for fewer runs (50k and less) and which are longer lasting and more expensive runs (in terms of memory consumption). In case your operating system allows forking, your data does not need to be picklable. If you choose `use_pool=False` you can also make use of the *cap* values, see below.

- `freeze_pool_input`

  Can be set to `True` if the run function as well as all additional arguments are immutable. This will prevent the trajectory from getting pickled again and again. Thus, the run function, the trajectory as well as all arguments are passed to the pool at initialisation.

- `queue_maxsize`

  Maximum size of the Storage Queue, in case of `'QUEUE'` wrapping. `0` means infinite, `-1` (default) means the educated guess of `2 * ncores`.

- `cpu_cap`

  If `multiproc=True` and `use_pool=False` you can specify a maximum CPU utilization between 0.0 (excluded) and 100.0 (included) as fraction of maximum capacity. If the current CPU usage is above the specified level (averaged across all cores), *pypet* will not spawn a new process and wait until activity falls below the threshold again. Note that in order to avoid deadlock at least one process will always be running regardless of the current utilization. If the threshold is crossed a warning will be issued. The warning won't be repeated as long as the threshold remains crossed.

  For example let us assume you chose `cpu_cap=70.0`, `ncores=3`, and currently on average 80 percent of your CPU are used. Moreover, at the moment only 2 processes are computing single runs simultaneously. Due to the usage of 80 percent of your CPU, *pypet* will wait until CPU usage drops below (or equal to) 70 percent again until it starts a third process to carry out another single run.

  The parameters `memory_cap` and `swap_cap` are analogous. These three thresholds are combined to determine whether a new process can be spawned. Accordingly, if only one of these thresholds is crossed, no new processes will be spawned.

  To disable the cap limits simply set all three values to 100.0.

  You need the psutil package to use this cap feature. If not installed and you choose cap values different from 100.0 a ValueError is thrown.

- `memory_cap`

  Cap value of RAM usage. If more RAM than the threshold is currently in use, no new processes are spawned. Can also be a tuple `(limit, memory_per_process)`, first value is the cap value (between 0.0 and 100.0), second one is the estimated memory per process in mega bytes (MB). If an estimate is given a new process is not started if the threshold would be crossed including the estimate.

- `swap_cap`

  Analogous to `cpu_cap` but the swap memory is considered.

- `wrap_mode`

If `multiproc` is `True`, specifies how storage to disk is handled via the storage service. Since PyTables HDF5 is not thread safe, the HDF5 storage service needs to be wrapped with a helper class to allow the interaction with multiple processes.

There are two options:

`pypet.pypetconstants.MULTIPROC_MODE_QUEUE`: ('QUEUE')

Another process for storing the trajectory is spawned. The sub processes running the individual single runs will add their results to a multiprocessing queue that is handled by an additional process.

`pypet.pypetconstants.MULTIPROC_MODE_LOCK`: ('LOCK')

Each individual process takes care about storage by itself. Before carrying out the storage, a lock is placed to prevent the other processes to store data.

If you don't want wrapping at all use `pypet.pypetconstants.MULTIPROC_MODE_NONE` ('NONE').

If you have no clue what I am talking about, you might want to take a look at multiprocessing in python to learn more about locks, queues and thread safety and so forth.

- `clean_up_runs`

  In case of single core processing, whether all results under `results.runs.run_XXXXXXXX` and `derived_parameters.runs.run_XXXXXXXX` should be removed after the completion of the run. Note in case of multiprocessing this happens anyway since the trajectory container will be destroyed after finishing of the process.

  Moreover, if set to `True` after post-processing run data is also cleaned up.

- `immediate_postproc`

  If you use post- and multiprocessing, you can immediately start analysing the data as soon as the trajectory runs out of tasks, i.e. is fully explored but the final runs are not completed. Thus, while executing the last batch of parameter space points, you can already analyse the finished runs. This is especially helpful if you perform some sort of adaptive search within the parameter space.

  The difference to normal post-processing is that you do not have to wait until all single runs are finished, but your analysis already starts while there are still runs being executed. This can be a huge time saver especially if your simulation time differs a lot between individual runs. Accordingly, you don't have to wait for a very long run to finish to start post-processing.

  Note that after the execution of the final run, your post-processing routine will be called again as usual.

- `continuable`

  Whether the environment should take special care to allow to resume or continue crashed trajectories. Default is `False`.

  You need to install dill to use this feature. dill will make snapshots of your simulation function as well as the passed arguments. **Be aware** that dill is still rather experimental!

  Assume you run experiments that take a lot of time. If during your experiments there is a power failure, you can resume your trajectory after the last single run that was still successfully stored via your storage service.

  The environment will create several *.ecnt* and *.rcnt* files in a folder that you specify (see below). Using this data you can continue crashed trajectories.

  In order to resume trajectories use *`f_continue()`*.

  Your individual single runs must be completely independent of one another to allow continuing to work. Thus, they should **not** be based on shared data that is manipulated during runtime (like a multiprocessing manager list) in the positional and keyword arguments passed to the run function.

If you use postprocessing, the expansion of trajectories and continuing of trajectories is *not* supported properly. There is no guarantee that both work together.

- `continue_folder`

  The folder where the continue files will be placed. Note that *pypet* will create a sub-folder with the name of the environment.

- `delete_continue`

  If true, *pypet* will delete the continue files after a successful simulation.

- `storage_service`

  Pass a given storage service or a class constructor (default is *HDF5StorageService*) if you want the environment to create the service for you. The environment will pass additional keyword arguments you provide directly to the constructor. If the trajectory already has a service attached, the one from the trajectory will be used. For the additional keyword arguments, see below.

- `git_repository`

  If your code base is under git version control you can specify the path (relative or absolute) to the folder containing the *.git* directory. See also *Git Integration*.

- `git_message`

  Message passed onto git command.

- `git_fail`

  If *True* the program fails instead of triggering a commit if there are not committed changes found in the code base. In such a case a *GitDiffError* is raised.

- `do_single_runs`

  Whether you intend to actually to compute single runs with the trajectory. If you do not intend to carry out single runs (probably because you loaded an old trajectory for data analysis), than set to `False` and the environment won't add config information like number of processors to the trajectory.

- `lazy_debug`

  If `lazy_debug=True` and in case you debug your code (aka you use *pydevd* and the expression `'pydevd' in sys.modules` is `True`), the environment will use the *LazyStorageService* instead of the HDF5 one. Accordingly, no files are created and your trajectory and results are not saved. This allows faster debugging and prevents *pypet* from blowing up your hard drive with trajectories that you probably not want to use anyway since you just debug your code.

If you use the standard *HDF5StorageService* you can pass the following additional keyword arguments to the environment. These are handed over to the service:

- `filename`

  The name of the hdf5 file. If none is specified, the default *./hdf5/the_name_of_your_trajectory.hdf5* is chosen. If `filename` contains only a path like `filename='./myfolder/'`, it is changed to `filename='./myfolder/the_name_of_your_trajectory.hdf5'`.

- `file_title`

  Title of the hdf5 file (only important if file is created new)

- `overwrite_file`

  If the file already exists it will be overwritten. Otherwise the trajectory will simply be added to the file and already existing trajectories are not deleted.

- `encoding`

  Encoding for unicode characters. The default `'utf8'` is highly recommended.

- `complevel`

  You can specify your compression level. 0 means no compression and 9 is the highest compression level. By default the level is set to 9 to reduce the size of the resulting HDF5 file. See PyTables Compression for a detailed explanation.

- `complib`

  The library used for compression. Choose between *zlib*, *blosc*, and *lzo*. Note that 'blosc' and 'lzo' are usually faster than 'zlib' but it may be the case that you can no longer open your hdf5 files with third-party applications that do not rely on PyTables.

- `shuffle`

  Whether or not to use the shuffle filters in the HDF5 library. This normally improves the compression ratio.

- `fletcher32`

  Whether or not to use the *Fletcher32* filter in the HDF5 library. This is used to add a checksum on hdf5 data.

- `pandas_format`

  How to store pandas data frames. Either in 'fixed' ('f') or 'table' ('t') format. Fixed format allows fast reading and writing but disables querying the hdf5 data and appending to the store (with other 3rd party software other than *pypet*).

- `purge_duplicate_comments`

  If you add a result via *`f_add_result()`* or a derived parameter *`f_add_derived_parameter()`* and you set a comment, normally that comment would be attached to each and every instance. This can produce a lot of unnecessary overhead if the comment is the same for every result over all runs. If `hdf5.purge_duplicate_comments=True` than only the comment of the first result or derived parameter instance created is stored, or comments that differ from this first comment. You might want to take a look at *HDF5 Purging of Duplicate Comments*.

- `summary_tables`

  Whether summary tables should be created. These give overview about 'derived_parameters_runs_summary', and 'results_runs_summary'. They give an example about your results by listing the very first computed result. If you want to `purge_duplicate_comments` you will need the `summary_tables`. You might want to check out *HDF5 Overview Tables*.

- `small_overview_tables`

  Whether the small overview tables should be created. Small tables are giving overview about 'config', 'parameters', 'derived_parameters_trajectory', 'results_trajectory'.

- `large_overview_tables`

  Whether to add large overview tables. These encompass information about every derived parameter and result and the explored parameters in every single run. If you want small HDF5 files set to `False` (default).

- `results_per_run`

  Expected results you store per run. If you give a good/correct estimate, storage to HDF5 file is much faster in case you want `large_overview_tables`.

  Default is 0, i.e. the number of results is not estimated!

- `derived_parameters_per_run`

  Analogous to the above.

Finally, you can also pass properties of the trajectory, like `v_auto_load=True` (you can leave the prefix `v_`, i.e. `auto_load` works, too). Thus, you can change the settings of the trajectory immediately.

Config Data added by the Environment

The Environment will automatically add some config settings to your trajectory. Thus, you can always look up how your trajectory was run. This encompasses many of the above named parameters as well as some information about the environment. This additional information includes a timestamp and a SHA-1 hash code that uniquely identifies your environment. If you use git integration (*Git Integration*), the SHA-1 hash code will be the one from your git commit. Otherwise the code will be calculated from the trajectory name, the current time, and your current *pypet* version.

The environment will be named *environment_XXXXXXX_XXXX_XX_XX_XXhXXmXXs*. The first seven *X* are the first seven characters of the SHA-1 hash code followed by a human readable timestamp.

All information about the environment can be found in your trajectory under `config.environment.environment_XXXXXXX_XXXX_XX_XX_XXhXXmXXs`. Your trajectory could potentially be run by several environments due to merging or extending an existing trajectory. Thus, you will be able to track how your trajectory was built over time.

Logging

pypet comes with a full fledged logging environment.

Per default the environment will created loggers and stores all logged messages to log files. This includes also everything written to the standard stream `stdout`, like `print` statements, for instance. To disable logging of the standard streams set `log_stdout=False`. Note that you should always do this in case you use an interactive console like *IPython*. Otherwise your console output will be garbled.

After your experiments are finished you can disable logging to files via *`f_disable_logging()`*. This also restores the standard stream.

You can tweak the standard logging settings via passing the following arguments to the environment. *log_folder* specifies a folder where all log-files are stored. *logger_names* is a list of logger names to which the standard settings apply. *log_levels* is a list of levels with which the specified loggers should be logged.

```
import logging
from pypet import Environment

env =  Environment(trajectory='mytraj',
                   log_folder = './logs/',
                   logger_nmes = ('pypet', 'MyCustomLogger'),
                   log_levels=(logging.ERROR, logging.INFO),
                   log_stdout=True)
```

Furthermore, if the standard settings don't suite you at all, you can fine grain logging via a logging config file passed via `log_config='/test/ini.'`. This file has to follow the logging configurations of the logging module.

Additionally, if you create file handlers you can use the following wildcards in the filenames which are replaced during runtime:

`LOG_ENV` ($env) is replaces by the name of the trajectory's environment.

`LOG_TRAJ` ($traj) is replaced by the name of the trajectory.

`LOG_RUN` ($run) is replaced by the name of the current run.

`LOG_SET` ($set) is replaced by the name of the current run set.

`LOG_PROC` ($proc) is replaced by the name fo the current process.

Note that in contrast to the standard logging package, *pypet* will automatically create folders for your log-files if these don't exist.

You can further specify settings for multiprocessing logging which will overwrite your current settings within each new process. To specify settings only used for multiprocessing, simply append *multiproc_* to the sections of the *.ini* file.

An example logging *ini* file including multiprocessing is given below.

Download: `default.ini`

```
[loggers]
keys=root

[logger_root]
handlers=file_main,file_error,stream
level=INFO

[formatters]
keys=file,stream

[formatter_file]
format=%(asctime)s %(name)s %(levelname)-8s %(message)s

[formatter_stream]
format=%(processName)-10s %(name)s %(levelname)-8s %(message)s

[handlers]
keys=file_main, file_error, stream

[handler_file_error]
class=FileHandler
level=ERROR
args=('logs/$traj/$env/ERROR.txt',)
formatter=file

[handler_file_main]
class=FileHandler
args=('logs/$traj/$env/LOG.txt',)
formatter=file

[handler_stream]
class=StreamHandler
level=INFO
args=()
formatter=stream


[multiproc_loggers]
keys=root

[multiproc_logger_root]
handlers=file_main,file_error
level=INFO

[multiproc_formatters]
keys=file

[multiproc_formatter_file]
format=%(asctime)s %(name)s %(levelname)-8s %(message)s

[multiproc_handlers]
keys=file_main,file_error

[multiproc_handler_file_error]
class=FileHandler
level=ERROR
args=('logs/$traj/$env/$run_$proc_ERROR.txt',)
formatter=file

[multiproc_handler_file_main]
```

```
class=FileHandler
args=('logs/$traj/$env/$run_$proc_LOG.txt',)
formatter=file
```

Furthermore, an environment can also be used as a context manager such that logging is automatically disabled in the end:

```
import logging
from pypet import Environment

with Environment(trajectory='mytraj',
                 log_config='DEFAULT,
                 log_stdout=True) as env:
    traj = env.v_trajectory

    # do your complex experiment...
```

This is equivalent to:

```
import logging
from pypet import Environment

env = Environment(trajectory='mytraj',
                  log_config='DEFAULT'
                  log_stdout=True)
traj = env.v_trajectory

# do your complex experiment...

env.f_disable_logging()
```

Multiprocessing

For an example on multiprocessing see *Multiprocessing*.

The following code snippet shows how to enable multiprocessing with 4 CPUs, a pool, and a queue.

```
env = Environment(self, trajectory='trajectory',
              comment='',
              dynamic_imports=None,
              log_folder='../log/',
              use_hdf5=True,
              filename='../experiments.h5',
              file_title='experiment',
              multiproc=True,
              ncores=4,
              use_pool=True,
              wrap_mode='QUEUE')
```

Setting `use_pool=True` will create a pool of `ncores` worker processes which perform your simulation runs.

IMPORTANT: Python multiprocessing does not work well with multi-threading of openBLAS. If your simulation relies on openBLAS, you need to make sure that multi-threading is disabled. For disabling set the environment variables `OPENBLAS_NUM_THREADS=1` and `OMP_NUM_THREADS=1` before starting python and using *pypet*. For instance, numpy and matplotlib (!) use openBLAS to solve linear algebra operations. If your simulation relies on these packages, make sure the environment variables are changed appropriately. Otherwise your program might crash or get stuck in an infinite loop.

IMPORTANT: In order to allow multiprocessing with a pool (or in general under **Windows**), all your data and objects of your simulation need to be serialized with pickle. But don't worry, most of the python stuff you use is automatically *picklable*.

If you come across the situation that your data cannot be pickled (which is the case for some BRIAN networks, for example), don't worry either. Set `use_pool=False` (and also `continuable=False`) and for every simulation run *pypet* will spawn an entirely new subprocess. The data is than passed to the subprocess by forking on OS level and not by pickling. However, this only works under **Linux**. If you use **Windows** and choose `use_pool=False` you still need to rely on pickle because **Windows** does not support forking of python processes.

Besides, as a general rule of thumb when to use `use_pool` or don't: Use the former if you perform many runs (50k and more) which are in terms of memory and runtime inexpensive. Use **no** pool (`use_pool=False`) for fewer runs (50k and less) and which are longer lasting and more expensive runs (in terms of memory consumption). In case your operating system allows forking, your data does not need to be picklable. Furthermore, if your trajectory contains many parameters and you want to avoid that your trajectory gets pickled over and over again you can set `freeze_pool_input=True`. The trajectory, the run function as well as the all additional function arguments are passed to the multiprocessing pool at initialization. Be aware that the run function as well as the the additional arguments must be immutable, otherwise your individual runs are no longer independent.

Moreover, if you **enable** multiprocessing and **disable** pool usage, besides the maximum number of utilized processors `ncores`, you can specify usage cap levels with `cpu_cap`, `memory_cap`, and `swap_cap` as fractions of the maximum capacity. Values must be chosen larger than 0.0 and smaller or equal to 100.0. If any of these thresholds is crossed no new processes will be started by *pypet*. For instance, if you want to use 3 cores aka `ncores=3` and set a memory cap of `memory_cap=90.` and let's assume that currently only 2 processes are started with currently 95 percent of you RAM are occupied. Accordingly, *pypet* will not start the third process until RAM usage drops again below (or equal to) 90 percent.

In addition, (only) the `memory_cap` argument can alternatively be a tuple with two entries: `(cap, memory_per_process)`. First entry is the cap value between 0.0 and 100.0 and the second one is the estimated memory per process in mega-bytes (MB). If you specify such an estimate, starting a new process is suspended if the threshold would be reached including the estimated memory.

Moreover, to prevent dead-lock *pypet* will regardless of the cap values always start at least one process. To disable the cap levels, simply set all three to 100.0 (which is default, anyway). *pypet* does not check if the processes themselves obey the cap limit. Thus, if one of the process that computes your single runs needs more RAM/Swap or CPU power than the cap value, this is its very own problem. The process will **not** be terminated by *pypet*. The process will only cause *pypet* to not start new processes until the utilization falls below the threshold again. In order to use this cap feature, you need the psutil package.

Note that HDF5 is not thread safe, so you cannot use the standard HDF5 storage service out of the box. However, if you want multiprocessing, the environment will automatically provide wrapper classes for the HDF5 storage service to allow safe data storage. There are two different modes that are supported. You can choose between them via setting `wrap_mode`. You can select between `'QUEUE'`, `'LOCK'`, and `'PIPE'` wrapping. If you have your own service that is already thread safe you can also choose `'NONE'` to skip wrapping.

If you chose the `'QUEUE'` mode, there will be an additional process spawned that is the only one writing to the HDF5 file. Everything that is supposed to be stored is send over a queue to the process. This has the advantage that your worker processes are only busy with your simulation and are not bothered with writing data to a file. More important, they don't spend time waiting for other processes to release a thread lock to allow file writing. The disadvantages are that you can only store but not load data and storage relies a lot on pickling of data, so often your entire trajectory is send over the queue. Moreover, in case of `'QUEUE'` wrapping you can choose the `queue_maxsize` of elements that can be put on the queue. To few means that your worker processes may need to wait until they can put more data on the queue. To many could blow up your memory in cases the single runs are actually faster than the storage of the data. `0` means a queue of infinite size. Default is `-1` meaning *pypet* makes a conservative estimate of twice te number of processes (i.e. `2 * ncores`). This doesn't sound a lot. However, keep in mind that a single element on the queue might already be quite large like the entire data gathered in a single run.

If you chose the `'LOCK'` mode, every process will place a lock before it opens the HDF5 file for writing data. Thus, only one process at a time stores data. The advantages are the possibility to load data and that your data does not need to be send over a queue over and over again. Yet, your simulations might take longer since processes have to wait often for each other to release locks.

`'PIPE'` wrapping is a rather experimental mode where all processes feed their data into a shared multiprocessing pipe. This can be much faster than a queue. However, no data integrity checks are made. So there's no guarantee

that all you data is really saved. Use this if you go for many runs that just produce small results, and use it carefully.

Finally, there also exist a lightweight multiprocessing environment *MultiprocContext*. It allows to use trajectories in a multiprocess safe setting without the need of a full *Environment*. For instance, you might use this if you also want to analyse the trajectory with multiprocessing. You can find an example here: *Lightweight Multiprocessing*.

Git Integration

The environment can make use of version control. If you manage your code with git, you can trigger automatic commits with the environment to get a proper snapshot of the code you actually use. This ensures that your experiments are repeatable. In order to use the feature of git integration, you additionally need GitPython.

To trigger an automatic commit simply pass the arguments `git_repository` and `git_message` to the *Environment* constructor. `git_repository` specifies the path to the folder containing the *.git* directory. `git_message` is optional and adds the corresponding message to the commit. Note that the message will always be augmented with some short information about the trajectory you are running. The commit SHA-1 hash and some other information about the commit will be added to the config subtree of your trajectory, so you can easily recall that commit from git later on.

The automatic commit functionality will only commit changes in files that are currently tracked by your git repository, it will **not** add new files. So make sure to put new files into your repository before running an experiment. Moreover, a commit will only be triggered if your working copy contains changes. If there are no changes detected, information about the previous commit will be added to the trajectory. By the way, the autocommit function is similar to calling `$ git add -u` and `$ git commit -m 'Some Message'` in your console.

If you want git version control but no automatic commits of your code base in case of changes, you can pass the option *git_fail=True* to the environment. Instead of triggering a new commit in case of changed code, the program will throw a `GitDiffError`.

Sumatra Integration

The environment can make use of a Sumatra experimental lab-book.

Just pass the argument `sumatra_project` - which should specify the path to your root sumatra folder - to the *Environment* constructor. You can additionally pass a `sumatra_reason`, a string describing the reason for you sumatra simulation. *pypet* will automatically add the name, comment, and the names of all explored parameters to the reason. You can also pick a `sumatra_label`, set this to `None` if you want Sumatra to pick a label for you. Moreover, *pypet* automatically adds all parameters to the sumatra record. The explored parameters are added with their full range instead of the default values.

In contrast to the automatic git commits (see above), which are done as soon as the environment is created, a sumatra record is only created and stored if you actually perform single runs. Hence, records are stored if you use one of following three functions: *f_run()*, or *f_pipeline()*, or *f_continue()* and your simulation succeeds and does not crash.

HDF5 Overview Tables

The *HDF5StorageService* creates summarizing information about your trajectory that can be found in the `overview` group within your HDF5 file. These overview tables give you a nice summary about all *parameters* and *results* you needed and computed during your simulations.

The following tables are created depending of your choice of `large_overview_tables` and `small_overview_tables`:

- An *info* table listing general information about your trajectory (needed internally)
- A *runs* table summarizing the single runs (needed internally)

- An *explorations* table listing only the names of explored parameters (needed internally)
- The branch tables:

  parameters_overview

  Containing all parameters, and some information about comments, length etc.

  config_overview,

  As above, but config parameters

  results_overview

  All results to reduce memory size only a short value summary and the name is given. Per default this table is switched off, to enable it pass `large_overview_tables=True` to your environment.

  results_summary

  Only the very first result with a particular **comment** is listed. For instance, if you create the result 'my_result' in all with the comment `'Contains my important data'`. Only the very first result having this comment is put into the summary table.

  If you use this table, you can purge duplicate comments, see *HDF5 Purging of Duplicate Comments*.

  derived_parameters_overview

  derived_parameters_summary

  Both are analogous to the result overviews above

- The *explored_parameters_overview* overview table showing the explored parameter ranges

IMPORTRANT: Be aware that *overview* and *summary* tables are **only** for eye-balling of data. You should **never** rely on data in these tables because it might be truncated or outdated. Moreover, the size of these tables is restricted to 1000 entries. If you add more parameters or results, these are no longer listed in the *overview* tables. Finally, deleting or merging information does not affect the overview tables. Thus, deleted data remains in the table and is not removed. Again, the overview tables are unreliable and their only purpose is to provide a quick glance at your data for eye-balling.

HDF5 Purging of Duplicate Comments

Adding a result with the same comment in every single run, may create a lot of overhead. Since the very same comment would be stored in every node in the HDF5 file. To get rid of this overhead use the option `purge_duplicate_comments=True` and `summary_tables=True`.

For instance, during a single run you call `traj.f_add_result('my_result', 42, comment='Mostly harmless!')` and the result will be renamed to `results.runs.run_00000000.my_result`. After storage of the result into your HDF5 file, you will find the comment `'Mostly harmless!'` in the corresponding HDF5 group node. If you call `traj.f_add_result('my_result',-55, comment='Mostly harmless!')` in another run again, let's say run_00000001, the name will be mapped to `results.runs.run_00000001.my_result`. But this time the comment will not be saved to disk, since `'Mostly harmless!'` is already part of the very first result with the name 'my_result'.

Furthermore, if you reload your data from the example above, the result instance `results.runs.run_00000001.my_result` won't have a comment only the instance `results.runs.run_00000000.my_result`.

IMPORTANT: If you use multiprocessing, the comment of the first result that was stored is used. Since runs are performed synchronously there is no guarantee that the comment of the result with the lowest run index is kept.

IMPORTANT Purging of duplicate comments requires overview tables. Since there are no overview tables for *group* nodes, this feature does not work for comments in *group* nodes. So try to avoid to adding the same comments over and over again in *group* nodes within single runs.

Using a Config File

You are not limited to specify the logging environment within an *.ini* file. You can actually specify all settings of the environment and already add some basic parameters or config data yourself. Simply pass `config='my_config_file.ini` to the environment. If your *.ini* file encompasses logging settings, you don't have to pass another `log_config`.

Anything found in an *environment*, *trajectory* or *storage_service* section is directly passed to the environment constructor. Yet, you can still specify other setting of the environment. Settings passed to the constructor directly take precedence over settings specified in the ini file.

Anything found under *parameters* or *config* is added to the trajectory as parameter or config data.

An example *ini* file including logging can be found below.

Download: `environment_config.ini`

```
######### Environment ##############
[trajectory]
trajectory='ConfigTest'
add_time=True
comment=''
auto_load=True
v_with_links=True
v_lazy_adding=True

[environment]
automatic_storing=True
log_stdout=('STDOUT', 50)
report_progress = (10, 'pypet', 50)
multiproc=True
ncores=2
use_pool=True
cpu_cap=100.0
memory_cap=100.0
swap_cap=100.0
wrap_mode='LOCK'
clean_up_runs=True
immediate_postproc=False
continuable=False
continue_folder=None
delete_continue=True
storage_service='pypet.HDF5StorageService'
do_single_runs=True
lazy_debug=False

[storage_service]
filename='test_overwrite'
file_title=None
overwrite_file=False
encoding='utf-8'
complevel=4
complib='zlib'
shuffle=False
fletcher32=True
pandas_format='t'
purge_duplicate_comments=False
summary_tables=False
small_overview_tables=False
large_overview_tables=True
results_per_run=1000
derived_parameters_per_run=1000
display_time=50
```

```
###### Config and Parameters ######
[config]
test.testconfig=True, 'This is a test config'

[parameters]
test.x=42
y=43, 'This is the second variable'


############ Logging  ###############
[loggers]
keys=root

[logger_root]
handlers=file_main,file_error,stream
level=INFO

[formatters]
keys=file,stream

[formatter_file]
format=%(asctime)s %(name)s %(levelname)-8s %(message)s

[formatter_stream]
format=%(processName)-10s %(name)s %(levelname)-8s %(message)s

[handlers]
keys=file_main, file_error, stream

[handler_file_error]
class=FileHandler
level=ERROR
args=('$temp$traj/$env/ERROR.txt',)
formatter=file

[handler_file_main]
class=FileHandler
args=('$temp$traj/$env/LOG.txt',)
formatter=file

[handler_stream]
class=StreamHandler
level=ERROR
args=()
formatter=stream


[multiproc_loggers]
keys=root

[multiproc_logger_root]
handlers=file_main,file_error
level=INFO

[multiproc_formatters]
keys=file

[multiproc_formatter_file]
format=%(asctime)s %(name)s %(levelname)-8s %(message)s

[multiproc_handlers]
keys=file_main, file_error
```

```
[multiproc_handler_file_error]
class=FileHandler
level=ERROR
args=('$temp$traj/$env/$run_$proc_ERROR.txt',)
formatter=file

[multiproc_handler_file_main]
class=FileHandler
args=('$temp$traj/$env/$run_$proc_LOG.txt',)
formatter=file
```

Example usage:

```
env = Environment(config='path/to/my_config.ini',
                  multiproc = False # This will set multiproc to `False` regardless of the
                  # setting within the `my_config.ini` file.
                  )
```

Running an Experiment

In order to run an experiment, you need to define a job or a top level function that specifies your simulation. This function gets as first positional argument the: *Trajectory* container (see *More on Trajectories*), and optionally other positional and keyword arguments of your choice.

```
def myjobfunc(traj, *args, **kwargs)
    #Do some sophisticated simulations with your trajectory
    ...
    return 'fortytwo'
```

In order to run this simulation, you need to hand over the function to the environment. You can also specify the additional arguments and keyword arguments using *f_run()*:

```
env.f_run(myjobfunc, *args, **kwargs)
```

The argument list `args` and keyword dictionary `kwargs` are directly handed over to the `myjobfunc` during runtime.

The *f_run()* will return a list of tuples. Whereas the first tuple entry is the index of the corresponding run and the second entry of the tuple is the result returned by your run function (for the example above this would simply always be the string `'fortytwo'`, i.e. `((0, 'fortytwo'), (1, 'fortytwo'),...)`). In case you use multiprocessing these tuples are **not** in the order of the run indices but in the order of their finishing time!

Adding Post-Processing

You can add a post-processing function that is called after the execution of all the single runs via *f_add_postprocessing()*.

Your post processing function must accept the trajectory container as the first argument, a list of tuples (containing the run indices and results), and arbitrary positional and keyword arguments. In order to pass arbitrary arguments to your post-processing function, simply pass these first to *f_add_postprocessing()*.

For example:

```
def mypostprocfunc(traj, result_list, extra_arg1, extra_arg2):
    # do some postprocessing here
    ...
```

Whereas in your main script you can call

```
env.f_add_postproc(mypostprocfunc, 42, extra_arg2=42.5)
```

which will later on pass `42` as `extra_arg1` and `42.4` as `extra_arg2`. It is the very same principle as before for your run function. The post-processing function will be called after the completion of all single runs.

Moreover, please note that your trajectory usually does **not** contain the data computed during the single runs, since this has been removed after the single runs to save RAM. If your post-processing needs access to this data, you can simply load it via one of the many loading functions (`f_load_child()`, `f_load_item()`, `f_load()`) or even turn on *Automatic Loading*.

Note that your post-processing function should **not** return any results, since these will simply be lost. However, there is one particular result that can be returned, see below.

Expanding your Trajectory via Post-Processing

If your post-processing function expands the trajectory via `f_expand()` or if your post-processing function returns a dictionary of lists that can be interpreted to expand the trajectory, *pypet* will start the single runs again and explore the expanded trajectory. Of course, after this expanded exploration, your post-processing function will be called again. Likewise, you could potentially expand again, and after the next expansion post-processing will be executed again (and again, and again, and again, I guess you get it). Thus, you can use post-processing for an adaptive search within your parameter space.

IMPORTANT: All changes you apply to your trajectory, like setting auto-loading or changing fast access, are propagated to the new single runs. So try to undo all changes before finishing the post-processing if you plan to trigger new single runs.

Expanding your Trajectory and using Multiprocessing

If you use multiprocessing and you want to adaptively expand your trajectory, it can be a waste of precious time to wait until all runs have finished. Accordingly, you can set the argument `immediate_postproc` to `True` when you create your environment. Then your post-processing function is called as soon as *pypet* runs out of jobs for single runs. Thus, you can expand your trajectory while the last batch of single runs is still being executed.

To emphasize this a bit more and to not be misunderstood: Your post-processing function is **not** called as soon as a single run finishes and the first result is available but as soon as there are **no more** single runs available to start new processes. Still, that does not mean you have to wait until *all* single runs are finished (as for normal post-processing), but you can already add new single runs to the trajectory while the final *n* runs are still being executed. Where *n* is determined by the number of cores (`ncores`) and probably the *cap values* you have chosen (see *Multiprocessing*).

pypet will **not** start a new process for your post-processing. Your post-processing function is executed in the main process (this makes writing actual post-processing functions much easier because you don't have to wrap your head around dead-locks). Accordingly, post-processing should be rather quick in comparison to your single runs, otherwise post-processing will become the bottleneck in your parallel simulations.

Using a Experiment Pipeline

Your numerical experiments usually work like the following: You add some parameters to your trajectory, you mark a few of these for exploration, and you pass your main function to the environment via `f_run()`. Accordingly, this function will be executed with all parameter combinations. Maybe you want some post-processing in the end and that's about it. However, sometimes even the addition of parameters can be fairly complex. Thus, you want this part under the supervision of an environment, too. For instance, because you have a Sumatra lab-book and adding of parameters should also account as runtime. Thus, to have your entire experiment and not only the exploration of the parameter space managed by *pypet* you can use the `f_pipeline()` function, see also *Post-Processing and Pipelining (from the Tutorial)*.

You have to pass a so called *pipeline* function to `f_pipeline()` that defines your entire experiment. Accordingly, your pipeline function is only allowed to take a single parameter, that is the trajectory container. Next, your pipeline function can fill in some parameters and do some pre-processing. Afterwards your pipeline function needs to return the run function, the corresponding arguments and potentially a post-processing function with

arguments. To be more precise your pipeline function needs to return two tuples with at most 3 entries each, for example:

```
def myjobfunc(traj, extra_arg1, extra_arg2, extra_arg3)
    # do some sophisticated simulation stuff
    solve_p_equals_np(traj, extra_arg1)
    disproof_spock(traj, extra_arg2, extra_arg3)
    ...

def mypostproc(traj, postproc_arg1, postproc_arg2, postproc_arg3)
    # do some analysis here
    ...

    exploration_dict={'ncards' : [100, 200]}

    if maybe_i_should_explore_more_cards:
        return exploration_dict
    else
        return None

def mypipeline(traj):
    # add some parameters
    traj.f_add_parameter('poker.ncards', 7, comment='Usually we play 7-card-stud')
    ...
    # Explore the trajectory
    traj.f_explore({'ncards': range(42)})

    # Finally return the tuples
    args = (myarg1, myarg2) # myargX can be anything form ints to strings to complex objects
    kwargs = {'extra_arg3': myarg3}
    postproc_args = (some_other_arg1,) # Check out the comma here! Important to make it a tuple
    postproc_kwargs = {'postproc_arg2' : some_other_arg2,
                       'postproc_arg3' : some_other_arg3}
    return (myjobfunc, args, kwargs), (mypostproc, postproc_args, postproc_kwargs)
```

The first entry of the first tuple is you run or top-level execution function, followed by a list or tuple defining the positional arguments and, thirdly, a dictionary defining the keyword arguments. The second tuple has to contain the post-processing function and positional arguments and keyword arguments. If you do not have any positional arguments pass an empty tuple `()`, if you do not have any keyword arguments pass an empty dictionary `{}`.

If you do not need postprocessing at all, your pipeline function can simply return the run function followed by the positional and keyword arguments:

```
def mypipeline(traj):
    #...
    return myjobfunc, args, kwargs
```

Continuing or Resuming a Crashed Experiment

In order to use this feature you need dill. Careful, dill is rather experimental and still in alpha status!

If all of your data can be handled by dill, you can use the config parameter `continuable=True` passed to the *Environment* constructor. This will create a continue directory (name specified by you via `continue_folder`) and a sub-folder with the name of the trajectory. This folder is your safety net for data loss due to a computer crash. If for whatever reason your day or week-long lasting simulation was interrupted, you can resume it without recomputing already obtained results. Note that this works only if the HDF5 file is not corrupted and for interruptions due to computer crashes, like power failure etc. If your simulations crashed due to errors in your code, there is no way to restore that!

You can resume a crashed trajectory via *f_continue()* with the name of the continue folder (not the subfolder) and the name of the trajectory:

```
env = Environment(continuable=True)

env.f_continue(trajectory_name='my_traj_2015_10_21_04h29m00s',
                          continue_folder='./experiments/continue/')
```

The neat thing here is, that you create a novel environment for the continuation. Accordingly, you can set different environmental settings, like changing the number of cores, etc. You *cannot* change any HDF5 settings or even change the whole storage service.

When does continuing not work?

Continuing will **not** work if your top-level simulation function or the arguments passed to your simulation function are altered between individual runs. For instance, if you use multiprocessing and you want to write computed data into a shared data list (like `multiprocessing.Manager().list()`, see *Sharing Data during Multiprocessing*), these changes will be lost and cannot be captured by the continue snapshots.

A work around here would be to not manipulate the arguments but pass these values as results of your top-level simulation function. Everything that is returned by your top-level function will be part of the snapshots and can be reconstructed after a crash.

Continuing *might not* work if you use post-processing that expands the trajectory. Since you are not limited in how you manipulate the trajectory within your post-processing, there are potentially many side effects that remain undetected by the continue snapshots. You can try to use both together, but there is **no** guarantee whatsoever that continuing a crashed trajectory and post-processing with expanding will work together.

1.4.7 Using BRIAN with pypet

IMPORTANT

Although the general *pypet* API is supposed to remain stable, this promise excludes the BRIAN part. The *pypet* BRIAN subpackage is still considered to be **alpha**. I probably won't change too much within the `pypet.brian.parameter` module, but expect the `pypet.brian.network` module to undergo many changes and updates. Furthermore, the *pypet.brian* package has only been used with BRIAN **1.X**, no guarantee for BRIAN 2.

pypet and BRIAN

BRIAN as it comes is nice for small scripts and quick simulations, but it can be really hard to manage and maintain large scale projects based on very complicated networks with many parts and components. So I wrote a *pypet* extension that allows easier handling of more sophisticated BRIAN networks.

All of this can be found in `pypet.brian` sub-package. The package contains a `parameter.py` file that includes specialized containers for BRIAN data, like the *`BrianParameter`*, the *`BrianResult`* (both for BRIAN Quantities), and the *`BrianMonitorResult`* (extracts data from any kind of BRIAN Monitor).

These can be used in conjunction with the network management system in the `network.py` file within the `pypet.brian` package.

In the following I want to explain how to use the `network.py` framework to run large scale simulations. An example of such a large scale simulation can be found in *Large scale BRIAN simulation* which is an implementation of the Litwin-Kumar and Doiron paper from 2012.

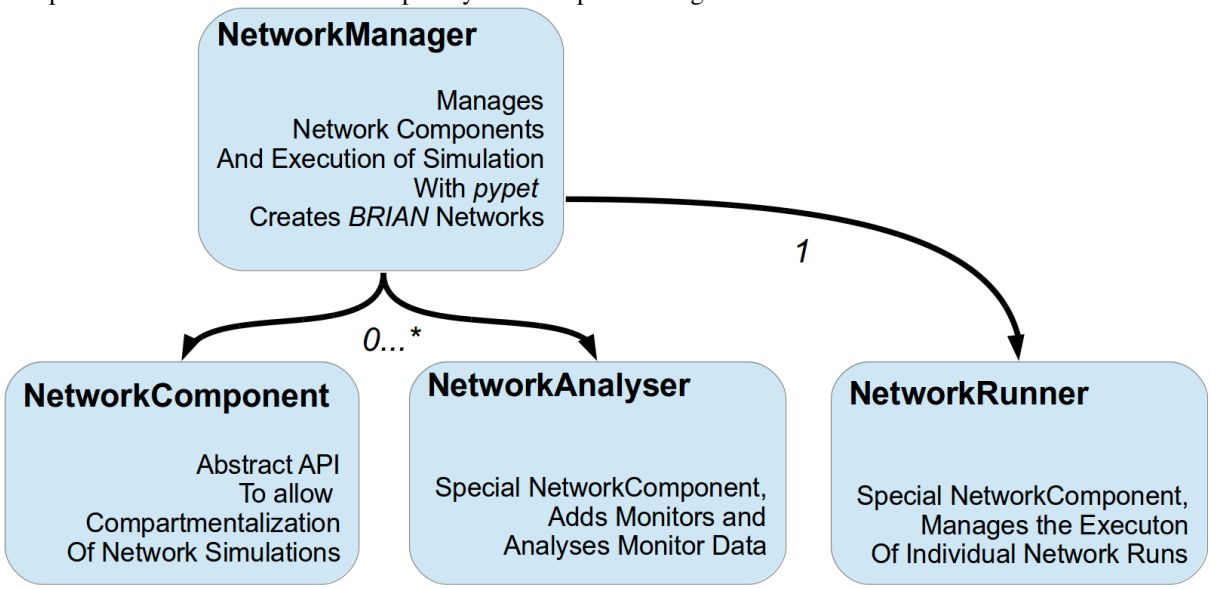
The BRIAN network framework

The core idea behind my framework is that simulated spiking neural network are not in one giant piece but compartmentalize. Networks consist of NeuronGroups, Connections or Synapses, Monitors and so on and so forth. Thus, it would be neat if these parts can be easily replaced or augmented without rewriting a whole simulation. You want to add STDP to your network? Just plug-in an STDP component. You do not want to record anymore from the inhibitory neurons? Just throw away a recording component.

To abstract this idea, the whole simulation framework evolves around the *NetworkComponent* class. This specifies an abstract API that any component (which you as a user implement) should agree on to make them easy to replace and communicate with each other.

There are two specialisation of this *NetworkComponent* API: The *NetworkAnalyser* and the *NetworkRunner*. Implementations of the former deal with the analysis of network output. This might range from simply adding and removing Monitors to evaluating the monitor data and computing statistics about the network activity. An instance of the latter is usually only created once and takes care about the running of a network simulation.

All these three types of components are managed by the *NetworkManager* that also creates BRIAN networks and passes these to the runner. Conceptually this is depicted in figure below.



Main Script

In your main script that you use to create an environment and start the parameter exploration, you also need to include these following steps.

- **Create a *NetworkRunner* and your** *NetworkComponent* instances and *NetworkAnalyser* instances defining the layout and structure of your network and simulation.

  What components are and how to implement these will be discussed in the next section.

- Create a *NetworkManager*:

  Pass your *NetworkRunner* (as first argument *network_runner*), all your *NetworkComponent* instances as a list (as second argument `component_list`) and all *NetworkAnalyser* instances (as third argument `analyser_list`) to the constructor of the manager.

  Be aware that the order of components and analysers matter. The building of components, addition, removal, and analysis (for analyser) is executed in the order they are passed in the `component_list` and `analyser_list`, respectively. If a component *B* depends on *A* and *C*, make *B* appear after *A* and *C* in the list.

  For instance, you have an excitatory neuron group, an inhibitory one, and a connection between the two. Accordingly, your *NetworkComponent* creating the connection must be listed after the components responsible for creating the neuron groups.

  For now on let's call the network manager instance `my_manager`.

- Call `my_manager.add_parameters(traj)`:

  This automatically calls `add_parameters(traj)` for all components, all analysers and the runner. So that they can add all their necessary parameters to the the trajectory `traj`.

- *(Optionally)* call `my_manager.pre_build(traj)`:

  This will automatically trigger the `pre_build` function of your components, analysers and the network runner.

  This is useful if you have some components that do not change during parameter exploration, but which are costly to create and can be so in advance.

  For example, you might have different neuron layers in your network and parts of the network do not change during the runtime of your simulation. For instance, your connections from an LGN neuron group to a V1 neuron group is fixed. Yet, the computation of the connection pattern is costly, so you can do this in `pre_build` to save some time instead of building these over and over again in every single run.

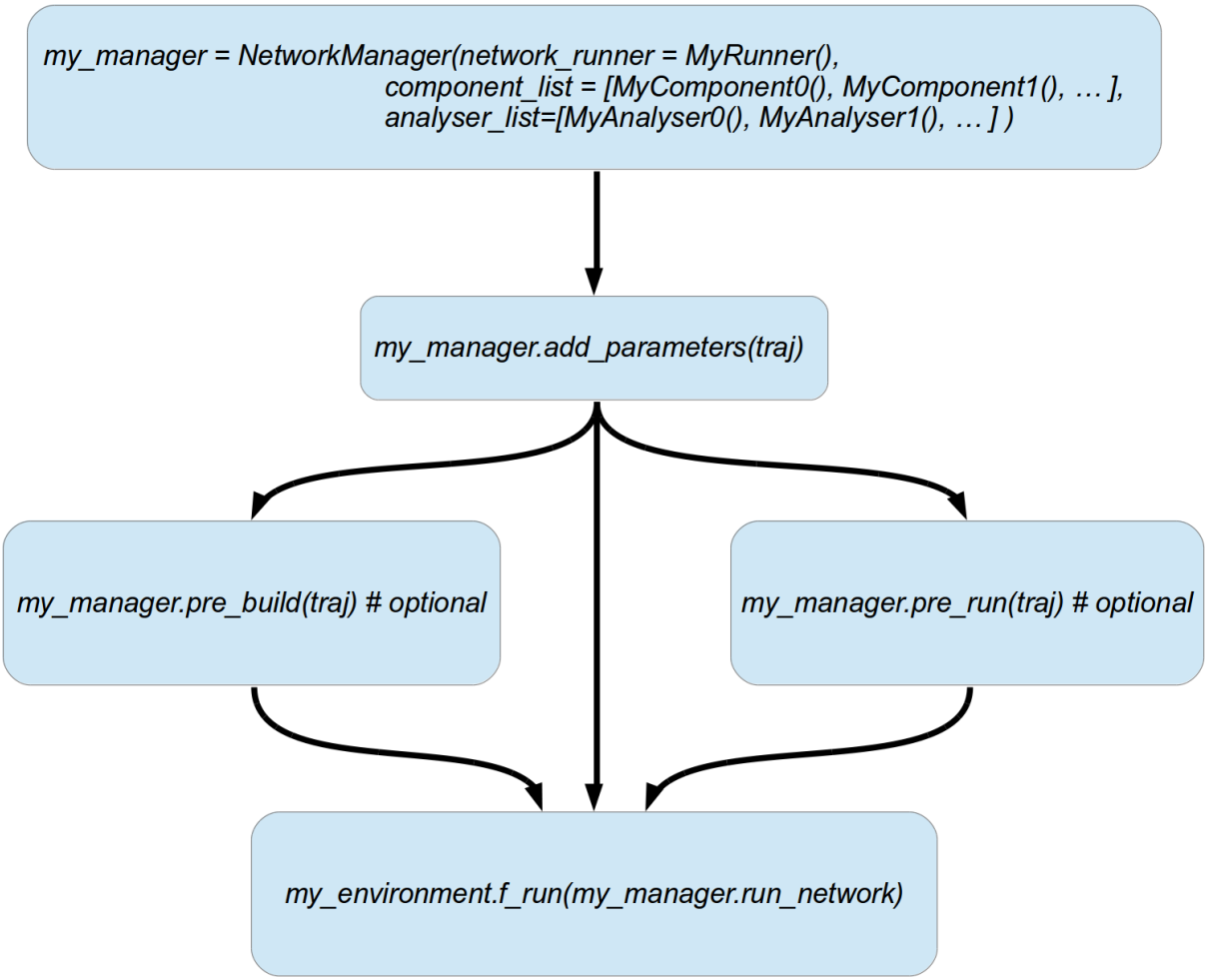
- *(Optionally)* call `my_manager.pre_run_network(traj)`

  This will trigger a *pre run* of the network. First `my_manager.pre_build` is called (so you do not have to call it yourself if you intend a *pre run*). Then a novel BRIAN network instance is created from the `brian_list` (see below). This network is simulated by your runner. The state after the *pre run* is preserved for all coming simulation runs during parameter exploration.

  This is useful if your parameter exploration does not involve modifications of the network per se. For instance, you explore different input stimuli which are tested on the very same network. Moreover, you have the very same initialisation run for every stimulus experiment. Instead of re-simulating the init run over and over again for every stimulus, you can perform it once as a *pre run* and use the network after the *pre run* for every stimulus input.

- Pass the `run_network()` to your environment's `f_run()` to start parameter exploration. This will automatically initiate the `build(traj)` method for all your components, analysers and your runner in every single run. Subsequently, your network will be simulated with he help of your network runner.

These steps are also depicted in the figure below.



An example *main script* might look like the following:

```
env = Environment(trajectory='Clustered_Network',
                  filename='experiments/example_11/HDF5/',
                  log_folder='experiments/example_11/LOGS/',
                  continuable=False,
                  multiproc=True,
                  ncores=2,
                  use_pool=False)

#Get the trajectory container
traj = env.v_trajectory

# We create a Manager and pass all our components to the Manager.
# Note the order, MyNeuronGroupsComponent are scheduled before MyConnectionsComponent,
# and the Fano Factor computation depends on the MyMonitorAnalysisComponent
my_manager = NetworkManager(network_runner=MyNetworkRunner(),
                    component_list=(MyNeuronGroupsComponent(), MyConnectionsComponent()),
                    analyser_list=(MyMonitorAnalysisComponent(), MyFanoFactorComputingComponent()))

# Add parameters
my_manager.add_parameters(traj)

# Explore different values of a parameter
explore_list = np.arange(0.0, 42.0, 0.5).tolist()
traj.f_explore({'some.random.parameter.of.my.network' : explore_list})

# Pre-build network components
my_manager.pre_build(traj)
```

```
# Run the network simulation
env.f_run(my_manager.run_network)
```

Multiprocessing and Iterative Processing

The framework is especially designed to allow for multiprocessing and to distribute parameter exploration of network simulations onto several cpus. Even if parts of your network cannot be pickled, multiprocessing can be easily achieved by setting `use_pool=False` for your *Environment*.

Yet, single core processing is more subtle. In fact if you want to `pre_build` parts of your network or even *pre run* a whole network, you can no longer use iterative computation of the single runs of your parameter exploration. The reason for this lies in the deep inner parts of BRIAN. The problem is that BRIAN networks are not well encapsulated objects, but are strongly dependent on the whole BRIAN runtime environment. As a consequence, you cannot take *snapshots* of a network in order to *rerun* a given network. In case of parameter exploration, a BRIAN network changes after each single run. The starting condition of the second run are the network state after (!) the first run and not before the first run. The only solution to this problem is to not only copy the BRIAN network but also the whole BRIAN runtime environment. The straightforward way to do this is simply to fork a new process. This is the reason why you cannot run single core processing on *pre-built* networks.

If you want to come close to single core processing use `multiproc=True` and `ncores=1` with your environment. If you really do not care about messed up initial conditions - maybe since you just debug your code - you can enforce true single core processing by passing `force_single_core=True` when you create your *NetworkManager*.

Next, I'll go a bit more into detail about components and finally you will learn which steps are involved in a network simulation.

Network Components

Network components are the basic building blocks of a *pypet* BRIAN experiment. There exist three types:

1. Ordinary *NetworkComponent*
2. *NetworkAnalyser* for data analysis and recording
3. *NetworkRunner* for simulation execution.

And these are written by YOU (eventually except for the network runner). The classes above are only abstract and define the API that can be implemented to make *pypet*'s BRIAN framework do its job.

By subclassing these, you define components that build and create BRIAN objects. For example, you could have your own *ExcNeuronGroupComponent* that creates a NeuronGroup of excitatory neurons. Your *ExcNeuronSynapsesComponent* creates BRIAN Synapses to make recurrent connections within the excitatory neuron group. These brian objects (NeuronGroup and Synapses) are then taken by the network manager to construct a BRIAN network.

Every component can implement these 5 methods:

- *add_parameters()*:

  This function should only add parameters necessary for your component to your trajectory `traj`.

- *pre_build()* and/or *build()*

  Both are very similar and should trigger the construction of objects relevant to BRIAN like NeuronGroups or Connections. However, they differ in when they are executed. The former is initiated either by you directly (aka `my_manger.pre_build(traj)`), or by a *pre run* (`my_manager.pre_run_network(traj)`). The latter is called during your single runs for parameter exploration, before the BRIAN network is simulated by your runner.

  The two methods provide the following arguments:

  - `traj`

Trajectory container, you can gather all parameters you need from here.

- `brian_list`

  A non-nested (!) list of objects relevant to BRIAN.

  Your component has to add BRIAN objects to this list if these objects should be added to the BRIAN network at network creation. Your manager will create a BRIAN network via `Network(*brian_list)`.

- `network_dict`

  Add any item to this dictionary that should be shared or accessed by all your components and which are not part of the trajectory container. It is recommended to also put all items from the `brian_list` into the dictionary for completeness.

For convenience I suggest documenting the implementation of `build` and `pre-build` and the other component methods in your subclass like the following. Use statements like *Adds* for items that are added to the list and dictionary and *Expects* for what is needed to be part of the `network_dict` in order to build the current component.

For instance:

brian_list:

Adds:

4 Connections, between all types of neurons (e->e, e->i, i->e, i->i)

network_dict:

Expects:

'neurons_i': Inhibitory neuron group

'neurons_e': Excitatory neuron group

Adds:

'connections' [List of 4 Connections,] between all types of neurons (e->e, e->i, i->e, i->i)

- *`add_to_network()`*:

  This method is called shortly before a *subrun* of your simulation (see below).

  Maybe you did not want to add a BRIAN object directly to the `network` on its creation, but sometime later. Here you have the chance to do that.

  For instance, you have a SpikeMonitor that should not record the initial first *subrun* but the second one. Accordingly, you did not pass it to the `brian_list` in *`pre_build()`* or *`build()`*. You can now add your monitor to the `network` via its `add` functionality, see the the BRIAN network class.

  The *`add_to_network()`* relies on the following arguments

  - `traj`

    Trajectoy container

  - `network`

    BRIAN network created by your manager. Elements can be added via *add(...)*.

  - `current_subrun`

    `BrianParameter` specifying the very next *subrun* to be simulated. See next section for *subruns*.

  - `subrun_list`

    List of *`BrianParameter`* objects that are to be simulated after the current *subrun*.

- `network_dict`

  Dictionary of items shared by all components.

- *`remove_from_network()`*

  This method is analogous to *`add_to_network()`*. It is called after a *subrun* (and after analysis, see below), and gives you the chance to remove items from a network.

  For instance, you might want to remove a particular BRIAN Monitor to skip recording of coming *subruns*.

Be aware that these functions **can** be implemented, but they do not have to be. If your custom component misses one of these, there is **no** error thrown. Instead, simply *pass* is executed (see the source code!).

NetworkAnalyser

The *`NetworkAnalyser`* is a subclass of an ordinary component. It augments the component API by the function *`analyse()`*. The very same parameters as for *`add_to_network()`* are passed to the analyse function. As the name suggests, you can run some analysis here. This might involve extracting data from monitors or computing statistics like Fano Factors, etc.

NetworkRunner

The *`NetworkRunner`* is another subclass of an ordinary component. The given *`NetworkRunner`* does not define an API but provides functionality to execute a network experiment. There's no need for creating your own subclass. Yet, I still suggest subclassing the *`NetworkRunner`*, but just implement the *`add_parameters()`* method. There you can add *`BrianParameter`* instances to your trajectory to define how long a network simulation lasts and in how many *subruns* it is divided.

A Simulation Run and Subruns

A single run of a network simulation is further subdivided into so called *subruns*. This holds for a *pre run* triggered by `my_manager.pre_run_network(traj)` as well as an actual single run during parameter exploration.

The subdivision of a single run into further *subruns* is necessary to allow having different phases of a simulation. For instance, you might want to run your network for an initial phase (subrun) of 500 milliseconds. Then one of your analyser components checks for pathological activity like too high firing rates. If this activity is detected, you cancel all further subruns and skip the rest of the single run. You can do this by simply removing all *subruns* from the `subrun_list`. You could also add further *`BrianParameter`* instances to the list to make your simulations last longer.

The `subrun_list` (as it is passed to *`add_to_network()`*, *`remove_from_network()`*, or *`analyse()`*) is populated by your network runner at the beginning of every single run (or *pre-run*) in your parameter exploration. The network runner searches for *`BrianParameter`* instances in a specific group in your trajectory. By default this group is *traj.parameters.simulation.durations* (or *traj.parameters.simulation.pre_durations* for a *pre-run*), but you can pick another group name when you create a *`NetworkRunner`* instance. The order of the subruns is inferred from the `v_annotations.order` attribute of the *`BrianParameter`* instances. The subruns are executed in increasing order. The orders do not need to be consecutive, but a RuntimeError is thrown in case two subruns have the same order. There is also an Error raised if there exists a parameter where `order` cannot be found in it's `v_annotations` property.

In previous versions of *pypet.brian* there was a so called `BrianDurationParameter` that possessed a special attribute `v_order`. This was basically a normal *`BrianParameter`* with a little bit of overhead. Thus, the `BrianDurationParameter` became a victim of refactoring. There is still an implementation left for backwards-compatibility. Please, do *NOT* use the old `BrianDurationParameter`, but a normal *`BrianParameter`* and replace calls to `v_order` with `v_annotations.order`.

For instance, in `traj.parameter.simulation.durations` there are three *`BrianParameter`* instances.

```
>>> init_run = traj.parameter.simulation.durations.f_add_parameter('init_run', 500 * ms)
>>> init_run.v_annotations.order=0
>>> third_run = traj.parameter.simulation.durations.f_add_parameter('third_run', 1.25 * second)
>>> third_run.v_annotations.order=42
>>> measurement_run = traj.parameter.simulation.durations.f_add_parameter('measurement_run', 15 *
>>> measurement_run.v_annotations.order=1
```

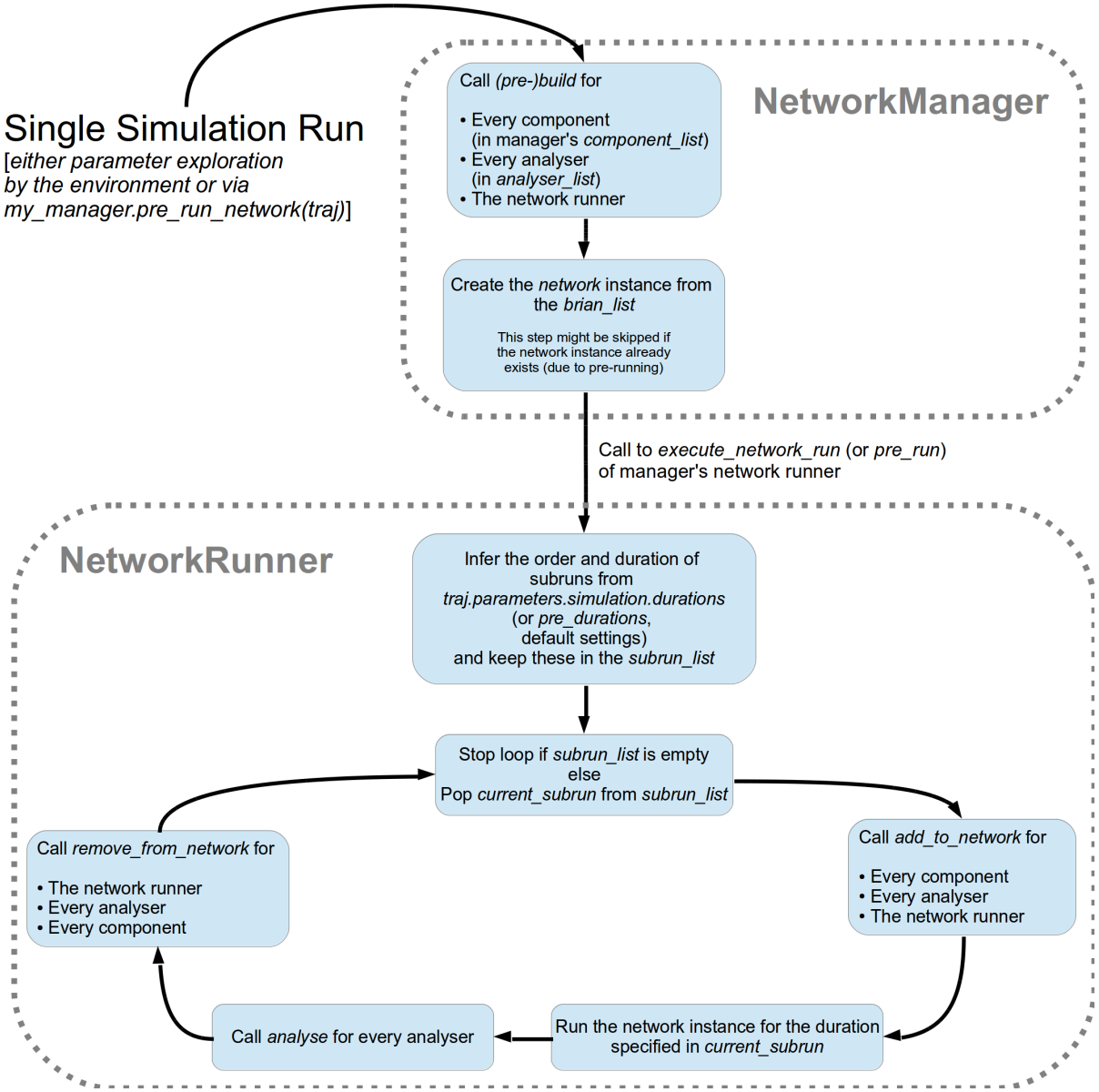
One is called *init_run*, has `v_annotations.order=0` and lasts 500 milliseconds (this is not cpu runtime but BRIAN simulation time). Another one is called *third_run* lasts 1.25 seconds and has order 42. The third one is named *measurement_run* lasts 5 seconds and has order 1. Thus, a single run involves three *subruns*. They are executed in the order: *init_run* involving running the network for 0.5 seconds, *measurement_run* for 5 seconds, and finally *third_run* for 1.25 seconds, because 0 < 1 < 42.

The `current_subrun` *BrianParameter* is taken from the `subrun_list`. In every subrun the *NetworkRunner* will call

1. *add_to_network()*
   - for all ordinary components
   - for all analysers
   - for the network runner itself
2. `run(duration)` from the BRIAN network created by the manager.

   Where the `duration` is simply the data handled by the `current_subrun` which is a *BrianParameter*.
3. *analyse()* for all analysers
4. *remove_from_network()*
   - for the network runner itself
   - for all analysers
   - for all ordinary components

The workflow of network simulation run is also depicted in the figure below.



I recommend taking a look at the source code in the `pypet.brian.network` python file for a better understanding how the *pypet* BRIAN framework can be used. Especially, check the `_execute_network_run()` method that performs the steps mentioned above.

Finally, despite the risk to repeat myself too much, there is an example on how to use *pypet* with BRIAN based on the paper by Litwin-Kumar and Doiron paper from 2012, see *Large scale BRIAN simulation*.

Cheers,

Robert

1.5 Examples

Here you can find some example code how to use the *pypet*. All examples were written and tested with python 2.7 and most of them also work under python 3.

1.5.1 Basic Concepts

First Steps

Download: `example_01_first_steps.py`

This is a basic overview about the usage of the tool, nothing fancy.

```
__author__ = 'Robert Meyer'

import os # To allow file paths working under Windows and Linux

from pypet import Environment
from pypet.utils.explore import cartesian_product

def multiply(traj):
    """Example of a sophisticated simulation that involves multiplying two values.

    :param traj:

        Trajectory containing
        the parameters in a particular combination,
        it also serves as a container for results.

    """
    z = traj.x * traj.y
    traj.f_add_result('z', z, comment='Result of our simulation!')


# Create an environment that handles running
filename = os.path.join('hdf5','example_01.hdf5')
env = Environment(trajectory='Multiplication',
                  filename=filename,
                  file_title='Example_01_First_Steps',
                  comment='The first example!',
                  large_overview_tables=True,  # To see a nice overview of all
                  # computed `z` values in the resulting HDF5 file.
                  # Per default disabled for more compact HDF5 files.
                  )

# The environment has created a trajectory container for us
traj = env.v_trajectory

# Add both parameters
traj.f_add_parameter('x', 1, comment='I am the first dimension!')
traj.f_add_parameter('y', 1, comment='I am the second dimension!')

# Explore the parameters with a cartesian product
traj.f_explore(cartesian_product({'x':[1,2,3,4], 'y':[6,7,8]}))

# Run the simulation
env.f_run(multiply)



# Now let's see how we can reload the stored data from above.
# We do not need an environment for that, just a trajectory.
from pypet.trajectory import Trajectory

# So, first let's create a new trajectory and pass it the path and name of the HDF5 file.
# Yet, to be very clear let's delete all the old stuff.
del traj
# Before deleting the environment let's disable logging and close all log-files
```

```
env.f_disable_logging()
del env

traj = Trajectory(filename=filename)

# Now we want to load all stored data.
traj.f_load(index=-1, load_parameters=2, load_results=2)

# Above `index` specifies that we want to load the trajectory with that particular index
# within the HDF5 file. We could instead also specify a `name`.
# Counting works also backwards, so `-1` yields the last or newest trajectory in the file.
#
# Next we need to specify how the data is loaded.
# Therefore, we have to set the keyword arguments `load_parameters` and `load_results`,
# here we chose both to be `2`.
# `0` would mean we do not want to load anything at all.
# `1` would mean we only want to load the empty hulls or skeletons of our parameters
# or results. Accordingly, we would add parameters or results to our trajectory
# but they would not contain any data.
# Instead `2` means we want to load the parameters and results including the data they contain.

# Finally we want to print a result of a particular run.
# Let's take the second run named `run_00000001` (Note that counting starts at 0!).
print('The result of `run_00000001` is: ')
print(traj.run_00000001.z)
```

Natural Naming, Storage and Loading

Download: `example_02_trajectory_access_and_storage.py`

The following code snippet shows how natural naming works, and how you can store and load a trajectory.

```
__author__ = 'Robert Meyer'

import os # To allow pathnames under Windows and Linux

from pypet import Trajectory, NotUniqueNodeError


# We first generate a new Trajectory
filename = os.path.join('hdf5', 'example_02.hdf5')
traj = Trajectory('Example', filename=filename,
                  comment='Access and Storage!')


# We add our first parameter with the data 'Harrison Ford'
traj.f_add_parameter('starwars.characters.han_solo', 'Harrison Ford')

# This automatically added the groups 'starwars' and the subgroup 'characters'
# Let's get the characters subgroup
characters = traj.parameters.starwars.characters

# Since characters is unique we could also use shortcuts
characters = traj.characters

# Or the get method
characters = traj.f_get('characters')

# Or square brackets
characters = traj['characters']

# Lets add another character
```

```
characters.f_add_parameter('luke_skywalker', 'Mark Hamill', comment='May the force be with you!')

#The full name of luke skywalker is now `parameters.starwars.characters.luke_skywalker`:
print('The full name of the new Skywalker Parameter is %s' %
      traj.f_get('luke_skywalker').v_full_name)

#Lets see what happens if we have not unique entries:
traj.f_add_parameter_group('spaceballs.characters')

# Now our shortcuts no longer work, since we have two character groups!
try:
    traj.characters
except NotUniqueNodeError as e:
    print('Damn it, there are two characters groups in the trajectory: %s' % e._msg)

# But if we are more specific we have again a unique finding
characters = traj.starwars.characters

# Now let's see what fast access is:
print('The name of the actor playing Luke is %s.' % traj.luke_skywalker)

# And now what happens if you forbid it
traj.v_fast_access=False
print('The object found for luke_skywalker is `%s`.' % str(traj.luke_skywalker))

#Let's store the trajectory:
traj.f_store()

# That was easy, let's assume we already completed a simulation and now we add a veeeery large
# result that we want to store to disk immediately and than empty it
traj.f_add_result('starwars.gross_income_of_film', amount=10.1 ** 11, currency='$$$',
                  comment='George Lucas is rich, dude!')
# This is a large number, we better store it and than free the memory:
traj.f_store_item('gross_income_of_film')
traj.gross_income_of_film.f_empty()


# Now lets reload the trajectory
del traj
traj = Trajectory(filename=filename)
# We want to load the last trajectory in the file, therefore index = -1
# We want to load the parameters, therefore load_parameters=2
# We only want to load the skeleton of the results, so load_results=1
traj.f_load(index=-1, load_parameters=2, load_results=1)

# Let's check if our result is really empty
if traj.gross_income_of_film.f_is_empty():
    print('Nothing there!')
else:
    print('I found something!')

# Ok, let's manually reload the result
traj.f_load_item('gross_income_of_film')
if traj.gross_income_of_film.f_is_empty():
    print('Still empty :-(')
else:
    print('George Lucas earned %s%s!' %(str(traj.gross_income_of_film.amount),
                                        traj.gross_income_of_film.currency))

# And that's how it works! If you wish, you can inspect the
# experiments/example_02/HDF5/example_02.hdf5 file to take a look at the tree structure
```

Using Links

Download: `example_14_links.py`

You can also link between different nodes of your *Trajectory*:

```
__author__ = 'Robert Meyer'

import os # To allow file paths working under Windows and Linux

from pypet import Environment, Result

def multiply(traj):
    """Example of a sophisticated simulation that involves multiplying two values.

    :param traj:

        Trajectory containing
        the parameters in a particular combination,
        it also serves as a container for results.

    """
    z=traj.mylink1*traj.mylink2 # And again we now can also use the different names
    # due to the creation of links
    traj.res = Result('runs.$.z', z, 'Result of our simulation!')


# Create an environment that handles running
filename = os.path.join('hdf5','example_14.hdf5')
env = Environment(trajectory='Multiplication',
                  filename=filename,
                  file_title='Example_14_Links',
                  comment='How to use links')

# The environment has created a trajectory container for us
traj = env.v_trajectory

# Add both parameters
traj.v_lazy_adding = True
traj.par.x = 1, 'I am the first dimension!'
traj.par.y = 1, 'I am the second dimension!'

# Explore just two points
traj.f_explore({'x': [3, 4]})

# So far everything was as in the first example. However now we add links:
traj.f_add_link('mylink1', traj.f_get('x'))
# Note the `f_get` here to ensure to get the parameter instance, not the value 1
# This allows us now to access x differently:
print('x=' + str(traj.mylink1))
# We can try to avoid fast access as well, and recover the original parameter
print(str(traj.f_get('mylink1')))
# And also colon notation is allowed that creates new groups on the fly
traj.f_add_link('parameters.mynewgroup.mylink2', traj.f_get('y'))



# And, of course, we can also use the links during run:
env.f_run(multiply)

# Finally disable logging and close all log-files
env.f_disable_logging()
```

Adding Data to the Trajectory

Download: `example_15_more_ways_to_add_data.py`

Here are the different ways to add data to your *Trajectory* container:

```
__author__ = 'Robert Meyer'

from pypet import Trajectory, Result, Parameter


traj = Trajectory()

# There are more ways to add data,
# 1st the standard way:
traj.f_add_parameter('x', 1, comment='I am the first dimension!')
# 2nd by providing a new parameter/result instance, be aware that the data is added where
# you specify it. There are no such things as shortcuts for parameter creation:
traj.parameters.y = Parameter('y', 1, comment='I am the second dimension!')
# 3rd as before, but if our new leaf has NO name it will be renamed accordingly:
traj.parameters.t = Parameter('', 1, comment='Third dimension')
# See:
print('t=' + str(traj.t))

# What happens if our new parameter's name does not match the name passed to the constructor?
traj.parameters.subgroup = Parameter('v', 2, comment='Fourth dimension')
# Well, since 'subgroup' != 'v', 'subgroup' becomes just another group node created on the fly
print(traj.parameters.subgroup)
# This even works for already existing groups and with the well known *dot* notation:
traj.parameters = Parameter('subgroup.subsubgroup.w', 2)
# See
print('w='+str(traj.par.subgroup.subsubgroup.w))


# There's a lazy version which does not require a constructor.
# This can be turned on via
traj.v_lazy_adding = True
# And you can add a new parameter via
traj.parameters.u = 1, 'Fourth dimension'
print('u=' + str(traj.u))
# However, now you can no longer change values of existing parameters,
# because this is interpreted as a new parameter addition, so this fails:
try:
    traj.parameters.u = 2
    print('I won`t be reached')
except AttributeError as exc:
    print('Told you: `%s`' % repr(exc))
# See:
print('u=' + str(traj.par.u))

# But disabling the new adding method makes this work again
traj.v_lazy_adding = False
traj.f_get('u').f_unlock()
traj.parameters.u = 3
# now we simply change `u` to be 3

# There's also a lazy version to add new group nodes:
from pypet import new_group
traj.v_lazy_adding=True
traj.im_new = new_group
# And `im_new` is a new group node:
print(traj.im_new)
```

```
# Finally, there's one more thing. Using this notation we can also add links.
# Simply use the `=` assignment with objects that already exist in your trajectory:
traj.mylink = traj.f_get('x')
# now `mylink` links to parameter `x`, also fast access works:
print('Linking to x gives: ' + str(traj.mylink))
```

Multiprocessing

Download: `example_04_multiprocessing.py`

This code snippet shows how to use multiprocessing with locks. In order to use the queue based multiprocessing one simply needs to make the following change for the environment creation:

```
wrap_mode=pypetconstants.WRAP_MODE_QUEUE.
```

```
__author__ = 'Robert Meyer'

import os # For path names being viable under Windows and Linux
import logging

from pypet import Environment, cartesian_product
from pypet import pypetconstants


# Let's reuse the simple multiplication example
def multiply(traj):
    """Sophisticated simulation of multiplication"""
    z=traj.x*traj.y
    traj.f_add_result('z',z=z, comment='I am the product of two reals!')


def main():
    """Main function to protect the *entry point* of the program.

    If you want to use multiprocessing under Windows you need to wrap your
    main code creating an environment into a function. Otherwise
    the newly started child processes will re-execute the code and throw
    errors (also see https://docs.python.org/2/library/multiprocessing.html#windows).

    """

    # Create an environment that handles running.
    # Let's enable multiprocessing with 2 workers.
    filename = os.path.join('hdf5', 'example_04.hdf5')
    env = Environment(trajectory='Example_04_MP',
                      filename=filename,
                      file_title='Example_04_MP',
                      log_stdout=True,
                      comment='Multiprocessing example!',
                      multiproc=True,
                      ncores=4,
                      use_pool=True,  # Our runs are inexpensive we can get rid of overhead
                      # by using a pool
                      wrap_mode=pypetconstants.WRAP_MODE_QUEUE)

    # Get the trajectory from the environment
    traj = env.v_trajectory

    # Add both parameters
    traj.f_add_parameter('x', 1.0, comment='I am the first dimension!')
    traj.f_add_parameter('y', 1.0, comment='I am the second dimension!')
```

```
    # Explore the parameters with a cartesian product, but we want to explore a bit more
    traj.f_explore(cartesian_product({'x':[float(x) for x in range(20)],
                                      'y':[float(y) for y in range(20)]}))

    # Run the simulation
    env.f_run(multiply)

    # Finally disable logging and close all log-files
    env.f_disable_logging()


if __name__ == '__main__':
    # This will execute the main function in case the script is called from the one true
    # main process and not from a child processes spawned by your environment.
    # Necessary for multiprocessing under Windows.
    main()
```

Storing and Loading Large Results (or just parts of them)

Download: `example_09_large_results.py`

Want to know how to load large results in parts? See below:

```
__author__ = 'Robert Meyer'

import numpy as np
import os # For path names being viable under Windows and Linux

from pypet.trajectory import Trajectory
from pypet import pypetconstants

# Here I show how to store and load results in parts if they are quite large.
# I will skip using an environment and only work with a trajectory.

# We can create a trajectory and hand it a filename directly and it will create an
# HDF5StorageService for us:
filename = os.path.join('hdf5', 'example_09.hdf5')
traj = Trajectory(name='example_09_huge_data',
                  filename=filename)

# Now we directly add a huge result. Note that we could do the exact same procedure during
# a single run, there is no syntactical difference.
# However, the sub branch now is different, the result will be found under `traj.results.trajecto
# instead of `traj.results.run_XXXXXXXX` (where XXXXXXX is the current run index, e.g. 00000007).
# We will add two large matrices a 100 by 100 by 100 one and 1000 by 1000 one, both containing
# random numbers. They are called `mat1` and `mat2` and are handled by the same result object
# called `huge_matrices`:
traj.f_add_result('huge_matrices',
                  mat1 = np.random.rand(100,100,100),
                  mat2 = np.random.rand(1000,1000))

# Note that the result will not support fast access since it contains more than a single
# data item. Even if there was only `mat1`, because the name is `mat1` instead of `huge_matrices`
# (as the result object itself is called), fast access does not work either.
# Yet, we can access data via natural naming using the names `mat1` and `mat2` e.g.:
val_mat1 = traj.huge_matrices.mat1[10,10,10]
val_mat2 = traj.huge_matrices.mat2[42,13]
print('mat1 contains %f at position [10,10,10]' % val_mat1)
print('mat2 contains %f at position [42,13]' % val_mat2)

# Ok that was enough analysis of the data and should be sufficient for a phd thesis (in economics
# Let's store our trajectory and subsequently free the space for something completely different.
```

```
traj.f_store()

# We free the data:
traj.huge_matrices.f_empty()

# Check if the data was deleted
if traj.huge_matrices.f_is_empty():
    print('As promised: Nothing there!')
else:
    print('What a disappointing peace of crap this software is!')

# Lucky, it worked.
# Ok we could it add some more stuff to the result object if we want to:
traj.huge_matrices.f_set(monty='Always look on the bright side of life!')

# Next we can store our new string called monty to disk. Since our trajectory was already
# stored to disk once, we can make use of the functionality to store individual items:
traj.f_store_item('huge_matrices')

# Neat, hu? Ok now let's load some of it back, for educational purposes let's start with a fresh
# trajectory. Let's keep the old trajectory name in mind. The current time is added to the
# trajectory name on creation (if you do not want this, just say `add_time=False`).
# Thus, the name is not `example_09_huge_data`, but `example_09_huge_data_XXXX_XX_XX_XXhXXmXXs`:
old_traj_name = traj.v_name
del traj
traj = Trajectory(filename=filename)

# We only want to load the skeleton but not the data:
traj.f_load(name=old_traj_name, load_results=pypetconstants.LOAD_SKELETON)

# Check if we only loaded the skeleton, that means the `huge_matrices` result must be empty:
if traj.huge_matrices.f_is_empty():
    print('Told you!')
else:
    print('Unbelievable, this sucks!')

# Now let's only load `monty` and `mat1`.
# We can do this by passing the keyword argument `load_only` to the load item function:
traj.f_load_item('huge_matrices', load_only=['monty','mat1'])

# Check if this worked:
if ('monty' in traj.huge_matrices and
     'mat1' in traj.huge_matrices and
      not 'mat2' in traj.huge_matrices ):

    val_mat1 = traj.huge_matrices.mat1[10,10,10]
    print('mat1 contains %f at position [10,10,10]' % val_mat1)
    print('And do not forget: %s' % traj.huge_matrices.monty)
else:
    print('That\'s it, I quit! I cannot work like this!')

# Thanks for your attention!
```

Post-Processing and Pipelining (from the Tutorial)

Here you find an example of post-processing.

It consists of a main script *main.py* for the three phases *pre-processing*, *run phase* and *post-processing* of a single neuron simulation and a *analysis.py* file giving an example of a potential data analysis encompassing plotting the results. Moreover, there exists a *pipeline.py* file to crunch all first three phases into a single function.

A detail explanation of the example can be found in the *Tutorial* section.

Download: `main.py`

Download: `analysis.py`

Download: `pipeline.py`

Main

```
__author__ = 'robert'

import numpy as np
import pandas as pd
import logging
import os # For path names working under Linux and Windows

from pypet import Environment, cartesian_product


def run_neuron(traj):
    """Runs a simulation of a model neuron.

    :param traj:

        Container with all parameters.

    :return:

        An estimate of the firing rate of the neuron

    """

    # Extract all parameters from `traj`
    V_init = traj.par.neuron.V_init
    I = traj.par.neuron.I
    tau_V = traj.par.neuron.tau_V
    tau_ref = traj.par.neuron.tau_ref
    dt = traj.par.simulation.dt
    duration = traj.par.simulation.duration

    steps = int(duration / float(dt))
    # Create some containers for the Euler integration
    V_array = np.zeros(steps)
    V_array[0] = V_init
    spiketimes = []  # List to collect all times of action potentials

    # Do the Euler integration:
    print('Starting Euler Integration')
    for step in range(1, steps):
        if V_array[step-1] >= 1:
            # The membrane potential crossed the threshold and we mark this as
            # an action potential
            V_array[step] = 0
            spiketimes.append((step-1)*dt)
        elif spiketimes and step * dt - spiketimes[-1] <= tau_ref:
            # We are in the refractory period, so we simply clamp the voltage
            # to 0
            V_array[step] = 0
        else:
            # Euler Integration step:
            dV = -1/tau_V * V_array[step-1] + I
            V_array[step] = V_array[step-1] + dV*dt
```

```
    print('Finished Euler Integration')

    # Add the voltage trace and spike times
    traj.f_add_result('neuron.$', V=V_array, nspikes=len(spiketimes),
                      comment='Contains the development of the membrane potential over time '
                              'as well as the number of spikes.')
    # This result will be renamed to `traj.results.neuron.run_XXXXXXXX`.

    # And finally we return the estimate of the firing rate
    return len(spiketimes) / float(traj.par.simulation.duration) *1000
    # *1000 since we have defined duration in terms of milliseconds


def neuron_postproc(traj, result_list):
    """Postprocessing, sorts computed firing rates into a table

    :param traj:

        Container for results and parameters

    :param result_list:

        List of tuples, where first entry is the run index and second is the actual
        result of the corresponding run.

    :return:
    """

    # Let's create a pandas DataFrame to sort the computed firing rate according to the
    # parameters. We could have also used a 2D numpy array.
    # But a pandas DataFrame has the advantage that we can index into directly with
    # the parameter values without translating these into integer indices.
    I_range = traj.par.neuron.f_get('I').f_get_range()
    ref_range = traj.par.neuron.f_get('tau_ref').f_get_range()

    I_index = sorted(set(I_range))
    ref_index = sorted(set(ref_range))
    rates_frame = pd.DataFrame(columns=ref_index, index=I_index)
    # This frame is basically a two dimensional table that we can index with our
    # parameters

    # Now iterate over the results. The result list is a list of tuples, with the
    # run index at first position and our result at the second
    for result_tuple in result_list:
        run_idx = result_tuple[0]
        firing_rates = result_tuple[1]
        I_val = I_range[run_idx]
        ref_val = ref_range[run_idx]
        rates_frame.loc[I_val, ref_val] = firing_rates # Put the firing rate into the
        # data frame

    # Finally we going to store our new firing rate table into the trajectory
    traj.f_add_result('summary.firing_rates', rates_frame=rates_frame,
                      comment='Contains a pandas data frame with all firing rates.')


def add_parameters(traj):
    """Adds all parameters to `traj`"""
    print('Adding Parameters')

    traj.f_add_parameter('neuron.V_init', 0.0,
                         comment='The initial condition for the '
                                 'membrane potential')
```

```
    traj.f_add_parameter('neuron.I', 0.0,
                         comment='The externally applied current.')
    traj.f_add_parameter('neuron.tau_V', 10.0,
                         comment='The membrane time constant in milliseconds')
    traj.f_add_parameter('neuron.tau_ref', 5.0,
                        comment='The refractory period in milliseconds '
                                'where the membrane potnetial '
                                'is clamped.')

    traj.f_add_parameter('simulation.duration', 1000.0,
                         comment='The duration of the experiment in '
                                 'milliseconds.')
    traj.f_add_parameter('simulation.dt', 0.1,
                         comment='The step size of an Euler integration step.')


def add_exploration(traj):
    """Explores different values of `I` and `tau_ref`."""

    print('Adding exploration of I and tau_ref')

    explore_dict = {'neuron.I': np.arange(0, 1.01, 0.01).tolist(),
                    'neuron.tau_ref': [5.0, 7.5, 10.0]}

    explore_dict = cartesian_product(explore_dict, ('neuron.tau_ref', 'neuron.I'))
    # The second argument, the tuple, specifies the order of the cartesian product,
    # The variable on the right most side changes fastest and defines the
    # 'inner for-loop' of the cartesian product

    traj.f_explore(explore_dict)


def main():

    filename = os.path.join('hdf5', 'FiringRate.hdf5')
    env = Environment(trajectory='FiringRate',
                      comment='Experiment to measure the firing rate '
                              'of a leaky integrate and fire neuron. '
                              'Exploring different input currents, '
                              'as well as refractory periods',
                      add_time=False, # We don't want to add the current time to the name,
                      log_stdout=True,
                      log_config='DEFAULT',
                      multiproc=True,
                      ncores=2, #My laptop has 2 cores ;-)
                      wrap_mode='QUEUE',
                      filename=filename,
                      overwrite_file=True)

    traj = env.v_trajectory

    # Add parameters
    add_parameters(traj)

    # Let's explore
    add_exploration(traj)

    # Ad the postprocessing function
    env.f_add_postprocessing(neuron_postproc)

    # Run the experiment
    env.f_run(run_neuron)
```

```
    # Finally disable logging and close all log-files
    env.f_disable_logging()

if __name__ == '__main__':
    main()
```

Analysis

```
__author__ = 'robert'

import os

from pypet import Trajectory
import matplotlib.pyplot as plt


def main():

    # This time we don't need an environment since we just going to look
    # at data in the trajectory
    traj = Trajectory('FiringRate', add_time=False)

    # Let's load the trajectory from the file
    # Only load the parameters, we will load the results on the fly as we need them
    filename = os.path.join('hdf5', 'FiringRate.hdf5')
    traj.f_load(load_parameters=2, load_derived_parameters=0, load_results=0,
                load_other_data=0, filename=filename)

    # We'll simply use auto loading so all data will be loaded when needed.
    traj.v_auto_load = True

    rates_frame = traj.res.summary.firing_rates.rates_frame
    # Here we load the data automatically on the fly

    plt.figure()
    plt.subplot(2,1,1)
    #Let's iterate through the columns and plot the different firing rates :
    for tau_ref, I_col in rates_frame.iteritems():
        plt.plot(I_col.index, I_col, label='Avg. Rate for tau_ref=%s' % str(tau_ref))

    # Label the plot
    plt.xlabel('I')
    plt.ylabel('f[Hz]')
    plt.title('Firing as a function of input current `I`')
    plt.legend(loc='best')

    # Also let's plot an example run, how about run 13 ?
    example_run = 13

    traj.v_idx = example_run # We make the trajectory behave as a single run container.
    # This short statement has two major effects:
    # a) all explored parameters are set to the value of run 13,
    # b) if there are tree nodes with names other than the current run aka `run_00000013`
    # they are simply ignored, if we use the `$` sign or the `crun` statement,
    # these are translated into `run_00000013`.

    # Get the example data
    example_I = traj.I
    example_tau_ref = traj.tau_ref
    example_V = traj.results.neuron.crun.V # Here crun stands for run_00000013
```

```
    # We need the time step...
    dt = traj.dt
    # ...to create an x-axis for the plot
    dt_array = [irun * dt for irun in range(len(example_V))]

    # And plot the development of V over time,
    # Since this is rather repetitive, we only
    # plot the first eighth of it.
    plt.subplot(2,1,2)
    plt.plot(dt_array, example_V)
    plt.xlim((0, dt*len(example_V)/8))

    # Label the axis
    plt.xlabel('t[ms]')
    plt.ylabel('V')
    plt.title('Example of development of V for I=%s, tau_ref=%s in run %d' %
              (str(example_I), str(example_tau_ref), traj.v_idx))

    # And let's take a look at it
    plt.show()

    # Finally revoke the `traj.v_idx=13` statement and set everything back to normal.
    # Since our analysis is done here, we could skip that, but it is always a good idea
    # to do that.
    traj.f_restore_default()


if __name__ == '__main__':
    main()
```

Pipelining

Additionally, you can use pipelining.

Since these three steps pre-processing, run-phase, post-processing define a common pipeline, you can actually also make *pypet* supervise all three steps at once.

You can define a pipeline function, that does the pre-processing and returns the job function plus some optional arguments and the post-processing function with some other optional arguments.

So, you could define the following pipeline function. The pipeline function has to only accept the trajectory as first argument and has to return 2 tuples, one for the run function and one for the post-processing. Since none of our functions takes any other arguments than the trajectory (and the pos-processing function the result list) we simply return an empty tuple `()` for no arguments and an empty dictionary `{}` for no keyword arguments.

And that's it, than everything including the pre-processing and addition of parameters is supervised by *pypet*. Check out the source code below:

```
__author__ = 'robert'

import logging
import os # For path names working under Windows and Linux

from main import add_parameters, add_exploration, run_neuron, neuron_postproc
from pypet import Environment


def mypipeline(traj):
    """A pipeline function that defines the entire experiment

    :param traj:
```

```
        Container for results and parameters

    :return:

        Two tuples. First tuple contains the actual run function plus additional
        arguments (yet we have none). Second tuple contains the
        postprocessing function including additional arguments.

    """
    add_parameters(traj)
    add_exploration(traj)
    return (run_neuron,(),{}), (neuron_postproc,(),{})

def main():
    filename = os.path.join('hdf5', 'FiringRate.hdf5')
    env = Environment(trajectory='FiringRatePipeline',
                      comment='Experiment to measure the firing rate '
                            'of a leaky integrate and fire neuron. '
                            'Exploring different input currents, '
                            'as well as refractory periods',
                      add_time=False, # We don't want to add the current time to the name,
                      log_stdout=True,
                      multiproc=True,
                      ncores=2, #My laptop has 2 cores ;-)
                      filename=filename,
                      overwrite_file=True)

    env.f_pipeline(mypipeline)

    # Finally disable logging and close all log-files
    env.f_disable_logging()

if __name__ == '__main__':
    main()
```

Wrapping an Existing Project (Cellular Automata Inside!)

Here you can find out how to wrap *pypet* around an already existing simulation. The original project (`original.py`) simulates elementary cellular automata.

The code explores different starting conditions and automata rules. `pypetwrap.py` shows how to include *pypet* into the project without changing much of the original code. Basically, the core code of the simulation is left untouched. Only the *boilerplate* of the main script changes and a short wrapper function is needed that passes parameters from the *trajectory* to the core simulation.

Moreover, introducing *pypet* allows much easier exploration of the parameter space. Now exploring different parameter sets requires no more code changes.

Download: `original.py`

Download: `pypetwrap.py`

Original Project

```
""" This module contains a simulation of 1 dimensional cellular automata

We also simulate famous rule 110: http://en.wikipedia.org/wiki/Rule_110

"""

__author__ = 'Robert Meyer'
```

```
import numpy as np
import os
import matplotlib.pyplot as plt
import pickle

from pypet import progressbar #  I don't want to write another progressbar, so I use this here


def convert_rule(rule_number):
    """ Converts a rule given as an integer into a binary list representation.

    It reads from left to right (contrary to the Wikipedia article given below),
    i.e. the 2**0 is found on the left hand side and 2**7 on the right.

    For example:

        ``convert_rule(30)`` returns [0, 1, 1, 1, 1, 0, 0, 0]


    The resulting binary list can be interpreted as
    the following transition table:

         neighborhood  new cell state
                000     0
                001     1
                010     1
                011     1
                100     1
                101     0
                110     0
                111     0

    For more information about this rule
    see: http://en.wikipedia.org/wiki/Rule_30

    """
    binary_rule = [(rule_number // pow(2,i)) % 2 for i in range(8)]
    return np.array(binary_rule)


def make_initial_state(name, ncells, seed=42):
    """ Creates an initial state for the automaton.

    :param name:

        Either ``'single'`` for a single live cell in the middle of the cell ring,
        or ``'random'`` for uniformly distributed random pattern of zeros and ones.

    :param ncells: Number of cells in the automaton

    :param seed: Random number seed for the ``#random'`` condition

    :return: Numpy array of zeros and ones (or just a one lonely one surrounded by zeros)

    :raises: ValueError if the ``name`` is unknown

    """
    if name == 'single':
        just_one_cell = np.zeros(ncells)
        just_one_cell[int(ncells/2)] = 1.0
        return just_one_cell
    elif name == 'random':
```

```
        np.random.seed(seed)
        random_init = np.random.randint(2, size=ncells)
        return random_init
    else:
        raise ValueError('I cannot handel your initial state `%s`.' % name)


def plot_pattern(pattern, rule_number, filename):
    """ Plots an automaton ``pattern`` and stores the image under a given ``filename``.

    For axes labels the ``rule_number`` is also required.

    """
    plt.figure()
    plt.imshow(pattern)
    plt.xlabel('Cell No.')
    plt.ylabel('Time Step')
    plt.title('CA with Rule %s' % str(rule_number))
    plt.savefig(filename)
    #plt.show()
    plt.close()


def cellular_automaton_1D(initial_state, rule_number, steps):
    """ Simulates a 1 dimensional cellular automaton.

    :param initial_state:

        The initial state of *dead* and *alive* cells as a 1D numpy array.
        It's length determines the size of the simulation.

    :param rule_number:

        The update rule as an integer from 0 to 255.

    :param steps:

        Number of cell iterations

    :return:

        A 2D numpy array (steps x len(initial_state)) containing zeros and ones representing
        the automaton development over time.

    """

    ncells = len(initial_state)
    # Create an array for the full pattern
    pattern = np.zeros((steps, ncells))

    # Pass initial state:
    pattern[0,:] = initial_state

    # Get the binary rule list
    binary_rule = convert_rule(rule_number)

    # Conversion list to get the position in the binary rule list
    neighbourhood_factors = np.array([1, 2, 4])

    # Iterate over all steps to compute the CA
    all_cells = range(ncells)
    for step in range(steps-1):
        current_row = pattern[step, :]
```

```
        next_row = pattern[step+1, :]
        for irun in all_cells:
            # Get the neighbourhood
            neighbour_indices = range(irun - 1, irun + 2)
            neighbourhood = np.take(current_row, neighbour_indices, mode='wrap')
            # Convert neighborhood to decimal
            decimal_neighborhood = int(np.sum(neighbourhood * neighbourhood_factors))
            # Get next state from rule book
            next_state = binary_rule[decimal_neighborhood]
            # Update next state of cell
            next_row[irun] = next_state

    return pattern


def main():
    """ Main simulation function """
    rules_to_test = [10, 30, 90, 110, 184]  # rules we want to explore:
    steps = 250  # cell iterations
    ncells = 400  # number of cells
    seed = 100042  # RNG seed
    initial_states = ['single', 'random']  # Initial states we want to explore

    # create a folder for the plots and the data
    folder = os.path.join(os.getcwd(), 'experiments', 'ca_patterns_original')
    if not os.path.isdir(folder):
        os.makedirs(folder)
    filename = os.path.join(folder, 'all_patterns.p')

    print('Computing all patterns')
    all_patterns = []  # list containing the simulation results
    for idx, rule_number in enumerate(rules_to_test):
        # iterate over all rules
        for initial_name in initial_states:
            # iterate over the initial states

            # make the initial state
            initial_state = make_initial_state(initial_name, ncells, seed=seed)
            # simulate the automaton
            pattern = cellular_automaton_1D(initial_state, rule_number, steps)
            # keep the resulting pattern
            all_patterns.append((rule_number, initial_name, pattern))

        # Print a progressbar, because I am always impatient
        #  (ok that's already from pypet, but it's really handy!)
        progressbar(idx, len(rules_to_test), reprint=True)

    # Store all patterns to disk
    with open(filename, 'wb') as file:
        pickle.dump(all_patterns, file=file)

    # Finally print all patterns
    print('Plotting all patterns')
    for idx, pattern_tuple in enumerate(all_patterns):
        rule_number, initial_name, pattern = pattern_tuple
        # Plot the pattern
        filename = os.path.join(folder, 'rule_%s_%s.png' % (str(rule_number), initial_name))
        plot_pattern(pattern, rule_number, filename)
        progressbar(idx, len(all_patterns), reprint=True)


if __name__ == '__main__':
    main()
```

Using *pypet*

```
""" Module that shows how to wrap *pypet* around an existing project

Thanks to *pypet* the module is now very flexible.
You can immediately start exploring different sets
of parameters, like different seeds or cell numbers.
Accordingly, you can simply change ``exp_dict`` to explore different sets.

On the contrary, this is tedious in the original code
and requires some effort of refactoring.

"""

__author__ = 'Robert Meyer'

import os
import logging

from pypet import Environment, cartesian_product, progressbar

# Lets import the stuff we already have:
from original import cellular_automaton_1D, make_initial_state, plot_pattern

def make_filename(traj):
    """ Function to create generic filenames based on what has been explored """
    explored_parameters = traj.f_get_explored_parameters()
    filename = ''
    for param in explored_parameters.values():
        short_name = param.v_name
        val = param.f_get()
        filename += '%s_%s__' % (short_name, str(val))

    return filename[:-2] + '.png' # get rid of trailing underscores and add file type

def wrap_automaton(traj):
    """ Simple wrapper function for compatibility with *pypet*.

    We will call the original simulation functions with data extracted from ``traj``.

    The resulting automaton patterns wil also be stored into the trajectory.

    :param traj: Trajectory container for data

    """
    # Make initial state
    initial_state = make_initial_state(traj.initial_name, traj.ncells, traj.seed)
    # Run simulation
    pattern = cellular_automaton_1D(initial_state, traj.rule_number, traj.steps)
    # Store the computed pattern
    traj.f_add_result('pattern', pattern, comment='Development of CA over time')


def main():
    """ Main *boilerplate* function to start simulation """
    # Now let's make use of logging
    logger = logging.getLogger()

    # Create folders for data and plots
    folder = os.path.join(os.getcwd(), 'experiments', 'ca_patterns_pypet')
    if not os.path.isdir(folder):
        os.makedirs(folder)
```

```
    filename = os.path.join(folder, 'all_patterns.hdf5')

    # Create an environment
    env = Environment(trajectory='cellular_automata',
                      multiproc=True,
                      ncores=4,
                      wrap_mode='QUEUE',
                      filename=filename,
                      overwrite_file=True)

    # extract the trajectory
    traj = env.v_traj

    traj.v_lazy_adding = True
    traj.par.ncells = 400, 'Number of cells'
    traj.par.steps = 250, 'Number of timesteps'
    traj.par.rule_number = 30, 'The ca rule'
    traj.par.initial_name = 'random', 'The type of initial state'
    traj.par.seed = 100042, 'RNG Seed'


    # Explore
    exp_dict = {'rule_number' : [10, 30, 90, 110, 184],
                'initial_name' : ['single', 'random'],}
    # # You can uncomment the ``exp_dict`` below to see that changing the
    # # exploration scheme is now really easy:
    # exp_dict = {'rule_number' : [10, 30, 90, 110, 184],
    #             'ncells' : [100, 200, 300],
    #             'seed': [333444555, 123456]}
    exp_dict = cartesian_product(exp_dict)
    traj.f_explore(exp_dict)

    # Run the simulation
    logger.info('Starting Simulation')
    env.f_run(wrap_automaton)

    # Load all data
    traj.f_load(load_data=2)

    logger.info('Printing data')
    for idx, run_name in enumerate(traj.f_iter_runs()):
        # Plot all patterns
        filename = os.path.join(folder, make_filename(traj))
        plot_pattern(traj.crun.pattern, traj.rule_number, filename)
        progressbar(idx, len(traj), logger=logger)

    # Finally disable logging and close all log-files
    env.f_disable_logging()


if __name__ == '__main__':
    main()
```

Large Explorations with Many Runs

Download: `example_18_many_runs.py`

How to group many results into buckets

```
"""Exploring more than 20000 runs may slow down *pypet*.

HDF5 has problems handling nodes with more than 10000 children.
```

```
To overcome this problem, simply group your runs into buckets or sets
using the `$set` wildcard.

"""

__author__ = 'Robert Meyer'


import os # To allow file paths working under Windows and Linux

from pypet import Environment
from pypet.utils.explore import cartesian_product

def multiply(traj):
    """Example of a sophisticated simulation that involves multiplying two values."""
    z = traj.x * traj.y
    # Since we perform many runs we will group results into sets of 1000 each
    # using the `$set` wildcard
    traj.f_add_result('$set.$.z', z, comment='Result of our simulation '
                                             'sorted into buckets of '
                                             '1000 runs each!')

def main():
    # Create an environment that handles running
    filename = os.path.join('hdf5','example_18.hdf5')
    env = Environment(trajectory='Multiplication',
                      filename=filename,
                      file_title='Example_18_Many_Runs',
                      comment='Contains many runs',
                      multiproc=True,
                      use_pool=True,
                      ncores=2,
                      wrap_mode='QUEUE')

    # The environment has created a trajectory container for us
    traj = env.v_trajectory

    # Add both parameters
    traj.f_add_parameter('x', 1, comment='I am the first dimension!')
    traj.f_add_parameter('y', 1, comment='I am the second dimension!')

    # Explore the parameters with a cartesian product, yielding 2500 runs
    traj.f_explore(cartesian_product({'x': range(50), 'y': range(50)}))

    # Run the simulation
    env.f_run(multiply)

    # Disable logging
    env.f_disable_logging()

    # turn auto loading on, since results have not been loaded, yet
    traj.v_auto_load = True
    # Use the `v_idx` functionality
    traj.v_idx = 2042
    print('The result of run %d is: ' % traj.v_idx)
    # Now we can rely on the wildcards
    print(traj.res.crunset.crun.z)
    traj.v_idx = -1

if __name__ == '__main__':
    main()
```

1.5.2 Advanced Concepts

Merging of Trajectories

Download: `example_03_trajectory_merging.py`

The code snippet below shows how to merge trajectories.

```
__author__ = 'Robert Meyer'

import os # For using pathnames under Windows and Linux

from pypet import Environment, cartesian_product


# Let's reuse the simple multiplication example
def multiply(traj):
    """Sophisticated simulation of multiplication"""
    z=traj.x*traj.y
    traj.f_add_result('z',z=z, comment='I am the product of two reals!',)



# Create 2 environments that handle running
filename = os.path.join('hdf5', 'example_03.hdf5')
env1 = Environment(trajectory='Traj1',filename=filename,
                   file_title='Example_03',
                   comment='I will be increased!')

env2 = Environment(trajectory='Traj2',filename=filename,
                   file_title='Example_03', log_config=None, # One environment keeping log files
                   # is enough
                   comment = 'I am going to be merged into some other trajectory!')

# Get the trajectories from the environment
traj1 = env1.v_trajectory
traj2 = env2.v_trajectory

# Add both parameters
traj1.f_add_parameter('x', 1.0, comment='I am the first dimension!')
traj1.f_add_parameter('y', 1.0, comment='I am the second dimension!')
traj2.f_add_parameter('x', 1.0, comment='I am the first dimension!')
traj2.f_add_parameter('y', 1.0, comment='I am the second dimension!')

# Explore the parameters with a cartesian product for the first trajectory:
traj1.f_explore(cartesian_product({'x':[1.0,2.0,3.0,4.0], 'y':[6.0,7.0,8.0]}))
# Let's explore slightly differently for the second:
traj2.f_explore(cartesian_product({'x':[3.0,4.0,5.0,6.0], 'y':[7.0,8.0,9.0]}))


# Run the simulations with all parameter combinations
env1.f_run(multiply)
env2.f_run(multiply)

# Now we merge them together into traj1
# We want to remove duplicate entries
# like the parameter space point x=3.0, y=7.0.
# Several points have been explored by both trajectories and we need them only once.
# Therefore, we set remove_duplicates=True (Note this takes O(N1*N2)!).
# We also want to backup both trajectories, but we let the system choose the filename.
# Accordingly we choose backup_filename=True instead of providing a filename.
# We want to move the hdf5 nodes from one trajectory to the other.
# Thus we set move_nodes=True.
```

```
# Finally,we want to delete the other trajectory afterwards since we already have a backup.
traj1.f_merge(traj2, remove_duplicates=True, backup_filename=True,
              move_data=True, delete_other_trajectory=True)

# And that's it, now we can take a look at the new trajectory and print all x,y,z triplets.
# But before that we need to load the data we computed during the runs from disk.
# We choose load_parameters=2 and load_results=2 since we want to load all data and not only
# the skeleton
traj1.f_load(load_parameters=2, load_results=2)

for run_name in traj1.f_get_run_names():
    # We can make the trajectory belief it is a single run. All parameters will
    # be treated as they were in the specific run. And we can use the `crun` wildcard.
    traj1.f_as_run(run_name)
    x=traj1.x
    y=traj1.y
    # We need to specify the current run, because there exists more than one z value
    z=traj1.crun.z
    print('%s: x=%f, y=%f, z=%f' % (run_name, x, y, z))

# Don't forget to reset you trajectory to the default settings, to release its belief to
# be the last run.
traj1.f_restore_default()

# As you can see duplicate parameter space points have been removed.
# If you wish you can take a look at the files and backup files in
# the experiments/example_03/HDF5 directory

# Finally, disable logging and close log files
env1.f_disable_logging()
```

Custom Parameter (Strange Attractor Inside!)

Download: `example_05_custom_parameter.py`

Here you can see an example of a custom parameter and how to reload results and use them for analysis. We will simulate the Lorenz Attractor and integrate with a simple Euler method. We will explore three different initial conditions.

```
__author__ = 'Robert Meyer'

import numpy as np
import inspect
import os # For path names being viable under Windows and Linux

from pypet import Environment, Parameter, ArrayParameter, Trajectory
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D


# Here we will see how we can write our own custom parameters and how we can use
# it with a trajectory.

# Now we want to do a more sophisticated simulations, we will integrate a differential equation
# with an Euler scheme

# Let's first define our job to do
def euler_scheme(traj, diff_func):
    """Simulation function for Euler integration.

    :param traj:
```

```
        Container for parameters and results

    :param diff_func:

        The differential equation we want to integrate

    """

    steps = traj.steps
    initial_conditions = traj.initial_conditions
    dimension = len(initial_conditions)

    # This array will collect the results
    result_array = np.zeros((steps,dimension))
    # Get the function parameters stored into `traj` as a dictionary
    # with the (short) names as keys :
    func_params_dict = traj.func_params.f_to_dict(short_names=True, fast_access=True)
    # Take initial conditions as first result
    result_array[0] = initial_conditions

    # Now we compute the Euler Scheme steps-1 times
    for idx in range(1,steps):
        result_array[idx] = diff_func(result_array[idx-1], **func_params_dict) * traj.dt + \
                            result_array[idx-1]
    # Note the **func_params_dict unzips the dictionary, it's the reverse of **kwargs in function
    # definitions!

    #Finally we want to keep the results
    traj.f_add_result('euler_evolution', data=result_array, comment='Our time series data!')


# Ok, now we want to make our own (derived) parameter that stores source code of python functions
# We do NOT want a parameter that stores an executable function. This would complicate
# the problem a lot. If you have something like that in mind, you might wanna take a look
# at the marshal (http://docs.python.org/2/library/marshal) module
# or dill (https://pypi.python.org/pypi/dill) package.
# Our intention here is to define a parameter that we later on use as a derived parameter
# to simply keep track of the source code we use ('git' would be, of course, the better solution
# but this is just an illustrative example)
class FunctionParameter(Parameter):
    # We can go for a a cheap solution and make use of the function `_convert_data` of the parent
    # This gets called before adding data to the parameter to turn numpy arrays
    # into read-only numpy arrays. But we will use the function for our purpose to extract
    # the source code:
    def _convert_data(self, val):
        if callable(val):
            return inspect.getsource(val)
        else:
            return super(FunctionParameter,self)._convert_data(val)

    # For more complicate parameters you might consider implementing:
    # `f_supports` (we do not need it since we convert the data to stuff the parameter already
    #    supports, and that is strings!)
    #
    # and
    # the private functions
    #
    # `_values_of_same_type` (to tell whether data is similar, i.e. of two data items agree in th
    #   type, this is important to only allow exploration within the same dimension.
    #   For instance, a parameter that stores integers, should only explore integers etc.)
    #
    # and
    #
```

```
    # `_equal_values` (to tell if two data items are equal. This is important for merging if you
    #       want to erase duplicate parameter points. The trajectory needs to know when a
    #       parameter space point was visited before.)
    #
    # and
    #
    # `_store` (to be able to turn complex data into basic types understood by the storage servic
    #
    # and
    #
    # `_load` (to be able to recover your complex data form the basic types understood by the sto
    # service)
    #
    # But for now we will rely on the parent functions and hope for the best!



# Ok now let's follow the ideas in the final section of the cookbook and let's
# have a part in our simulation that only defines the parameters.
def add_parameters(traj):
    """Adds all necessary parameters to the `traj` container"""

    traj.f_add_parameter('steps', 10000, comment='Number of time steps to simulate')
    traj.f_add_parameter('dt', 0.01, comment='Step size')

    # Here we want to add the initial conditions as an array parameter. We will simulate
    # a 3-D differential equation, the Lorenz attractor.
    traj.f_add_parameter(ArrayParameter,'initial_conditions', np.array([0.0,0.0,0.0]),
                         comment = 'Our initial conditions, as default we will start from'
                                   ' origin!')

    # We will group all parameters of the Lorenz differential equation into the group 'func_param
    traj.f_add_parameter('func_params.sigma', 10.0)
    traj.f_add_parameter('func_params.beta', 8.0/3.0)
    traj.f_add_parameter('func_params.rho', 28.0)

    #For the fun of it we will annotate the  group
    traj.func_params.v_annotations.info='This group contains as default the original values chose
                                    'by Edward Lorenz in 1963. Check it out on wikipedia ' \
                                    '(https://en.wikipedia.org/wiki/Lorenz_attractor)!'


# We need to define the lorenz function, we will assume that the value array is 3 dimensional,
# First dimension contains the x-component, second y-component, and third the z-component
def diff_lorenz(value_array, sigma, beta, rho):
    """The Lorenz attractor differential equation

    :param value_array: 3d array containing the x,y, and z component values.
    :param sigma: Constant attractor parameter
    :param beta: FConstant attractor parameter
    :param rho: Constant attractor parameter

    :return: 3d array of the Lorenz system evaluated at `value_array`

    """
    diff_array = np.zeros(3)
    diff_array[0] = sigma * (value_array[1]-value_array[0])
    diff_array[1] = value_array[0] * (rho - value_array[2]) - value_array[1]
    diff_array[2] = value_array[0] * value_array[1] - beta * value_array[2]

    return diff_array
```

```
# And here goes our main function
def main():

    filename = os.path.join('hdf5', 'example_05.hdf5')
    env = Environment(trajectory='Example_05_Euler_Integration',
                      filename=filename,
                      file_title='Example_05_Euler_Integration',
                      comment='Go for Euler!')


    traj = env.v_trajectory
    trajectory_name = traj.v_name

    # 1st a) phase parameter addition
    add_parameters(traj)

    # 1st b) phase preparation
    # We will add the differential equation (well, its source code only) as a derived parameter
    traj.f_add_derived_parameter(FunctionParameter,'diff_eq', diff_lorenz,
                                 comment='Source code of our equation!')

    # We want to explore some initial conditions
    traj.f_explore({'initial_conditions' : [
        np.array([0.01,0.01,0.01]),
        np.array([2.02,0.02,0.02]),
        np.array([42.0,4.2,0.42])
    ]})
    # 3 different conditions are enough for an illustrative example

    # 2nd phase let's run the experiment
    # We pass `euler_scheme` as our top-level simulation function and
    # the Lorenz equation 'diff_lorenz' as an additional argument
    env.f_run(euler_scheme, diff_lorenz)

    # We don't have a 3rd phase of post-processing here

    # 4th phase analysis.
    # I would recommend to do post-processing completely independent from the simulation,
    # but for simplicity let's do it here.

    # Let's assume that we start all over again and load the entire trajectory new.
    # Yet, there is an error within this approach, do you spot it?
    del traj
    traj = Trajectory(filename=filename)

    # We will only fully load parameters and derived parameters.
    # Results will be loaded manually later on.
    try:
        # However, this will fail because our trajectory does not know how to
        # build the FunctionParameter. You have seen this coming, right?
        traj.f_load(name=trajectory_name, load_parameters=2, load_derived_parameters=2,
                    load_results=1)
    except ImportError as e:

        print('That did\'nt work, I am sorry: %s ' % str(e))

        # Ok, let's try again but this time with adding our parameter to the imports
        traj = Trajectory(filename=filename,
                          dynamically_imported_classes=FunctionParameter)

        # Now it works:
        traj.f_load(name=trajectory_name, load_parameters=2, load_derived_parameters=2,
                    load_results=1)
```

```
    #For the fun of it, let's print the source code
    print('\n ---------- The source code of your function ---------- \n %s' % traj.diff_eq)

    # Let's get the exploration array:
    initial_conditions_exploration_array = traj.f_get('initial_conditions').f_get_range()
    # Now let's plot our simulated equations for the different initial conditions:
    # We will iterate through the run names
    for idx, run_name in enumerate(traj.f_get_run_names()):

        #Get the result of run idx from the trajectory
        euler_result = traj.results.f_get(run_name).euler_evolution
        # Now we manually need to load the result. Actually the results are not so large and we
        # could load them all at once. But for demonstration we do as if they were huge:
        traj.f_load_item(euler_result)
        euler_data = euler_result.data

        #Plot fancy 3d plot
        fig = plt.figure(idx)
        ax = fig.gca(projection='3d')
        x = euler_data[:,0]
        y = euler_data[:,1]
        z = euler_data[:,2]
        ax.plot(x, y, z, label='Initial Conditions: %s' % str(initial_conditions_exploration_array
        plt.legend()
        plt.show()

        # Now we free the data again (because we assume its huuuuuuge):
        del euler_data
        euler_result.f_empty()

    # You have to click through the images to stop the example_05 module!

    # Finally disable logging and close all log-files
    env.f_disable_logging()


if __name__ == '__main__':
    main()
```

Parameter Presetting

Download: `example_06_parameter_presetting.py`

We will reuse some stuff from the previous example *Custom Parameter (Strange Attractor Inside!)*:

- Our main euler simulation job *euler_scheme*
- The *FunctionParameter* to store source code

We will execute the same euler simulation as before, but now with a different differential equation yielding the Roessler Attractor. If you erase the statement

traj.f_preset_parameter('diff_name', 'diff_roessler')

you will end up with the same results as in the previous example.

```
__author__ = 'Robert Meyer'

import numpy as np
import os # For path names being viable under Windows and Linux

# Let's reuse the stuff from the previous example
```

```
from example_05_custom_parameter import euler_scheme, FunctionParameter, diff_lorenz

from pypet import Environment, ArrayParameter
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Now we will add some control flow to allow to switch between the differential equations
def add_parameters(traj):
    """Adds all necessary parameters to the `traj` container.

    You can choose between two parameter sets. One for the Lorenz attractor and
    one for the Roessler attractor.
    The former is chosen for `traj.diff_name=='diff_lorenz'`, the latter for
    `traj.diff_name=='diff_roessler'`.
    You can use parameter presetting to switch between the two cases.

    :raises: A ValueError if `traj.diff_name` is none of the above

    """
    traj.f_add_parameter('steps', 10000, comment='Number of time steps to simulate')
    traj.f_add_parameter('dt', 0.01, comment='Step size')

    # Here we want to add the initial conditions as an array parameter, since we will simulate
    # a 3-D differential equation, that is the Roessler attractor
    # (https://en.wikipedia.org/wiki/R%C3%B6ssler_attractor)
    traj.f_add_parameter(ArrayParameter,'initial_conditions', np.array([0.0,0.0,0.0]),
                         comment = 'Our initial conditions, as default we will start from'
                                   ' origin!')

    # Per default we choose the name `'diff_lorenz'` as in the last example
    traj.f_add_parameter('diff_name','diff_lorenz', comment= 'Name of our differential equation')

    # We want some control flow depending on which name we really choose
    if traj.diff_name == 'diff_lorenz':
        # These parameters are for the Lorenz differential equation
        traj.f_add_parameter('func_params.sigma', 10.0)
        traj.f_add_parameter('func_params.beta', 8.0/3.0)
        traj.f_add_parameter('func_params.rho', 28.0)
    elif traj.diff_name == 'diff_roessler':
        # If we use the Roessler system we need different parameters
        traj.f_add_parameter('func_params.a', 0.1)
        traj.f_add_parameter('func_params.c', 14.0)
    else:
        raise ValueError('I don\'t know what %s is.' % traj.diff_name)



# We need to define the Roessler function, we will assume that the value array is 3 dimensional,
# First dimension is x-component, second y-component, and third the z-component
def diff_roessler(value_array, a, c):
    """The Roessler attractor differential equation

    :param value_array: 3d array containing the x,y, and z component values.
    :param a: Constant attractor parameter
    :param c: Constant attractor parameter

    :return: 3d array of the Roessler system evaluated at `value_array`

    """
    b=a
    diff_array = np.zeros(3)
    diff_array[0] = -value_array[1] - value_array[2]
    diff_array[1] = value_array[0] + a * value_array[1]
```

```
    diff_array[2] = b + value_array[2] * (value_array[0] - c)

    return diff_array


# And here goes our main function
def main():

    filename = os.path.join('hdf5', 'example_06.hdf5')
    env = Environment(trajectory='Example_06_Euler_Integration',
                      filename=filename,
                      file_title='Example_06_Euler_Integration',
                      comment = 'Go for Euler!')


    traj = env.v_trajectory

    # 1st a) phase parameter addition
    # Remember we have some control flow in the `add_parameters` function, the default parameter
    # set we choose is the `'diff_lorenz'` one, but we want to deviate from that and use the
    # `'diff_roessler'`.
    # In order to do that we can preset the corresponding name parameter to change the
    # control flow:
    traj.f_preset_parameter('diff_name', 'diff_roessler') # If you erase this line, you will get
                                                          # again the lorenz attractor
    add_parameters(traj)

    # 1st b) phase preparation
    # Let's check which function we want to use
    if traj.diff_name=='diff_lorenz':
        diff_eq = diff_lorenz
    elif traj.diff_name=='diff_roessler':
        diff_eq = diff_roessler
    else:
        raise ValueError('I don\'t know what %s is.' % traj.diff_name)
    # And add the source code of the function as a derived parameter.
    traj.f_add_derived_parameter(FunctionParameter, 'diff_eq', diff_eq,
                                 comment='Source code of our equation!')

    # We want to explore some initial conditions
    traj.f_explore({'initial_conditions' : [
        np.array([0.01,0.01,0.01]),
        np.array([2.02,0.02,0.02]),
        np.array([42.0,4.2,0.42])
    ]})
    # 3 different conditions are enough for now

    # 2nd phase let's run the experiment
    # We pass 'euler_scheme' as our top-level simulation function and
    # the Roessler function as an additional argument
    env.f_run(euler_scheme, diff_eq)

    # Again no post-processing

    # 4th phase analysis.
    # I would recommend to do the analysis completely independent from the simulation
    # but for simplicity let's do it here.
    # We won't reload the trajectory this time but simply update the skeleton
    traj.f_load_skeleton()

    #For the fun of it, let's print the source code
    print('\n ---------- The source code of your function ---------- \n %s' % traj.diff_eq)
```

```
    # Let's get the exploration array:
    initial_conditions_exploration_array = traj.f_get('initial_conditions').f_get_range()
    # Now let's plot our simulated equations for the different initial conditions.
    # We will iterate through the run names
    for idx, run_name in enumerate(traj.f_get_run_names()):

        # Get the result of run idx from the trajectory
        euler_result = traj.results.f_get(run_name).euler_evolution
        # Now we manually need to load the result. Actually the results are not so large and we
        # could load them all at once, but for demonstration we do as if they were huge:
        traj.f_load_item(euler_result)
        euler_data = euler_result.data

        # Plot fancy 3d plot
        fig = plt.figure(idx)
        ax = fig.gca(projection='3d')
        x = euler_data[:,0]
        y = euler_data[:,1]
        z = euler_data[:,2]
        ax.plot(x, y, z, label='Initial Conditions: %s' % str(initial_conditions_exploration_array
        plt.legend()
        plt.show()

        # Now we free the data again (because we assume its huuuuuuge):
        del euler_data
        euler_result.f_empty()

    # Finally disable logging and close all log-files
    env.f_disable_logging()


if __name__ == '__main__':
    main()
```

Using the f_find_idx Function

Download: `example_08_f_find_idx.py`

Here you can see how you can search for particular parameter combinations and the corresponding run indices using the `f_find_idx()` function.

```
__author__ = 'Robert Meyer'

import os # For path names being viable under Windows and Linux

from pypet import Environment, cartesian_product
from pypet import pypetconstants


def multiply(traj):
    """Sophisticated simulation of multiplication"""
    z=traj.x*traj.y
    traj.f_add_result('z',z, comment='I am the product of two reals!')



# Create an environment that handles running
filename = os.path.join('hdf5', 'example_08.hdf5')
env = Environment(trajectory='Example08',filename=filename,
                  file_title='Example08',
                  comment='Another example!')
```

```
# Get the trajectory from the environment
traj = env.v_trajectory

# Add both parameters
traj.f_add_parameter('x', 1, comment='I am the first dimension!')
traj.f_add_parameter('y', 1, comment='I am the second dimension!')

# Explore the parameters with a cartesian product:
traj.f_explore(cartesian_product({'x':[1,2,3,4], 'y':[6,7,8]}))

# Run the simulation
env.f_run(multiply)

# We load all results
traj.f_load(load_results=pypetconstants.LOAD_DATA)

# And now we want to find som particular results, the ones where x was 2 or y was 8.
# Therefore, we use a lambda function
my_filter_predicate= lambda x,y: x==2 or y==8

# We can now use this lambda function to search for the run indexes associated with x==2 OR y==8.
# We need a list specifying the names of the parameters and the predicate to do this.
# Note that names need to be in the order as listed in the lambda function, here 'x' and 'y':
idx_iterator = traj.f_find_idx(['x','y'], my_filter_predicate)

# Now we can print the corresponding results:
print('The run names and results for parameter combinations with x==2 or y==8:')
for idx in idx_iterator:
    # We focus on one particular run. This is equivalent to calling `traj.f_as_run(idx)`.
    traj.v_idx=idx
    run_name = traj.v_as_run
    # and print everything nicely
    print('%s: x=%d, y=%d, z=%d' %(run_name, traj.x, traj.y, traj.crun.z))

# And we do not forget to set everything back to normal
traj.f_restore_default()

# Finally disable logging and close all log-files
env.f_disable_logging()
```

Accessing Results from All Runs at Once

Download: `example_10_get_items_from_all_runs.py`

Want to know how to access all data from results at once? Check out *f_get_from_runs()* and the code below:

```
__author__ = 'Robert Meyer'

from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
import numpy as np
import os # For path names working under Windows ans Linux

from pypet import Environment, cartesian_product
from pypet import pypetconstants


def multiply(traj):
    """Sophisticated simulation of multiplication"""
    z=traj.x * traj.y
    traj.f_add_result('z', z, comment='I am the product of two reals!')
```

```
# Create an environment that handles running
filename = os.path.join('hdf5', 'example_10.hdf5')
env = Environment(trajectory='Example10', filename=filename,
                  file_title='Example10',
                  comment='Another example!')

# Get the trajectory from the environment
traj = env.v_trajectory

# Add both parameters
traj.f_add_parameter('x', 1, comment='I am the first dimension!')
traj.f_add_parameter('y', 1, comment='I am the second dimension!')

# Explore the parameters with a cartesian product:
x_length = 12
y_length = 12
traj.f_explore(cartesian_product({'x': range(x_length), 'y': range(y_length)}))

# Run the simulation
env.f_run(multiply)

# We load all results
traj.f_load(load_results=pypetconstants.LOAD_DATA)

# We access the ranges for plotting
xs = traj.f_get('x').f_get_range()
ys = traj.f_get('y').f_get_range()

# Now we want to directly get all numbers z from all runs
# for plotting.
# We use `fast_access=True` to directly get access to
# the values.
# Moreover, since `f_get_from_runs` returns an ordered dictionary
# `values()` gives us all values already in the correct order of the runs.
zs = list(traj.f_get_from_runs(name='z', fast_access=True).values())
# We also make sure it's a list (because in python 3 ``value()`` returns an
# iterator instead of a list)

# Convert the lists to numpy 2D arrays
x_mesh = np.reshape(np.array(xs), (x_length, y_length))
y_mesh = np.reshape(np.array(ys), (x_length, y_length))
z_mesh = np.reshape(np.array(zs), (x_length, y_length))

# Make fancy 3D plot
fig=plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_wireframe(x_mesh, y_mesh, z_mesh, rstride=1, cstride=1)
plt.show()

# Finally disable logging and close all log-files
env.f_disable_logging()
```

Sharing Data during Multiprocessing

Here we show how data can be shared among multiple processes. Mind however, that this is conceptually a rather bad design since the single runs are no longer independent of each other. A better solution would be to simply return the data and sort it into a list during post-processing.

Download: `example_12_sharing_data_between_processes.py`

```
__author__ = 'Robert Meyer'

import multiprocessing as mp
import numpy as np
import os # For path names working under Windows and Linux

from pypet import Environment, cartesian_product


def multiply(traj, result_list):
    """Example of a sophisticated simulation that involves multiplying two values.

    This time we will store tha value in a shared list and only in the end add the result.

    :param traj:

        Trajectory containing
        the parameters in a particular combination,
        it also serves as a container for results.


    """
    z=traj.x*traj.y
    result_list[traj.v_idx] = z


def main():
    # Create an environment that handles running
    filename = os.path.join('hdf5', 'example_12.hdf5')
    env = Environment(trajectory='Multiplication',
                      filename=filename,
                      file_title='Example_12_Sharing_Data',
                      comment='The first example!',
                      continuable=False, # We have shared data in terms of a multiprocessing list,
                      # so we CANNOT use the continue feature.
                      multiproc=True,
                      ncores=2)

    # The environment has created a trajectory container for us
    traj = env.v_trajectory

    # Add both parameters
    traj.f_add_parameter('x', 1, comment='I am the first dimension!')
    traj.f_add_parameter('y', 1, comment='I am the second dimension!')

    # Explore the parameters with a cartesian product
    traj.f_explore(cartesian_product({'x':[1,2,3,4], 'y':[6,7,8]}))

    # We want a shared list where we can put all out results in. We use a manager for this:
    result_list = mp.Manager().list()
    # Let's make some space for potential results
    result_list[:] =[0 for _dummy in range(len(traj))]

    # Run the simulation
    env.f_run(multiply, result_list)

    # Now we want to store the final list as numpy array
    traj.f_add_result('z', np.array(result_list))

    # Finally let's print the result to see that it worked
    print(traj.z)

    #Disable logging and close all log-files
```

```
    env.f_disable_logging()

if __name__ == '__main__':
    main()
```

Lightweight Multiprocessing

Download: `example_16_multiproc_context.py`

This example shows you how to use a *MultiprocContext*.

```
__author__ = 'Robert Meyer'

import os
import multiprocessing as mp
import logging

from pypet import Trajectory, MultiprocContext


def manipulate_multiproc_safe(traj):
    """ Target function that manipulates the trajectory.

    Stores the current name of the process into the trajectory and
    **overwrites** previous settings.

    :param traj:

        Trajectory container with multiprocessing safe storage service

    """

    # Manipulate the data in the trajectory
    traj.last_process_name = mp.current_process().name
    # Store the manipulated data
    traj.results.f_store(store_data=3)  # Overwrites data on disk
    # Not recommended, here only for demonstration purposes :-)


def main():
    # We don't use an environment so we enable logging manually
    logging.basicConfig(level=logging.INFO)

    filename = os.path.join('hdf5','example_16.hdf5')
    traj = Trajectory(filename=filename, overwrite_file=True)

    # The result that will be manipulated
    traj.f_add_result('last_process_name', 'N/A',
                      comment='Name of the last process that manipulated the trajectory')

    with MultiprocContext(trajectory=traj, wrap_mode='LOCK') as mc:
        # The multiprocessing context manager wraps the storage service of the trajectory
        # and passes the wrapped service to the trajectory.
        # Also restores the original storage service in the end.
        # Moreover, wee need to use the `MANAGER_LOCK` wrapping because the locks
        # are pickled and send to the pool for all function executions

        # Start a pool of processes manipulating the trajectory
        iterable = (traj for x in range(20))
        pool = mp.Pool(processes=4)
        # Pass the trajectory and the function to the pool and execute it 20 times
        pool.map_async(manipulate_multiproc_safe, iterable)
```

```
        pool.close()
        # Wait for all processes to join
        pool.join()

    # Reload the data from disk and overwrite the existing result in RAM
    traj.results.f_load(load_data=3)
    # Print the name of the last process the trajectory was manipulated by
    print('The last process to manipulate the trajectory was: `%s`' % traj.last_process_name)


if __name__ == '__main__':
    main()
```

1.5.3 BRIAN Examples

Short BRIAN Example

Download: `example_07_brian_network.py`

Find an example usage with BRIAN below.

```
__author__ = 'Robert Meyer'

import logging
import os # For path names being viable under Windows and Linux

from pypet.environment import Environment
from pypet.brian.parameter import BrianParameter,BrianMonitorResult
from pypet.utils.explore import cartesian_product
# Don't do this at home:
from brian import *

# We define a function to set all parameter
def add_params(traj):
    """Adds all necessary parameters to `traj`."""

    # We set the BrianParameter to be the standard parameter
    traj.v_standard_parameter=BrianParameter
    traj.v_fast_access=True

    # Add parameters we need for our network
    traj.f_add_parameter('Sim.defaultclock', 0.01*ms)
    traj.f_add_parameter('Net.C',281*pF)
    traj.f_add_parameter('Net.gL',30*nS)
    traj.f_add_parameter('Net.EL',-70.6*mV)
    traj.f_add_parameter('Net.VT',-50.4*mV)
    traj.f_add_parameter('Net.DeltaT',2*mV)
    traj.f_add_parameter('Net.tauw',40*ms)
    traj.f_add_parameter('Net.a',4*nS)
    traj.f_add_parameter('Net.b',0.08*nA)
    traj.f_add_parameter('Net.I',.8*nA)
    traj.f_add_parameter('Net.Vcut',traj.VT+5*traj.DeltaT) # practical threshold condition
    traj.f_add_parameter('Net.N',50)

    eqs='''
    dvm/dt=(gL*(EL-vm)+gL*DeltaT*exp((vm-VT)/DeltaT)+I-w)/C : volt
    dw/dt=(a*(vm-EL)-w)/tauw : amp
    Vr:volt
    '''
    traj.f_add_parameter('Net.eqs', eqs)
    traj.f_add_parameter('reset', 'vm=Vr;w+=b')
```

```
# This is our job that we will execute
def run_net(traj):
    """Creates and runs BRIAN network based on the parameters in `traj`."""

    # We want to give every network a fresh start
    clear(True, True)

    defaultclock.dt=traj.defaultclock

    # We let BRIAN grasp the parameters from the local namespace
    C=traj.C
    gL=traj.gL
    EL=traj.EL
    VT=traj.VT
    DeltaT=traj.DeltaT
    tauw=traj.tauw
    a=traj.a
    b=traj.b
    I=traj.I
    Vcut=traj.Vcut
    N=traj.N

    eqs=traj.eqs

    # Create the Neuron Group
    neuron=NeuronGroup(N,model=eqs,threshold=Vcut,reset=traj.reset)
    neuron.vm=EL
    neuron.w=a*(neuron.vm-EL)
    neuron.Vr=linspace(-48.3*mV,-47.7*mV,N) # bifurcation parameter

    # Run the network initially for 100 milliseconds
    print 'Initial Run'
    run(100*msecond,report='text') # we discard the first spikes

    # Create a Spike Monitor
    MSpike=SpikeMonitor(neuron, delay = 1*ms)
    # Create a State Monitor for the membrane voltage, record from neurons 1-3
    MStateV = StateMonitor(neuron,'vm',record=[1,2,3])


    # Now record for 500 milliseconds
    print 'Measurement run'
    run(500*msecond,report='text')

    # Add the BRAIN monitors
    traj.v_standard_result = BrianMonitorResult
    traj.f_add_result('SpikeMonitor',MSpike)
    traj.f_add_result('StateMonitorV', MStateV)


def main():
    # Let's be very verbose!
    logging.basicConfig(level = logging.INFO)


    # Let's do multiprocessing this time with a lock (which is default)
    filename = os.path.join('hdf5', 'example_07.hdf5')
    env = Environment(trajectory='Example_07_BRIAN',
                      filename=filename,
                      file_title='Example_07_Brian',
                      comment = 'Go Brian!',
                      dynamically_imported_classes=[BrianMonitorResult, BrianParameter],
```

```
                      multiproc=True,
                      wrap_mode='QUEUE',
                      ncores=2)

    traj = env.v_trajectory

    # 1st a) add the parameters
    add_params(traj)

    # 1st b) prepare, we want to explore the different network sizes and different tauw time scal
    traj.f_explore(cartesian_product({traj.f_get('N').v_full_name:[50,60],
                            traj.f_get('tauw').v_full_name:[30*ms,40*ms]}))

    # 2nd let's run our experiment
    env.f_run(run_net)

    # You can take a look at the results in the hdf5 file if you want!

    # Finally disable logging and close all log-files
    env.f_disable_logging()


if __name__ == '__main__':
    main()
```

Large scale BRIAN simulation

This example involves a large scale simulation of a BRIAN network *Using BRIAN with pypet*. The example is taken from the Litwin-Kumar and Doiron paper from Nature neuroscience 2012.

It is split into three different modules: The *clusternet.py* file containing the network specification, the *runscript.py* file to start a simulation (you have to be patient, BRIAN simulations can take some time), and the *plotff.py* to plot the results of the parameter exploration, i.e. the Fano Factor as a function of the clustering parameter *R_ee*.

Download: `clusternet.py`

Download: `runscript.py`

Download: `plotff.py`

Clusternet

```
"""Module to run the clustered Neural Network Simulations as in Litwin-Kumar & Doiron 2012"""

__author__ = 'Robert Meyer'

import os
import numpy as np
import matplotlib.pyplot as plt

from pypet.trajectory import Trajectory
from pypet.brian.parameter import BrianParameter, BrianMonitorResult
from pypet.brian.network import NetworkComponent, NetworkRunner, NetworkAnalyser
from brian.stdunits import ms

from brian import NeuronGroup, rand, Connection, Equations, Network, SpikeMonitor, second, \
    raster_plot, show, StateMonitor, clear, reinit_default_clock


def _explored_parameters_in_group(traj, group_node):
    """Checks if one the parameters in `group_node` is explored.
```

```
    :param traj: Trajectory container
    :param group_node: Group node
    :return: `True` or `False`

    """
    explored = False
    for param in traj.f_get_explored_parameters():
            if param in group_node:
                explored = True
                break

    return explored

class CNNeuronGroup(NetworkComponent):
    """Class to create neuron groups.

    Creates two groups of excitatory and inhibitory neurons.

    """

    @staticmethod
    def add_parameters(traj):
        """Adds all neuron group parameters to `traj`."""
        assert(isinstance(traj,Trajectory))

        scale = traj.simulation.scale


        traj.v_standard_parameter = BrianParameter

        model_eqs = '''dV/dt= 1.0/tau_POST * (mu - V) + I_syn : 1
                       mu : 1
                       I_syn =  - I_syn_i + I_syn_e : Hz
                    '''

        conn_eqs = '''I_syn_PRE = x_PRE/(tau2_PRE-tau1_PRE) : Hz
                      dx_PRE/dt = -(normalization_PRE*y_PRE+x_PRE)*invtau1_PRE : 1
                      dy_PRE/dt = -y_PRE*invtau2_PRE : 1
                   '''

        traj.f_add_parameter('model.eqs', model_eqs,
                             comment='The differential equation for the neuron model')

        traj.f_add_parameter('model.synaptic.eqs', conn_eqs,
                             comment='The differential equation for the synapses. '
                                     'PRE will be replaced by `i` or `e` depending '
                                     'on the source population')

        traj.f_add_parameter('model.synaptic.tau1', 1*ms, comment = 'The decay time')
        traj.f_add_parameter('model.synaptic.tau2_e', 3*ms, comment = 'The rise time, excitatory')
        traj.f_add_parameter('model.synaptic.tau2_i', 2*ms, comment = 'The rise time, inhibitory')

        traj.f_add_parameter('model.V_th', 1.0, comment = "Threshold value")
        traj.f_add_parameter('model.reset_func', 'V=0.0',
                             comment = "String representation of reset function")
        traj.f_add_parameter('model.refractory', 5*ms, comment = "Absolute refractory period")

        traj.f_add_parameter('model.N_e', int(4000*scale), comment = "Amount of excitatory neurons
        traj.f_add_parameter('model.N_i', int(1000*scale), comment = "Amount of inhibitory neurons

        traj.f_add_parameter('model.tau_e', 15*ms, comment = "Membrane time constant, excitatory"
        traj.f_add_parameter('model.tau_i', 10*ms, comment = "Membrane time constant, inhibitory"
```

```
        traj.f_add_parameter('model.mu_e_min', 1.1, comment = "Lower bound for bias, excitatory")
        traj.f_add_parameter('model.mu_e_max', 1.2, comment = "Upper bound for bias, excitatory")

        traj.f_add_parameter('model.mu_i_min', 1.0, comment = "Lower bound for bias, inhibitory")
        traj.f_add_parameter('model.mu_i_max', 1.05, comment = "Upper bound for bias, inhibitory"


    @staticmethod
    def _build_model_eqs(traj):
        """Computes model equations for the excitatory and inhibitory population.

        Equation objects are created by fusing `model.eqs` and `model.synaptic.eqs`
        and replacing `PRE` by `i` (for inhibitory) or `e` (for excitatory) depending
        on the type of population.

        :return: Dictionary with 'i' equation object for inhibitory neurons and 'e' for excitator

        """
        model_eqs = traj.model.eqs
        post_eqs={}
        for name_post in ['i','e']:
            variables_dict ={}
            new_model_eqs=model_eqs.replace('POST', name_post)
            for name_pre in ['i', 'e']:
                conn_eqs = traj.model.synaptic.eqs
                new_conn_eqs = conn_eqs.replace('PRE', name_pre)
                new_model_eqs += new_conn_eqs

                tau1 = traj.model.synaptic['tau1']
                tau2 = traj.model.synaptic['tau2_'+name_pre]

                normalization = (tau1-tau2) / tau2
                invtau1=1.0/tau1
                invtau2 = 1.0/tau2

                variables_dict['invtau1_'+name_pre] = invtau1
                variables_dict['invtau2_'+name_pre] = invtau2
                variables_dict['normalization_'+name_pre] = normalization
                variables_dict['tau1_'+name_pre] = tau1
                variables_dict['tau2_'+name_pre] = tau2

            variables_dict['tau_'+name_post] = traj.model['tau_'+name_post]

            post_eqs[name_post] = Equations(new_model_eqs, **variables_dict)

        return post_eqs

    def pre_build(self, traj, brian_list, network_dict):
        """Pre-builds the neuron groups.

        Pre-build is only performed if none of the
        relevant parameters is explored.

        :param traj: Trajectory container

        :param brian_list:

            List of objects passed to BRIAN network constructor.

            Adds:

            Inhibitory neuron group
```

```
            Excitatory neuron group

        :param network_dict:

            Dictionary of elements shared among the components

            Adds:

            'neurons_i': Inhibitory neuron group

            'neurons_e': Excitatory neuron group

        """
        self._pre_build = not _explored_parameters_in_group(traj, traj.parameters.model)

        if self._pre_build:
            self._build_model(traj, brian_list, network_dict)


    def build(self, traj, brian_list, network_dict):
        """Builds the neuron groups.

        Build is only performed if neuron group was not
        pre-build before.

        :param traj: Trajectory container

        :param brian_list:

            List of objects passed to BRIAN network constructor.

            Adds:

            Inhibitory neuron group

            Excitatory neuron group

        :param network_dict:

            Dictionary of elements shared among the components

            Adds:

            'neurons_i': Inhibitory neuron group

            'neurons_e': Excitatory neuron group

        """
        if not hasattr(self, '_pre_build') or not self._pre_build:
            self._build_model(traj, brian_list, network_dict)


    def _build_model(self, traj, brian_list, network_dict):
        """Builds the neuron groups from `traj`.

        Adds the neuron groups to `brian_list` and `network_dict`.

        """

        model = traj.parameters.model

        # Create the equations for both models
```

```
        eqs_dict = self._build_model_eqs(traj)

        # Create inhibitory neurons
        eqs_i = eqs_dict['i']
        neurons_i = NeuronGroup(N=model.N_i,
                                model = eqs_i,
                                threshold=model.V_th,
                                reset=model.reset_func,
                                refractory=model.refractory,
                                freeze=True,
                                compile=True,
                                method='Euler')

        # Create excitatory neurons
        eqs_e = eqs_dict['e']
        neurons_e = NeuronGroup(N=model.N_e,
                                model = eqs_e,
                                threshold=model.V_th,
                                reset=model.reset_func,
                                refractory=model.refractory,
                                freeze=True,
                                compile=True,
                                method='Euler')


        # Set the bias terms
        neurons_e.mu =rand(model.N_e) * (model.mu_e_max - model.mu_e_min) + model.mu_e_min
        neurons_i.mu =rand(model.N_i) * (model.mu_i_max - model.mu_i_min) + model.mu_i_min

        # Set initial membrane potentials
        neurons_e.V = rand(model.N_e)
        neurons_i.V = rand(model.N_i)

        # Add both groups to the `brian_list` and the `network_dict`
        brian_list.append(neurons_i)
        brian_list.append(neurons_e)
        network_dict['neurons_e']=neurons_e
        network_dict['neurons_i']=neurons_i



class CNConnections(NetworkComponent):
    """Class to connect neuron groups.

    In case of no clustering `R_ee=1,0` there are 4 connection instances (i->i, i->e, e->i, e->e)

    Otherwise there are 3 + 3*N_c-2 connections with N_c the number of clusters
    (i->i, i->e, e->i, N_c conns within cluster, 2*N_c-2 connections from cluster to outside).

    """

    @staticmethod
    def add_parameters(traj):
        """Adds all neuron group parameters to `traj`."""
        assert(isinstance(traj,Trajectory))

        traj.v_standard_parameter = BrianParameter
        scale = traj.simulation.scale

        traj.f_add_parameter('connections.R_ee', 1.0, comment='Scaling factor for clustering')

        traj.f_add_parameter('connections.clustersize_e', 80, comment='Size of a cluster')
        traj.f_add_parameter('connections.strength_factor', 1.9,
```

```
                            comment='Factor for scaling cluster weights')

        traj.f_add_parameter('connections.p_ii', 0.5,
                             comment='Connection probability from inhibitory to inhibitory' )
        traj.f_add_parameter('connections.p_ei', 0.5,
                             comment='Connection probability from inhibitory to excitatory' )
        traj.f_add_parameter('connections.p_ie', 0.5,
                             comment='Connection probability from excitatory to inhibitory' )
        traj.f_add_parameter('connections.p_ee', 0.2,
                             comment='Connection probability from excitatory to excitatory' )

        traj.f_add_parameter('connections.J_ii', 0.057/np.sqrt(scale),
                             comment='Connection strength from inhibitory to inhibitory')
        traj.f_add_parameter('connections.J_ei', 0.045/np.sqrt(scale),
                             comment='Connection strength from inhibitory to excitatroy')
        traj.f_add_parameter('connections.J_ie', 0.014/np.sqrt(scale),
                             comment='Connection strength from excitatory to inhibitory')
        traj.f_add_parameter('connections.J_ee', 0.024/np.sqrt(scale),
                             comment='Connection strength from excitatory to excitatory')


    def pre_build(self, traj, brian_list, network_dict):
        """Pre-builds the connections.

        Pre-build is only performed if none of the
        relevant parameters is explored and the relevant neuron groups
        exist.

        :param traj: Trajectory container

        :param brian_list:

            List of objects passed to BRIAN network constructor.

            Adds:

            Connections, amount depends on clustering

        :param network_dict:

            Dictionary of elements shared among the components

            Expects:

            'neurons_i': Inhibitory neuron group

            'neurons_e': Excitatory neuron group

            Adds:

            Connections, amount depends on clustering

        """
        self._pre_build = not _explored_parameters_in_group(traj, traj.parameters.connections)

        self._pre_build = (self._pre_build and 'neurons_i' in network_dict and
                           'neurons_e' in network_dict)

        if self._pre_build:
            self._build_connections(traj, brian_list, network_dict)


    def build(self, traj, brian_list, network_dict):
```

```
        """Builds the connections.

        Build is only performed if connections have not
        been pre-build.

        :param traj: Trajectory container

        :param brian_list:

            List of objects passed to BRIAN network constructor.

            Adds:

            Connections, amount depends on clustering

        :param network_dict:

            Dictionary of elements shared among the components

            Expects:

            'neurons_i': Inhibitory neuron group

            'neurons_e': Excitatory neuron group

            Adds:

            Connections, amount depends on clustering

        """
        if not hasattr(self, '_pre_build') or not self._pre_build:
            self._build_connections(traj, brian_list, network_dict)


    def _build_connections(self, traj, brian_list, network_dict):
        """Connects neuron groups `neurons_i` and `neurons_e`.

        Adds all connections to `brian_list` and adds a list of connections
        with the key 'connections' to the `network_dict`.

        """

        connections = traj.connections

        neurons_i = network_dict['neurons_i']
        neurons_e = network_dict['neurons_e']

        print 'Connecting ii'
        self.conn_ii = Connection(neurons_i,neurons_i, state='y_i',
                                  weight=connections.J_ii,
                                  sparseness=connections.p_ii)

        print 'Connecting ei'
        self.conn_ei = Connection(neurons_i,neurons_e,state='y_i',
                                  weight=connections.J_ei,
                                  sparseness=connections.p_ei)

        print 'Connecting ie'
        self.conn_ie = Connection(neurons_e,neurons_i,state='y_e',
                                  weight=connections.J_ie,
                                  sparseness=connections.p_ie)

        conns_list = [self.conn_ii, self.conn_ei, self.conn_ie]
```

```
    if connections.R_ee > 1.0:
        # If we come here we want to create clusters

        cluster_list=[]
        cluster_conns_list=[]
        model=traj.model

        # Compute the number of clusters
        clusters = model.N_e/connections.clustersize_e
        traj.f_add_derived_parameter('connections.clusters', clusters, comment='Number of clus

        # Compute outgoing connection probability
        p_out = (connections.p_ee*model.N_e) / \
                (connections.R_ee*connections.clustersize_e+model.N_e- connections.clustersize

        # Compute within cluster connection probability
        p_in = p_out * connections.R_ee

        # We keep these derived parameters
        traj.f_add_derived_parameter('connections.p_ee_in', p_in ,
                                     comment='Connection prob within cluster')
        traj.f_add_derived_parameter('connections.p_ee_out', p_out ,
                                     comment='Connection prob to outside of cluster')


        low_index = 0
        high_index = connections.clustersize_e
        # Iterate through cluster and connect within clusters and to the rest of the neurons
        for irun in range(clusters):

            cluster = neurons_e[low_index:high_index]

            # Connections within cluster
            print 'Connecting ee cluster #%d of %d' % (irun, clusters)
            conn = Connection(cluster,cluster,state='y_e',
                              weight=connections.J_ee*connections.strength_factor,
                              sparseness=p_in)

            cluster_conns_list.append(conn)

            # Connections reaching out from cluster
            # A cluster consists of `clustersize_e` neurons with consecutive indices.
            # So usually the outside world consists of two groups, neurons with lower
            # indices than the cluster indices, and neurons with higher indices.
            # Only the clusters at the index boundaries project to neurons with only either
            # lower or higher indices
            if low_index > 0:
                rest_low = neurons_e[0:low_index]
                print 'Connecting cluster with other neurons of lower index'
                low_conn = Connection(cluster,rest_low,state='y_e',
                                      weight=connections.J_ee,
                                      sparseness=p_out)

                cluster_conns_list.append(low_conn)

            if high_index < model.N_e:
                rest_high = neurons_e[high_index:model.N_e]
                print 'Connecting cluster with other neurons of higher index'
                high_conn = Connection(cluster,rest_high,state='y_e',
                                      weight=connections.J_ee,
                                      sparseness=p_out)
```

```
                    cluster_conns_list.append(high_conn)

                low_index=high_index
                high_index+=connections.clustersize_e

            self.cluster_conns=cluster_conns_list
            conns_list+=cluster_conns_list
        else:
            # Here we don't cluster and connection probabilities are homogeneous
            print 'Connectiong ee'
            self.conn_ee = Connection(neurons_e,neurons_e,state='y_e',
                                      weight=connections.J_ee,
                                      sparseness=connections.p_ee)

            conns_list.append(self.conn_ee)


        # Add the connections to the `brian_list` and the network dict
        brian_list.extend(conns_list)
        network_dict['connections'] = conns_list


class CNNetworkRunner(NetworkRunner):
    """Runs the network experiments.

    Adds two BrianParameters, one for an initial run, and one for a run
    that is actually measured.

    """


    def add_parameters(self, traj):
        """Adds all necessary parameters to `traj` container."""
        par= traj.f_add_parameter(BrianParameter,'simulation.durations.initial_run', 500*ms,
                              comment='Initialisation run for more realistic '
                                      'measurement conditions.')

        par.v_annotations.order=0
        par=traj.f_add_parameter(BrianParameter,'simulation.durations.measurement_run', 2000*ms,
                              comment='Measurement run that is considered for '
                                      'statistical evaluation')
        par.v_annotations.order=1



class CNFanoFactorComputer(NetworkAnalyser):
    """Computes the FanoFactor if the MonitorAnalyser has extracted data"""

    def add_parameters(self, traj):
        traj.f_add_parameter('analysis.statistics.time_window', 0.1 , 'Time window for FF computat
        traj.f_add_parameter('analysis.statistics.neuron_ids', tuple(range(500)),
                              comment= 'Neurons to be taken into account to compute FF')

    @staticmethod
    def _compute_fano_factor(spike_table, neuron_id, time_window, start_time, end_time):
        """Computes Fano Factor for one neuron.

        :param spike_table:

            DataFrame containing the spiketimes of all neurons

        :param neuron_id:
```

```
            Index of neuron for which FF is computed

        :param time_window:

            Length of the consecutive time windows to compute the FF

        :param start_time:

            Start time of measurement to consider

        :param end_time:

            End time of measurement to consider

        :return:

            Fano Factor (float) or
            returns 0 if mean firing activity is 0.

        """
        assert(end_time >= start_time+time_window)

        # Number of time bins
        bins = (end_time-start_time)/float(time_window)
        bins = int(np.floor(bins))

        # Arrays for binning of spike counts
        binned_spikes = np.zeros(bins)

        # DataFrame only containing spikes of the particular neuron
        spike_table_neuron = spike_table[spike_table.neuron==neuron_id]

        for bin in range(bins):
            # We iterate over the bins to calculate the spike counts
            lower_time = start_time+time_window*bin
            upper_time = start_time+time_window*(bin+1)

            # Filter the spikes
            spike_table_interval = spike_table_neuron[spike_table_neuron.spiketimes >= lower_time
            spike_table_interval = spike_table_interval[spike_table_interval.spiketimes < upper_t

            # Add count to bins
            spikes = len(spike_table_interval)
            binned_spikes[bin]=spikes


        var = np.var(binned_spikes)
        avg = np.mean(binned_spikes)

        if avg > 0:
            return var/float(avg)
        else:
            return 0

    @staticmethod
    def _compute_mean_fano_factor( neuron_ids, spike_table, time_window, start_time, end_time):
        """Computes average Fano Factor over many neurons.

        :param neuron_ids:

            List of neuron indices to average over
```

```
        :param spike_table:

            DataFrame containing the spiketimes of all neurons

        :param time_window:

            Length of the consecutive time windows to compute the FF

        :param start_time:

            Start time of measurement to consider

        :param end_time:

            End time of measurement to consider

        :return:

            Average fano factor

        """
        ffs = np.zeros(len(neuron_ids))

        for idx, neuron_id in enumerate(neuron_ids):
            ff=CNFanoFactorComputer._compute_fano_factor(
                            spike_table, neuron_id, time_window, start_time, end_time)
            ffs[idx]=ff

        mean_ff = np.mean(ffs)
        return mean_ff

    def analyse(self, traj, network, current_subrun, subrun_list, network_dict):
        """Calculates average Fano Factor of a network.

        :param traj:

            Trajectory container

            Expects:

            `results.monitors.spikes_e`: Data from SpikeMonitor for excitatory neurons

            Adds:

            `results.statistics.mean_fano_factor`: Average Fano Factor

        :param network:

            The BRIAN network

        :param current_subrun:

            BrianParameter

        :param subrun_list:

            Upcoming subruns, analysis is only performed if subruns is empty,
            aka the final subrun has finished.

        :param network_dict:

            Dictionary of items shared among componetns
```

```
        """
        #Check if we finished all subruns
        if len(subrun_list)==0:
            spikes_e = traj.results.monitors.spikes_e

            time_window = traj.parameters.analysis.statistics.time_window
            start_time = float(traj.parameters.simulation.durations.initial_run)
            end_time = start_time+float(traj.parameters.simulation.durations.measurement_run)
            neuron_ids = traj.parameters.analysis.statistics.neuron_ids

            mean_ff = self._compute_mean_fano_factor(
                neuron_ids, spikes_e.spikes, time_window, start_time, end_time)

            traj.f_add_result('statistics.mean_fano_factor', mean_ff, comment='Average Fano '
                                                                              'Factor over all '
                                                                              'exc neurons')

            print 'R_ee: %f, Mean FF: %f' % (traj.R_ee, mean_ff)


class CNMonitorAnalysis(NetworkAnalyser):
    """Adds monitors for recoding and plots the monitor output."""

    @staticmethod
    def add_parameters( traj):
        traj.f_add_parameter('analysis.neuron_records',(0,1,100,101),
                             comment='Neuron indices to record from.')
        traj.f_add_parameter('analysis.plot_folder',
                             os.path.join('experiments', 'example_11', 'PLOTS'),
                             comment='Folder for plots')
        traj.f_add_parameter('analysis.show_plots', 0, comment='Whether to show plots.')
        traj.f_add_parameter('analysis.make_plots', 1, comment='Whether to make plots.')

    def add_to_network(self, traj, network, current_subrun, subrun_list, network_dict):
        """Adds monitors to the network if the measurement run is carried out.

        :param traj: Trajectory container

        :param network: The BRIAN network

        :param current_subrun: BrianParameter

        :param subrun_list: List of coming subrun_list

        :param network_dict:

            Dictionary of items shared among the components

            Expects:

            'neurons_e': Excitatory neuron group

            Adds:

            'monitors': List of monitors

                0. SpikeMonitor of excitatory neurons

                1. StateMonitor of membrane potential of some excitatory neurons
                (specified in `neuron_records`)

                2. StateMonitor of excitatory synaptic currents of some excitatory neurons
```

```
            3. State monitor of inhibitory currents of some excitatory neurons

        """
        if current_subrun.v_annotations.order == 1:
            self._add_monitors(traj, network, network_dict)

    def _add_monitors(self, traj,  network, network_dict):
        """Adds monitors to the network"""

        neurons_e = network_dict['neurons_e']

        monitor_list = []

        # Spiketimes
        self.spike_monitor = SpikeMonitor(neurons_e, delay=0*ms)
        monitor_list.append(self.spike_monitor)

        # Membrane Potential
        self.V_monitor = StateMonitor(neurons_e,'V',
                                              record=list(traj.neuron_records))

        monitor_list.append(self.V_monitor)

        # Exc. syn .Current
        self.I_syn_e_monitor = StateMonitor(neurons_e, 'I_syn_e',
                                            record=list(traj.neuron_records))
        monitor_list.append(self.I_syn_e_monitor)

        # Inh. syn. Current
        self.I_syn_i_monitor = StateMonitor(neurons_e, 'I_syn_i',
                                            record=list(traj.neuron_records))
        monitor_list.append(self.I_syn_i_monitor)

        # Add monitors to network and dictionary
        network.add(*monitor_list)
        network_dict['monitors'] = monitor_list

    def _make_folder(self, traj):
        """Makes a subfolder for plots.

        :return: Path name to print folder

        """
        print_folder = os.path.join(traj.analysis.plot_folder,
                                    traj.v_trajectory_name, traj.v_crun)
        print_folder = os.path.abspath(print_folder)
        if not os.path.isdir(print_folder):
            os.makedirs(print_folder)

        return print_folder

    def _plot_result(self, traj, result_name):
        """Plots a state variable graph for several neurons into one figure"""
        result = traj.f_get(result_name)
        values = result.values
        varname = result.varname
        unit = result.unit
        times = result.times

        record = result.record

        for idx, celia_neuron in enumerate(record):
            plt.subplot(len(record), 1, idx+1)
```

```
            plt.plot(times, values[idx,:])
            if idx==0:
                plt.title('%s' % varname)
            if idx==1:
                plt.ylabel('%s/%s' % ( varname,unit))
            if idx == len(record)-1:
                plt.xlabel('t/ms')

    def _print_graphs(self, traj):
        """Makes some plots and stores them into subfolders"""
        print_folder = self._make_folder(traj)

        # If we use BRIAN's own raster_plot functionality we
        # need to sue the SpikeMonitor directly
        raster_plot(self.spike_monitor, newfigure=True, xlabel='t', ylabel='Exc. Neurons',
                    title='Spike Raster Plot')

        filename=os.path.join(print_folder,'spike.png')

        print 'Current plot: %s ' % filename
        plt.savefig(filename)
        plt.close()

        fig=plt.figure()
        self._plot_result(traj, 'monitors.V')
        filename=os.path.join(print_folder,'V.png')
        print 'Current plot: %s ' % filename
        fig.savefig(filename)
        plt.close()

        plt.figure()
        self._plot_result(traj, 'monitors.I_syn_e')
        filename=os.path.join(print_folder,'I_syn_e.png')
        print 'Current plot: %s ' % filename
        plt.savefig(filename)
        plt.close()

        plt.figure()
        self._plot_result(traj, 'monitors.I_syn_i')
        filename=os.path.join(print_folder,'I_syn_i.png')
        print 'Current plot: %s ' % filename
        plt.savefig(filename)
        plt.close()

        if not traj.analysis.show_plots:
            plt.close('all')
        else:
            plt.show()


    def analyse(self, traj, network, current_subrun, subrun_list, network_dict):
        """Extracts monitor data and plots.

        Data extraction is done if all subruns have been completed,
        i.e. `len(subrun_list)==0`

        First, extracts results from the monitors and stores them into `traj`.

        Next, uses the extracted data for plots.

        :param traj:

            Trajectory container
```

```
            Adds:

            Data from monitors

        :param network: The BRIAN network

        :param current_subrun: BrianParameter

        :param subrun_list: List of coming subruns

        :param network_dict: Dictionary of items shared among all components

        """
        if len(subrun_list)==0:

            traj.f_add_result(BrianMonitorResult, 'monitors.spikes_e', self.spike_monitor,
                              comment = 'The spiketimes of the excitatory population')

            traj.f_add_result(BrianMonitorResult, 'monitors.V', self.V_monitor,
                              comment = 'Membrane voltage of four neurons from 2 clusters')

            traj.f_add_result(BrianMonitorResult, 'monitors.I_syn_e', self.I_syn_e_monitor,
                              comment = 'I_syn_e of four neurons from 2 clusters')

            traj.f_add_result(BrianMonitorResult, 'monitors.I_syn_i', self.I_syn_i_monitor,
                              comment = 'I_syn_i of four neurons from 2 clusters')

            print 'Plotting'

            if traj.parameters.analysis.make_plots:
                self._print_graphs(traj)
```

Runscript

```
"""Starting script to run a network simulation of the clustered network
by Litwin-Kumar and Doiron (Nature neuroscience 2012).

The network has been implemented using the *pypet* network framework.

"""

__author__ = 'Robert Meyer'

import numpy as np
import os # To allow path names work under Windows and Linux

from pypet.environment import Environment
from pypet.brian.network import NetworkManager, run_network

from clusternet import CNMonitorAnalysis, CNNeuronGroup, CNNetworkRunner, CNConnections,\
    CNFanoFactorComputer


def main():
    filename = os.path.join('hdf5', 'Clustered_Network.hdf5')
    env = Environment(trajectory='Clustered_Network',
                      add_time=False,
                      filename=filename,
                      continuable=False,
                      lazy_debug=False,
                      multiproc=True,
```

```
                      ncores=2,
                      use_pool=False, # We cannot use a pool, our network cannot be pickled
                      wrap_mode='QUEUE',
                      overwrite_file=True)

    #Get the trajectory container
    traj = env.v_trajectory

    # We introduce a `meta` parameter that we can use to easily rescale our network
    scale = 0.5  # To obtain the results from the paper scale this to 1.0
    # Be aware that your machine will need a lot of memory then!
    traj.f_add_parameter('simulation.scale', scale,
            comment='Meta parameter that can scale default settings. '
                    'Rescales number of neurons and connections strenghts, but '
                    'not the clustersize.')


    # We create a Manager and pass all our components to the Manager.
    # Note the order, CNNeuronGroups are scheduled before CNConnections,
    # and the Fano Factor computation depends on the CNMonitorAnalysis
    clustered_network_manager = NetworkManager(network_runner=CNNetworkRunner(),
                                component_list=(CNNeuronGroup(), CNConnections()),
                                analyser_list=(CNMonitorAnalysis(),CNFanoFactorComputer()))




    # Add original parameters (but scaled according to `scale`)
    clustered_network_manager.add_parameters(traj)

    # We need `tolist` here since our parameter is a python float and not a
    # numpy float.
    explore_list = np.arange(1.0, 2.6, 0.2).tolist()
    # Explore different values of `R_ee`
    traj.f_explore({'R_ee' : explore_list})

    # Pre-build network components
    clustered_network_manager.pre_build(traj)


    # Run the network simulation
    traj.f_store() # Let's store the parameters already before the run
    env.f_run(clustered_network_manager.run_network)

    # Finally disable logging and close all log-files
    env.f_disable_logging()


if __name__=='__main__':
    main()
```

Plotff

```
"""Script to plot the fano factor graph for a given simulation
stored as a trajectory to an HDF5 file.

"""

__author__ = 'Robert Meyer'

import os
```

```
import matplotlib.pyplot as plt

from pypet import Trajectory, Environment
from pypet.brian.parameter import BrianMonitorResult, BrianParameter, BrianDurationParameter



def main():

    filename = os.path.join('hdf5', 'Clustered_Network.hdf5')
    # If we pass a filename to the trajectory a new HDF5StorageService will
    # be automatically created
    traj = Trajectory(filename=filename,
                      dynamically_imported_classes=[BrianDurationParameter,
                                                    BrianMonitorResult,
                                                    BrianParameter])

    # Let's create and fake environment to enable logging:
    env = Environment(traj, do_single_runs=False)


    # Load the trajectory, but onyl laod the skeleton of the results
    traj.f_load(index=-1, load_parameters=2, load_derived_parameters=2, load_results=1)

    # Find the result instances related to the fano factor
    fano_dict = traj.f_get_from_runs('mean_fano_factor', fast_access=False)

    # Load the data of the fano factor results
    ffs = fano_dict.values()
    traj.f_load_items(ffs)

    # Extract all values and R_ee values for each run
    ffs_values = [x.f_get() for x in ffs]
    Rees = traj.f_get('R_ee').f_get_range()

    # Plot average fano factor as a function of R_ee
    plt.plot(Rees, ffs_values)
    plt.xlabel('R_ee')
    plt.ylabel('Avg. Fano Factor')
    plt.show()

    # Finally disable logging and close all log-files
    env.f_disable_logging()


if __name__ == '__main__':
    main()
```

1.6 Optimization Tips

1.6.1 Group your Results into Buckets/Sets

HDF5 has a hard time managing nodes with more than 20,000 children. Accordingly, file I/O and reading or writing data can become very inefficient if one of your trajectory groups has more than 20,000 children. For instance, this may happen to you if you explore many runs.

Suppose in every run you add the following result:

```
>>> traj.f_add_result('some_group.$.z', 42, comment='Universal answer.')
```

If this line is executed in each of your, let's say 100,000 runs, the node `some_group` will have at least 100k children. Hence, storage and loading becomes extremely slow.

The simplest way around this problem is to group your results into buckets using the `'$set'` wildcard, see also *More on Wildcards*. Accordingly, your result addition becomes:

```
>>> traj.f_add_result('some_group.$set.$.z', 42, comment='Universal answer.')
```

Hence, even running 100k runs, *some_group* has only 100 children, each having only 1000 children themselves.

1.6.2 Huge Explorations

Yet, this approach will still fall short in case you have parameter exploration of more than 1,000,000 runs, because loading meta-data of your trajectory may already take more than a minute. And this can be annoying. In case of such huge explorations, I would advise you to tailor your parameter space and split it among several individual trajectories.

1.6.3 Collect Small Results

In case you compute only small results during your runs, like a single value, but you do this quite often (100k+), it might be more convenient to return the result instead of storing it into the trajectory directly. As a consequence, you can collect these single values later on during the post-processing phase and store all of them together into a single result. This has also been done for the estimated firing rate in the *Tutorial*.

1.6.4 Many and Fast Single Runs

In case you perform many single runs and milliseconds matter, use a pool (`use_pool=True`) in combination with a queue (`wrap_mode='QUEUE'`, see *Multiprocessing*) or the even faster - but potentially unreliable - method of using a shared pipe (`wrape_mode='PIPE'`). Moreover, to avoid re-pickling of unnecessary data of your trajectory, store and remove all data that is not needed during single runs.

For instance, if you don't really need config data during the runs, use the following **before** calling the environment's *f_run()* function:

```
traj.f_store()
traj.config.f_remove(recursive=True)
```

This may save a couple of milliseconds each run because the config data no longer needs to be pickled and send over the queue for storage.

Moreover, you can further avoid unnecessary pickling for the pool by setting `freeze_pool_input=True`. Accordingly, the trajectory, your target function, and all additional arguments are passed to each pool process at initialisation and not for each run individually. However, in order to use this feature, you must make sure that neither your target function nor the additional arguments are mutated over the course of your runs.

1.7 FAQs and Known Issues

1.7.1 Tools

Q: How can I open and inspect an HDF5 file created by *pypet*?

A: For inspection I mostly use these two tools: HDFview and ViTables.

1.7.2 Performance Issues

Q: Exploring many runs (10k+) *pypet* becomes incredibly slow when it comes to loading and storing data!?

> **A:** HDF5 has a hard time managing nodes with many children. To avoid this simply group your result into buckets using the *'$set'* wildcard. See also the *Optimization Tips*.

Q: *pypet* produces enormously large files of several Gigabytes despite them containing almost no data!?

> **A:** Your HDF5 version is too old (most likely you are using 1.8.5). Please update to 1.8.9 or newer.

1.7.3 Infinite Loops

Q: My program does not terminate (i.e. it does not crash but runs forever) when I use *pypet* in multiprocessing mode in combination with *matplotlib* and *savefig*!?

> **A:** *Matplotlib* uses *numpy* for linear algebra operations, these operations are often necessary when plotting. So, to solve the issues take a look at the next question.

Q: My program does not terminate (i.e. it does not crash but runs forever) when I use *pypet* in multiprocessing mode in combination with *numpy* and *linalg.inv* or some similar function!?

> **A:** Numpy uses openBLAS (http://www.openblas.net/) to solve linear algebra operations. Yet, there are many issues with openBLAS and Python multiprocessing. To resolve this set the environment variables `OPENBLAS_NUM_THREADS=1` and `OMP_NUM_THREADS=1`.

1.7.4 Crashes and Errors

Q: GitPython does not work. If I specify my repository `git_repository='./myrepo'`, pypet crashes with an `AttributeError: 'Repo' object has no attribute 'index'`. What should I do?

> **A:** You probably have an older version of GitPython (likely 0.1.7), install a newer one. If `pip install GitPython` still downloads the old version, try `pip install --pre GitPython` or if you simply want to upgrade, use `pip install --upgrade --pre GitPython`.

Q: My program crashes with `TypeError: [..] dtype: float64 its type is <class 'pandas.core.series.Series'>.`!?

> **A:** You are using pandas version `0.13.x`. Unfortunately, pandas performs some unwanted upcasting that cannot be handled by *pypet* (see https://github.com/pydata/pandas/issues/6526/). This unwanted upcasting did not happen in previous pandas versions and will be, or more precisely, has already been removed in the next pandas version. So either downgrade pandas to version `0.12.0` or upgrade to `0.14.1` or newer.

Q: My program crashes if I try to store a Trajectory containing an ArrayParameter!?

> **A:** Look at the previous answer, you are using pandas `0.13.x`, please up or downgrade your pandas package.

1.7.5 Other Problems

Q: If I create and environment in an *IPython* console everything becomes gibberish!?

> **A:** Pypet will redirect `stdout` to files. Unfortunately, this messes with the *IPython* console. To avoid this simply disable logging of this stream setting the `log_stdout` option to `False`: `env = Environment( ..., log_stdout=False, ...)`.

Q: I have large data sets that are not stored if I use multiprocessing and the lock wrapping!?

> **A:** Probably, you use an older HDF5 version (`< 1.8.7`) that does not allow simultaneous openings of a single HDF5 file. Either install a newer version or switch to queue wrapping.

CHAPTER 2

Miscellaneous

2.1 Publication Information

2.1.1 Citation Policy

If you use *pypet* in your research, it would be very kind of you to cite this in your amazing work. A research article about *pypet* is currently in preparation which will be the basis for citations in the future. In the meantime you can cite the software as given below. For *bibtex* you can use:

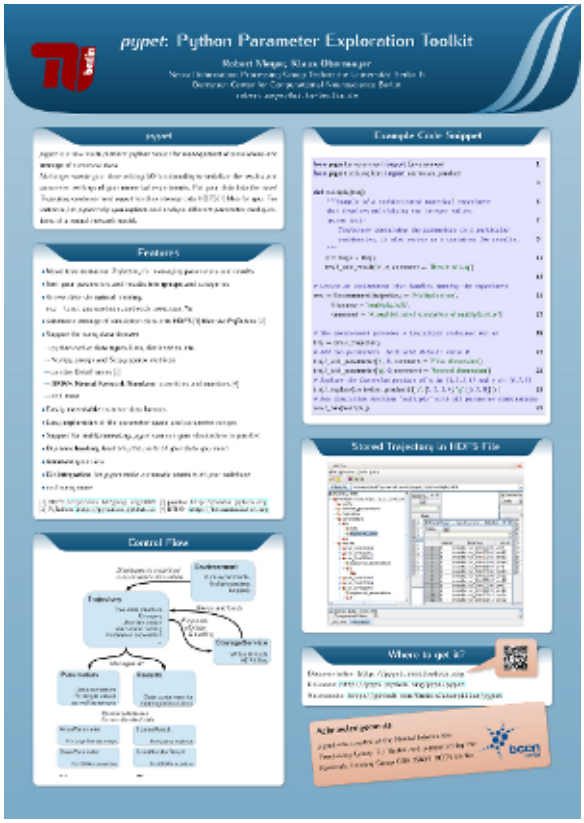
```
@misc{rmeyer2015,
    author = {Robert Meyer and Klaus Obermayer},
    year = {2015},
    title = {pypet: {T}he {P}ython {P}arameter {E}xploration {T}oolkit},
    note = {\url{http://pypet.readthedocs.org/}},
    institution = {Technische Universität Berlin, Neural Information Processing Group}
}
```

Otherwise you can cite it as:

- Robert Meyer and Klaus Obermayer. *pypet*: The Python Parameter Exploration Toolkit, 2015. http://pypet.readthedocs.org/.

2.1.2 Brain Days and EuroPython Poster

There is a poster about *pypet* that was shown at the Berlin Brain Days 2013 and the EuroPython 2014.



Download:

- CLICK ME for PDF DOWNLOAD
- CLICK ME for PNG DOWNLOAD

2.2 Acknowledgments

- Thanks to Robert Pröpper and Philipp Meier for answering all my python questions

  You might want to check out their SpykeViewer tool for visualization of MEA recordings and NEO data

- Thanks to Owen Mackwood for his SNEP toolbox which provided the initial ideas for this project
- Thanks to the BCCN Berlin, the Research Training Group GRK 1589/1, and the Neural Information Processing Group for support

2.3 Tests

Tests can be found in *pypet/tests*. Note that they involve heavy file IO and you need privileges to write files to a temporary folder. The test suites will make use of the `tempfile.gettempdir()` function to create such a temporary folder.

Each test module can be run individually, for instance `$ python trajectory_test.py`.

You can run **all** tests with `$ python all_tests.py` which can also be found under *pypet/tests*. You can pass additional arguments as `$ python all_tests.py -k --folder=myfolder/` with `-k` to keep the HDF5 and log files created by the tests (if you want to inspect them, otherwise they will be deleted after the completed tests), and `--folder=` to specify a folder where to store the HDF5 files instead of the temporary one. If the folder cannot be created, the program defaults to `tempfile.gettempdir()`.

If you do not want to browse to your installation folder, you can also download the `all_tests.py` script.

Running all tests can take up to 20 minutes and might temporarily take up to 1 GB of disk space. The test suite encompasses more than **700** tests and has a code coverage of about **90%**!

pypet is constantly tested with Python 2.6, 2.7, 3.3 and 3.4 for **Linux** using Travis-CI. Testing for **Windows** platforms is performed via Appveyor. The source code is available at github.com/SmokinCaterpillar/pypet.

2.4 Changelog

pypet 0.1b.12

- Renaming of the MultiprocContext's *start* function to *f_start*
- BUG FIX: Correct lock acquiring for multiprocessing with StorageContextManager
- BUG FIX: *v_full_copy* is now *False* by default
- BUG FIX: *v_full_copy* is no longer automatically set to *True* when using *freeze_pool_input*.
- New *consecutive_merge* parameter for *f_merge* to allow faster merging of several trajectories.

pypet 0.1b.11

- If one wants the old logging method, *log_config* should not be specified, i.e. setting it to *None* is no longer sufficient‘
- BUG FIX: Connection loss between the queue manager and the pool/processes has been resolved. This caused a minor slowing down of multiprocessing using a queue.
- New `freeze_pool_input` functionality for faster starting of single runs if using a pool.

pypet 0.1b.10

- New *v_crun_* property simply returning `'run_ALL'` if `v_crun` is `None`.
- BUG FIX: Removed recursive evaluation due to usage of *itertools.chain* during recursive node traversal
- max_depth is now also supported by *f_store*, *f_store_child*, *f_load*, *f_load_child*
- Loading and Storing internally are no longer truly recursive but iteratively handled.
- New *v_auto_run_prepend* property of the trajectory to switch off auto run name prepending if desired.
- The trajectory no longer relies on evil *eval* to construct a class. Instead it relies on the global scope.
- Better counting of loading and storing nodes to display rate in nodes/s
- BUG FIX: Minor bug in the progressbar has been fixed to detect automatic resets.
- Now support for non-nested empty containers: Empty dictionary, empty list, empty tuple and empty numpy array. All of them supported by the ArrayParameter and normal Results.
- Support for Sparse matrices containing *NO* data (i.e. only zeros).
- Performance optimization for storage and loading
- Improved test handling and parsing in *pypet.tests*
- Environment now supports *git_fail* option to fail if there are not committed changes instead of triggering a new commit.
- Users can now define their own functions to produce run-names
- Likewise users can define their onw wildcards
- The lazy version of adding elements (*traj.par.x = 42, 'A comment') now needs to be turned on by the user via ('traj.v_lazy_adding = True*) before it can be used.
- HDF5_STRCOL_MAX_ARRAY_LENGTH has been renamed to HDF5_STRCOL_MAX_RANGE_LENGTH
- The summary tables have been shortened. Now there's no distinction anymore between the actual runs and everything else.

- Moreover, data added to summary tables is no longer deleted. There also exists a maximum length for these tables (1000).
- The overview about the explored parameters in each run has been removed (due to size)
- Summary tables are now only based on the comments not the names!
- One can pass an estimate for memory that each run needs to better protect the memory cap.
- All tree nodes except the trajectory now use __slots__ for faster and more compact creation.
- You can now request to load a trajectory without *run_information* to save time for huge trajectories
- Trajectories use ordered dictionaries to remember what was added during a single run. Accordingly, now every data added during a single run regardless if they were added below a group *run_XXXXXXXX* is stored.
- BUG FIX: The *'QUEUE'* wrapping no longer waits for chunks of data, but starts storing immediately. Thus, if you have fast simulations, the storage service no longer waits until the end of you simulation before it starts storing data. In order to avoid overhead, the hdf5 is simply kept open until the queue is closed.
- BUG FIX: If *log_stdout=True*, the original stream is restored instead of *sys.__stdout__*. Thus, using another Python interpreter with a redirection of *stdout* and calling *f_disable_logging* no longer disables *print* completely.
- Refactored 'QUEUE' wrapping. The user can now decide the maximum size of the Storage Queue.
- CAP values are now in %, so choose values between 0.0 and 100.0
- BUG FIX: Links removed during single runs are now no longer stored
- BUG FIX: *pypet.* is no longer prepended to unpickled logger names. Accordingly, pypet logger names are now fully qualified names like *pypet.trajectory.Trajectory*.

pypet 0.1b.9

- BUG FIX: Fixed backwards compatibility
- BUG FIX: Metadata is loaded only once
- Results no longer support the `v_no_data_string` property
- Data of Results is no longer sorted in case of calling *f_val_to_string*
- In accordance with the python default to call *__repr__* for displaying contained objects, *f_val_to_str* calls *repr* on the contained data in parameters and results.
- Added informative __repr__ for the most commonly used classes
- The (annoyingly long) keyword *dynamically_imported_classes* is changed to *dynamic_imports*. For backwards compatibility, the old keyword can still be used.
- New *f_get_default* method, now one can specify a default value that should be returned if the requested data is not found in the trajectory
- *progressbar* displays the run and remaining time.
- New LINK features allowing group nodes to refer to other parts of the tree
- The SingleRun has been removed and all interactions are with real Trajectory objects, but the API remained the same.
- All *pypet* relevant imported packages will be stored by the trajectory
- Internally the queue no longer needs to be re-passed to the QueueSender, allowing for easier custom multiprocessing
- New MultiprocessWrapper (aka a light-weight environment for multiprocessing) for custom multiprocessing
- StorageServices provide an `multiproc_safe` attribute to enable the user to check if they work in a multi-process safe environment
- Environments can be used as context managers to disable the logging to files after the experiment.

- Environments provide the `v_log_path` property to get the current log path
- BUG FIX: Trajectories with only a single explored parameter can now be merged several times
- Backwards search no longer supported!
- *f_get_all* now supports shortcuts and abbreviations like *crun* or *par*
- *$* always translates to the run the trajectory is set to, also for adding new items to the tree
- If the current run is not set, `traj.v_crun` is set to `None`
- Moreover, `f_iter_nodes` and `f_iter_leaves` is no longer affected by the setting of a current run and always return all nodes and leaves
- The iteration functions from above now allow for a predicate function to filter potential nodes
- Storing a leaf or a group via `traj.f_store_item(item, overwrite=True)` now also replaces all annotations and comments
- Passing *overwrite_file=True* to an environment will overwrite the hdf5 file.
- *remove_empty_groups* is no longer supported
- All messages logged by *pypet* are now no longer using the root logger but one called 'pypet'.
- Better customization of logging. The user can now pass a list of `logger_names` and corresponding `log_levels` which are logged to files in the `log_path`.
- The environment no longer adds config information about hdf5 to the trajectory directly. This is now done by the service itself.
- The keyword arguments passed to the environment regarding the storage service are no longer handled by the environment but are directly passed to the service.
- BUG FIX: Fixed merging of result summaries that are only found in one trajectory
- BUG FIX: Log files are now closed when the handlers are removed
- BUG FIX: `max_depth` is now really always in relation to the start node and not in relation to intermediate results
- API change for *f_migrate* to match new concept of storage service
- Short function names for item additions like *f_apar* besides *f_add_parameter*.
- Abbreviations like *par* and *dpar* can now also be used for item creation and are always translated
- To streamline the API you can now no longer specify the name of backup files for merging
- Locked parameters can no longer be loaded and must be unlocked before.
- Parameters are no longer required to implement `__len__` because it can be ambiguous, instead they must implement `f_get_range_length` function.
- BUG FIX: `crun` is now also accepted for adding of data and not only for requests
- Setting *ncores=0* lets *pypet* determine the number of CPUs automatically (requires psutil).

pypet 0.1b.8

- Support for python 3.3 and 3.4!
- Proper handling of unicode strings (well, see above^^)
- Checking if names of leaf and group nodes only contain alphanumeric characters
- PickleParameter and PickleResult now explicitly store the pickle protocol because retrieval from the pickle dump is not always possible in python 3.
- Children of groups are no longer listed via __dir__ in case of debugging to prevent unwanted locking.
- Better support for PyTables 2 and 3 with same code base.
- pypet and pypet.brian now provide the __all__ list.

pypet 0.1b.7

- StreamToLogger has moved to the pypetlogging.py module. The mplogging module was deleted.
- The Network Manager now accepts custom network constructors
- A SingleRun can now provide a *v_filename* and *v_as_run* property. Both cannot be changed and the latter simply returns the name of the run.
- Better testing on travis
- Better support for pandas 0.14.1
- Now you can import most of the objects directly from the *pypet* package, e.g. `from pypet import Trajectory` instead of `from pypet.trajectory import Trajectory`.

pypet 0.1b.6

- The storage service prints status updates for loading and storing trees
- v_as_run is not longer *None* for a trajectory but *run_ALL*
- The Queue storage writer now stores batches of single runs to avoid re-opening of files as much as possible
- Faster Loading of data
- Support for PyTables 3.1.1
- *pypet* stores the name of the main script as a config parameter
- Data of Parameters can be accessed via *.data* or *param['data']*. Same holds for results that only contain a single data item.
- Parameters provide the function *f_get_default* to return the default value if the parameter is not empty.
- Large dictionaries and Object Tables are now split into chunks of 512 items
- In case an object table has more than 32 columns, another table is created to store the data item types (which is faster than storing all of the types as hdf5 node attributes)

pypet 0.1b.5

- New auto load feature
- BUG FIX: When parameters are emptied, the default value is set to None (and no longer kept)
- Now items are only saved once, if the node already exist on disk, storage is refused (Previously new data was added if it was not within the leaf before, but this can lead to strange inconsistencies).
- BUG FIX: *f_has_children* of a group node, now returns the correct result
- Refactored continuing of trajectories. Now this is based on *dill* and works also with data that cannot be pickled. *f_continue_run* is renamed *f_continue* to emphasize this change in API
- Picking the search strategy and using *v_check_uniqueness* is no longer supported. Sorry for the inconvenience. So forward search will always check if it finds 2 nodes with the same name within the same depth, and skip search if the next tree depth is entered.
- *f_contains* of group nodes has per default *shortcuts=False*
- There exists now the abstract class *HasLogger* in pypetlogging.py that establishes a unified logging framework
- Likewise the loggers of all network components are now private objects *_logger* and so is the function *_set_logger*.
- BUG FIX: *f_get_run_information* now works without passing any arguments
- Trajectories no longer accept a *file_tile* on initialisation
- One can now decide if trajectories should be automatically stored and if data should be cleaned up after every run

- BUG FIX: Storage of individual items during a single run do no longer require a full storage of the single run container.
- If automatic storage is enabled, trajectories are now stored at the end of the experiment, no longer before the starting of the single runs
- You can use the *$* character to decide where the HDF5 file tree should branch out for the individual runs
- *v_creator_name* is now called *v_run_branch* (since single runs can also create items that are not part of a run branch, so this is no longer misleading‘.
- Results and parameters now issue a warning when they have been stored and you change their data
- Parameters now have a property *v_explored* which is True for explored parameters even if the range has been removed
- By default *backwards_search* is turned off!
- Brian parameters no longer store the *storage_mode* explicitly
- BUG FIX: Wildcard character now always defaults to *run_ALL* if trajectory is not set to particular run
- BUG FIX: Now names with *XXXrun_* are again allowed only *run_* are forbidden.

pypet 0.1b.4

- Annotations and Results now support __setitem__, which is analogue to *f_get* and *f_set*
- Group Nodes can now contain comments as well
- Comments are only stored to HDF5 if they are not the empty string
- Large Overview Tables are off by default
- BrianDurationParameter was removed and the annotations are used instead. Use a normal BrianParameter and instead of *v_order* use *v_annotations.order*
- The user is advised to use *environment.f_run(manager.run_network)*, instead of *environment.f_run(run_network, manager)*
- Now there is the distinction between *small*, *large*, and *summary* tables
- BrianMonitorResult: Mean and variance values of State and MultiState Monitors are only extracted if they were recorded (which for newer BRIAN versions is only the case if you do NOT record traces)
- Log Level can be passed to environment
- BUG FIX: Scalars are no longer autoconverted by the storage service to zero-length numpy arrays
- Names of loggers have been shortened
- The trajectory now contains the functions *f_delete_item* and *f_delete_items* to erase stuff from disk. *f_remove_items* and *f_remove_item* no longer delete data from disk.
- Loading and deletions of items can now be made with SingleRuns as well.
- f_iter_nodes now iterates by default recursively all nodes
- A group node now supports __iter__ which simply calls f_iter_nodes NON recursively
- The structure of the trees are slightly changed. Results and derived parameters added with the trajectory are no longer assigned the prefix *trajectory*. Results and derived parameters added during single runs are now sorted into *runs.run_XXXXXXXXX*.
- Useless shortcuts have been removed.
- New *Backwards* search functionality
- New *f_get_all* functionality to find all items in a tree matching a particular search string
- Pandas Series and Panels are now supported, too!
- Now Pypet allows compression of HDF5 files, this can yield a massive reduction in memory space.

- *tr*, *cr*, *current_run*, *param*, *dparam* are no longer supported as a shortcuts
- __getitem__ is equivalent to __getattr__
- Now one can specify a maximum depth for accessing items.
- Now one can specify if shortcuts, i.e. hopping over parts of the tree, are allowed.
- New trajectroy attributes *v_backwards_search*, *v_max_depth*, *v_shortcuts* and *v_iter_recursive*. *v_iter_recursive* specifies the behavior of *__iter__*.
- *__contains__* now greps is arguments from the attributes *v_max_depth* and *v_shortcuts*.
- *log_stdout* parameter of the Environment determines if STDOUT and STDERROR will be logged to files. Can be disabled to allow better usage of pypet with interactive consoles.
- git commits only happen on changes and not all the time
- Now one can specify CPU, RAM and Swap cap levels if no pool is used. If these cap levels are crossed *pypet* won't start new processes.
- *f_load* now has an argument *load_all* to quickly load all subtrees with the same setting. Also *f_load* now accepts a filename as well
- New post-processing and pipeline modes!

pypet 0.1b.3

- BUG FIX: Now *f_run* and *f_continue_run* of an environment return the results produced by *runfunc*
- You can enforce a type convert for exploration
- Added lazy_debug option for the environment
- If you don't specify a filename, the environment defaults to a file with the name of the trajectory
- New multiprocessing mode (*use_pool=False* for environment) to spawn individual processes for each run. Useful if data cannot be pickled.
- New Brian framework with NetworkManager, NetworkComponent, NetworkAnalyser, NetworkRunner and DurationParameter
- Components, Analysers and the network runner of the manager are now publicly available
- Every component now provides the function *set_logger* to enable logging and instantiate a logger for *self.logger*

pypet 0.1b.2

- DefaultReplacementError is now called PresettingError
- Now the runtime of single runs is measured and stored.
- __getitem__ of the trajectory always returns the instance and fast access is not applied
- PickleResult and PickleParameter support the choice of protocol
- Explored Sparse matrices are stored under a slightly different name to disk.
- BUG FIX: BFS works properly
- BUG FIX: f_iter_runs is now affected if f_as_run is chosen
- Annotations support __iter__
- Annotations support __getitem__
- BrianMonitorResult, the property and columns 'times' for the Spike and StateSpikeMonitor has been renamed 'spiketimes'.
- Results support __iter__

- BrianMonitorResult, the name of state variables in array mode is changed to varname+'_%Xd', instead of varname+'_idx%08d', and 'spiketimes_%08d' to 'spiketimes_%Xd' and X is chosen in accordance with the number of neurons
- BUG FIX: nested_equal now supports Object Tables containing numpy arrays
- Better categorizations of the utility functions
- Comments are no longer limited in size
- New Brian Result
- Storage Service - in case of purging - now sets the comment to the result with the lowest index, in case of multiprocessing.
- Old API names are kept, but emit a depricated warning.
- The exploration array is now termed range. Accordingly, the function *f_is_array* is renamed *f_has_range* and *f_get_array* renamed to *f_get_range*.
- v_leaf renamed to v_is_leaf
- f_is_root renamed to v_is_root and changed to property
- v_fast_accessible renamed to f_supports_fast_access and changed to function
- v_parameter changed to v_is_parameter

pypet 0.1b.1

- Support for *long* types
- Documentation for the *f_find_idx* function
- The parameters *trajectory_name* and *trajectory_index* in f_load have been renamed to *name* and *index*

pypet 0.1b.0

- Group nodes support __getitem__
- SparseResult
- If you merge a trajectory, all environment settings of both are kept.
- More information about the environment is added to the trajectory
- BUG FIX:

  Recall of trajectory comments from disks yielded numpy strings instead of python strings This could cause trouble if the comment is empty!
- Git Integration, you can now make autocommits for every experiment
- New Sparse Parameter, for scipy sparse matrices
- BUG FIX: Loading of Trajectory metadata, now v_time is loaded correctly
- BUG FIX: Expand no longer repeats already run experiments
- More fine grain overview tables
- Comments for runs are only added once and not every run
- The overview tables are now found in the group *overview*
- Test are operating in a temp directory
- Now you can have fast access with results if they contain only a single entry with the name of the result
- New trajectory function *f_as_run* that makes the trajectory belief it is a particular single run and results and derived parameters of other runs are blinded out.
- Every group node now knows how to store and load itself via *f_load* and *f_store*.

- Storage of data is now analogous to loading with constants in (1,2,3). 1 Storing data only of items not been stored before, 2 for storing data as previously known. 3 For overwriting data. For instance, `traj.f_load(store_data=3)` overwrites all data on disk.
- f_update_skeleton is now f_load_skeleton to be more in line with naming scheme.
- setattr no longer supports shortcuts, i.e. `traj.x = 4` only works if `x` is directly below the trajectory root.
- Using setattr with a tuple of exactly length 2 whereas the second element is a string, sets the value as well as a comment

pypet 0.1a.6

- BUG FIX: (HDF5StorageService): storing a trajectory several times increased run and info table

pypet 0.1a.5

- Removed unnecessary imports
- Better documentation

pypet 0.1a.4

- Adding positional results will add them with the result name

pypet 0.1a.3

- Better handling of filenames, now relative paths are considered

pypet 0.1a.2

- Added automatic version grapping in setup.py

pypet 0.1a.1

- BaseParameter supports now __getitem__ if it is an array

CHAPTER 3

Library Reference

3.1 The Environment

genindex

3.1.1 Quicklinks

Environment	The environment to run a parameter exploration.
f_run	Runs the experiments and explores the parameter space.
f_continue	Resumes crashed trajectories.
f_pipeline	You can make *pypet* supervise your whole experiment by defining a pipeline.
v_trajectory	The trajectory of the Environment

3.1.2 Environment

Module containing the environment to run experiments.

An *Environment* provides an interface to run experiments based on parameter exploration.

The environment contains and might even create a *Trajectory* container which can be filled with parameters and results (see *pypet.parameter*). Instance of this trajectory are distributed to the user's job function to perform a single run of an experiment.

An *Environment* is the handyman for scheduling, it can be used for multiprocessing and takes care of organizational issues like logging.

class pypet.environment.**Environment**(*trajectory='trajectory'*, *add_time=True*, *comment=''*, *dynamic_imports=None*, *wildcard_functions=None*, *automatic_storing=True*, *log_config='DEFAULT'*, *log_stdout=('STDOUT'*, *20)*, *report_progress=(5*, *'pypet'*, *20)*, *multiproc=False*, *ncores=1*, *use_pool=False*, *freeze_pool_input=False*, *queue_maxsize=-1*, *cpu_cap=100.0*, *memory_cap=100.0*, *swap_cap=100.0*, *wrap_mode='LOCK'*, *clean_up_runs=True*, *immediate_postproc=False*, *continuable=False*, *continue_folder=None*, *delete_continue=True*, *storage_service=<class 'pypet.storageservice.HDF5StorageService'>*, *git_repository=None*, *git_message=''*, *git_fail=False*, *sumatra_project=None*, *sumatra_reason=''*, *sumatra_label=None*, *do_single_runs=True*, *lazy_debug=False*, ***kwargs*)

The environment to run a parameter exploration.

The first thing you usually do is to create and environment object that takes care about the running of the experiment. You can provide the following arguments:

Parameters

- **trajectory** – String or trajectory instance. If a string is supplied, a novel trajectory is created with that name. Note that the comment and the dynamically imported classes (see below) are only considered if a novel trajectory is created. If you supply a trajectory instance, these fields can be ignored.
- **add_time** – If True the current time is added to the trajectory name if created new.
- **comment** – Comment added to the trajectory if a novel trajectory is created.
- **dynamic_imports** – Only considered if a new trajectory is created. If you've written custom parameters or results that need to be loaded dynamically during runtime, the module containing the class needs to be specified here as a list of classes or strings naming classes and there module paths.

  For example: *dynamic_imports = ['pypet.parameter.PickleParameter', MyCustomParameter]*

  If you only have a single class to import, you do not need the list brackets: *dynamic_imports = 'pypet.parameter.PickleParameter'*
- **wildcard_functions** – Dictionary of wildcards like *$* and corresponding functions that are called upon finding such a wildcard. For example, to replace the *$* aka *crun* wildcard, you can pass the following: `wildcard_functions = {('$', 'crun'): myfunc}`.

  Your wildcard function *myfunc* must return a unique run name as a function of a given integer run index. Moreover, your function must also return a unique *dummy* name for the run index being *-1*.

  Of course, you can define your own wildcards like *wildcard_functions = {('$mycard', 'mycard'): myfunc)}. These are not required to return a unique name for each run index, but can be used to group runs into buckets by returning the same name for several run indices. Yet, all wildcard functions need to return a dummy name for the index '-1*.
- **automatic_storing** – If *True* the trajectory will be stored at the end of the simulation and single runs will be stored after their completion. Be aware of data loss if you set this to *False* and not manually store everything.
- **log_config** – Can be path to a logging *.ini* file specifying the logging configuration. For an example of such a file see *Logging*. Can also be a dictionary that is accepted by the built-in logging module. Set to *None* if you don't want *pypet* to configure logging.

  If not specified, the default settings are used. Moreover, you can manually tweak the default settings without creating a new *ini* file. Instead of the *log_config* parameter, pass a `log_folder`, a list of *logger_names* and corresponding *log_levels* to fine grain the loggers to which the default settings apply.

  For example:

  ```
  log_folder='logs', logger_names='('pypet',
  'MyCustomLogger'), log_levels=(logging.ERROR,
  logging.INFO)
  ```
- **log_stdout** – Whether the output of `stdout` should be recorded into the log files. Disable if only logging statement should be recorded. Note if you work with an interactive console like *IPython*, it is a good idea to set `log_stdout=False` to avoid messing up the console output.

  Can also be a tuple: ('mylogger', 10), specifying a logger name as well as a log-level. The log-level defines with what level *stdout* is logged, it is *not* a filter.

- **report_progress** – If progress of runs and an estimate of the remaining time should be shown. Can be *True* or *False* or a triple `(10, 'pypet', logging.Info)` where the first number is the percentage and update step of the resulting progressbar and the second one is a corresponding logger name with which the progress should be logged. If you use *'print'*, the *print* statement is used instead. The third value specifies the logging level (level of logging statement *not* a filter) with which the progress should be logged.

  Note that the progress is based on finished runs. If you use the *QUEUE* wrapping in case of multiprocessing and if storing takes long, the estimate of the remaining time might not be very accurate.

- **multiproc** – Whether or not to use multiprocessing. Default is `False`. Besides the wrap_mode (see below) that deals with how storage to disk is carried out in case of multiprocessing, there are two ways to do multiprocessing. By using a fixed pool of processes (choose *use_pool=True*, default option) or by spawning an individual process for every run and parameter combination (*use_pool=False*). The former will only spawn not more than *ncores* processes and all simulation runs are sent over to to the pool one after the other. This requires all your data to be pickled.

  If your data cannot be pickled (which could be the case for some BRIAN networks, for instance) choose *use_pool=False* (also make sure to set *continuable=False*). This will also spawn at most *ncores* processes at a time, but as soon as a process terminates a new one is spawned with the next parameter combination. Be aware that you will have as many logfiles in your logfolder as processes were spawned. If your simulation returns results besides storing results directly into the trajectory, these returned results still need to be pickled.

- **ncores** – If multiproc is `True`, this specifies the number of processes that will be spawned to run your experiment. Note if you use QUEUE mode (see below) the queue process is not included in this number and will add another extra process for storing. If you have *psutil* installed, you can set *ncores=0* to let *psutil* determine the number of CPUs available.

- **use_pool** – Whether to use a fixed pool of processes or whether to spawn a new process for every run. Use the former if you perform many runs (50k and more) which are in terms of memory and runtime inexpensive. Be aware that everything you use must be picklable. Use the latter for fewer runs (50k and less) and which are longer lasting and more expensive runs (in terms of memory consumption). In case your operating system allows forking, your data does not need to be picklable. If you choose `use_pool=False` you can also make use of the *cap* values, see below.

- **freeze_pool_input** – Can be set to `True` if the run function as well as all additional arguments are immutable. This will prevent the trajectory from getting pickled again and again. Thus, the run function, the trajectory as well as all arguments are passed to the pool at initialisation.

- **queue_maxsize** – Maximum size of the Storage Queue, in case of `'QUEUE'` wrapping. `0` means infinite, `-1` (default) means the educated guess of `2 * ncores`.

- **cpu_cap** – If *multiproc=True* and *use_pool=False* you can specify a maximum cpu utilization between 0.0 (excluded) and 100.0 (included) as fraction of maximum capacity. If the current cpu usage is above the specified level (averaged across all cores), *pypet* will not spawn a new process and wait until activity falls below the threshold again. Note that in order to avoid dead-lock at least one process will always be running regardless of the current utilization. If the threshold is crossed a warning will be issued. The warning won't be repeated as long as the threshold remains crossed.

  For example *cpu_cap=70.0*, *ncores=3*, and currently on average 80 percent of your cpu are used. Moreover, let's assume that at the moment only 2 processes are computing single runs simultaneously. Due to the usage of 80 percent of your cpu, *pypet* will wait until cpu usage drops below (or equal to) 70 percent again until it starts a third process to carry out another single run.

The parameters *memory_cap* and *swap_cap* are analogous. These three thresholds are combined to determine whether a new process can be spawned. Accordingly, if only one of these thresholds is crossed, no new processes will be spawned.

To disable the cap limits simply set all three values to 100.0.

You need the psutil package to use this cap feature. If not installed and you choose cap values different from 100.0 a ValueError is thrown.

- **memory_cap** – Cap value of RAM usage. If more RAM than the threshold is currently in use, no new processes are spawned. Can also be a tuple `(limit, memory_per_process)`, first value is the cap value (between 0.0 and 100.0), second one is the estimated memory per process in mega bytes (MB). If an estimate is given a new process is not started if the threshold would be crossed including the estimate.

- **swap_cap** – Analogous to *cpu_cap* but the swap memory is considered.

- **wrap_mode** – If multiproc is `True`, specifies how storage to disk is handled via the storage service.

  There are two options:

  WRAP_MODE_QUEUE: ('QUEUE')

  Another process for storing the trajectory is spawned. The sub processes running the individual single runs will add their results to a multiprocessing queue that is handled by an additional process. Note that this requires additional memory since the trajectory will be pickled and send over the queue for storage!

  WRAP_MODE_LOCK: ('LOCK')

  Each individual process takes care about storage by itself. Before carrying out the storage, a lock is placed to prevent the other processes to store data. Accordingly, sometimes this leads to a lot of processes waiting until the lock is released. Yet, single runs do not need to be pickled before storage!

  If you don't want wrapping at all use *WRAP_MODE_NONE* ('NONE')

- **clean_up_runs** – In case of single core processing, whether all results under groups named *run_XXXXXXXX* should be removed after the completion of the run. Note in case of multiprocessing this happens anyway since the single run container will be destroyed after finishing of the process.

  Moreover, if set to `True` after post-processing it is checked if there is still data under *run_XXXXXXXX* and this data is removed if the trajectory is expanded.

- **immediate_postproc** – If you use post- and multiprocessing, you can immediately start analysing the data as soon as the trajectory runs out of tasks, i.e. is fully explored but the final runs are not completed. Thus, while executing the last batch of parameter space points, you can already analyse the finished runs. This is especially helpful if you perform some sort of adaptive search within the parameter space.

  The difference to normal post-processing is that you do not have to wait until all single runs are finished, but your analysis already starts while there are still runs being executed. This can be a huge time saver especially if your simulation time differs a lot between individual runs. Accordingly, you don't have to wait for a very long run to finish to start post-processing.

  In case you use immediate postprocessing, the storage service of your trajectory is still multiprocessing safe. Moreover, internally the lock securing the storage service will be supervised by a multiprocessing manager. Accordingly, you could even use multiprocessing in your immediate post-processing phase if you dare, like use a multiprocessing pool, for instance.

  Note that after the execution of the final run, your post-processing routine will be called again as usual.

- **continuable** – Whether the environment should take special care to allow to resume or continue crashed trajectories. Default is `False`.

  You need to install dill to use this feature. *dill* will make snapshots of your simulation function as well as the passed arguments. BE AWARE that dill is still rather experimental!

  Assume you run experiments that take a lot of time. If during your experiments there is a power failure, you can resume your trajectory after the last single run that was still successfully stored via your storage service.

  The environment will create several *.ecnt* and *.rcnt* files in a folder that you specify (see below). Using this data you can continue crashed trajectories.

  In order to resume trajectories use `f_continue()`.

  Be aware that your individual single runs must be completely independent of one another to allow continuing to work. Thus, they should **NOT** be based on shared data that is manipulated during runtime (like a multiprocessing manager list) in the positional and keyword arguments passed to the run function.

  If you use post-processing, the expansion of trajectories and continuing of trajectories is NOT supported properly. There is no guarantee that both work together.

- **continue_folder** – The folder where the continue files will be placed. Note that *pypet* will create a sub-folder with the name of the environment.

- **delete_continue** – If true, *pypet* will delete the continue files after a successful simulation.

- **storage_service** – Pass a given storage service or a class constructor (default `HDF5StorageService`) if you want the environment to create the service for you. The environment will pass the additional keyword arguments you pass directly to the constructor. If the trajectory already has a service attached, the one from the trajectory will be used.

- **git_repository** – If your code base is under git version control you can specify here the path (relative or absolute) to the folder containing the *.git* directory as a string. Note in order to use this tool you need GitPython.

  If you set this path the environment will trigger a commit of your code base adding all files that are currently under version control. Similar to calling *git add -u* and *git commit -m 'My Message'* on the command line. The user can specify the commit message, see below. Note that the message will be augmented by the name and the comment of the trajectory. A commit will only be triggered if there are changes detected within your working copy.

  This will also add information about the revision to the trajectory, see below.

- **git_message** – Message passed onto git command. Only relevant if a new commit is triggered. If no changes are detected, the information about the previous commit and the previous commit message are added to the trajectory and this user passed message is discarded.

- **git_fail** – If *True* the program fails instead of triggering a commit if there are not committed changes found in the code base. In such a case a *GitDiffError* is raised.

- **sumatra_project** – If your simulation is managed by sumatra, you can specify here the path to the *sumatra* root folder. Note that you have to initialise the *sumatra* project at least once before via `smt init MyFancyProjectName`.

  pypet will automatically ad ALL parameters to the *sumatra* record. If a parameter is explored, the WHOLE range is added instead of the default value.

  pypet will add the label and reason (only if provided, see below) to your trajectory as config parameters.

- **sumatra_reason** – You can add an additional reason string that is added to the *sumatra* record. Regardless if *sumatra_reason* is empty, the name of the trajectory, the comment as well as a list of all explored parameters is added to the *sumatra* record.

  Note that the augmented label is not stored into the trajectory as config parameter, but the original one (without the name of the trajectory, the comment, and the list of explored parameters) in case it is not the empty string.

- **sumatra_label** – The label or name of your sumatra record. Set to *None* if you want sumatra to choose a label in form of a timestamp for you.

- **do_single_runs** – Whether you intend to actually to compute single runs with the trajectory. If you do not intend to do single runs, than set to `False` and the environment won't add config information like number of processors to the trajectory.

- **lazy_debug** – If `lazy_debug=True` and in case you debug your code (aka you use pydevd and the expression `'pydevd' in sys.modules` is `True`), the environment will use the *LazyStorageService* instead of the HDF5 one. Accordingly, no files are created and your trajectory and results are not saved. This allows faster debugging and prevents *pypet* from blowing up your hard drive with trajectories that you probably not want to use anyway since you just debug your code.

The Environment will automatically add some config settings to your trajectory. Thus, you can always look up how your trajectory was run. This encompasses most of the above named parameters as well as some information about the environment. This additional information includes a timestamp as well as a SHA-1 hash code that uniquely identifies your environment. If you use git integration, the SHA-1 hash code will be the one from your git commit. Otherwise the code will be calculated from the trajectory name, the current time, and your current *pypet* version.

The environment will be named *environment_XXXXXXX_XXXX_XX_XX_XXhXXmXXs*. The first seven *X* are the first seven characters of the SHA-1 hash code followed by a human readable timestamp.

All information about the environment can be found in your trajectory under *config.environment.environment_XXXXXXX_XXXX_XX_XX_XXhXXmXXs*. Your trajectory could potentially be run by several environments due to merging or extending an existing trajectory. Thus, you will be able to track how your trajectory was built over time.

Git information is added to your trajectory as follows:

- git.commit_XXXXXXX_XXXX_XX_XX_XXh_XXm_XXs.hexsha

  The SHA-1 hash of the commit. *commit_XXXXXXX_XXXX_XX_XX_XXhXXmXXs* is mapped to the first seven items of the SHA-1 hash and the formatted data of the commit, e.g. *commit_7ef7hd4_2015_10_21_16h29m00s*.

- git.commit_XXXXXXX_XXXX_XX_XX_XXh_XXm_XXs.name_rev

  String describing the commits hexsha based on the closest reference

- git.commit_XXXXXXX_XXXX_XX_XX_XXh_XXm_XXs.committed_date

  Commit date as Unix Epoch data

- git.commit_XXXXXXX_XXXX_XX_XX_XXh_XXm_XXs.message

  The commit message

Moreover, if you use the standard `HDF5StorageService` you can pass the following keyword arguments in `**kwargs`:

Parameters

- **filename** – The name of the hdf5 file. If none is specified the default *./hdf5/the_name_of_your_trajectory.hdf5* is chosen. If *filename* contains only a path like *filename='./myfolder/'*, *it is changed to 'filename='./myfolder/the_name_of_your_trajectory.hdf5'*.

- **file_title** – Title of the hdf5 file (only important if file is created new)

- **overwrite_file** – If the file already exists it will be overwritten. Otherwise, the trajectory will simply be added to the file and already existing trajectories are **not** deleted.

- **encoding** – Format to encode and decode unicode strings stored to disk. The default `'utf8'` is highly recommended.

- **complevel** – You can specify your compression level. 0 means no compression and 9 is the highest compression level. See PyTables Compression for a detailed description.

- **complib** – The library used for compression. Choose between *zlib*, *blosc*, and *lzo*. Note that 'blosc' and 'lzo' are usually faster than 'zlib' but it may be the case that you can no longer open your hdf5 files with third-party applications that do not rely on PyTables.

- **shuffle** – Whether or not to use the shuffle filters in the HDF5 library. This normally improves the compression ratio.

- **fletcher32** – Whether or not to use the *Fletcher32* filter in the HDF5 library. This is used to add a checksum on hdf5 data.

- **pandas_format** – How to store pandas data frames. Either in 'fixed' ('f') or 'table' ('t') format. Fixed format allows fast reading and writing but disables querying the hdf5 data and appending to the store (with other 3rd party software other than *pypet*).

- **purge_duplicate_comments** – If you add a result via `f_add_result()` or a derived parameter `f_add_derived_parameter()` and you set a comment, normally that comment would be attached to each and every instance. This can produce a lot of unnecessary overhead if the comment is the same for every instance over all runs. If *purge_duplicate_comments=1* than only the comment of the first result or derived parameter instance created in a run is stored or comments that differ from this first comment.

  For instance, during a single run you call *traj.f_add_result('my_result',42, comment='Mostly harmless!')*' and the result will be renamed to *results.run_00000000.my_result*. After storage in the node associated with this result in your hdf5 file, you will find the comment *'Mostly harmless!'* there. If you call *traj.f_add_result('my_result',-43, comment='Mostly harmless!')* in another run again, let's say run 00000001, the name will be mapped to *results.run_00000001.my_result*. But this time the comment will not be saved to disk since *'Mostly harmless!'* is already part of the very first result with the name 'results.run_00000000.my_result'. Note that the comments will be compared and storage will only be discarded if the strings are exactly the same.

  If you use multiprocessing, the storage service will take care that the comment for the result or derived parameter with the lowest run index will be considered regardless of the order of the finishing of your runs. Note that this only works properly if all comments are the same. Otherwise the comment in the overview table might not be the one with the lowest run index.

  You need summary tables (see below) to be able to purge duplicate comments.

  This feature only works for comments in *leaf* nodes (aka Results and Parameters). So try to avoid to add comments in *group* nodes within single runs.

- **summary_tables** – Whether the summary tables should be created, i.e. the 'derived_parameters_runs_summary', and the *results_runs_summary*.

  The 'XXXXXX_summary' tables give a summary about all results or derived parameters. It is assumed that results and derived parameters with equal names in individual runs are similar and only the first result or derived parameter that was created is shown as an example.

  The summary table can be used in combination with *purge_duplicate_comments* to only store a single comment for every result with the same name in each run, see above.

- **small_overview_tables** – Whether the small overview tables should be created. Small tables are giving overview about 'config','parameters', 'derived_parameters_trajectory', , 'results_trajectory', 'results_runs_summary'.

  Note that these tables create some overhead. If you want very small hdf5 files set *small_overview_tables* to False.

- **large_overview_tables** – Whether to add large overview tables. This encompasses information about every derived parameter, result, and the explored parameter in every single run. If you want small hdf5 files set to `False` (default).

- **results_per_run** – Expected results you store per run. If you give a good/correct estimate storage to hdf5 file is much faster in case you store LARGE overview tables.

  Default is 0, i.e. the number of results is not estimated!

- **derived_parameters_per_run** – Analogous to the above.

Finally, you can also pass properties of the trajectory, like `v_with_links=True` (you can leave the prefix `v_`, i.e. `with_links` works, too). Thus, you can change the settings of the trajectory immediately.

f_add_postprocessing(*postproc*, **args*, ***kwargs*)

Adds a post processing function.

The environment will call this function via `postproc(traj, result_list, *args, **kwargs)` after the completion of the single runs.

This function can load parts of the trajectory id needed and add additional results.

Moreover, the function can be used to trigger an expansion of the trajectory. This can be useful if the user has an *optimization* task.

Either the function calls *f_expand* directly on the trajectory or returns an dictionary. If latter *f_expand* is called by the environment.

Note that after expansion of the trajectory, the post-processing function is called again (and again for further expansions). Thus, this allows an iterative approach to parameter exploration.

Note that in case post-processing is called after all runs have been executed, the storage service of the trajectory is no longer multiprocessing safe. If you want to use multiprocessing in your post-processing you can still manually wrap the storage service with the `MultiprocessWrapper`.

Nonetheless, in case you use **immediate** post-processing, the storage service is still multiprocessing safe. In fact, it has to be because some single runs are still being executed and write data to your HDF5 file. Accordingly, you can also use multiprocessing during the immediate post-processing without having to use the `MultiprocessWrapper`.

You can easily check in your post-processing function if the storage service is multiprocessing safe via the `multiproc_safe` attribute, i.e. `traj.v_storage_service.multiproc_safe`.

Parameters

- **postproc** – The post processing function
- **args** – Additional arguments passed to the post-processing function
- **kwargs** – Additional keyword arguments passed to the postprocessing function

Returns

f_continue(*trajectory_name=None*, *continue_folder=None*)

Resumes crashed trajectories.

Parameters

- **trajectory_name** – Name of trajectory to resume, if not specified the name passed to the environment is used. Be aware that if *add_time=True* the name you passed to the environment is altered and the current date is added.

- **continue_folder** – The folder where continue files can be found. Do not pass the name of the sub-folder with the trajectory name, but to the name of the parental folder. If not specified the continue folder passed to the environment is used.

Returns

List of the individual results returned by your run function.

Returns a LIST OF TUPLES, where first entry is the run idx and second entry is the actual result. In case of multiprocessing these are not necessarily ordered according to their run index, but ordered according to their finishing time.

Does not contain results stored in the trajectory! In order to access these simply interact with the trajectory object, potentially after calling‘~pypet.trajectory.Trajectory.f_update_skeleton‘ and loading all results at once with `f_load()` or loading manually with `f_load_items()`.

Even if you use multiprocessing without a pool the results returned by *runfunc* still need to be pickled.

f_disable_logging(*remove_all_handlers=True*)

Removes all logging handlers and stops logging to files and logging stdout.

Parameters **remove_all_handlers** – If *True* all logging handlers are removed. If you want to keep the handlers set to *False*.

f_pipeline(*pipeline*)

You can make *pypet* supervise your whole experiment by defining a pipeline.

pipeline is a function that defines the entire experiment. From pre-processing including setting up the trajectory over defining the actual simulation runs to post processing.

The *pipeline* function needs to return TWO tuples with a maximum of three entries each.

For example:

```
return (runfunc, args, kwargs), (postproc, postproc_args, postproc_kwargs)
```

Where *runfunc* is the actual simulation function thet gets passed the trajectory container and potentially additional arguments *args* and keyword arguments *kwargs*. This will be run by your environment with all parameter combinations.

postproc is a post processing function that handles your computed results. The function must accept as arguments the trajectory container, a list of results (list of tuples (run idx, result)) and potentially additional arguments *postproc_args* and keyword arguments *postproc_kwargs*.

As for `f_add_postproc()`, this function can potentially extend the trajectory.

If you don't want to apply post-processing, your pipeline function can also simply return the run function and the arguments:

```
return runfunc, args, kwargs
```

Or

```
return runfunc, args
```

Or

```
return runfunc
```

`return runfunc, kwargs` does NOT work, if you don't want to pass *args* do `return runfunc, (), kwargs`.

Analogously combinations like

```
return (runfunc, args), (postproc,)
```

work as well.

Parameters

pipeline – The pipleine function, taking only a single argument *traj*. And returning all functions necessary for your experiment.

Returns

List of the individual results returned by *runfunc*.

Returns a LIST OF TUPLES, where first entry is the run idx and second entry is the actual result. In case of multiprocessing these are not necessarily ordered according to their run index, but ordered according to their finishing time.

Does not contain results stored in the trajectory! In order to access these simply interact with the trajectory object, potentially after calling `f_update_skeleton()` and loading all results at once with `f_load()` or loading manually with `f_load_items()`.

Even if you use multiprocessing without a pool the results returned by *runfunc* still need to be pickled.

Results computed from *postproc* are not returned. *postproc* should not return any results except dictionaries if the trajectory should be expanded.

f_run(*runfunc*, **args*, ***kwargs*)

Runs the experiments and explores the parameter space.

Parameters

- **runfunc** – The task or job to do
- **args** – Additional arguments (not the ones in the trajectory) passed to *runfunc*
- **kwargs** – Additional keyword arguments (not the ones in the trajectory) passed to *runfunc*

Returns

List of the individual results returned by *runfunc*.

Returns a LIST OF TUPLES, where first entry is the run idx and second entry is the actual result. In case of multiprocessing these are not necessarily ordered according to their run index, but ordered according to their finishing time.

Does not contain results stored in the trajectory! In order to access these simply interact with the trajectory object, potentially after calling‘~pypet.trajectory.Trajectory.f_update_skeleton‘ and loading all results at once with `f_load()` or loading manually with `f_load_items()`.

If you use multiprocessing without a pool the results returned by *runfunc* still need to be pickled.

f_set_large_overview(*switch*)

Switches large overview tables on (*switch=True*) or off (*switch=False*).

f_set_small_overview(*switch*)

Switches small overview tables on (*switch=True*) or off (*switch=False*).

f_set_summary(*switch*)

Switches summary tables on (*switch=True*) or off (*switch=False*).

f_switch_off_all_overview()

Switches all tables off.

DEPRECATED: Please pass whether to use the tables to the environment constructor.

f_switch_off_large_overview()

Switches off the tables consuming the most memory.

- Single Run Result Overview
- Single Run Derived Parameter Overview
- Explored Parameter Overview in each Single Run

DEPRECATED: Please pass whether to use the tables to the environment constructor.

f_switch_off_small_overview()
Switches off small overview tables and switches off *purge_duplicate_comments*.

DEPRECATED: Please pass whether to use the tables to the environment constructor.

v_hexsha
The SHA1 identifier of the environment.

It is identical to the SHA1 of the git commit. If version control is not used, the environment hash is computed from the trajectory name, the current timestamp and your current *pypet* version.

v_log_path
The full path to the (sub) folder where log files are stored

v_name
Name of the Environment

v_time
Time of the creation of the environment, human readable.

v_timestamp
Time of creation as python datetime float

v_traj
Equivalent to env.v_trajectory

v_trajectory
The trajectory of the Environment

3.1.3 MultiprocContext

class pypet.environment.**MultiprocContext**(*trajectory*, *wrap_mode='LOCK'*, *full_copy=None*, *manager=None*, *use_manager=True*, *lock=None*, *queue=None*, *queue_maxsize=0*, *log_config=None*, *log_stdout=False*)
A lightweight environment that allows the usage of multiprocessing.

Can be used if you don't want a full-blown *Environment* to enable multiprocessing or if you want to implement your own custom multiprocessing.

This Wrapper tool will take a trajectory container and take care that the storage service is multiprocessing safe. Supports the 'LOCK' as well as the 'QUEUE' mode. In case of the latter an extra queue process is created if desired. This process will handle all storage requests and write data to the hdf5 file.

Not that in case of 'QUEUE' wrapping data can only be stored not loaded, because the queue will only be read in one direction.

Parameters

- **trajectory** – The trajectory which storage service should be wrapped
- **wrap_mode** – There are two options:

  WRAP_MODE_QUEUE: ('QUEUE')

  If desired another process for storing the trajectory is spawned. The sub processes running the individual trajectories will add their results to a multiprocessing queue that is handled by an additional process. Note that this requires additional memory since data will be pickled and send over the queue for storage!

  WRAP_MODE_LOCK: ('LOCK')

Each individual process takes care about storage by itself. Before carrying out the storage, a lock is placed to prevent the other processes to store data. Accordingly, sometimes this leads to a lot of processes waiting until the lock is released. Yet, data does not need to be pickled before storage!

- **full_copy** – In case the trajectory gets pickled (sending over a queue or a pool of processors) if the full trajectory should be copied each time (i.e. all parameter points) or only a particular point. A particular point can be chosen beforehand with `f_as_run()`.

  Leave `full_copy=None` if the setting from the passed trajectory should be used. Otherwise `v_full_copy` of the trajectory is changed to your chosen value.

- **manager** – You can pass an optional multiprocessing manager here, if you already have instantiated one. Leave `None` if you want the wrapper to create one.

- **use_manager** – If your lock and queue should be created with a manager or if wrapping should be created from the multiprocessing module directly.

  For example: `multiprocessing.Lock()` or via a manager `multiprocessing.Manager().Lock()` (if you specified a manager, this manager will be used).

  The former is usually faster whereas the latter is more flexible and can be used in an environment where fork is not available, for instance.

- **lock** – You can pass a multiprocessing lock here, if you already have instantiated one. Leave `None` if you want the wrapper to create one in case of `'LOCK'` wrapping.

- **queue** – You can pass a multiprocessing queue here, if you already instantiated one. Leave `None` if you want the wrapper to create one in case of ''*QUEUE*'' wrapping.

- **queue_maxsize** – Maximum size of queue if created new. 0 means infinite.

- **log_config** – Path to logging config file or dictionary to configure logging for the spawned queue process. Thus, only considered if the queue wrap mode is chosen.

- **log_stdout** – If stdout of the queue process should also be logged.

For an usage example see *Lightweight Multiprocessing*.

f_finalize()
Restores the original storage service.

If a queue process and a manager were used both are shut down.

Automatically called when used as context manager.

f_start()
Starts the multiprocess wrapping.

Automatically called when used as context manager.

3.2 The Trajectory and Group Nodes

3.2.1 Quicklinks

Here are some links to important functions:

Trajectory	The trajectory manages results and parameters.
f_add_parameter	Adds a parameter under the current node.
f_add_derived_parameter	Adds a derived parameter under the current group.
f_add_result	Adds a result under the current node.

Continued on next page

Table 3.2 – continued from previous page

f_add_link	Adds a link to an existing node.
f_add_leaf	Adds an empty generic leaf under the current node.
f_iter_leaves	Iterates (recursively) over all leaves hanging below the current group.
f_iter_nodes	Iterates recursively (default) over nodes hanging below this group.
f_get	Searches and returns an item (parameter/result/group node) with the given *name*.
f_store_child	Stores a child or recursively a subtree to disk.
f_store	Stores a group node to disk
f_load_child	Loads a child or recursively a subtree from disk.
f_load	Loads a group from disk.
f_explore	Prepares the trajectory to explore the parameter space.
f_store	Stores the trajectory to disk and recursively all data in the tree.
f_load	Loads a trajectory via the storage service.
f_load_skeleton	Loads the full skeleton from the storage service.
f_preset_parameter	Presets parameter value before a parameter is added.
f_get_from_runs	Searches for all occurrences of *name* in each run.
f_load_items	Loads parameters and results specified in *iterator*.
f_store_items	Stores individual items to disk.
f_remove_items	Removes parameters, results or groups from the trajectory.
f_delete_items	Deletes items from storage on disk.
f_find_idx	Finds a single run index given a particular condition on parameters.
f_get_run_information	Returns a dictionary containing information about a single run.
v_crun	Run name if you want to access the trajectory as a single run.
v_idx	Index if you want to access the trajectory as during a single run.
v_standard_parameter	The standard parameter used for parameter creation
v_standard_result	The standard result class used for result creation
v_annotations	Annotation feature of a trajectory node.
load_trajectory	Helper function that creates a novel trajectory and loads it from disk.

3.2.2 Trajectory

class pypet.trajectory.**Trajectory**(*name='my_trajectory'*, *add_time=True*, *comment=''*, *dynamic_imports=None*, *wildcard_functions=None*, *storage_service=None*, ***kwargs*)

The trajectory manages results and parameters.

The trajectory provides all functionality to define how the parameter space of your simulation should be explored. During single runs based on a particular parameter point, the functionality fo the trajectory is reduced.

You can add four types of data to the trajectory:

•Config:

These are special parameters specifying modalities of how to run your simulations. Changing a config parameter should NOT have any influence on the results you obtain from your simulations.

They specify runtime environment parameters like how many CPUs you use for multiprocessing etc.

In fact, if you use the default runtime environment of this project, the environment will add some config parameters to your trajectory.

The method to add more config is *f_add_config()*

Config parameters are put into the subtree *traj.config* (with *traj* being your trajectory instance).

•Parameters:

These are your primary ammunition in numerical simulations. They specify how your simulation works. They can only be added before the actual running of the simulation exploring the parameter space. They can be added via `f_add_parameter()` and be explored using `f_explore()`. Or to expand an existing trajectory use `f_expand()`.

Your parameters should encompass all values that completely define your simulation, I recommend also storing random number generator seeds as parameters to guarantee that a simulation can be repeated exactly the way it was run the first time.

Parameters are put into the subtree *traj.parameters*.

•Derived Parameters:

They are not much different from parameters except that they can be added anytime.

Conceptually this encompasses stuff that is intermediately computed from the original parameters. For instance, as your original parameters you have a random number seed and some other parameters. From these you compute a connection matrix for a neural network. This connection matrix could be stored as a derived parameter.

Derived parameters are added via `f_add_derived_parameter()`.

Derived parameters are put into the subtree *traj.derived_parameters*. They are further sorted into *traj.derived_parameters.runs.run_XXXXXXXX* if they were added during a single run. *XXXXXXXX* is replaced by the index of the corresponding run, for example *run_00000001*.

•Results:

Result are added via the `f_add_result()`. They are kept under the subtree *traj.results* and are further sorted into *traj.results.runs.run_XXXXXXXX* if they are added during a single run.

There are several ways to access the parameters and results, to learn about these, fast access, and natural naming see *Accessing Data in the Trajectory*.

In case you create a new trajectory you can pass the following arguments:

Parameters

- **name** – Name of the trajectory, if *add_time=True* the current time is added as a string to the parameter name.
- **add_time** – Boolean whether to add the current time in human readable format to the trajectory name.
- **comment** – A useful comment describing the trajectory.
- **dynamic_imports** – If you've written a custom parameter that needs to be loaded dynamically during runtime, this needs to be specified here as a list of classes or strings naming classes and there module paths. For example: *dynamic_imports = ['pypet.parameter.PickleParameter',MyCustomParameter]*

  If you only have a single class to import, you do not need the list brackets: *dynamic_imports = 'pypet.parameter.PickleParameter'*
- **wildcard_functions** – Dictionary of wildcards like *$* and corresponding functions that are called upon finding such a wildcard.
- **storage_service** – Pass a storage service used by the Trajectory. Alternatively, pass a constructor and other `**kwargs` are passed onto the constructor.
- **kwargs** – Other arguments passed to the storage service constructor

Raises

ValueError: If the name of the trajectory contains invalid characters ornot all additional keyword arguments are used.

TypeError: If the dynamically imported classes are not classes or strings.

Example usage:

```
>>> traj = Trajectory('ExampleTrajectory', dynamic_imports=['Some.custom.class'],    comment
```

f_add_config(**args*, ***kwargs*)

Adds a config parameter under the current group.

Similar to *f_add_parameter()*.

If current group is the trajectory the prefix *'config'* is added to the name.

ATTENTION: This function is not available during a single run!

f_add_config_group(**args*, ***kwargs*)

Adds an empty config group under the current node.

Adds the full name of the current node as prefix to the name of the group. If current node is the trajectory (root), the prefix *'config'* is added to the full name.

The *name* can also contain subgroups separated via colons, for example: *name=subgroup1.subgroup2.subgroup3*. These other parent groups will be automatically be created.

ATTENTION: This function is not available during a single run!

f_add_parameter(**args*, ***kwargs*)

Adds a parameter under the current node.

There are two ways to add a new parameter either by adding a parameter instance:

```
>>> new_parameter = Parameter('group1.group2.myparam', data=42, comment='Example!')
>>> traj.f_add_parameter(new_parameter)
```

Or by passing the values directly to the function, with the name being the first (non-keyword!) argument:

```
>>> traj.f_add_parameter('group1.group2.myparam', data=42, comment='Example!')
```

If you want to create a different parameter than the standard parameter, you can give the constructor as the first (non-keyword!) argument followed by the name (non-keyword!):

```
>>> traj.f_add_parameter(PickleParameter,'group1.group2.myparam', data=42, comment='E
```

The full name of the current node is added as a prefix to the given parameter name. If the current node is the trajectory the prefix *'parameters'* is added to the name.

ATTENTION: This function is not available during a single run!

f_add_parameter_group(**args*, ***kwargs*)

Adds an empty parameter group under the current node.

Can be called with `f_add_parameter_group('MyName', 'this is an informative comment')` or `f_add_parameter_group(name='MyName', comment='This is an informative comment')` or with a given new group instance: `f_add_parameter_group(ParameterGroup('MyName', comment='This is a comment'))`.

Adds the full name of the current node as prefix to the name of the group. If current node is the trajectory (root), the prefix *'parameters'* is added to the full name.

The *name* can also contain subgroups separated via colons, for example: *name=subgroup1.subgroup2.subgroup3*. These other parent groups will be automatically created.

ATTENTION: This function is not available during a single run!

f_add_to_dynamic_imports(*dynamic_imports*)
Adds classes or paths to classes to the trajectory to create custom parameters.

param dynamic_imports If you've written custom parameter that needs to be loaded dynamically during runtime, this needs to be specified here as a list of classes or strings naming classes and there module paths. For example: *dynamic_imports = ['pypet.parameter.PickleParameter',MyCustomParameter]*

If you only have a single class to import, you do not need the list brackets: *dynamic_imports = 'pypet.parameter.PickleParameter'*

ATTENTION: This function is not available during a single run!

f_add_wildcard_functions(*func_dict*)
#TODO

f_backup(***kwargs*)
Backs up the trajectory with the given storage service.

Arguments of `kwargs` are directly passed to the storage service, for the HDF5StorageService you can provide the following argument:

param backup_filename Name of file where to store the backup.

In case you use the standard HDF5 storage service and *backup_filename=None*, the file will be chosen automatically. The backup file will be in the same folder as your hdf5 file and named 'backup_XXXXX.hdf5' where 'XXXXX' is the name of your current trajectory.

ATTENTION: This function is not available during a single run!

f_delete_item(*item*, **args*, ***kwargs*)
Deletes a single item, see *f_delete_items()*

f_delete_items(*iterator*, **args*, ***kwargs*)
Deletes items from storage on disk.

Per default the item is NOT removed from the trajectory.

Links are NOT deleted on the hard disk, please delete links manually before deleting data!

Parameters

- **iterator** – A sequence of items you want to remove. Either the instances themselves or strings with the names of the items.
- **remove_from_trajectory** – If items should also be removed from trajectory. Default is *False*.
- **args** – Additional arguments passed to the storage service
- **kwargs** – Additional keyword arguments passed to the storage service

If you use the standard hdf5 storage service, you can pass the following additional keyword argument:

param delete_only You can partially delete leaf nodes. Specify a list of parts of the result node that should be deleted like *delete_only=['mystuff','otherstuff']*. This wil only delete the hdf5 sub parts *mystuff* and *otherstuff* from disk. BE CAREFUL, erasing data partly happens at your own risk. Depending on how complex the loading process of your result node is, you might not be able to reconstruct any data due to partially deleting some of it.

Be aware that you need to specify the names of parts as they were stored to HDF5. Depending on how your leaf construction works, this may differ from the names the data might have in your leaf in the trajectory container.

If the hdf5 nodes you specified in *delete_only* cannot be found a warning is issued.

Note that massive deletion will fragment your HDF5 file. Try to avoid changing data on disk whenever you can.

If you want to erase a full node, simply ignore this argument or set to *None*.

param remove_from_item If data that you want to delete from storage should also be removed from the items in *iterator* if they contain these. Default is *False*.

param recursive If you want to delete a group node and it has children you need to set *recursive* to *True. Default is 'False*.

f_delete_link(*link*, *remove_from_trajectory=False*)
Deletes a single link see *f_delete_links()*

f_delete_links(*iterator_of_links*, *remove_from_trajectory=False*)
Deletes several links from the hard disk.

Links can be passed as a string `'groupA.groupB.linkA'` or as a tuple containing the node from which the link should be removed and the name of the link `(groupWithLink, 'linkA')`.

f_expand(*build_dict*, *fail_safe=True*)

Similar to *f_explore()*, but can be used to enlargealready completed trajectories.

Please ensure before usage, that all explored parameters are loaded!

param build_dict Dictionary containing the expansion

param fail_safe If old ranges should be **deep-copied** in order to allow to restore the original exploration if something fails during expansion. Set to *False* if deep-copying your parameter ranges causes errors.

raisesTypeError: If not all explored parameters are enlarged

AttributeError: If keys of dictionary cannot be found in the trajectory

NotUniqueNodeError:

If dictionary keys do not unambiguously map to single parameters

ValueError: If not all explored parameter ranges are of the same length

ATTENTION: This function is not available during a single run!

f_explore(*build_dict*)
Prepares the trajectory to explore the parameter space.

To explore the parameter space you need to provide a dictionary with the names of the parameters to explore as keys and iterables specifying the exploration ranges as values.

All iterables need to have the same length otherwise a ValueError is raised. A ValueError is also raised if the names from the dictionary map to groups or results and not parameters.

If your trajectory is already explored but not stored yet and your parameters are not locked you can add new explored parameters to the current ones if their iterables match the current length of the trajectory.

Raises an AttributeError if the names from the dictionary are not found at all in the trajectory and NotUniqueNodeError if the keys not unambiguously map to single parameters.

Raises a TypeError if the trajectory has been stored already, please use *f_expand()* then instead.

Example usage:

```
>>> traj.f_explore({'groupA.param1' : [1,2,3,4,5], 'groupA.param2':['a','b','c','d','
```

Could also be called consecutively:

```
>>> traj.f_explore({'groupA.param1' : [1,2,3,4,5]})
>>> traj.f_explore({'groupA.param2':['a','b','c','d','e']})
```

NOTE:

Since parameters are very conservative regarding the data they accept (see *Values supported by Parameters*), you sometimes won't be able to use Numpy arrays for exploration as iterables.

For instance, the following code snippet won't work:

```
import numpy a np
from pypet.trajectory import Trajectory
traj = Trajectory()
traj.f_add_parameter('my_float_parameter', 42.4,
                     comment='My value is a standard python float')

traj.f_explore( { 'my_float_parameter': np.arange(42.0, 44.876, 0.23) } )
```

This will result in a *TypeError* because your exploration iterable *np.arange(42.0, 44.876, 0.23)* contains *numpy.float64* values whereas you parameter is supposed to use standard python floats.

Yet, you can use Numpys *tolist()* function to overcome this problem:

```
traj.f_explore( { 'my_float_parameter': np.arange(42.0, 44.876, 0.23).tolist() } )
```

Or you could specify your parameter directly as a numpy float:

```
traj.f_add_parameter('my_float_parameter', np.float64(42.4),
                         comment='My value is a numpy 64 bit float')
```

ATTENTION: This function is not available during a single run!

f_find_idx(*name_list*, *predicate*)

Finds a single run index given a particular condition on parameters.

ONLY useful for a single run if `v_full_copy` was set to ``True`. Otherwise a TypeError is thrown.

Parameters

- **name_list** – A list of parameter names the predicate applies to, if you have only a single parameter name you can omit the list brackets.
- **predicate** – A lambda predicate for filtering that evaluates to either `True` or `False`

ReturnsA generator yielding the matching single run indices

Example:

```
>>> predicate = lambda param1, param2: param1==4 and param2 in [1.0, 2.0]
>>> iterator = traj.f_find_idx(['groupA.param1', 'groupA.param2'], predicate)
>>> [x for x in iterator]
[0, 2, 17, 36]
```

f_get_config(*fast_access=False*, *copy=True*)

Returns a dictionary containing the full config names as keys and the config parameters or the config parameter data items as values.

Parameters

- **fast_access** – Determines whether the parameter objects or their values are returned in the dictionary.

- **copy** – Whether the original dictionary or a shallow copy is returned. If you want the real dictionary please do not modify it at all! Not Copying and fast access do not work at the same time! Raises ValueError if fast access is true and copy false.

Returns Dictionary containing the config data

Raises ValueError

f_get_derived_parameters (*fast_access=False*, *copy=True*)

Returns a dictionary containing the full parameter names as keys and the parameters or the parameter data items as values.

Parameters

- **fast_access** – Determines whether the parameter objects or their values are returned in the dictionary.
- **copy** – Whether the original dictionary or a shallow copy is returned. If you want the real dictionary please do not modify it at all! Not Copying and fast access do not work at the same time! Raises ValueError if fast access is true and copy false.

Returns Dictionary containing the parameters.

Raises ValueError

f_get_explored_parameters (*fast_access=False*, *copy=True*)

Returns a dictionary containing the full parameter names as keys and the parameters or the parameter data items as values.

IMPORTANT: This dictionary always contains all explored parameters as keys. Even when they are not loaded, in this case the value is simply *None*. *fast_access* only works if all explored parameters are loaded.

Parameters

- **fast_access** – Determines whether the parameter objects or their values are returned in the dictionary.
- **copy** – Whether the original dictionary or a shallow copy is returned. If you want the real dictionary please do not modify it at all! Not Copying and fast access do not work at the same time! Raises ValueError if fast access is true and copy false.

Returns Dictionary containing the parameters.

Raises ValueError

f_get_from_runs (*name*, *include_default_run=True*, *use_indices=False*, *fast_access=False*, *with_links=True*, *shortcuts=True*, *max_depth=None*, *auto_load=False*)

Searches for all occurrences of *name* in each run.

Generates an ordered dictionary with the run names or indices as keys and found items as values.

Example:

```
>>> traj.f_get_from_runs(self, 'deep.universal_answer', use_indices=True, fast_access
OrderedDict([(0, 42), (1, 42), (2, 'fortytwo), (3, 43)])
```

param name String description of the item(s) to find. Cannot be full names but the part of the names that are below a *run_XXXXXXXXX* group.

param include_default_run If results found under `run_ALL` should be accounted for every run or simply be ignored.

param use_indices If *True* the keys of the resulting dictionary are the run indices (e.g. 0,1,2,3), otherwise the keys are run names (e.g. *run_00000000*, *run_000000001*)

param fast_access Whether to return parameter or result instances or the values handled by these.

param with_links If links should be considered

param shortcuts If shortcuts are allowed and the trajectory can *hop* over nodes in the path.

param max_depth Maximum depth (relative to start node) how search should progress in tree. *None* means no depth limit. Only relevant if *shortcuts* are allowed.

param auto_load If data should be loaded from the storage service if it cannot be found in the current trajectory tree. Auto-loading will load group and leaf nodes currently not in memory and it will load data into empty leaves. Be aware that auto-loading does not work with shortcuts.

returnOrdered dictionary with run names or indices as keys and found items as values. Will only include runs where an item was actually found.

ATTENTION: This function is not available during a single run!

f_get_parameters(*fast_access=False*, *copy=True*)

Returns a dictionary containing the full parameter names as keys and the parametersor the parameter data items as values.

Parameters

- **fast_access** – Determines whether the parameter objects or their values are returned in the dictionary.
- **copy** – Whether the original dictionary or a shallow copy is returned. If you want the real dictionary please do not modify it at all! Not Copying and fast access do not work at the same time! Raises ValueError if fast access is true and copy false.

ReturnsDictionary containing the parameters.

RaisesValueError

f_get_results(*fast_access=False*, *copy=True*)

Returns a dictionary containing the full result names as keys and the corresponding result objects or result data items as values.

Parameters

- **fast_access** – Determines whether the result objects or their values are returned in the dictionary. Works only for results if they contain a single item with the name of the result.
- **copy** – Whether the original dictionary or a shallow copy is returned. If you want the real dictionary please do not modify it at all! Not Copying and fast access do not work at the same time! Raises ValueError if fast access is true and copy false.

ReturnsDictionary containing the results.

RaisesValueError

f_get_run_information(*name_or_idx=None*, *copy=True*)

Returns a dictionary containing information about a single run.

ONLY useful during a single run if `v_full_copy` was set to ``True`. Otherwise only the current run is available.

The information dictionaries have the following key, value pairings:

•completed: Boolean, whether a run was completed

•idx: Index of a run

•timestamp: Timestamp of the run as a float

•time: Formatted time string

•finish_timestamp: Timestamp of the finishing of the run

•runtime: Total runtime of the run in human readable format

•name: Name of the run

•parameter_summary:

A string summary of the explored parameter settings for the particular run

•short_environment_hexsha: The short version of the environment SHA-1 code

If no name or idx is given then a nested dictionary with keys as run names and info dictionaries as values is returned.

Parameters

•**name_or_idx** – str or int

•**copy** – Whether you want the dictionary used by the trajectory or a copy. Note if you want the real thing, please do not modify it, i.e. popping or adding stuff. This could mess up your whole trajectory.

ReturnsA run information dictionary or a nested dictionary of information dictionaries with the run names as keys.

f_get_run_names(*sort=True*)

Returns a list of run names.

ONLY useful for a single run during multiprocessing if `v_full_copy` was set to ``True``. Otherwise only the current run is available.

Parameters**sort** – Whether to get them sorted, will only require O(N) [and not O(N*log N)] since we use (sort of) bucket sort.

f_get_wildcards()

Returns a list of all defined wildcards

f_idx_to_run(*name_or_idx*)

Converts an integer idx to the corresponding single run name and vice versa.

Note during a single run ONLY useful if `v_full_copy` was set to True.

Parameters**name_or_idx** – Name of a single run or an integer index

ReturnsThe corresponding idx or name of the single run

Example usage:

```
>>> traj.f_idx_to_run(4)
'run_00000004'
>>> traj.f_idx_to_run('run_00000000')
0
```

f_is_completed(*name_or_id=None*)

Whether or not a given run is completed.

If no run is specified it is checked whether all runs were completed.

param name_or_id Nam or id of a run to check

returnTrue or False

ATTENTION: This function is not available during a single run!

f_is_empty()

Whether no results nor parameters have been added yet to the trajectory(ignores config).

ATTENTION: This function is not available during a single run!

f_is_wildcard(*wildcard*)
Checks if a given *wildcard* is really a wildcard.

f_iter_runs()
Makes the trajectory iterate over all runs.

Note that after a full iteration, the trajectory is set back to normal.

Thus, the following code snippet

```
for run_name in traj.f_iter_runs():

     # Do some stuff here...
```

is equivalent to

```
for run_name in traj.f_get_run_names(sort=True):
    traj.f_set_crun(run_name)

    # Do some stuff here...

traj.f_set_crun(None)
```

returnIterator over runs. The iterator itself will return the run names but modify the trajectory in each iteration and set it back do normal in the end.

ATTENTION: This function is not available during a single run!

f_load(*name=None*, *index=None*, *as_new=False*, *load_parameters=2*, *load_derived_parameters=1*, *load_results=1*, *load_other_data=1*, *recursive=True*, *load_data=None*, *max_depth=None*, *force=False*, *dynamic_imports=None*, *with_run_information=True*, *storage_service=None*, ***kwargs*)
Loads a trajectory via the storage service.

If you want to load individual results or parameters manually, you can take a look at *f_load_items()*. To only load subtrees check out *f_load_child()*.

For *f_load* you can pass the following arguments:

param name Name of the trajectory to be loaded. If no name or index is specified the current name of the trajectory is used.

param index If you don't specify a name you can specify an integer index instead. The corresponding trajectory in the hdf5 file at the index position is loaded (counting starts with 0). Negative indices are also allowed counting in reverse order. For instance, *-1* refers to the last trajectory in the file, *-2* to the second last, and so on.

param as_new Whether you want to rerun the experiments. So the trajectory is loaded only with parameters. The current trajectory name is kept in this case, which should be different from the trajectory name specified in the input parameter *name*. If you load *as_new=True* all parameters are unlocked. If you load *as_new=False* the current trajectory is replaced by the one on disk, i.e. name, timestamp, formatted time etc. are all taken from disk.

param load_parameters How parameters and config items are loaded

param load_derived_parameters How derived parameters are loaded

param load_results How results are loaded

You can specify how to load the parameters, derived parameters and results as follows:

- *pypet.pypetconstants.LOAD_NOTHING*: (0)

  Nothing is loaded.

- *pypet.pypetconstants.LOAD_SKELETON*: (1)

  The skeleton is loaded including annotations (See *Annotations*). This means that only empty *parameter* and *result* objects will be created and you can manually load the data into them afterwards. Note that *pypet.annotations.Annotations* do not count as data and they will be loaded because they are assumed to be small.

- *pypet.pypetconstants.LOAD_DATA*: (2)

  The whole data is loaded. Note in case you have non-empty leaves already in RAM, these are left untouched.

- *pypet.pypetconstants.OVERWRITE_DATA*: (3)

  As before, but non-empty nodes are emptied and reloaded.

Note that in all cases except *pypet.pypetconstants.LOAD_NOTHING*, annotations will be reloaded if the corresponding instance is created or the annotations of an existing instance were emptied before.

param recursive If data should be loaded recursively. If set to *None*, this is equivalent to set all data loading to *:const:'pypet.pypetconstants.LOAD_NOTHING*.

param load_data As the above, per default set to *None*. If not *None* the setting of *load_data* will overwrite the settings of *load_parameters*, *load_derived_parameters*, *load_results*, and *load_other_data*. This is more or less or shortcut if all types should be loaded the same.

param max_depth Maximum depth to load nodes (inclusive).

param force*pypet* will refuse to load trajectories that have been created using *pypet* with a different version number or a different python version. To force the load of a trajectory from a previous version simply set `force = True`. Note that it is not checked if other versions of packages differ from previous experiments, i.e. numpy, scipy, etc. But you can check for this manually. The versions of other packages can be found under `'config.environment.name_of_environment.versions.package_name'`.

param dynamic_imports If you've written a custom parameter that needs to be loaded dynamically during runtime, this needs to be specified here as a list of classes or strings naming classes and there module paths. For example: *dynamic_imports = ['pypet.parameter.PickleParameter',MyCustomParameter]*

If you only have a single class to import, you do not need the list brackets: *dynamic_imports = 'pypet.parameter.PickleParameter'*

The classes passed here are added for good and will be kept by the trajectory. Please add your dynamically imported classes only once.

param with_run_information If information about the individual runs should be loaded. If you have many runs, like 1,000,000 or more you can spare time by setting *with_run_information=False*. Note that *f_get_run_information* and *f_idx_to_run* do not work in such a case. Moreover, setting *v_idx* does not work either. If you load the trajectory without this information, be careful, this is not recommended.

param storage_service Pass a storage service used by the trajectory. Alternatively pass a constructor and other `**kwargs` are passed onto the constructor. Leave

None in combination with using no other kwargs, if you don't want to change the service the trajectory is currently using.

param kwargs Other arguments passed to the storage service constructor. Don't pass any other kwargs and `storage_service=None`, if you don't want to change the current service.

ATTENTION: This function is not available during a single run!

f_load_item(*item*, **args*, ***kwargs*)
Loads a single item, see also *f_load_items()*

f_load_items(*iterator*, **args*, ***kwargs*)
Loads parameters and results specified in *iterator*.

You can directly list the Parameter objects or just their names.

If names are given the *~pypet.naturalnaming.NNGroupNode.f_get* method is applied to find the parameters or results in the trajectory. Accordingly, the parameters and results you want to load must already exist in your trajectory (in RAM), probably they are just empty skeletons waiting desperately to handle data. If they do not exist in RAM yet, but have been stored to disk before, you can call `f_update_skeleton()` in order to bring your trajectory tree skeleton up to date. In case of a single run you can use the *f_load_child()* method to recursively load a subtree without any data. Then you can load the data of individual results or parameters one by one.

If want to load the whole trajectory at once or ALL results and parameters that are still empty take a look at *f_load()*. As mentioned before, to load subtrees of your trajectory you might want to check out *f_load_child()*.

To load a list of parameters or results with *f_load_items* you can pass the following arguments:

Parameters

- **iterator** – A list with parameters or results to be loaded.
- **only_empties** – Optional keyword argument (boolean), if *True* only empty parameters or results are passed to the storage service to get loaded. Non-empty parameters or results found in *iterator* are simply ignored.
- **args** – Additional arguments directly passed to the storage service
- **kwargs** – Additional keyword arguments directly passed to the storage service (except the kwarg *only_empties*)

If you use the standard hdf5 storage service, you can pass the following additional keyword arguments:

param load_only If you load a result, you can partially load it and ignore the rest of data items. Just specify the name of the data you want to load. You can also provide a list, for example *load_only='spikes'*, *load_only=['spikes','membrane_potential']*.

Be aware that you need to specify the names of parts as they were stored to HDF5. Depending on how your leaf construction works, this may differ from the names the data might have in your leaf in the trajectory container.

A warning is issued if data specified in *load_only* cannot be found in the instances specified in *iterator*.

param load_except Analogous to the above, but everything is loaded except names or parts specified in *load_except*. You cannot use *load_only* and *load_except* at the same time. If you do a ValueError is thrown.

A warning is issued if names listed in *load_except* are not part of the items to load.

f_load_skeleton()
Loads the full skeleton from the storage service.

This needs to be done after a successful exploration in order to update the trajectory tree with all results and derived parameters from the individual single runs. This will only add empty results and derived parameters (i.e. the skeleton) and load annotations.

ATTENTION: This function is not available during a single run!

f_lock_derived_parameters()
Locks all non-empty derived parameters

ATTENTION: This function is not available during a single run!

f_lock_parameters()
Locks all non-empty parameters

ATTENTION: This function is not available during a single run!

f_merge(*other_trajectory*, *trial_parameter=None*, *remove_duplicates=False*, *ignore_data=()*, *backup=True*, *move_data=False*, *delete_other_trajectory=False*, *keep_info=True*, *keep_other_trajectory_info=True*, *merge_config=True*, *consecutive_merge=False*)
Merges another trajectory into the current trajectory.

Both trajectories must live in the same space. This means both need to have the same parameters with similar types of values.

Note that links are also merged. There are exceptions: Links found under a generic run group called *run_ALL* or links linking to a node under such a group are NOT merged and simply skipped, because there is no straightforward way to resolve the link.

param other_trajectory Other trajectory instance to merge into the current one.

param trial_parameter If you have a particular parameter that specifies only the trial number, i.e. an integer parameter running form 0 to T1 and 0 to T2, the parameter is modified such that after merging it will cover the range 0 to T1+T2+1. T1 is the number of individual trials in the current trajectory and T2 number of trials in the other trajectory.

param remove_duplicates Whether you want to remove duplicate parameter points. Requires N1 * N2 (quadratic complexity in single runs). A ValueError is raised if no runs would be merged.

param ignore_data List of full names of data that should be ignored and not merged.

param backup If `True`, backs up both trajectories into files chosen automatically by the storage services. If you want to customize your backup use the *f_backup* function instead.

param move_data Tells the storage service to move data from one trajectory to the other instead of copying it.

If you use the HDF5 storage service and both trajectories are stored in the same file, merging is performed fast directly within the file. You can choose if you want to copy nodes ('move_nodes=False') from the other trajectory to the current one, or if you want to move them. Accordingly, the stored data is no longer accessible in the other trajectory.

param delete_other_trajectory If you want to delete the other trajectory after merging.

param keep_info If *True*, information about the merge is added to the trajectory *config* tree under *config.merge*.

param merge_config Whether or not to merge all config parameters under *.config.git*, *.config.environment*, and *.config.merge* of the other trajectory into the current one.

param keep_other_trajectory_info Whether to keep information like length, name, etc. of the other trajectory in case you want to keep all the information. Setting of *keep_other_trajectory_info* is irrelevant in case *keep_info=False*.

param consecutive_merge Can be set to *True* if you are about to merge several trajectories into the current one within a loop to avoid quadratic complexity. But remember to store your trajectory manually after all merges. Also make sure that all parameters and derived parameters are available in your current trajectory and load them before the consecutive merging. Also avoid specifying a *trial_parameter* and set *backup=False* to avoid quadratic complexity in case of consecutive merges.

If you cannot directly merge trajectories within one HDF5 file, a slow merging process is used. Results are loaded, stored, and emptied again one after the other. Might take some time!

Annotations of parameters and derived parameters under *.derived_parameters.trajectory* are NOT merged. If you wish to extract the annotations of these parameters you have to do that manually before merging. Note that annotations of results and derived parameters of single runs are copied, so you don't have to worry about these.

ATTENTION: This function is not available during a single run!

f_merge_many(*other_trajectories*, *ignore_data=()*, *move_data=False*, *delete_other_trajectory=False*, *keep_info=True*, *keep_other_trajectory_info=True*, *merge_config=True*, *backup=True*)
Can be used to merge several *other_trajectories* into your current one.

IMPORTANT *backup=True* only backs up the current trajectory not any of the *other_trajectories*. If you need a backup of these, do it manually.

Parameters as for `f_merge()`.

ATTENTION: This function is not available during a single run!

f_migrate(*new_name=None*, *in_store=False*, *new_storage_service=None*, ***kwargs*)
Can be called to rename and relocate the trajectory.

param new_name New name of the trajectory, None if you do not want to change the name.

param in_store Set this to True if the trajectory has been stored with the new name at the new file before and you just want to "switch back" to the location. If you migrate to a store used before and you do not set *in_store=True*, the storage service will throw a RuntimeError in case you store the Trajectory because it will assume that you try to store a new trajectory that accidentally has the very same name as another trajectory. If set to *True* and trajectory is not found in the file, the trajectory is simply stored to the file.

param new_storage_service New service where you want to migrate to. Leave none if you want to keep the olde one.

param kwargs Additional keyword arguments passed to the service. For instance, to change the file of the trajectory use `filename='my_new_file.hdf5`.

ATTENTION: This function is not available during a single run!

f_preset_config(*config_name*, **args*, ***kwargs*)
Similar to func:*~pypet.trajectory.Trajectory.f_preset_parameter*

ATTENTION: This function is not available during a single run!

f_preset_parameter(*param_name*, **args*, ***kwargs*)
Presets parameter value before a parameter is added.

Can be called before parameters are added to the Trajectory in order to change the values that are stored into the parameter on creation.

After creation of a parameter, the instance of the parameter is called with *param.f_set(*args,**kwargs)* with **args*, and ***kwargs* provided by the user with *f_preset_parameter*.

Before an experiment is carried out it is checked if all parameters that were marked were also preset.

param param_name The full name (!) of the parameter that is to be changed after its creation.

param args Arguments that will be used for changing the parameter's data

param kwargs Keyword arguments that will be used for changing the parameter's data

Example:

```
>>> traj.f_preset_parameter('groupA.param1', data=44)
>>> traj.f_add_parameter('groupA.param1', data=11)
>>> traj.parameters.groupA.param1
44
```

ATTENTION: This function is not available during a single run!

f_remove(*recursive=True*, *predicate=None*)
Recursively removes all children of the trajectory

Parameters

- **recursive** – Only here for consistency with signature of parent method. Cannot be set to *False* because the trajectory root node cannot be removed.
- **predicate** – Predicate which can evaluate for each node to `True` in order to remove the node or `False` if the node should be kept. Leave `None` if you want to remove all nodes.

f_remove_item(*item*, *recursive=False*)
Removes a single item, see *f_remove_items()*

f_remove_items(*iterator*, *recursive=False*)
Removes parameters, results or groups from the trajectory.

This function ONLY removes items from your current trajectory and does not delete data stored to disk. If you want to delete data from disk, take a look at *f_delete_items()*.

This will also remove all links if items are linked.

Parameters

- **iterator** – A sequence of items you want to remove. Either the instances themselves or strings with the names of the items.
- **recursive** – In case you want to remove group nodes, if the children should be removed, too.

f_restore_default()

Restores the default value in all explored parameters and sets thev_idx property back to -1 and v_crun to None.

ATTENTION: This function is not available during a single run!

f_set_crun(*name_or_idx*)
Can make the trajectory behave as during a particular single run.

It allows easier data analysis.

Has the following effects:

- *v_idx* and *v_crun* are set to the appropriate index and run name
- All explored parameters are set to the corresponding value in the exploration ranges, i.e. when you call *f_get()* (or fast access) on them you will get in return the value at the corresponding *v_idx* position in the exploration range.

- If you perform a search in the trajectory tree, the trajectory will only search the run subtree under *results* and *derived_parameters* with the corresponding index. For instance, if you use *f_set_crun('run_00000007')* or *f_set_crun(7)* and search for *traj.results.z* this will search for *z* only in the subtree *traj.results.run_00000007*. Yet, you can still explicitly name other subtrees, i.e. *traj.results.run_00000004.z* will still work.

ATTENTION: This function is not available during a single run!

f_set_properties(***kwargs*)
Sets properties like `v_fast_access`.

For example: `traj.f_set_properties(v_fast_access=True, v_auto_load=False)`

f_shrink(*force=False*)

Shrinks the trajectory and removes all exploration ranges from the parameters. Only possible if the trajectory has not been stored to disk before or was loaded as new.

param force Usually you cannot shrink the trajectory if it has been stored to disk, because there's no guarantee that it is actually shrunk if there still exist explored parameters on disk. In case you are certain that you did not store explored parameters to disk set or you deleted all of them from disk set *force=True*.

raisesTypeError if the trajectory was stored before.

ATTENTION: This function is not available during a single run!

f_store(*only_init=False*, *store_data=2*, *max_depth=None*)
Stores the trajectory to disk and recursively all data in the tree.

Parameters

- **only_init** – If you just want to initialise the store. If yes, only meta information about the trajectory is stored and none of the groups/leaves within the trajectory. Alternatively, you can pass *recursive=False*.
- **store_data** – Only considered if `only_init=False`. Choose of the following:
  - *pypet.pypetconstants.STORE_NOTHING*: (0)

    Nothing is store.
  - *pypet.pypetconstants.STORE_DATA_SKIPPING*: (1)

    Speedy version of normal `STORE_DATA` will entirely skip groups (but not their children) and leaves if they have been stored before. No new data is added in this case.
  - *pypet.pypetconstants.STORE_DATA*: (2)

    Stores every group and leave node. If they contain data that is not yet stored to disk it is added.
  - *pypet.pypetconstants.OVERWRITE_DATA*: (3)

    Stores all groups and leave nodes and will delete all data on disk and overwrite it with the current data in RAM.

    NOT RECOMMENDED! Overwriting data on disk fragments the HDF5 file and yields badly compressed large files. Better stick to the concept write once and read many!

If you use the HDF5 Storage Service usually (STORE_DATA (2)) only novel data is stored to disk. If you have results that have been stored to disk before only new data items are added and already present data is NOT overwritten.

Overwriting (OVERWRITE_DATA (3)) existing data with the HDF5 storage service is not recommended due to fragmentation of the HDF5 file. Better stick to the concept write once, but read often.

If you want to store individual parameters or results, you might want to take a look at `f_store_items()`. To store whole subtrees of your trajectory check out *f_store_child()*. Note both functions require that your trajectory was stored to disk with *f_store* at least once before.

ATTENTION: Calling *f_store* during a single run the behavior is different.

To avoid re-storing the full trajectory in every single run, which is redundant, only sub-trees of the trajectory are really stored.

The storage serivce looks for new data that is added below groups called *run_XXXXXXXXX* and stores it where *XXXXXXXXX* is the index of this run. The *only_init* parameter is ignored in this case. You can avoid this behavior by using the argument from below.

Parameters **max_depth** – Maximum depth to store tree (inclusive). During single runs *max_depth* is also counted from root.

f_store_item(*item*, **args*, ***kwargs*)
Stores a single item, see also *f_store_items()*.

f_store_items(*iterator*, **args*, ***kwargs*)
Stores individual items to disk.

This function is useful if you calculated very large results (or large derived parameters) during runtime and you want to write these to disk immediately and empty them afterwards to free some memory.

Instead of storing individual parameters or results you can also store whole subtrees with *f_store_child()*.

You can pass the following arguments to *f_store_items*:

Parameters

- **iterator** – An iterable containing the parameters or results to store, either their names or the instances. You can also pass group instances or names here to store the annotations of the groups.
- **non_empties** – Optional keyword argument (boolean), if *True* will only store the subset of provided items that are not empty. Empty parameters or results found in *iterator* are simply ignored.
- **args** – Additional arguments passed to the storage service
- **kwargs** – If you use the standard hdf5 storage service, you can pass the following additional keyword argument:

  param overwrite List names of parts of your item that should be erased and overwritten by the new data in your leaf. You can also set *overwrite=True* to overwrite all parts.

  For instance:

```
>>> traj.f_add_result('mygroup.myresult', partA=42, partB=44, partC=46)
>>> traj.f_store()
>>> traj.mygroup.myresult.partA = 333
>>> traj.mygroup.myresult.partB = 'I am going to change to a string'
>>> traj.f_store_item('mygroup.myresult', overwrite=['partA', 'partB'])
```

  Will store *'mygroup.myresult'* to disk again and overwrite the parts *'partA'* and *'partB'* with the new values *333* and *'I am going to change to a string'*. The data stored as *partC* is not changed.

  Be aware that you need to specify the names of parts as they were stored to HDF5. Depending on how your leaf construction works, this may differ from the names the data might have in your leaf in the trajectory container.

  Note that massive overwriting will fragment and blow up your HDF5 file. Try to avoid changing data on disk whenever you can.

RaisesTypeError:

If the (parent) trajectory has never been stored to disk. In this case use `pypet.trajectory.f_store()` first.

ValueError: If no item could be found to be stored.

Note if you use the standard hdf5 storage service, there are no additional arguments or keyword arguments to pass!

f_to_dict(*fast_access=False*, *short_names=False*, *copy=True*, *with_links=True*)

Returns a dictionary with pairings of (full) names as keys and instances/values.

Parameters

- **fast_access** – If True, parameter values are returned instead of the instances. Works also for results if they contain a single item with the name of the result.
- **short_names** – If true, keys are not full names but only the names. Raises a ValueError if the names are not unique.
- **copy** – If *fast_access=False* and *short_names=False* you can access the original data dictionary if you set *copy=False*. If you do that, please do not modify anything! Raises ValueError if *copy=False* and *fast_access=True* or *short_names=True*.
- **with_links** – If links should be ignored

Returnsdictionary

RaisesValueError

f_wildcard(*wildcard='$'*, *run_idx=None*)

#TODO

v_auto_load

Whether the trajectory should attempt to load data on the fly.

v_auto_run_prepend

If during run the *runs.run_XXXXXXXX* should be prepended if it is missing.

Is not considered for *f_add_leaf* and *f_add_group* which never prepend.

v_comment

Should be a nice descriptive comment

v_crun

Run name if you want to access the trajectory as a single run.

You can turn the trajectory to behave as during a single run if you set *v_crun* to a particular run name. Note that only string values are appropriate here, not indices. Check the *v_idx* property if you want to provide an index.

Alternatively instead of directly setting *v_crun* you can call `f_set_crun:()`.

Set to *None* to make the trajectory to turn everything back to default.

v_crun_

" Similar to `v_crun` but returns `'run_ALL'` if `v_crun` is `None`.

v_environment_hexsha

If the trajectory is used with an environment this returns the SHA-1 code of the environment.

v_environment_name

If the trajectory is used with an environment this returns the name of the environment.

v_fast_access

Whether parameter instances (False) or their values (True) are returned via natural naming.

Works also for results if they contain a single item with the name of the result.

Default is True.

v_filename

The name and path of the hdf5 file in case you use the HDF4StorageService

v_full_copy
Whether trajectory is copied fully during pickling or only the current parameter space point.

Note if the trajectory is copied as a whole, also during a single run you can access the full parameter space.

Changing *v_full_copy* will also change *v_full_copy* of all explored parameters!

v_idx
Index if you want to access the trajectory as during a single run.

You can turn the trajectory to behave as if during the execution of your runs if you set *v_idx* to a particular index. Note that only integer values are appropriate here, not names of runs.

Alternatively instead of directly setting *v_idx* you can call `f_set_crun:()`.

Set to *-1* to make the trajectory turn everything back to default.

v_is_run
`True` mak if trajectory is used during a single run initiated by an environment.

Accordingly, the functionality of the trajectory is reduced.

v_iter_recursive
Whether using *__iter__* should iterate only immediate children or recursively all nodes.

v_lazy_adding
If lazy additions are allowed.

I.e. *traj.par.x = 42* which adds a new parameter with value 42

v_max_depth
The maximum depth the tree should be searched if shortcuts are allowed.

Set to *None* if there should be no depth limit.

v_python
The version of python as a string that was used to create the trajectory

v_shortcuts
Whether shortcuts are allowed if accessing data via natural naming or squared bracket indexing.

v_standard_leaf
The standard constructor used if you add a generic leaf.

The constructor is only used if you do not add items under the usual four subtrees (*parameters*, *derived_parameters*, *config*, *results*).

v_standard_parameter
The standard parameter used for parameter creation

v_standard_result
The standard result class used for result creation

v_storage_service
The service that can store the trajectory to disk or wherever.

Default is None or if a filename was provided on construction the *HDF5StorageService*.

v_time
Formatted time string of the time the trajectory or run was created.

v_timestamp
Float timestamp of creation time

v_trajectory_name
Name of the (parent) trajectory

v_trajectory_time
Time (parent) trajectory was created

v_trajectory_timestamp
Float timestamp when (parent) trajectory was created

v_version
The version of *pypet* that was used to create the trajectory

v_with_links
Whether links should be considered in case using natural naming or squared bracket indexing

pypet.trajectory.**load_trajectory**(*name=None*, *index=None*, *as_new=False*, *load_parameters=2*, *load_derived_parameters=1*, *load_results=1*, *load_other_data=1*, *recursive=True*, *load_data=None*, *max_depth=None*, *force=False*, *dynamic_imports=None*, *new_name='my_trajectory'*, *add_time=True*, *wildcard_functions=None*, *with_run_information=True*, *storage_service=<class 'pypet.storageservice.HDF5StorageService'>*, ***kwargs*)

Helper function that creates a novel trajectory and loads it from disk.

For the parameters see *f_load()*.

`new_name` and `add_time` are only used in case `as_new` is `True`. Accordingly, they determine the new name of trajectory.

3.2.3 NNGroupNode

class pypet.naturalnaming.**NNGroupNode**(*full_name=''*, *trajectory=None*, *comment=''*)

A group node hanging somewhere under the trajectory or single run root node.

You can add other groups or parameters/results to it.

f_add_group(**args*, ***kwargs*)

Adds an empty generic group under the current node.

You can add to a generic group anywhere you want. So you are free to build your parameter tree with any structure. You do not necessarily have to follow the four subtrees *config*, *parameters*, *derived_parameters*, *results*.

If you are operating within these subtrees this simply calls the corresponding adding function.

Be aware that if you are within a single run and you add items not below a group *run_XXXXXXXX* that you have to manually save the items. Otherwise they will be lost after the single run is completed.

f_add_leaf(**args*, ***kwargs*)

Adds an empty generic leaf under the current node.

You can add to a generic leaves anywhere you want. So you are free to build your trajectory tree with any structure. You do not necessarily have to follow the four subtrees *config*, *parameters*, *derived_parameters*, *results*.

If you are operating within these subtrees this simply calls the corresponding adding function.

Be aware that if you are within a single run and you add items not below a group *run_XXXXXXXX* that you have to manually save the items. Otherwise they will be lost after the single run is completed.

f_add_link(*name_or_item*, *full_name_or_item=None*)

Adds a link to an existing node.

Can be called as `node.f_add_link(other_node)` this will add a link the *other_node* with the link name as the name of the node.

Or can be called as `node.f_add_link(name, other_node)` to add a link to the *other_node* and the given *name* of the link.

In contrast to addition of groups and leaves, colon separated names are **not** allowed, i.e. `node.f_add_link('mygroup.mylink', other_node)` does not work.

f_ann_to_str()
Returns annotations as string

Equivalent to *v_annotations.f_ann_to_str()*

f_ann_to_string()
Returns annotations as string

Equivalent to *v_annotations.f_ann_to_str()*

DEPRECATED: Please use *f_ann_to_str()* instead.

f_children()
Returns the number of children of the group

f_contains(*item*, *with_links=True*, *shortcuts=False*, *max_depth=None*)
Checks if the node contains a specific parameter or result.

It is checked if the item can be found via the `f_get()` method.

Parameters

- **item** – Parameter/Result name or instance.

  If a parameter or result instance is supplied it is also checked if the provided item and the found item are exactly the same instance, i.e. *id(item)==id(found_item)*.
- **with_links** – If links are considered.
- **shortcuts** – Shortcuts is *False* the name you supply must be found in the tree WITHOUT hopping over nodes in between. If *shortcuts=False* and you supply a non colon separated (short) name, than the name must be found in the immediate children of your current node. Otherwise searching via shortcuts is allowed.
- **max_depth** – If shortcuts is *True* than the maximum search depth can be specified. *None* means no limit.

Returns True or False

f_debug()
Creates a dummy object containing the whole tree to make unfolding easier.

This method is only useful for debugging purposes. If you use an IDE and want to unfold the trajectory tree, you always need to open the private attribute *_children*. Use to this function to create a new object that contains the tree structure in its attributes.

Manipulating the returned object does not change the original tree!

f_get(*name*, *fast_access=False*, *with_links=True*, *shortcuts=True*, *max_depth=None*, *auto_load=False*)
Searches and returns an item (parameter/result/group node) with the given *name*.

Parameters

- **name** – Name of the item (full name or parts of the full name)
- **fast_access** – Whether fast access should be applied.
- **with_links** – If links are considered. Cannot be set to `False` if `auto_load` is `True`.
- **shortcuts** – If shortcuts are allowed and the trajectory can *hop* over nodes in the path.
- **max_depth** – Maximum depth relative to starting node (inclusive). *None* means no depth limit.
- **auto_load** – If data should be loaded from the storage service if it cannot be found in the current trajectory tree. Auto-loading will load group and leaf nodes currently not in memory and it will load data into empty leaves. Be aware that auto-loading does not work with shortcuts.

Returns The found instance (result/parameter/group node) or if fast access is True and you found a parameter or result that supports fast access, the contained value is returned.

Raises AttributeError: If no node with the given name can be found

NotUniqueNodeError
In case of forward search if more than one candidate node is found within a particular depth of the tree. In case of backwards search if more than one candidate is found regardless of the depth.

DataNotInStorageError:

In case auto-loading fails
Any exception raised by the StorageService in case auto-loading is enabled

f_get_all(*name*, *max_depth=None*, *shortcuts=True*)
Searches for all occurrences of *name* under *node*.

Links are NOT considered since nodes are searched bottom up in the tree.

Parameters
- **node** – Start node
- **name** – Name of what to look for, can be separated by colons, i.e. `'mygroupA.mygroupB.myparam'`.
- **max_depth** – Maximum search depth relative to start node. *None* for no limit.
- **shortcuts** – If shortcuts are allowed, otherwise the stated name defines a consecutive name.For instance. `'mygroupA.mygroupB.myparam'` would also find `mygroupA.mygroupX.mygroupB.mygroupY.myparam` if shortcuts are allowed, otherwise not.

ReturnsList of nodes that match the name, empty list if nothing was found.

f_get_annotations(**args*)
Returns annotations

Equivalent to *v_annotations.f_get(*args)*

f_get_children(*copy=True*)
Returns a children dictionary.

Parameters
copy – Whether the group's original dictionary or a shallow copy is returned. If you want the real dictionary please do not modify it at all!
ReturnsDictionary of nodes

f_get_class_name()
Returns the class name of the parameter or result or group.

Equivalent to *obj.__class__.__name__*

f_get_default(*name*, *default=None*, *fast_access=True*, *with_links=True*, *shortcuts=True*, *max_depth=None*, *auto_load=False*)
Similar to *f_get*, but returns the default value if *name* is not found in the trajectory.

This function uses the *f_get* method and will return the default value in case *f_get* raises an AttributeError or a DataNotInStorageError. Other errors are not handled.

In contrast to *f_get*, fast access is True by default.

f_get_groups(*copy=True*)
Returns a dictionary of groups hanging immediately below this group.

Parameters
copy – Whether the group's original dictionary or a shallow copy is returned. If you want the real dictionary please do not modify it at all!
ReturnsDictionary of nodes

f_get_leaves(*copy=True*)
Returns a dictionary of all leaves hanging immediately below this group.

Parameters
copy – Whether the group's original dictionary or a shallow copy is returned. If you want the real dictionary please do not modify it at all!
ReturnsDictionary of nodes

f_get_links(*copy=True*)
Returns a link dictionary.

Parameters
copy – Whether the group's original dictionary or a shallow copy is returned. If you want the real dictionary please do not modify it at all!
ReturnsDictionary of nodes

f_get_parent()
Returns the parent of the node.

Raises a TypeError if current node is root.

f_get_root()
Returns the root node of the tree.

Either a full trajectory or a single run container.

f_groups()
Returns the number of immediate groups of the group

f_has_children()
Checks if node has children or not

f_has_groups()
Checks if node has groups or not

f_has_leaves()
Checks if node has leaves or not

f_has_links()
Checks if node has children or not

f_is_root()
Whether the group is root (True for the trajectory and a single run object)

DEPRECATED: Please use property v_is_root!

f_iter_leaves(*with_links=True*)
Iterates (recursively) over all leaves hanging below the current group.
Parameters**with_links** – If links should be ignored, leaves hanging below linked nodes are not listed.
ReturnsIterator over all leaf nodes

f_iter_nodes(*recursive=True*, *with_links=True*, *max_depth=None*, *predicate=None*)
Iterates recursively (default) over nodes hanging below this group.
Parameters
- **recursive** – Whether to iterate the whole sub tree or only immediate children.
- **with_links** – If links should be considered
- **max_depth** – Maximum depth in search tree relative to start node (inclusive)
- **predicate** – A predicate function that is applied for each node and only returns the node if it evaluates to `True`. If `False` and you iterate recursively also the children are spared.

  Leave to *None* if you don't want to filter and simply iterate over all nodes.

  For example, to iterate only over groups you could use:

```
>>> traj.f_iter_nodes(recursive=True, predicate=lambda x: x.v_is_group)
```

  To blind out all runs except for a particular set, you can simply pass a tuple of run indices with -1 referring to the `run_ALL` node.

  For instance

```
>>> traj.f_iter_nodes(recursive=True, predicate=(0,3,-1))
```

  Will blind out all nodes hanging below a group named `run_XXXXXXXXX` (including the group) except `run_00000000`, `run_00000003`, and `run_ALL`.

ReturnsIterator over nodes

f_leaves()
Returns the number of immediate leaves of the group

f_links()
Returns the number of links of the group

f_load(*recursive=True*, *load_data=2*, *max_depth=None*)
Loads a group from disk.

Parameters

- **recursive** – Default is `True`. Whether recursively all nodes below the current node should be loaded, too. Note that links are never evaluated recursively. Only the linked node will be loaded if it does not exist in the tree, yet. Any nodes or links of this linked node are not loaded.
- **load_data** – Flag how to load the data. For how to choose 'load_data' see *Loading*.
- **max_depth** – In case *recursive* is *True*, you can specify the maximum depth to load load data relative from current node.

Returns The node itself.

f_load_child(*name*, *recursive=False*, *load_data=2*, *max_depth=None*)

Loads a child or recursively a subtree from disk.

Parameters

- **name** – Name of child to load. If grouped ('groupA.groupB.childC') the path along the way to last node in the chain is loaded. Shortcuts are NOT allowed!
- **recursive** – Whether recursively all nodes below the last child should be loaded, too. Note that links are never evaluated recursively. Only the linked node will be loaded if it does not exist in the tree, yet. Any nodes or links of this linked node are not loaded.
- **load_data** – Flag how to load the data. For how to choose 'load_data' see *Loading*.
- **max_depth** – In case *recursive* is *True*, you can specify the maximum depth to load load data relative from current node. Leave *None* if you don't want to limit the depth.

Returns The loaded child, in case of grouping ('groupA.groupB.childC') the last node (here 'childC') is returned.

f_remove(*recursive=True*, *predicate=None*)

Recursively removes the group and all it's children.

Parameters

- **recursive** – If removal should be applied recursively. If not, node can only be removed if it has no children.
- **predicate** – In case of recursive removal, you can selectively remove nodes in the tree. Predicate which can evaluate for each node to `True` in order to remove the node or `False` if the node should be kept. Leave `None` if you want to remove all nodes.

f_remove_child(*name*, *recursive=False*, *predicate=None*)

Removes a child of the group.

Note that groups and leaves are only removed from the current trajectory in RAM. If the trajectory is stored to disk, this data is not affected. Thus, removing children can be only be used to free RAM memory!

If you want to free memory on disk via your storage service, use *f_delete_items()* of your trajectory.

Parameters

- **name** – Name of child, naming by grouping is NOT allowed ('groupA.groupB.childC'), child must be direct successor of current node.
- **recursive** – Must be true if child is a group that has children. Will remove the whole subtree in this case. Otherwise a Type Error is thrown.
- **predicate** – Predicate which can evaluate for each node to `True` in order to remove the node or `False` if the node should be kept. Leave `None` if you want to remove all nodes.

Raises TypeError if recursive is false but there are children below the node.

ValueError if child does not exist.

f_remove_link(*name*)

Removes a link from from the current group node with a given name.

Does not delete the link from the hard drive. If you want to do this, checkout *f_delete_links()*

f_set_annotations(**args*, ***kwargs*)

Sets annotations

Equivalent to calling *v_annotations.f_set(*args,**kwargs)*

f_store(*recursive=True*, *store_data=2*, *max_depth=None*)

Stores a group node to disk

Parameters

- **recursive** – Whether recursively all children should be stored too. Default is `True`.
- **store_data** – For how to choose 'store_data' see *Storing*.
- **max_depth** – In case *recursive* is *True*, you can specify the maximum depth to store data relative from current node. Leave *None* if you don't want to limit the depth.

f_store_child(*name*, *recursive=False*, *store_data=2*, *max_depth=None*)
Stores a child or recursively a subtree to disk.

Parameters
- **name** – Name of child to store. If grouped ('groupA.groupB.childC') the path along the way to last node in the chain is stored. Shortcuts are NOT allowed!
- **recursive** – Whether recursively all children's children should be stored too.
- **store_data** – For how to choose 'store_data' see *Storing*.
- **max_depth** – In case *recursive* is *True*, you can specify the maximum depth to store data relative from current node. Leave *None* if you don't want to limit the depth.

Raises ValueError if the child does not exist.

f_to_dict(*fast_access=False*, *short_names=False*, *with_links=True*)
Returns a dictionary with pairings of (full) names as keys and instances as values.

This will iteratively traverse the tree and add all nodes below this group to the dictionary.

Parameters
- **fast_access** – If True parameter or result values are returned instead of the instances.
- **short_names** – If true keys are not full names but only the names. Raises a ValueError if the names are not unique.
- **with_links** – If links should be considered

Returns dictionary

Raises ValueError

v_annotations
Annotation feature of a trajectory node.

Store some short additional information about your nodes here. If you use the standard HDF5 storage service, they will be stored as hdf5 node attributes.

v_branch
The name of the branch/subtree, i.e. the first node below the root.

The empty string in case of root itself.

v_comment
Should be a nice descriptive comment

v_depth
Depth of the node in the trajectory tree.

v_full_name
The full name, relative to the root node.

The full name of a trajectory or single run is the empty string since it is root.

v_is_group
Whether node is a group or not (i.e. it is a leaf node)

v_is_leaf
Whether node is a leaf or not (i.e. it is a group node)

v_is_root
Whether the group is root (True for the trajectory and a single run object)

v_leaf
Whether node is a leaf or not (i.e. it is a group node)

DEPRECATED: Please use v_is_leaf!

v_location
Location relative to the root node.

The location of a trajectory or single run is the empty string since it is root.

v_name
Name of the node

v_root
Link to the root of the tree, i.e. the trajectory

v_run_branch
If this node is hanging below a branch named *run_XXXXXXXXX*.

The branch name is either the name of a single run (e.g. 'run_00000009') or 'trajectory'.

v_stored
Whether or not this tree node has been stored to disk before.

3.2.4 ParameterGroup

class pypet.naturalnaming.**ParameterGroup**(*full_name=''*, *trajectory=None*, *comment=''*)
Group node in your trajectory, hanging below *traj.parameters*.

You can add other groups or parameters to it.

f_add_parameter(**args*, ***kwargs*)
Adds a parameter under the current node.

There are two ways to add a new parameter either by adding a parameter instance:

```
>>> new_parameter = Parameter('group1.group2.myparam', data=42, comment='Example!')
>>> traj.f_add_parameter(new_parameter)
```

Or by passing the values directly to the function, with the name being the first (non-keyword!) argument:

```
>>> traj.f_add_parameter('group1.group2.myparam', data=42, comment='Example!')
```

If you want to create a different parameter than the standard parameter, you can give the constructor as the first (non-keyword!) argument followed by the name (non-keyword!):

```
>>> traj.f_add_parameter(PickleParameter,'group1.group2.myparam', data=42, comment='Example!
```

The full name of the current node is added as a prefix to the given parameter name. If the current node is the trajectory the prefix *'parameters'* is added to the name.

f_add_parameter_group(**args*, ***kwargs*)
Adds an empty parameter group under the current node.

Can be called with `f_add_parameter_group('MyName', 'this is an informative comment')` or `f_add_parameter_group(name='MyName', comment='This is an informative comment')` or with a given new group instance: `f_add_parameter_group(ParameterGroup('MyName', comment='This is a comment'))`.

Adds the full name of the current node as prefix to the name of the group. If current node is the trajectory (root), the prefix *'parameters'* is added to the full name.

The *name* can also contain subgroups separated via colons, for example: *name=subgroup1.subgroup2.subgroup3*. These other parent groups will be automatically created.

f_apar(**args*, ***kwargs*)
Adds a parameter under the current node.

There are two ways to add a new parameter either by adding a parameter instance:

```
>>> new_parameter = Parameter('group1.group2.myparam', data=42, comment='Example!')
>>> traj.f_add_parameter(new_parameter)
```

Or by passing the values directly to the function, with the name being the first (non-keyword!) argument:

```
>>> traj.f_add_parameter('group1.group2.myparam', data=42, comment='Example!')
```

If you want to create a different parameter than the standard parameter, you can give the constructor as the first (non-keyword!) argument followed by the name (non-keyword!):

```
>>> traj.f_add_parameter(PickleParameter,'group1.group2.myparam', data=42, comment='Example
```

The full name of the current node is added as a prefix to the given parameter name. If the current node is the trajectory the prefix *'parameters'* is added to the name.

3.2.5 ConfigGroup

class pypet.naturalnaming.**ConfigGroup**(*full_name=''*, *trajectory=None*, *comment=''*)

Group node in your trajectory, hanging below *traj.config*.

You can add other groups or parameters to it.

f_aconf(**args*, ***kwargs*)

Adds a config parameter under the current group.

Similar to *f_add_parameter()*.

If current group is the trajectory the prefix *'config'* is added to the name.

f_add_config(**args*, ***kwargs*)

Adds a config parameter under the current group.

Similar to *f_add_parameter()*.

If current group is the trajectory the prefix *'config'* is added to the name.

f_add_config_group(**args*, ***kwargs*)

Adds an empty config group under the current node.

Adds the full name of the current node as prefix to the name of the group. If current node is the trajectory (root), the prefix *'config'* is added to the full name.

The *name* can also contain subgroups separated via colons, for example: *name=subgroup1.subgroup2.subgroup3*. These other parent groups will be automatically be created.

3.2.6 DerivedParameterGroup

class pypet.naturalnaming.**DerivedParameterGroup**(*full_name=''*, *trajectory=None*, *comment=''*)

Group node in your trajectory, hanging below *traj.derived_parameters*.

You can add other groups or parameters to it.

f_add_derived_parameter(**args*, ***kwargs*)

Adds a derived parameter under the current group.

Similar to *f_add_parameter()*

Naming prefixes are added as in *f_add_derived_parameter_group()*

f_add_derived_parameter_group(**args*, ***kwargs*)
Adds an empty derived parameter group under the current node.

Adds the full name of the current node as prefix to the name of the group. If current node is a single run (root) adds the prefix *'derived_parameters.runs.run_08%d%'* to the full name where *'08%d'* is replaced by the index of the current run.

The *name* can also contain subgroups separated via colons, for example: *name=subgroup1.subgroup2.subgroup3*. These other parent groups will be automatically be created.

f_adpar(**args*, ***kwargs*)
Adds a derived parameter under the current group.

Similar to *f_add_parameter()*

Naming prefixes are added as in *f_add_derived_parameter_group()*

3.2.7 ResultGroup

class pypet.naturalnaming.**ResultGroup**(*full_name=''*, *trajectory=None*, *comment=''*)
Group node in your trajectory, hanging below *traj.results*.

You can add other groups or results to it.

f_add_result(**args*, ***kwargs*)
Adds a result under the current node.

There are two ways to add a new result either by adding a result instance:

```
>>> new_result = Result('group1.group2.myresult', 1666, x=3, y=4, comment='Example!')
>>> traj.f_add_result(new_result)
```

Or by passing the values directly to the function, with the name being the first (non-keyword!) argument:

```
>>> traj.f_add_result('group1.group2.myresult', 1666, x=3, y=3,comment='Example!')
```

If you want to create a different result than the standard result, you can give the constructor as the first (non-keyword!) argument followed by the name (non-keyword!):

```
>>> traj.f_add_result(PickleResult,'group1.group2.myresult', 1666, x=3, y=3, comment='Ex
```

Additional arguments (here *1666*) or keyword arguments (here *x=3, y=3*) are passed onto the constructor of the result.

Adds the full name of the current node as prefix to the name of the result. If current node is a single run (root) adds the prefix *'results.runs.run_08%d%'* to the full name where *'08%d'* is replaced by the index of the current run.

f_add_result_group(**args*, ***kwargs*)
Adds an empty result group under the current node.

Adds the full name of the current node as prefix to the name of the group. If current node is a single run (root) adds the prefix *'results.runs.run_08%d%'* to the full name where *'08%d'* is replaced by the index of the current run.

The *name* can also contain subgroups separated via colons, for example: *name=subgroup1.subgroup2.subgroup3*. These other parent groups will be automatically be created.

f_ares(**args*, ***kwargs*)

Adds a result under the current node.

There are two ways to add a new result either by adding a result instance:

```
>>> new_result = Result('group1.group2.myresult', 1666, x=3, y=4, comment='Example!')
>>> traj.f_add_result(new_result)
```

Or by passing the values directly to the function, with the name being the first (non-keyword!) argument:

```
>>> traj.f_add_result('group1.group2.myresult', 1666, x=3, y=3,comment='Example!')
```

If you want to create a different result than the standard result, you can give the constructor as the first (non-keyword!) argument followed by the name (non-keyword!):

```
>>> traj.f_add_result(PickleResult,'group1.group2.myresult', 1666, x=3, y=3, comment='Ex
```

Additional arguments (here *1666*) or keyword arguments (here *x=3, y=3*) are passed onto the constructor of the result.

Adds the full name of the current node as prefix to the name of the result. If current node is a single run (root) adds the prefix *'results.runs.run_08%d%'* to the full name where *'08%d'* is replaced by the index of the current run.

3.3 Parameters and Results

This module contains implementations of result and parameter containers.

Results and parameters are the leaf nodes of the *Trajectory* tree. Instances of results can only be found under the subtree *traj.results*, whereas parameters are used to handle data kept under *traj.config*, *traj.parameters*, and *traj.derived_parameters*.

Result objects can handle more than one data item and heterogeneous data. On the contrary, parameters only handle single data items. However, they can contain ranges - arrays of homogeneous data items - to allow parameter exploration.

The module contains the following parameters:

- *BaseParameter*

  Abstract base class to define the parameter interface

- *Parameter*

  Standard parameter that handles a variety of different data types.

- *ArrayParameter*

  Parameter class for larger numpy arrays and python tuples

- *SparseParameter*

  Parameter for Scipy sparse matrices

- *PickleParameter*

  Parameter that can handle all objects that can be pickled

The module contains the following results:

- *BaseResult*

  Abstract base class to define the result interface

- *Result*

Standard result that handles a variety of different data types

- *SparseResult*

  Result that can handle Scipy sparse matrices

- *PickleResult*

  Result that can handle all objects that can be pickled

Moreover, part of this module is also the *ObjectTable*. This is a specification of pandas DataFrames which maintains data types. It prevents auto-conversion of data to numpy data types, like python integers to numpy 64 bit integers, for instance.

3.3.1 Parameter Quicklinks

f_set	Sets a data value for a parameter.
f_get	Returns the current data value of the parameter and locks the parameter.
f_empty	Erases all data in the parameter.
f_get_range	Returns a python tuple containing the exploration range.
f_has_range	If the parameter has a range.
f_supports	Checks if input data is supported by the parameter.

3.3.2 Result Quicklinks

f_set	Method to put data into the result.
f_get	Returns items handled by the result.
f_empty	Removes all data from the result or parameter.
f_to_dict	Returns all handled data as a dictionary.

3.3.3 Parameter

class pypet.parameter.**Parameter** (*full_name*, *data=None*, *comment=''*)

The standard container that handles access to simulation parameters.

Parameters are simple container objects for data values. They handle single values as well as the so called exploration range. An array containing multiple values which are accessed one after the other in individual simulation runs.

Parameter exploration is usually initiated through the trajectory see *:func:~pypet.trajectory.Trajectory.f_explore* and *:func:~pypet.trajectory.Trajectory.f_expand*.

To access the parameter's data value one can call the *f_get()* method.

Parameters support the concept of locking. Once a value of the parameter has been accessed, the parameter cannot be changed anymore unless it is explicitly unlocked using *f_unlock()*. Locking prevents parameters from being changed during runtime of a simulation.

Supported data values for the parameter are

•python natives (int, long, str, bool, float, complex),
•numpy natives, arrays and matrices of type np.int8-64, np.uint8-64, np.float32-64, np.complex, np.str
•python homogeneous non-nested tuples

Note that for larger numpy arrays it is recommended to use the *ArrayParameter*.

In case you create a new parameter you can pass the following arguments:

Parameters

•**full_name** – The full name of the parameter. Grouping can be achieved by using colons.

- **data** – A data value that is handled by the parameter. It is checked whether the parameter *f_supports()* the data. If not a TypeError is thrown. If the parameter becomes explored, the data value is kept as a default. After simulation the default value will be restored.

  The data can be accessed as follows:

```
>>> param.f_get()
42
```

  Or using >>> param.data 42

  [It is not *v_data* because the data is supposed to be part of the trajectory tree or extension of the natural naming scheme and not considered as an attribute/variable of the parameter container.]

  To change the data after parameter creation one can call *f_set()*:

```
>>> param.f_set(43)
>>> param.f_get()
43
```

- **comment** – A useful comment describing the parameter. The comment can be changed later on using the 'v_comment' variable.

```
>>> param.v_comment = 'Example comment'
>>> print param.v_comment
'Example comment'
```

RaisesTypeError: If *data* is not supported by the parameter.

Example usage:

```
>>> param = Parameter('traffic.mobiles.ncars',data=42, comment='I am a neat example')
```

f_ann_to_str()

Returns annotations as string

Equivalent to *v_annotations.f_ann_to_str()*

f_ann_to_string()

Returns annotations as string

Equivalent to *v_annotations.f_ann_to_str()*

DEPRECATED: Please use *f_ann_to_str()* instead.

f_empty()

Erases all data in the parameter.

Does not erase data from disk. So if the parameter has been stored with a service to disk and is emptied, it can be restored by loading from disk.

RaisesParameterLockedException: If the parameter is locked.

f_get()

Returns the current data value of the parameter and locks the parameter.

RaisesTypeError if the parameter is empty

Example usage:

```
>>> param = Parameter('groupA.groupB.myparam', comment='I am a neat example')
>>> param.f_set(44.0)
>>> param.f_get()
44.0:
```

f_get_annotations(**args*)
Returns annotations

Equivalent to *v_annotations.f_get(*args)*

f_get_array()
Returns an iterable to iterate over the values of the exploration range.

Note that the returned values should be either a copy of the exploration range or the array must be immutable, for example a python tuple.

ReturnsImmutable sequence

RaisesTypeError if the parameter is not explored

Example usage:

```
>>> param = Parameter('groupA.groupB.myparam',data=22, comment='I am a neat example')
>>> param._explore([42,43,43])
>>> param.f_get_array()
(42,43,44)
```

DEPRECATED: Use *f_get_range()* instead!

f_get_class_name()
Returns the name of the class i.e. *return self.__class__.__name__*

f_get_default()
Returns the default value of the parameter and locks it.

f_get_range()
Returns a python tuple containing the exploration range.

Example usage:

```
>>> param = Parameter('groupA.groupB.myparam',data=22, comment='I am a neat example')
>>> param._explore([42,43,43])
>>> param.f_get_range()
(42,43,44)
```

RaisesTypeError: If parameter is not explored.

f_get_range_length()
Returns the length of the parameter range.

Raises TypeError if the parameter has no range.

Does not need to be implemented if the parameter supports `__len__` appropriately.

f_has_range()
If the parameter has a range.

Does not have to be *True* if the parameter is explored. The range might be removed during pickling to save memory. Accordingly, *v_explored* remains *True* whereas *f_has_range* is *False*.

f_is_array()
Returns true if the parameter is explored and contains a range array.

DEPRECATED: Use *f_has_range()* instead.

f_is_empty()
True if no data has been assigned to the parameter.
Example usage:

```
>>> param = Parameter('myname.is.example', comment='I am _empty!')
>>> param.f_is_empty()
True
>>> param.f_set(444)
>>> param.f_is_empty()
False
```

True if no data has been assigned to the parameter.

Example usage:

```
>>> param = Parameter('myname.is.example', comment='I am _empty!')
>>> param.f_is_empty()
True
>>> param.f_set(444)
>>> param.f_is_empty()
False
```

f_is_root()

Whether the group is root (True for the trajectory and a single run object)

DEPRECATED: Please use property v_is_root!

f_lock()

Locks the parameter and forbids further manipulation.

Changing the data value or exploration range of the parameter are no longer allowed.

f_set(*data*)

Sets a data value for a parameter.

Example usage:

```
>>> param = Parameter('groupA.groupB.myparam', comment='I am a neat example')
>>> param.f_set(44.0)
>>> param.f_get()
44.0
```

RaisesParameterLockedException: If parameter is locked

TypeError: If the type of the data value is not supported by the parameter

f_set_annotations(**args*, ***kwargs*)

Sets annotations

Equivalent to calling *v_annotations.f_set(*args,**kwargs)*

f_supports(*data*)

Checks if input data is supported by the parameter.

f_supports_fast_access()

Checks if parameter supports fast access.

A parameter supports fast access if it is NOT empty!

f_unlock()

Unlocks the locked parameter.

Please use it very carefully, or best do not use this function at all. There should better be no reason to unlock a locked parameter! The only exception I can think of is to unlock a large derived parameter after usage to subsequently call *f_empty()* to clear memory.

f_val_to_str()

String summary of the value handled by the parameter.

Note that representing the parameter as a string accesses its value, but for simpler debugging, this does not lock the parameter or counts as usage!

Calls *__repr__* of the contained value.

v_annotations

Annotation feature of a trajectory node.

Store some short additional information about your nodes here. If you use the standard HDF5 storage service, they will be stored as hdf5 node attributes.

v_branch
The name of the branch/subtree, i.e. the first node below the root.

The empty string in case of root itself.

v_comment
Should be a nice descriptive comment

v_depth
Depth of the node in the trajectory tree.

v_explored
Whether parameter is explored.

Does not necessarily have to be similar to *f_has_range()* since the range can be deleted on pickling and the parameter remains explored.

v_fast_accessible
Whether or not fast access can be supported by the Parameter or Result

DEPRECATED: Please use function *f_supports_fast_access* instead!

v_full_copy
Whether or not the full parameter including the range or only the current data is copied during pickling.

If you run your simulations in multiprocessing mode, the whole trajectory and all parameters need to be pickled and are sent to the individual processes. Each process than runs an individual point in the parameter space. As a consequence, you do not need the full ranges during these calculations. Thus, if the full copy mode is set to *False* the parameter is pickled without the range array and you can save memory.

If you want to access the full range during individual runs, you need to set *v_full_copy* to *True*.

It is recommended NOT to do that in order to save memory and also do obey the philosophy that individual simulation runs are independent.

Example usage:

```
>>> import pickle
>>> param = Parameter('examples.fullcopy', data=333, comment='I show you how the copy mo
>>> param._explore([1,2,3,4])
>>> dump=pickle.dumps(param)
>>> newparam = pickle.loads(dump)
>>> newparam.f_get_range()
TypeError
```

```
>>> param.v_full_copy=True
>>> dump = pickle.dumps(param)
>>> newparam=pickle.loads(dump)
>>> newparam.f_get_range()
(1,2,3,4)
```

v_full_name
The full name, relative to the root node.

The full name of a trajectory or single run is the empty string since it is root.

v_is_group
Whether node is a group or not (i.e. it is a leaf node)

v_is_leaf
Whether node is a leaf or not (i.e. it is a group node)

v_is_parameter
Whether the node is a parameter or not (i.e. a result)

v_is_root
Whether the group is root (True for the trajectory and a single run object)

v_leaf
Whether node is a leaf or not (i.e. it is a group node)

DEPRECATED: Please use v_is_leaf!

v_location
Location relative to the root node.

The location of a trajectory or single run is the empty string since it is root.

v_locked
Whether or not the parameter is locked and prevents further modification

v_name
Name of the node

v_parameter
Whether the node is a parameter or not (i.e. a result)

DEPRECATED: Please use *v_is_parameter* instead!

v_run_branch
If this node is hanging below a branch named *run_XXXXXXXXX*.

The branch name is either the name of a single run (e.g. 'run_00000009') or 'trajectory'.

v_stored
Whether or not this tree node has been stored to disk before.

3.3.4 ArrayParameter

class pypet.parameter.**ArrayParameter** (*full_name*, *data=None*, *comment=''*)
Similar to the *Parameter*, but recommended for large numpy arrays and python tuples.

The array parameter is a bit smarter in memory management than the parameter. If a numpy array is used several times within an exploration, only one numpy array is stored by the default HDF5 storage service. For each individual run references to the corresponding numpy array are stored.

Since the ArrayParameter inherits from *Parameter* it also supports all other native python types.

IDENTIFIER = '__rr__'
Identifier to mark stored data as an array

f_supports (*data*)
Checks if input data is supported by the parameter.

3.3.5 SparseParameter

class pypet.parameter.**SparseParameter** (*full_name*, *data=None*, *comment=''*)
Parameter that handles Scipy csr, csc, bsr and dia sparse matrices.

Sparse Parameter inherits from *ArrayParameter* and supports arrays and native python data as well.

Uses similar memory management as its parent class.

DIA_NAME_LIST = ['format', 'data', 'offsets', 'shape']
Data names for serialization of dia matrices

IDENTIFIER = '__spsp__'
Identifier to mark stored data as a sparse matrix

OTHER_NAME_LIST = ['format', 'data', 'indices', 'indptr', 'shape']
Data names for serialization of csr, csc, and bsr matrices

f_supports(*data*)
Sparse matrices support Scipy csr, csc, bsr and dia matrices and everything their parent class the *ArrayParameter* supports.

3.3.6 PickleParameter

class pypet.parameter.**PickleParameter**(*full_name*, *data=None*, *comment=''*, *protocol=2*)
A parameter class that supports all picklable objects, and pickles everything!

If you use the default HDF5 storage service, the pickle dumps are stored to disk. Works similar to the array parameter regarding memory management (Equality of objects is based on object id).

There is no straightforward check to guarantee that data is picklable, so you have to take care that all data handled by the PickleParameter supports pickling.

You can pass the pickle protocol via *protocol=2* to the constructor or change it with the *v_protocol* property. Default protocol is 0. Note that after storage to disk changing the protocol has no effect. If the parameter is loaded, *v_protocol* is set to the protocol used to store the data.

f_supports(*data*)
There is no straightforward check if an object can be pickled and this function will always return *True*.

So you have to take care in advance that the item can be pickled.

v_protocol
The protocol used to pickle data, default is 0.

See pickle documentation for the protocols.

3.3.7 Result

class pypet.parameter.**Result**(*full_name*, **args*, ***kwargs*)
Light Container that stores basic python and numpy data.

Note that no sanity checks on individual data is made (only on outer data structure) and you have to take care, that your data is understood by the storage service. It is assumed that results tend to be large and therefore sanity checks would be too expensive.

Data that can safely be stored into a Result are:
- python natives (int, long, str, bool, float, complex),
- numpy natives, arrays and matrices of type np.int8-64, np.uint8-64, np.float32-64, np.complex, np.str
- python lists and tuples of the previous types (python natives + numpy natives and arrays) Lists and tuples are not allowed to be nested and must be homogeneous, i.e. only contain data of one particular type. Only integers, or only floats, etc.
- python dictionaries of the previous types (not nested!), data can be heterogeneous, keys must be strings. For example, one key-value pair of string and int and one key-value pair of string and float, and so on.
- pandas DataFrames
- *ObjectTable*

Note that containers should NOT be empty (like empty dicts or lists) at the time they are saved to disk. The standard HDF5 storage service cannot store empty containers! The Result emits a warning if you hand over an empty container.

Data is set on initialisation or with *f_set()*

Example usage:

```
>>> res = Result('supergroup.subgroup.myresult', comment='I am a neat example!'
```

[1000

In case you create a new result you can pass the following arguments:
Parameters

- **fullanme** – The fullname of the result, grouping can be achieved by colons,
- **comment** – A useful comment describing the result. The comment can later on be changed using the *v_comment* variable

```
>>> param.v_comment
'I am a neat example!'
```

- **args** – Data that is handled by the result. The first positional argument is stored with the name of the result. Following arguments are stored with *name_X* where *X* is the position of the argument.
- **kwargs** – Data that is handled by the result, it is kept by the result under the names specified by the keys of kwargs.

```
>>> res.f_get(0)
[1000,2000]
>>> res.f_get(1)
{'a':'b','c':'d'}
>>> res.f_get('myresult')
[1000,2000]
>>> res.f_get('hitchhiker')
'ArthurDent'
>>> res.f_get('myresult','hitchhiker')
([1000,2000], 'ArthurDent')
```

Can be changed or more can be added via *f_set()*

```
>>> result.f_set('Uno',x='y')
>>> result.f_get(0)
'Uno'
>>> result.f_get('x')
'y'
```

Alternative method to put and retrieve data from the result container is via `__getattr__` and `__setattr__`:

```
>>> res.ford = 'prefect'
>>> res.ford
'prefect'
```

RaisesTypeError:

If the data format in args or kwargs is not known to the result. Checks type of outer data structure, i.e. checks if you have a list or dictionary. But it does not check on individual values within dicts or lists.

f_ann_to_str()

Returns annotations as string

Equivalent to *v_annotations.f_ann_to_str()*

f_ann_to_string()

Returns annotations as string

Equivalent to *v_annotations.f_ann_to_str()*

DEPRECATED: Please use *f_ann_to_str()* instead.

f_empty()

Removes all data from the result or parameter.

If the result has already been stored to disk via a trajectory and a storage service, the data on disk is not affected by *f_empty*.

Yet, this function is particularly useful if you have stored very large data to disk and you want to free some memory on RAM but still keep the skeleton of your result or parameter.

Note that freeing RAM requires that all references to the data are deleted. If you reference the data somewhere else in your code, the data is not erased from RAM.

f_get(**args*)

Returns items handled by the result.

If only a single name is given, a single data item is returned. If several names are given, a list is returned. For integer inputs the result returns *resultname_X*.

If the result contains only a single entry you can call *f_get()* without arguments. If you call *f_get()* and the result contains more than one element a ValueError is thrown.

If the requested item(s) cannot be found an AttributeError is thrown.

Parameters **args** – strings-names or integers
Returns Single data item or tuple of data

Example:

```
>>> res = Result('supergroup.subgroup.myresult', comment='I am a neat example!'
>>> res.f_get('hitchhiker')
'Arthur Dent'
>>> res.f_get(0)
[1000,2000]
>>> res.f_get('hitchhiker', 'myresult')
('Arthur Dent', [1000,2000])
```

f_get_annotations(**args*)

Returns annotations

Equivalent to *v_annotations.f_get(*args)*

f_get_class_name()

Returns the class name of the parameter or result or group.

Equivalent to *obj.__class__.__name__*

f_is_empty()

True if no data has been put into the result.

Also True if all data has been erased via *f_empty()*.

f_is_root()

Whether the group is root (True for the trajectory and a single run object)

DEPRECATED: Please use property v_is_root!

f_remove(**args*)

Removes **args* from the result

f_set(**args*, ***kwargs*)

Method to put data into the result.

Parameters

- **args** – The first positional argument is stored with the name of the result. Following arguments are stored with *name_X* where *X* is the position of the argument.
- **kwargs** – Arguments are stored with the key as name.

Raises TypeError if outer data structure is not understood.

Example usage:

```
>>> res = Result('supergroup.subgroup.myresult', comment='I am a neat example!')
>>> res.f_set(333,42.0, mystring='String!')
>>> res.f_get('myresult')
333
>>> res.f_get('myresult_1')
42.0
>>> res.f_get(1)
```

```
        42.0
        >>> res.f_get('mystring')
        'String!'
```

f_set_annotations(**args*, ***kwargs*)
Sets annotations

Equivalent to calling *v_annotations.f_set(*args,**kwargs)*

f_set_single(*name*, *item*)
Sets a single data item of the result.

Raises TypeError if the type of the outer data structure is not understood. Note that the type check is shallow. For example, if the data item is a list, the individual list elements are NOT checked whether their types are appropriate.

Parameters
- **name** – The name of the data item
- **item** – The data item

RaisesTypeError

Example usage:

```
        >>> res.f_set_single('answer', 42)
        >>> res.f_get('answer')
        42
```

f_supports_fast_access()
Whether or not the result supports fast access.

A result supports fast access if it contains exactly one item with the name of the result.

f_to_dict(*copy=True*)
Returns all handled data as a dictionary.

Parameterscopy – Whether the original dictionary or a shallow copy is returned.

ReturnsData dictionary

f_translate_key(*key*)
Translates integer indeces into the appropriate names

f_val_to_str()
Summarizes data handled by the result as a string.

Calls *__repr__* on all handled data. Data is NOT ordered.

Truncates the string if it is longer than `pypetconstants.HDF5_STRCOL_MAX_VALUE_LENGTH`

Returnsstring

v_annotations
Annotation feature of a trajectory node.

Store some short additional information about your nodes here. If you use the standard HDF5 storage service, they will be stored as hdf5 node attributes.

v_branch
The name of the branch/subtree, i.e. the first node below the root.

The empty string in case of root itself.

v_comment
Should be a nice descriptive comment

v_depth
Depth of the node in the trajectory tree.

v_fast_accessible
Whether or not fast access can be supported by the Parameter or Result

DEPRECATED: Please use function *f_supports_fast_access* instead!

v_full_name
The full name, relative to the root node.

The full name of a trajectory or single run is the empty string since it is root.

v_is_group
Whether node is a group or not (i.e. it is a leaf node)

v_is_leaf
Whether node is a leaf or not (i.e. it is a group node)

v_is_parameter
Whether the node is a parameter or not (i.e. a result)

v_is_root
Whether the group is root (True for the trajectory and a single run object)

v_leaf
Whether node is a leaf or not (i.e. it is a group node)

DEPRECATED: Please use v_is_leaf!

v_location
Location relative to the root node.

The location of a trajectory or single run is the empty string since it is root.

v_name
Name of the node

v_no_data_string
Whether or not to give a short summarizing string when calling `f_val_to_str()` or *__str__*. Can be set to *False* if the evaluation of stored data into string is too costly.

DEPRECATED! Does not change anything. Data will always be printed.

v_parameter
Whether the node is a parameter or not (i.e. a result)

DEPRECATED: Please use *v_is_parameter* instead!

v_run_branch
If this node is hanging below a branch named *run_XXXXXXXXX*.

The branch name is either the name of a single run (e.g. 'run_00000009') or 'trajectory'.

v_stored
Whether or not this tree node has been stored to disk before.

3.3.8 SparseResult

class `pypet.parameter.`**SparseResult**(*full_name*, **args*, ***kwargs*)
Handles Scipy sparse matrices.

Supported Formats are csr, csc, bsr, and dia.

Subclasses the standard result and can also handle all data supported by `Result`.

IDENTIFIER = '__spsp__'
Identifier string to label sparse matrix data

f_ann_to_str()
Returns annotations as string

Equivalent to *v_annotations.f_ann_to_str()*

f_ann_to_string()
Returns annotations as string

Equivalent to *v_annotations.f_ann_to_str()*

DEPRECATED: Please use *f_ann_to_str()* instead.

f_empty()
Removes all data from the result or parameter.

If the result has already been stored to disk via a trajectory and a storage service, the data on disk is not affected by *f_empty*.

Yet, this function is particularly useful if you have stored very large data to disk and you want to free some memory on RAM but still keep the skeleton of your result or parameter.

Note that freeing RAM requires that all references to the data are deleted. If you reference the data somewhere else in your code, the data is not erased from RAM.

f_get(**args*)
Returns items handled by the result.

If only a single name is given, a single data item is returned. If several names are given, a list is returned. For integer inputs the result returns *resultname_X*.

If the result contains only a single entry you can call *f_get()* without arguments. If you call *f_get()* and the result contains more than one element a ValueError is thrown.

If the requested item(s) cannot be found an AttributeError is thrown.

Parameters **args** – strings-names or integers
Returns Single data item or tuple of data

Example:

```
>>> res = Result('supergroup.subgroup.myresult', comment='I am a neat example!'
>>> res.f_get('hitchhiker')
'Arthur Dent'
>>> res.f_get(0)
[1000,2000]
>>> res.f_get('hitchhiker', 'myresult')
('Arthur Dent', [1000,2000])
```

f_get_annotations(**args*)
Returns annotations

Equivalent to *v_annotations.f_get(*args)*

f_get_class_name()
Returns the class name of the parameter or result or group.

Equivalent to *obj.__class__.__name__*

f_is_empty()
True if no data has been put into the result.

Also True if all data has been erased via *f_empty()*.

f_is_root()
Whether the group is root (True for the trajectory and a single run object)

DEPRECATED: Please use property v_is_root!

f_remove(**args*)
Removes **args* from the result

f_set(**args*, ***kwargs*)
Method to put data into the result.

Parameters

- **args** – The first positional argument is stored with the name of the result. Following arguments are stored with *name_X* where *X* is the position of the argument.
- **kwargs** – Arguments are stored with the key as name.

Raises TypeError if outer data structure is not understood.

Example usage:

```
>>> res = Result('supergroup.subgroup.myresult', comment='I am a neat example!')
>>> res.f_set(333,42.0, mystring='String!')
>>> res.f_get('myresult')
333
>>> res.f_get('myresult_1')
42.0
>>> res.f_get(1)
42.0
>>> res.f_get('mystring')
'String!'
```

f_set_annotations(**args*, ***kwargs*)

Sets annotations

Equivalent to calling *v_annotations.f_set(*args,**kwargs)*

f_set_single(*name*, *item*)

Sets a single data item of the result.

Raises TypeError if the type of the outer data structure is not understood. Note that the type check is shallow. For example, if the data item is a list, the individual list elements are NOT checked whether their types are appropriate.

Parameters

- **name** – The name of the data item
- **item** – The data item

Raises TypeError

Example usage:

```
>>> res.f_set_single('answer', 42)
>>> res.f_get('answer')
42
```

f_supports_fast_access()

Whether or not the result supports fast access.

A result supports fast access if it contains exactly one item with the name of the result.

f_to_dict(*copy=True*)

Returns all handled data as a dictionary.

Parameters **copy** – Whether the original dictionary or a shallow copy is returned.

Returns Data dictionary

f_translate_key(*key*)

Translates integer indeces into the appropriate names

f_val_to_str()

Summarizes data handled by the result as a string.

Calls *__repr__* on all handled data. Data is NOT ordered.

Truncates the string if it is longer than `pypetconstants.HDF5_STRCOL_MAX_VALUE_LENGTH`

Returns string

v_annotations

Annotation feature of a trajectory node.

Store some short additional information about your nodes here. If you use the standard HDF5 storage service, they will be stored as hdf5 node attributes.

v_branch
The name of the branch/subtree, i.e. the first node below the root.

The empty string in case of root itself.

v_comment
Should be a nice descriptive comment

v_depth
Depth of the node in the trajectory tree.

v_fast_accessible
Whether or not fast access can be supported by the Parameter or Result

DEPRECATED: Please use function *f_supports_fast_access* instead!

v_full_name
The full name, relative to the root node.

The full name of a trajectory or single run is the empty string since it is root.

v_is_group
Whether node is a group or not (i.e. it is a leaf node)

v_is_leaf
Whether node is a leaf or not (i.e. it is a group node)

v_is_parameter
Whether the node is a parameter or not (i.e. a result)

v_is_root
Whether the group is root (True for the trajectory and a single run object)

v_leaf
Whether node is a leaf or not (i.e. it is a group node)

DEPRECATED: Please use v_is_leaf!

v_location
Location relative to the root node.

The location of a trajectory or single run is the empty string since it is root.

v_name
Name of the node

v_no_data_string
Whether or not to give a short summarizing string when calling `f_val_to_str()` or *__str__*.
Can be set to *False* if the evaluation of stored data into string is too costly.

DEPRECATED! Does not change anything. Data will always be printed.

v_parameter
Whether the node is a parameter or not (i.e. a result)

DEPRECATED: Please use *v_is_parameter* instead!

v_run_branch
If this node is hanging below a branch named *run_XXXXXXXXX*.

The branch name is either the name of a single run (e.g. 'run_00000009') or 'trajectory'.

v_stored
Whether or not this tree node has been stored to disk before.

3.3.9 PickleResult

class `pypet.parameter.`**PickleResult** (*full_name*, **args*, ***kwargs*)
Result that digest everything and simply pickles it!

Note that it is not checked whether data can be pickled, so take care that it works!

You can pass the pickle protocol via *protocol=2* to the constructor or change it with the *v_protocol* property. Default protocol is 0.

Note that after storage to disk changing the protocol has no effect. If the parameter is loaded, *v_protocol* is set to a protocol used to store an item. Note that items are reconstructed from a dictionary and the protocol is taken from the first one found in the dictionary. This is a rather arbitrary choice. Yet, the underlying assumption is that all items were pickled with the same protocol, which is the general case.

f_set_single(*name*, *item*)
Adds a single data item to the pickle result.

Note that it is NOT checked if the item can be pickled!

v_protocol
The protocol used to pickle data, default is 0.

See pickle documentation for the protocols.

3.3.10 Object Table

class pypet.parameter.**ObjectTable**(*data=None*, *index=None*, *columns=None*, *copy=False*)
Wrapper class for pandas DataFrames.

It creates data frames with *dtype=object*.

Data stored into an object table preserves its original type when stored to disk. For instance, a python int is not automatically converted to a numpy 64 bit integer (np.int64).

The object table serves as a data structure to hand data to a storage service.

Example Usage:

```
>>> ObjectTable(data={'characters':['Luke', 'Han', 'Spock'], 'Random_Values' :[42,43,44] })
```

Creates the following table:

Index	Random_Values	characters
0	42	Luke
1	43	Han
2	44	Spock

3.3.11 The Abstract Base Classes of Parameters and Results

These classes serve as a reference if you want to implement your own parameter or result. Therefore, also private functions are listed.

class pypet.parameter.**BaseParameter**(*full_name*, *comment=''*)
Abstract class that specifies the methods for a trajectory parameter.

Parameters are simple container objects for data values. They handle single values as well as ranges of potential values. These range arrays contain multiple values which are accessed one after the other in individual simulation runs.

Parameter exploration is usually initiated through the trajectory see *f_explore()* and *f_expand()*.

To access the parameter's data value one can call the *f_get()* method.

Parameters support the concept of locking. Once a value of the parameter has been accessed, the parameter cannot be changed anymore unless it is explicitly unlocked using *f_unlock()*. This prevents parameters from being changed during runtime of a simulation.

If multiprocessing is desired the parameter must be picklable!
Parameters

- **full_name** – The full name of the parameter in the trajectory tree, groupings are separated by a colon: *fullname = 'supergroup.subgroup.paramname'*
- **comment** – A useful comment describing the parameter: *comment = 'Some useful text, dude!'*

__all_slots__ = set(['_depth', '_full_copy', '_full_name', '_stored', '_is_parameter', '_logger', '_comment', '_is_

__class__
alias of MetaSlotMachine

__delattr__
x.__delattr__('name') <==> del x.name

__dir__()
Includes all slots in the *dir* method

__format__()
default object formatter

__getattribute__
x.__getattribute__('name') <==> x.name

__getitem__(*idx*)
Equivalent to *f_get_range[idx]*

Raises TypeError if parameter has no range

__getstate__()
Called for pickling.

Removes the logger to allow pickling and returns a copy of *__dict__*.

__hash__

__init__(*full_name*, *comment=''*)

__module__ = 'pypet.parameter'

__new__(*S*, ...) → a new object with type S, a subtype of T

__reduce__()
helper for pickle

__reduce_ex__()
helper for pickle

__repr__()

__setattr__
x.__setattr__('name', value) <==> x.name = value

__setstate__(*statedict*)
Called after loading a pickle dump.

Restores *__dict__* from *statedict* and adds a new logger.

__sizeof__() → int
size of object in memory, in bytes

__slots__ = ('_locked', '_full_copy', '_explored')

__str__()
String representation of the Parameter

Output format is:<*class_name*> *full_name (len:X, 'comment*): value'. If comment is the empty string, the comment is omitted. If the parameter is not explored the length is omitted.

__subclasshook__()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__weakref__
list of weak references to the object (if defined)

_annotations

_branch

_comment

_depth

_equal_values(*val1*, *val2*)
Checks if the parameter considers two values as equal.

This is important for the trajectory in case of merging. In case you want to delete duplicate parameter points, the trajectory needs to know when two parameters are equal. Since equality is not always implemented by values handled by parameters in the same way, the parameters need to judge whether their values are equal.

The straightforward example here is a numpy array. Checking for equality of two numpy arrays yields a third numpy array containing truth values of a piecewise comparison. Accordingly, the parameter could judge two numpy arrays equal if ALL of the numpy array elements are equal.

In this BaseParameter class values are considered to be equal if they obey the function *nested_equal()*. You might consider implementing a different equality comparison in your subclass.

RaisesTypeError: If both values are not supported by the parameter.

_expand(*iterable*)
Similar to *_explore()* but appends to the exploration range.

Parameters
iterable – An iterable specifying the exploration range.

RaisesParameterLockedException: If the parameter is locked

TypeError: If the parameter did not have a range before

Example usage:

```
>>> param = Parameter('groupA.groupB.myparam', data=3.13, comment='I am a neat example')
>>> param._explore([3.0,2.0,1.0])
>>> param._expand([42.0,43.0])
>>> param.f_get_range()
(3.0,2.0,1.0,42.0,43.0)
```

ABSTRACT: Needs to be defined in subclass

_explore(*iterable*)
The method to explore a parameter and create a range of entries.

Parameters
iterable – An iterable specifying the exploration range

For example:

```
>>> param = Parameter('groupA.groupB.myparam',data=22.33,                comment=
>>> param._explore([3.0,2.0,1.0])
```

RaisesParameterLockedException: If the parameter is locked

TypeError: If the parameter is already explored

ABSTRACT: Needs to be defined in subclass

_explored

_full_copy

_full_name

_is_leaf

_is_parameter

_load(*load_dict*)
Method called by the storage service to reconstruct the original result.

Data contained in the load_dict is equal to the data provided by the result or parameter when previously called with _store().

Parameters **load_dict** – The dictionary containing basic data structures, see also `_store()`.

ABSTRACT: Needs to be implemented by subclass

_load_flags()
Currently not used because I let the storage service infer how to load stuff from the data itself.

If you write your own parameter or result you can implement this function to make specifications on how to load data, see also *pypet.storageservice.HDF5StorageService.store()*.

Returns {} (Empty dictionary)

_locked

_logger

_name

_rename(*full_name*)
Renames the tree node

_restore_default()
Restores original data if changed due to exploration.

If a Parameter is explored, the actual data is changed over the course of different simulations. This method restores the original data assigned before exploration.

ABSTRACT: Needs to be defined in subclass

_run_branch

_set_details(*depth*, *branch*, *run_branch*)
Sets some details for internal handling.

_set_logger(*name=None*)
Adds a logger with a given *name*.

If no name is given, name is constructed as *type(self).__name__*.

_set_parameter_access(*idx=0*)
Sets the current value according to the *idx* in the exploration range.

Prepares the parameter for further usage, and tells it which point in the parameter space should be accessed by calls to *f_get()*.

Parameters **idx** – The index within the exploration range.

If the parameter has no range, the single data value is considered regardless of the value of *idx*. Raises ValueError if the parameter is explored and *idx>=len(param)*.

Raises ValueError:

If the parameter has a range and *idx* is larger or equal to the length of the parameter.

Example usage:

```
>>> param = Parameter('groupA.groupB.myparam',data=22.33, comment='I am a neat example')
>>> param._explore([42.0,43.0,44.0])
>>> param._set_parameter_access(idx=1)
>>> param.f_get()
43.0
```

ABSTRACT: Needs to be defined in subclass

_shrink()
If a parameter is explored, i.e. it has a range, the whole exploration range is deleted.

Note that this function does not erase data from disk. So if the parameter has been stored with a service to disk and is shrunk, it can be restored by loading from disk.

RaisesParameterLockedException: If the parameter is locked

TypeError: If the parameter has no range

ABSTRACT: Needs to be defined in subclass

_store()
Method called by the storage service for serialization.

The method converts the parameter's or result's value(s) into simple data structures that can be stored to disk. Returns a dictionary containing these simple structures.

Understood basic structures are

- •python natives (int, long, str,bool,float,complex)
- •python lists and tuples
- •numpy natives arrays, and matrices of type np.int8-64, np.uint8-64, np.float32-64, np.complex, np.str
- •python dictionaries of the previous types (flat not nested!)
- •pandas data frames
- •object tables (see *ObjectTable*)

ReturnsA dictionary containing basic data structures.

ABSTRACT: Needs to be implemented by subclass

_store_flags()
Currently not used because I let the storage service infer how to store stuff from the data itself.

If you write your own parameter or result you can implement this function to make specifications on how to store data, see also *pypet.storageservice.HDF5StorageService.store()*.

Returns{} (Empty dictionary)

_stored

_values_of_same_type(*val1*, *val2*)
Checks if two values agree in type.

For example, two 32 bit integers would be of same type, but not a string and an integer, nor a 64 bit and a 32 bit integer.

This is important for exploration. You are only allowed to explore data that is of the same type as the default value.

One could always come up with a trivial solution of *type(val1) is type(val2)*. But sometimes your parameter does want even more strict equality or less type equality.

For example, the *Parameter* has a stricter sense of type equality regarding numpy arrays. In order to have two numpy arrays of the same type, they must also agree in shape. However, the *ArrayParameter*, considers all numpy arrays as of being of same type regardless of their shape.

Moreover, the *SparseParameter* considers all supported sparse matrices (csc, csr, bsr, dia) as being of the same type. You can make explorations using all these four types at once.

The difference in how strict types are treated arises from the way parameter data is stored to disk and how the parameters hand over their data to the storage service (see *pypet.parameter.BaseParameter._store()*).

The *Parameter* puts all it's data in an *ObjectTable* which has strict constraints on the column sizes. This means that numpy array columns only accept numpy arrays with a particular size. In contrast, the array and sparse parameter hand over their data as individual items which yield individual entries in the hdf5 node. In order to see what I mean simply run an experiment with all 3 parameters, explore all of them, and take a look at the resulting hdf5 file!

However, this BaseParameter class implements the straightforward version of *type(val1) is type(val2)* to consider data to be of the same type.

RaisesTypeError: if both values are not supported by the parameter.

f_ann_to_str()

Returns annotations as string

Equivalent to *v_annotations.f_ann_to_str()*

f_ann_to_string()

Returns annotations as string

Equivalent to *v_annotations.f_ann_to_str()*

DEPRECATED: Please use *f_ann_to_str()* instead.

f_empty()

Erases all data in the parameter.

Does not erase data from disk. So if the parameter has been stored with a service to disk and is emptied, it can be restored by loading from disk.

RaisesParameterLockedException: If the parameter is locked.

ABSTRACT: Needs to be defined in subclass

f_get()

Returns the current data value of the parameter and locks the parameter.

RaisesTypeError if the parameter is empty

Example usage:

```
>>> param = Parameter('groupA.groupB.myparam', comment='I am a neat example')
>>> param.f_set(44.0)
>>> param.f_get()
44.0:
```

ABSTRACT: Needs to be defined in subclass

f_get_annotations(**args*)

Returns annotations

Equivalent to *v_annotations.f_get(*args)*

f_get_array()

Returns an iterable to iterate over the values of the exploration range.

Note that the returned values should be either a copy of the exploration range or the array must be immutable, for example a python tuple.

ReturnsImmutable sequence

RaisesTypeError if the parameter is not explored

Example usage:

```
>>> param = Parameter('groupA.groupB.myparam',data=22, comment='I am a neat example')
>>> param._explore([42,43,43])
>>> param.f_get_array()
(42,43,44)
```

DEPRECATED: Use *f_get_range()* instead!

f_get_class_name()

Returns the name of the class i.e. *return self.__class__.__name__*

f_get_default()

Returns the default value of the parameter and locks it.

f_get_range()

Returns an iterable to iterate over the values of the exploration range.

Note that the returned values should be either a copy of the exploration range or the array must be immutable, for example a python tuple.

Returns Immutable sequence

Raises TypeError if the parameter is not explored

Example usage:

```
>>> param = Parameter('groupA.groupB.myparam',data=22, comment='I am a neat example')
>>> param._explore([42,43,43])
>>> param.f_get_range()
(42,43,44)
```

ABSTRACT: Needs to be defined in subclass

f_get_range_length()

Returns the length of the parameter range.

Raises TypeError if the parameter has no range.

Does not need to be implemented if the parameter supports `__len__` appropriately.

f_has_range()

Returns true if the parameter contains a range array.

Not necessarily equal to *v_explored* if the range is removed on pickling due to *v_full_copy=False*.

ABSTRACT: Needs to be defined in subclass

f_is_array()

Returns true if the parameter is explored and contains a range array.

DEPRECATED: Use *f_has_range()* instead.

f_is_empty()

True if no data has been assigned to the parameter.

Example usage:

```
>>> param = Parameter('myname.is.example', comment='I am _empty!')
>>> param.f_is_empty()
True
>>> param.f_set(444)
>>> param.f_is_empty()
False
```

f_is_root()

Whether the group is root (True for the trajectory and a single run object)

DEPRECATED: Please use property v_is_root!

f_lock()

Locks the parameter and forbids further manipulation.

Changing the data value or exploration range of the parameter are no longer allowed.

f_set(*data*)

Sets a data value for a parameter.

Example usage:

```
>>> param = Parameter('groupA.groupB.myparam', comment='I am a neat example')
>>> param.f_set(44.0)
>>> param.f_get()
44.0
```

Raises ParameterLockedException: If parameter is locked

TypeError: If the type of the data value is not supported by the parameter

ABSTRACT: Needs to be defined in subclass

f_set_annotations(**args*, ***kwargs*)
Sets annotations

Equivalent to calling *v_annotations.f_set(*args,**kwargs)*

f_supports(*data*)
Checks whether the data is supported by the parameter.

f_supports_fast_access()
Checks if parameter supports fast access.

A parameter supports fast access if it is NOT empty!

f_unlock()
Unlocks the locked parameter.

Please use it very carefully, or best do not use this function at all. There should better be no reason to unlock a locked parameter! The only exception I can think of is to unlock a large derived parameter after usage to subsequently call `f_empty()` to clear memory.

f_val_to_str()
String summary of the value handled by the parameter.

Note that representing the parameter as a string accesses its value, but for simpler debugging, this does not lock the parameter or counts as usage!

Calls *__repr__* of the contained value.

v_annotations
Annotation feature of a trajectory node.

Store some short additional information about your nodes here. If you use the standard HDF5 storage service, they will be stored as hdf5 node attributes.

v_branch
The name of the branch/subtree, i.e. the first node below the root.

The empty string in case of root itself.

v_comment
Should be a nice descriptive comment

v_depth
Depth of the node in the trajectory tree.

v_explored
Whether parameter is explored.

Does not necessarily have to be similar to `f_has_range()` since the range can be deleted on pickling and the parameter remains explored.

v_fast_accessible
Whether or not fast access can be supported by the Parameter or Result

DEPRECATED: Please use function *f_supports_fast_access* instead!

v_full_copy
Whether or not the full parameter including the range or only the current data is copied during pickling.

If you run your simulations in multiprocessing mode, the whole trajectory and all parameters need to be pickled and are sent to the individual processes. Each process than runs an individual point in the parameter space. As a consequence, you do not need the full ranges during these calculations. Thus, if the full copy mode is set to *False* the parameter is pickled without the range array and you can save memory.

If you want to access the full range during individual runs, you need to set *v_full_copy* to *True*.

It is recommended NOT to do that in order to save memory and also do obey the philosophy that individual simulation runs are independent.

Example usage:

```
>>> import pickle
>>> param = Parameter('examples.fullcopy', data=333, comment='I show you how the copy mo
>>> param._explore([1,2,3,4])
>>> dump=pickle.dumps(param)
>>> newparam = pickle.loads(dump)
>>> newparam.f_get_range()
TypeError
```

```
>>> param.v_full_copy=True
>>> dump = pickle.dumps(param)
>>> newparam=pickle.loads(dump)
>>> newparam.f_get_range()
(1,2,3,4)
```

v_full_name
The full name, relative to the root node.

The full name of a trajectory or single run is the empty string since it is root.

v_is_group
Whether node is a group or not (i.e. it is a leaf node)

v_is_leaf
Whether node is a leaf or not (i.e. it is a group node)

v_is_parameter
Whether the node is a parameter or not (i.e. a result)

v_is_root
Whether the group is root (True for the trajectory and a single run object)

v_leaf
Whether node is a leaf or not (i.e. it is a group node)

DEPRECATED: Please use v_is_leaf!

v_location
Location relative to the root node.

The location of a trajectory or single run is the empty string since it is root.

v_locked
Whether or not the parameter is locked and prevents further modification

v_name
Name of the node

v_parameter
Whether the node is a parameter or not (i.e. a result)

DEPRECATED: Please use *v_is_parameter* instead!

v_run_branch
If this node is hanging below a branch named *run_XXXXXXXXX*.

The branch name is either the name of a single run (e.g. 'run_00000009') or 'trajectory'.

v_stored
Whether or not this tree node has been stored to disk before.

class pypet.parameter.**BaseResult** (*full_name*, *comment=''*)
Abstract base API for results.

Compared to parameters (see *BaseParameter*) results are also initialised with a full name and a comment. Yet, results can contain more than a single value and heterogeneous data.

__all_slots__ = set(['_depth', '_full_name', '_stored', '_is_parameter', '_logger', '_comment', '_is_leaf', '_name

__class__
alias of MetaSlotMachine

__delattr__
x.__delattr__('name') <==> del x.name

__dir__()
Includes all slots in the *dir* method

__format__()
default object formatter

__getattribute__
x.__getattribute__('name') <==> x.name

__getstate__()
Called for pickling.

Removes the logger to allow pickling and returns a copy of *__dict__*.

__hash__

__init__(*full_name*, *comment=''*)

__module__ = 'pypet.parameter'

__new__(*S*, ...) → a new object with type S, a subtype of T

__reduce__()
helper for pickle

__reduce_ex__()
helper for pickle

__repr__

__setattr__
x.__setattr__('name', value) <==> x.name = value

__setstate__(*statedict*)
Called after loading a pickle dump.

Restores *__dict__* from *statedict* and adds a new logger.

__sizeof__() → int
size of object in memory, in bytes

__slots__ = ()

__str__()
String representation of the parameter or result.

If not specified in subclass this is simply the full name.

__subclasshook__()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__weakref__
list of weak references to the object (if defined)

_annotations

_branch

_comment

_depth

_full_name

_is_leaf

_is_parameter

_load(*load_dict*)
Method called by the storage service to reconstruct the original result.

Data contained in the load_dict is equal to the data provided by the result or parameter when previously called with _store().

Parameters **load_dict** – The dictionary containing basic data structures, see also `_store()`.

ABSTRACT: Needs to be implemented by subclass

_load_flags()
Currently not used because I let the storage service infer how to load stuff from the data itself.

If you write your own parameter or result you can implement this function to make specifications on how to load data, see also *pypet.storageservice.HDF5StorageService.store()*.

Returns { } (Empty dictionary)

_logger

_name

_rename(*full_name*)
Renames the tree node

_run_branch

_set_details(*depth*, *branch*, *run_branch*)
Sets some details for internal handling.

_set_logger(*name=None*)
Adds a logger with a given *name*.

If no name is given, name is constructed as *type(self).__name__*.

_store()
Method called by the storage service for serialization.

The method converts the parameter's or result's value(s) into simple data structures that can be stored to disk. Returns a dictionary containing these simple structures.

Understood basic structures are

- •python natives (int, long, str,bool,float,complex)
- •python lists and tuples
- •numpy natives arrays, and matrices of type np.int8-64, np.uint8-64, np.float32-64, np.complex, np.str
- •python dictionaries of the previous types (flat not nested!)
- •pandas data frames
- •object tables (see *ObjectTable*)

Returns A dictionary containing basic data structures.

ABSTRACT: Needs to be implemented by subclass

_store_flags()
Currently not used because I let the storage service infer how to store stuff from the data itself.

If you write your own parameter or result you can implement this function to make specifications on how to store data, see also *pypet.storageservice.HDF5StorageService.store()*.

Returns { } (Empty dictionary)

_stored

f_ann_to_str()
Returns annotations as string

Equivalent to *v_annotations.f_ann_to_str()*

f_ann_to_string()
Returns annotations as string

Equivalent to *v_annotations.f_ann_to_str()*

DEPRECATED: Please use *f_ann_to_str()* instead.

f_empty()
Removes all data from the result or parameter.

If the result has already been stored to disk via a trajectory and a storage service, the data on disk is not affected by *f_empty*.

Yet, this function is particularly useful if you have stored very large data to disk and you want to free some memory on RAM but still keep the skeleton of your result or parameter.

Note that freeing RAM requires that all references to the data are deleted. If you reference the data somewhere else in your code, the data is not erased from RAM.

ABSTRACT: Needs to be implemented by subclass

f_get_annotations(**args*)
Returns annotations

Equivalent to *v_annotations.f_get(*args)*

f_get_class_name()
Returns the class name of the parameter or result or group.

Equivalent to *obj.__class__.__name__*

f_is_empty()
Returns true if no data is handled by a result or parameter.

ABSTRACT: Needs to be implemented by subclass

f_is_root()
Whether the group is root (True for the trajectory and a single run object)

DEPRECATED: Please use property v_is_root!

f_set_annotations(**args*, ***kwargs*)
Sets annotations

Equivalent to calling *v_annotations.f_set(*args,**kwargs)*

f_supports_fast_access()
Whether or not fast access can be supported by the parameter or result.

ABSTRACT: Needs to be implemented by subclass.

f_val_to_str()
Returns a string summarizing the data handled by the parameter or result

ABSTRACT: Needs to be implemented by subclass, otherwise the empty string is returned.

v_annotations
Annotation feature of a trajectory node.

Store some short additional information about your nodes here. If you use the standard HDF5 storage service, they will be stored as hdf5 node attributes.

v_branch
The name of the branch/subtree, i.e. the first node below the root.

The empty string in case of root itself.

v_comment
Should be a nice descriptive comment

v_depth
Depth of the node in the trajectory tree.

v_fast_accessible
Whether or not fast access can be supported by the Parameter or Result

DEPRECATED: Please use function *f_supports_fast_access* instead!

v_full_name
The full name, relative to the root node.

The full name of a trajectory or single run is the empty string since it is root.

v_is_group
Whether node is a group or not (i.e. it is a leaf node)

v_is_leaf
Whether node is a leaf or not (i.e. it is a group node)

v_is_parameter
Whether the node is a parameter or not (i.e. a result)

v_is_root
Whether the group is root (True for the trajectory and a single run object)

v_leaf
Whether node is a leaf or not (i.e. it is a group node)

DEPRECATED: Please use v_is_leaf!

v_location
Location relative to the root node.

The location of a trajectory or single run is the empty string since it is root.

v_name
Name of the node

v_parameter
Whether the node is a parameter or not (i.e. a result)

DEPRECATED: Please use *v_is_parameter* instead!

v_run_branch
If this node is hanging below a branch named *run_XXXXXXXXX*.

The branch name is either the name of a single run (e.g. 'run_00000009') or 'trajectory'.

v_stored
Whether or not this tree node has been stored to disk before.

3.4 Annotations

class pypet.annotations.**Annotations**
Simple container class for annotations.

Every tree node (*leaves* and *group* nodes) can be annotated. In case you use the standard *HDF5StorageService*, these annotations are stored in the attributes of the hdf5 nodes in the hdf5 file, you might wanna take a look at pytables attributes.

Annotations should be small (short strings or basic python data types) since their storage and retrieval is quite slow!

f_ann_to_str()
Returns all annotations lexicographically sorted as a concatenated string.

f_empty()
Removes all annotations from RAM

f_get(**args*)
Returns annotations

If len(args)>1, then returns a list of annotations.

f_get(X) with *X* integer will return the annotation with name *annotation_X*.

If the annotation contains only a single entry you can call *f_get()* without arguments. If you call *f_get()* and the annotation contains more than one element a ValueError is thrown.

f_is_empty()
Checks if annotations are empty

f_remove(*key*)
Removes *key* from annotations

f_set(**args*, ***kwargs*)
Sets annotations

Items in args are added as *annotation* and *annotation_X* where 'X' is the position in args for following arguments.

f_set_single(*name*, *data*)
Sets a single annotation.

f_to_dict(*copy=True*)
Returns annotations as dictionary.

Parameters **copy** – Whether to return a shallow copy or the real thing (aka _dict).

3.5 Utils

3.5.1 Exploration Functions

Module containing factory functions for parameter exploration

pypet.utils.explore.**cartesian_product**(*parameter_dict*, *combined_parameters=()*)
Generates a Cartesian product of the input parameter dictionary.

For example:

```
>>> print cartesian_product({'param1':[1,2,3], 'param2':[42.0, 52.5]})
{'param1':[1,1,2,2,3,3],'param2': [42.0,52.5,42.0,52.5,42.0,52.5]}
```

Parameters

- **parameter_dict** – Dictionary containing parameter names as keys and iterables of data to explore.
- **combined_parameters** – Tuple of tuples. Defines the order of the parameters and parameters that are linked together. If an inner tuple contains only a single item, you can spare the inner tuple brackets.

  For example:

```
>>> print cartesian_product( {'param1': [42.0, 52.5], 'param2':['a', 'b'], 'par
{param3':[1,1,2,2,3,3],'param1' : [42.0,52.5,42.0,52.5,42.0,52.5], 'param2':['a
```

Returns Dictionary with cartesian product lists.

pypet.utils.explore.**find_unique_points**(*explored_parameters*)

Takes a list of explored parameters and finds unique parameter combinations.

If parameter ranges are hashable operates in O(N), otherwise O(N**2).

Parameters **explored_parameters** – List of **explored** parameters

Returns List of tuples, first entry being the parameter values, second entry a list containing the run position of the unique combination.

3.5.2 Utility Functions

HDF5 File Compression

You can use the following function to compress an existing HDF5 file that already contains a trajectory. This only works under **Linux**.

pypet.**compact_hdf5_file**(*filename*, *name=None*, *index=None*, *keep_backup=True*)

Can compress an HDF5 to reduce file size.

The properties on how to compress the new file are taken from a given trajectory in the file. Simply calls `ptrepack` from the command line. (Se also https://pytables.github.io/usersguide/utilities.html#ptrepackdescr)

Currently only supported under Linux, no guarantee for Windows usage.

Parameters

- **filename** – Name of the file to compact
- **name** – The name of the trajectory from which the compression properties are taken
- **index** – Instead of a name you could also specify an index, i.e -1 for the last trajectory in the file.
- **keep_backup** – If a back up version of the original file should be kept. The backup file is named as the original but *_backup* is appended to the end.

Returns The return/error code of ptrepack

Progressbar

Simple progressbar that can be used during a for-loop (no initialisation necessary). It displays progress and estimates remaining time.

pypet.**progressbar**(*index*, *total*, *percentage_step=10*, *logger='print'*, *log_level=20*, *reprint=True*, *time=True*, *length=20*, *fmt_string=None*, *reset=False*)

Plots a progress bar to the given *logger* for large for loops.

To be used inside a for-loop at the end of the loop:

```
for irun in range(42):
    my_costly_job() # Your expensive function
    progressbar(index=irun, total=42, reprint=True) # shows a growing progressbar
```

There is no initialisation of the progressbar necessary before the for-loop. The progressbar will be reset automatically if used in another for-loop.

Parameters

- **index** – Current index of for-loop
- **total** – Total size of for-loop
- **percentage_step** – Steps with which the bar should be plotted
- **logger** – Logger to write to - with level INFO. If string 'print' is given, the print statement is used. Use `None` if you don't want to print or log the progressbar statement.
- **log_level** – Log level with which to log.
- **reprint** – If no new line should be plotted but carriage return (works only for printing)

- **time** – If the remaining time should be estimated and displayed
- **length** – Length of the bar in = signs.
- **fmt_string** – A string which contains exactly one *%s* in order to incorporate the progressbar. If such a string is given, `fmt_string % progressbar` is printed/logged.
- **reset** – If the progressbar should be restarted. If progressbar is called with a lower index than the one before, the progressbar is automatically restarted.

Returns The progressbar string or *None* if the string has not been updated.

3.5.3 General Equality Function and Comparisons of Parameters and Results

Module containing utility functions to compare parameters and results

`pypet.utils.comparisons.`**get_all_attributes**(*instance*)
Returns an attribute value dictionary much like *__dict__* but incorporates *__slots__*

`pypet.utils.comparisons.`**nested_equal**(*a*, *b*)
Compares two objects recursively by their elements.

Also handles numpy arrays, pandas data and sparse matrices.

First checks if the data falls into the above categories. If not, it is checked if a or b are some type of sequence or mapping and the contained elements are compared. If this is not the case, it is checked if a or b do provide a custom *__eq__* that evaluates to a single boolean value. If this is not the case, the attributes of a and b are compared. If this does not help either, normal == is used.

Assumes hashable items are not mutable in a way that affects equality. Based on the suggestion from HERE, thanks again Lauritz V. Thaulow :-)

`pypet.utils.comparisons.`**parameters_equal**(*a*, *b*)
Compares two parameter instances

Checks full name, data, and ranges. Does not consider the comment.

Returns True or False
Raises ValueError if both inputs are no parameter instances

`pypet.utils.comparisons.`**results_equal**(*a*, *b*)
Compares two result instances

Checks full name and all data. Does not consider the comment.

Returns True or False
Raises ValueError if both inputs are no result instances

3.6 Exceptions

Module containing all exceptions

exception `pypet.pypetexceptions.`**DataNotInStorageError**(*msg*)
Excpetion raise by Storage Service if data that is supposed to be loaded cannot be found on disk.

exception `pypet.pypetexceptions.`**GitDiffError**(*msg*)
Exception raised if there are uncommited changes.

exception `pypet.pypetexceptions.`**NoSuchServiceError**(*msg*)
Exception raised by the Storage Service if a specific operation is not supported, i.e. the message is not understood.

exception `pypet.pypetexceptions.`**NotUniqueNodeError**(*msg*)
Exception raised by the Natural Naming if a node can be found more than once.

exception `pypet.pypetexceptions.`**ParameterLockedException**(*msg*)
Exception raised if someone tries to modify a locked Parameter.

exception pypet.pypetexceptions.**PresettingError**(*msg*)
Exception raised if parameter presetting failed.

Probable cause might be a typo in the parameter name.

exception pypet.pypetexceptions.**TooManyGroupsError**(*msg*)
Exception raised by natural naming fast search if fast search cannot be applied.

exception pypet.pypetexceptions.**VersionMismatchError**(*msg*)
Exception raised if the current version of pypet does not match the version with which the trajectory was handled.

3.7 Global Constants

Here you can find global constants. These constants define the data supported by the storage service and the standard parameter, maximum length of comments, messages for storing and loading etc. This module contains constants defined for a global scale and used across most pypet modules.

It contains constants defining the maximum length of a parameter/result name or constants that are recognized by storage services to determine how to store and load data.

pypet.pypetconstants.**PARAMETERTYPEDICT = {<class '__name__'>: <Mock object at 0x7f03a6bdc5d0>, <class '__**
A Mapping (dict) from the the string representation of a type and the type.

These are the so far supported types of the storage service and the standard parameter!

pypet.pypetconstants.**PARAMETER_SUPPORTED_DATA = (<Mock object at 0x7f03a6bdc650>, <Mock object at 0x7f**
Set of supported scalar types by the storage service and the standard parameter

pypet.pypetconstants.**HDF5_STRCOL_MAX_NAME_LENGTH = 128**
Maximum length of a (short) name

pypet.pypetconstants.**HDF5_STRCOL_MAX_LOCATION_LENGTH = 512**
Maximum length of the location string

pypet.pypetconstants.**HDF5_STRCOL_MAX_VALUE_LENGTH = 64**
Maximum length of a value string

pypet.pypetconstants.**HDF5_STRCOL_MAX_COMMENT_LENGTH = 512**
Maximum length of a comment

pypet.pypetconstants.**HDF5_STRCOL_MAX_RANGE_LENGTH = 1024**
Maximum length of a parameter array summary

pypet.pypetconstants.**HDF5_STRCOL_MAX_RUNTIME_LENGTH = 18**
Maximum length of human readable runtime, 18 characters allows to display up to 999 days excluding the microseconds

pypet.pypetconstants.**HDF5_MAX_OVERVIEW_TABLE_LENGTH = 1000**
Maximum number of entries in an overview table

pypet.pypetconstants.**WRAP_MODE_QUEUE = 'QUEUE'**
For multiprocessing, queue multiprocessing mode

pypet.pypetconstants.**WRAP_MODE_LOCK = 'LOCK'**
Lock multiprocessing mode

pypet.pypetconstants.**WRAP_MODE_NONE = 'NONE'**
No multiprocessing wrapping for the storage service

pypet.pypetconstants.**WRAP_MODE_PIPE = 'PIPE'**
Pipe multiprocessing mode

pypet.pypetconstants.**LOAD_SKELETON = 1**
For trajectory loading, loads only the skeleton.

pypet.pypetconstants.**LOAD_DATA = 2**
Loads skeleton and data.

pypet.pypetconstants.**LOAD_NOTHING = 0**
Loads nothing

pypet.pypetconstants.**UPDATE_SKELETON = 1**
DEPRECATED: Updates skeleton, i.e. adds only items that are not part of your current trajectory.

pypet.pypetconstants.**UPDATE_DATA = 2**
DEPRECATED: Updates skeleton and data, adds only items that are not part of your current trajectory.

pypet.pypetconstants.**STORE_NOTHING = 0**
Stores nothing to disk

pypet.pypetconstants.**STORE_DATA_SKIPPING = 1**
Stores only data of instances that have not been stored before

pypet.pypetconstants.**STORE_DATA = 2**
Stored all data to disk adds to existing data

pypet.pypetconstants.**OVERWRITE_DATA = 3**
Overwrites data on disk

pypet.pypetconstants.**LEAF = 'LEAF'**
For trajectory or item storage, stores a *leaf* node, i.e. parameter or result object

pypet.pypetconstants.**TRAJECTORY = 'TRAJECTORY'**
Stores the whole trajectory

pypet.pypetconstants.**MERGE = 'MERGE'**
Merges two trajectories

pypet.pypetconstants.**GROUP = 'GROUP'**
Stores a group node, can be recursive.

pypet.pypetconstants.**LIST = 'LIST'**
Stores a list of different things, in order to avoid reopening and closing of the hdf5 file.

pypet.pypetconstants.**SINGLE_RUN = 'SINGLE_RUN'**
Stores a single run

pypet.pypetconstants.**PREPARE_MERGE = 'PREPARE_MERGE'**
Updates a trajectory before it is going to be merged

pypet.pypetconstants.**BACKUP = 'BACKUP'**
Backs up a trajectory

pypet.pypetconstants.**DELETE = 'DELETE'**
Removes an item from hdf5 file

pypet.pypetconstants.**DELETE_LINK = 'DELETE_LINK'**
Removes a soft link from hdf5 file

pypet.pypetconstants.**TREE = 'TREE'**
Stores a subtree of the trajectory

pypet.pypetconstants.**ACCESS_DATA = 'ACCESS_DATA'**
Access and manipulate data directly in the hdf5 file

pypet.pypetconstants.**CLOSE_FILE = 'CLOSE_FILE'**
Close a still opened HDF5 file

pypet.pypetconstants.**OPEN_FILE = 'OPEN_FILE'**
Opens an HDF5 file and keeps it open until *CLOSE_FILE* is passed.

pypet.pypetconstants.**FLUSH = 'FLUSH'**
Tells the storage to flush the file

pypet.pypetconstants.**FORMAT_ZEROS = 8**
Number of leading zeros

pypet.pypetconstants.**RUN_NAME = 'run_'**
Name of a single run

pypet.pypetconstants.**RUN_NAME_DUMMY = 'run_ALL'**
Dummy name if not created during run

pypet.pypetconstants.**FORMATTED_RUN_NAME = 'run_%08d'**
Name formatted with leading zeros

pypet.pypetconstants.**SET_FORMAT_ZEROS = 5**
Number of leading zeros for set

pypet.pypetconstants.**SET_NAME = 'run_set_'**
Name of a run set

pypet.pypetconstants.**SET_NAME_DUMMY = 'run_set_ALL'**
Dummy name if not created during run

pypet.pypetconstants.**FORMATTED_SET_NAME = 'run_set_%05d'**
Name formatted with leading zeros

pypet.pypetconstants.**ARRAY = 'ARRAY'**
Stored as array

pypet.pypetconstants.**CARRAY = 'CARRAY'**
Stored as carray

pypet.pypetconstants.**EARRAY = 'EARRAY'**
Stored as earray_e.

pypet.pypetconstants.**VLARRAY = 'VLARRAY'**
Stored as vlarray

pypet.pypetconstants.**TABLE = 'TABLE'**
Stored as pytable

pypet.pypetconstants.**DICT = 'DICT'**
Stored as dict.

In fact, stored as pytable, but the dictionary wil be reconstructed.

pypet.pypetconstants.**FRAME = 'FRAME'**
Stored as pandas DataFrame

pypet.pypetconstants.**SERIES = 'SERIES'**
Store data as pandas Series

pypet.pypetconstants.**PANEL = 'PANEL'**
Store data as pandas Panel(4D)

pypet.pypetconstants.**SPLIT_TABLE = 'SPLIT_TABLE'**
If a table was split due to too many columns

pypet.pypetconstants.**DATATYPE_TABLE = 'DATATYPE_TABLE'**
If a table contains the data types instead of the attrs

pypet.pypetconstants.**SHARED_DATA = 'SHARED_DATA_'**
An HDF5 data object for direct interaction

pypet.pypetconstants.**LOG_ENV = '$env'**
Wildcard replaced by name of environment

pypet.pypetconstants.**LOG_TRAJ = '$traj'**
Wildcard replaced by name of trajectory

pypet.pypetconstants.**LOG_RUN = '$run'**
Wildcard replaced by name of current run

pypet.pypetconstants.**LOG_PROC = '$proc'**
Wildcard replaced by the name of the current process

pypet.pypetconstants.**LOG_SET = '$set'**
Wildcard replaced by the name of the current run set

pypet.pypetconstants.**DEFAULT_LOGGING = 'DEFAULT'**
Default logging configuration

3.8 Slots

For performance reasons all tree nodes support slots. They all sub-class the `HasSlots` class, which is the top-level class of *pypet* (its direct descendant is `HasLogger`, see below). This class provides an `__all_slots__` property (with the help of the `MetaSlotMachine` metaclass) that lists all existing `__slots__` of a class including the inherited ones. Moreover, via `__getstate__` and `__setstate__` `HasSlots` takes care that all sub-classes can be pickled with the lowest protocol and don't need to implement `__getstate__` and `__setstate__` themselves even when they have `__slots__`. However, sub-classes that still implement these functions should call the parent ones via `super`. Sub-classes are not required to define `__slots__`. If they don't, `HasSlots` wil also automatically handle their `__dict__` in `__getstate__` and `__setstate__`.

class pypet.slots.**HasSlots**
Top-class that allows mixing of classes with and without slots.

Takes care that instances can still be pickled with the lowest protocol. Moreover, provides a generic *__dir__* method that lists all slots.

__dir__()
Includes all slots in the *dir* method

__setstate__(*state*)
Recalls state for items with slots

__weakref__
list of weak references to the object (if defined)

pypet.slots.**add_metaclass**(*metaclass*)
Adds a metaclass to a given class.

This decorator is used instead of *__metaclass__* to allow for Python 2 and 3 compatibility.

Inspired by the *six* module: (https://bitbucket.org/gutworth/six/src/784c6a213c4527ea18f86a800f51bf16bc1df5bc/six.py?at=d

For example:

```
@add_metaclass(MyMetaClass)
class MyClass(object):
    pass
```

is equivalent to

```
class MyClass(object):
    __metaclass__ = MyMetaClass
```

in Python 2 or

```
class MyClass(object, metaclass=MyMetaClass)
    pass
```

in Python 3.

pypet.slots.**get_all_slots**(*cls*)
Iterates through a class' (*cls*) mro to get all slots as a set.

class pypet.slots.**MetaSlotMachine**(*name*, *bases*, *dictionary*)
Meta-class that adds the attribute *__all_slots__* to a class.

__all_slots__ is a set that contains all unique slots of a class, including the ones that are inherited from parents.

3.9 Logging

`HasLogger` can be sub-classed to allow per class or even per instance logging. The logger is initialized via `_set_logger()` and is available via the `_logger` attribute. `HasLogger` also takes care that the logger does not get pickled when `__getstate__` and `__setstate__` are called. Thus, you are always advised in sub-classes that also implement these functions to call the parent ones via `super`. `HasLogger` is a direct sub-class of `HasSlots`. Hence, support for `__slots__` is ensured.

class pypet.pypetlogging.**HasLogger**
Abstract super class that automatically adds a logger to a class.

To add a logger to a sub-class of yours simply call `myobj._set_logger(name)`. If `name=None` the logger is chosen as follows:

```
self._logger = logging.getLogger(self.__class.__.__module__ +
'.'  + self.__class__.__name__)
```

The logger can be accessed via `myobj._logger`.

__getstate__()
Called for pickling.

Removes the logger to allow pickling and returns a copy of *__dict__*.

__setstate__(*statedict*)
Called after loading a pickle dump.

Restores *__dict__* from *statedict* and adds a new logger.

_set_logger(*name=None*)
Adds a logger with a given *name*.

If no name is given, name is constructed as *type(self).__name__*.

pypet.pypetlogging.**rename_log_file**(*traj*, *filename*, *process_name=None*)
Renames a given *filename* with valid wildcard placements.

LOG_ENV ($env) is replaces by the name of the trajectory's environment.

LOG_TRAJ ($traj) is replaced by the name of the trajectory.

LOG_RUN ($run) is replaced by the name of the current run. If the trajectory is not set to a run 'run_ALL' is used.

LOG_SET ($set) is replaced by the name of the current run set. If the trajectory is not set to a run 'run_set_ALL' is used.

LOG_PROC ($proc) is replaced by the name fo the current process.

Parameters

- **traj** – A trajectory container
- **filename** – A filename string
- **process_name** – The name of the desired process. If *None* the name of the current process is taken determined by the multiprocessing module.

ReturnsThe new filename

3.10 Storage Services

3.10.1 The HDF5 Storage Service

class pypet.storageservice.**HDF5StorageService**(*filename=None*, *file_title=None*, *overwrite_file=False*, *encoding='utf8'*, *complevel=9*, *complib='zlib'*, *shuffle=True*, *fletcher32=False*, *pandas_format='fixed'*, *purge_duplicate_comments=True*, *summary_tables=True*, *small_overview_tables=True*, *large_overview_tables=False*, *results_per_run=0*, *derived_parameters_per_run=0*, *display_time=20*, *trajectory=None*)

Storage Service to handle the storage of a trajectory/parameters/results into hdf5 files.

Normally you do not interact with the storage service directly but via the trajectory, see *pypet.trajectory.Trajectory.f_store()* and *pypet.trajectory.Trajectory.f_load()*.

The service is not thread safe. For multiprocessing the service needs to be wrapped either by the *LockWrapper* or with a combination of *QueueStorageServiceSender* and *QueueStorageServiceWriter*.

The storage service supports two operations *store* and *load*.

Requests for these two are always passed as *msg, what_to_store_or_load, *args, **kwargs*

For example:

```
>>> HDF5StorageService.load(pypetconstants.LEAF, myresult, load_only=['spikestimes','nspikes'
```

For a list of supported items see *store()* and *load()*.

The service accepts the following parameters

Parameters

- **filename** – The name of the hdf5 file. If none is specified the default *./hdf5/the_name_of_your_trajectory.hdf5* is chosen. If *filename* contains only a path like *filename='./myfolder/'*, it is changed to *'filename='./myfolder/the_name_of_your_trajectory.hdf5'*.
- **file_title** – Title of the hdf5 file (only important if file is created new)
- **overwrite_file** – If the file already exists it will be overwritten. Otherwise the trajectory will simply be added to the file and already existing trajectories are not deleted.
- **encoding** – Format to encode and decode unicode strings stored to disk. The default `'utf8'` is highly recommended.
- **complevel** – If you use HDF5, you can specify your compression level. 0 means no compression and 9 is the highest compression level. See PyTables Compression for a detailed description.
- **complib** – The library used for compression. Choose between *zlib*, *blosc*, and *lzo*. Note that 'blosc' and 'lzo' are usually faster than 'zlib' but it may be the case that you can no longer open your hdf5 files with third-party applications that do not rely on PyTables.
- **shuffle** – Whether or not to use the shuffle filters in the HDF5 library. This normally improves the compression ratio.
- **fletcher32** – Whether or not to use the *Fletcher32* filter in the HDF5 library. This is used to add a checksum on hdf5 data.

- **pandas_format** – How to store pandas data frames. Either in 'fixed' ('f') or 'table' ('t') format. Fixed format allows fast reading and writing but disables querying the hdf5 data and appending to the store (with other 3rd party software other than *pypet*).
- **purge_duplicate_comments** – If you add a result via `f_add_result()` or a derived parameter `f_add_derived_parameter()` and you set a comment, normally that comment would be attached to each and every instance. This can produce a lot of unnecessary overhead if the comment is the same for every instance over all runs. If *purge_duplicate_comments=1* than only the comment of the first result or derived parameter instance created in a run is stored or comments that differ from this first comment.

  For instance, during a single run you call *traj.f_add_result('my_result',42, comment='Mostly harmless!')* and the result will be renamed to *results.run_00000000.my_result*. After storage in the node associated with this result in your hdf5 file, you will find the comment *'Mostly harmless!'* there. If you call *traj.f_add_result('my_result',-43, comment='Mostly harmless!')* in another run again, let's say run 00000001, the name will be mapped to *results.run_00000001.my_result*. But this time the comment will not be saved to disk since *'Mostly harmless!'* is already part of the very first result with the name 'results.run_00000000.my_result'. Note that the comments will be compared and storage will only be discarded if the strings are exactly the same.

  If you use multiprocessing, the storage service will take care that the comment for the result or derived parameter with the lowest run index will be considered regardless of the order of the finishing of your runs. Note that this only works properly if all comments are the same. Otherwise the comment in the overview table might not be the one with the lowest run index.

  You need summary tables (see below) to be able to purge duplicate comments.

  This feature only works for comments in *leaf* nodes (aka Results and Parameters). So try to avoid to add comments in *group* nodes within single runs.
- **summary_tables** – Whether the summary tables should be created, i.e. the 'derived_parameters_runs_summary', and the *results_runs_summary*.

  The 'XXXXXX_summary' tables give a summary about all results or derived parameters. It is assumed that results and derived parameters with equal names in individual runs are similar and only the first result or derived parameter that was created is shown as an example.

  The summary table can be used in combination with *purge_duplicate_comments* to only store a single comment for every result with the same name in each run, see above.
- **small_overview_tables** – Whether the small overview tables should be created. Small tables are giving overview about 'config','parameters', 'derived_parameters_trajectory', , 'results_trajectory','results_runs_summary'.

  Note that these tables create some overhead. If you want very small hdf5 files set *small_overview_tables* to False.
- **large_overview_tables** – Whether to add large overview tables. This encompasses information about every derived parameter, result, and the explored parameter in every single run. If you want small hdf5 files, this is the first option to set to false.
- **results_per_run** – Expected results you store per run. If you give a good/correct estimate storage to hdf5 file is much faster in case you store LARGE overview tables.

  Default is 0, i.e. the number of results is not estimated!
- **derived_parameters_per_run** – Analogous to the above.
- **display_time** – How often status messages about loading and storing time should be displayed. Interval in seconds.

- **trajectory** – A trajectory container, the storage service will add the used parameter to the trajectory container.

ADD_ROW = 'ADD'
Adds a row to an overview table

REMOVE_ROW = 'REMOVE'
Removes a row from an overview table

MODIFY_ROW = 'MODIFY'
Changes a row of an overview table

COLL_TYPE = 'COLL_TYPE'
Type of a container stored to hdf5, like list,tuple,dict,etc

Must be stored in order to allow perfect reconstructions.

COLL_LIST = 'COLL_LIST'
Container was a list

COLL_TUPLE = 'COLL_TUPLE'
Container was a tuple

COLL_NDARRAY = 'COLL_NDARRAY'
Container was a numpy array

COLL_MATRIX = 'COLL_MATRIX'
Container was a numpy matrix

COLL_DICT = 'COLL_DICT'
Container was a dictionary

COLL_EMPTY_DICT = 'COLL_EMPTY_DICT'
Container was an empty dictionary

COLL_SCALAR = 'COLL_SCALAR'
No container, but the thing to store was a scalar

SCALAR_TYPE = 'SCALAR_TYPE'
Type of scalars stored into a container

NAME_TABLE_MAPPING = {'_overview_parameters': 'parameters_overview', '_overview_derived_parameters_summ
Mapping of trajectory config names to the tables

PR_ATTR_NAME_MAPPING = {'_derived_parameters_per_run': 'derived_parameters_per_run', '_purge_duplicate_
Mapping of Attribute names for hdf5_settings table

ATTR_LIST = ['complevel', 'complib', 'shuffle', 'fletcher32', 'pandas_format', 'encoding']
List of HDF5StorageService Attributes that have to be stored into the hdf5_settings table

STORAGE_TYPE = 'SRVC_STORE'
Flag, how data was stored

ARRAY = 'ARRAY'
Stored as array

CARRAY = 'CARRAY'
Stored as carray

EARRAY = 'EARRAY'
Stored as earray_e.

VLARRAY = 'VLARRAY'
Stored as vlarray

TABLE = 'TABLE'
Stored as pytable

DICT = 'DICT'
Stored as dict.

In fact, stored as pytable, but the dictionary wil be reconstructed.

FRAME = 'FRAME'
Stored as pandas DataFrame

SERIES = 'SERIES'
Store data as pandas Series

PANEL = 'PANEL'
Store data as pandas Panel(4D)

SPLIT_TABLE = 'SPLIT_TABLE'
If a table was split due to too many columns

DATATYPE_TABLE = 'DATATYPE_TABLE'
If a table contains the data types instead of the attrs

SHARED_DATA = 'SHARED_DATA_'
An HDF5 data object for direct interaction

TYPE_FLAG_MAPPING = {<Mock object at 0x7f03a6bdc810>: 'ARRAY', <class 'pypet.shareddata.SharedEArray'>:
Mapping from object type to storage flag

FORMATTED_COLUMN_PREFIX = 'SRVC_COLUMN_%s_'
Stores data type of a specific pytables column for perfect reconstruction

DATA_PREFIX = 'SRVC_DATA_'
Stores data type of a pytables carray or array for perfect reconstruction

ANNOTATION_PREFIX = 'SRVC_AN_'
Prefix to store annotations as node attributes

ANNOTATED = 'SRVC_ANNOTATED'
Whether an item was annotated

INIT_PREFIX = 'SRVC_INIT_'
Hdf5 attribute prefix to store class name of parameter or result

CLASS_NAME = 'SRVC_INIT_CLASS_NAME'
Name of a parameter or result class, is converted to a constructor

COMMENT = 'SRVC_INIT_COMMENT'
Comment of parameter or result

LENGTH = 'SRVC_INIT_LENGTH'
Length of a parameter if it is explored, no longer in use, only for backwards compatibility

LEAF = 'SRVC_LEAF'
Whether an hdf5 node is a leaf node

is_open
Normally the file is opened and closed after each insertion.

However, the storage service may provide the option to keep the store open and signals this via this property.

encoding
How unicode strings are encoded

display_time
Time interval in seconds, when to display the storage or loading of nodes

complib
Compression library used

complevel
Compression level used

fletcher32
Whether fletcher 32 should be used

shuffle
Whether shuffle filtering should be used

pandas_append
If pandas should create storage in append mode.

DEPRECATED. No longer used, please use shared data instead!

pandas_format
Format of pandas data. Applicable formats are 'table' (or 't') and 'fixed' (or 'f')

filename
The name and path of the underlying hdf5 file.

load(*msg*, *stuff_to_load*, **args*, ***kwargs*)
Loads a particular item from disk.

The storage service always accepts these parameters:

Parameters

- **trajectory_name** – Name of current trajectory and name of top node in hdf5 file.
- **trajectory_index** – If no *trajectory_name* is provided, you can specify an integer index. The trajectory at the index position in the hdf5 file is considered to loaded. Negative indices are also possible for reverse indexing.
- **filename** – Name of the hdf5 file

The following messages (first argument msg) are understood and the following arguments can be provided in combination with the message:

- *pypet.pypetconstants.TRAJECTORY* ('TRAJECTORY')
  Loads a trajectory.
  param stuff_to_load The trajectory
  param as_new Whether to load trajectory as new
  param load_parameters How to load parameters and config
  param load_derived_parameters How to load derived parameters
  param load_results How to load results
  param force Force load in case there is a pypet version mismatch
  You can specify how to load the parameters, derived parameters and results as follows:

  pypet.pypetconstants.LOAD_NOTHING: (0)
  Nothing is loaded
  pypet.pypetconstants.LOAD_SKELETON: (1)
  The skeleton including annotations are loaded, i.e. the items are empty. Non-empty items in RAM are left untouched.
  pypet.pypetconstants.LOAD_DATA: (2)
  The whole data is loaded. Only empty or in RAM non-existing instance are filled with the data found on disk.
  pypet.pypetconstants.OVERWRITE_DATA: (3)
  The whole data is loaded. If items that are to be loaded are already in RAM and not empty, they are emptied and new data is loaded from disk.
- *pypet.pypetconstants.LEAF* ('LEAF')
  Loads a parameter or result.
  param stuff_to_load The item to be loaded
  param load_data How to load data
  param load_only If you load a result, you can partially load it and ignore the rest of the data. Just specify the name of the data you want to load. You can also provide a list, for example *load_only='spikes'*, *load_only=['spikes','membrane_potential']*.

  Issues a warning if items cannot be found.

param load_except If you load a result you can partially load in and specify items that should NOT be loaded here. You cannot use *load_except* and *load_only* at the same time.

- `pypet.pyetconstants.GROUP`
  Loads a group a node (comment and annotations)
  param recursive Recursively loads everything below
  param load_data How to load stuff if `recursive=True` accepted values as above for loading the trajectory
  param max_depth Maximum depth in case of recursion. *None* for no limit.
- *`pypet.pypetconstants.TREE`* ('TREE')
  Loads a whole subtree
  param stuff_to_load The parent node (!) not the one where loading starts!
  param child_name Name of child node that should be loaded
  param recursive Whether to load recursively the subtree below child
  param load_data How to load stuff, accepted values as above for loading the trajectory
  param max_depth Maximum depth in case of recursion. *None* for no limit.
  param trajectory The trajectory object
- *`pypet.pypetconstants.LIST`* ('LIST')
  Analogous to *storing lists*

RaisesNoSuchServiceError if message or data is not understood

DataNotInStorageError if data to be loaded cannot be found on disk

`store`(*msg*, *stuff_to_store*, **args*, ***kwargs*)

Stores a particular item to disk.

The storage service always accepts these parameters:

Parameters

- **`trajectory_name`** – Name or current trajectory and name of top node in hdf5 file
- **`filename`** – Name of the hdf5 file
- **`file_title`** – If file needs to be created, assigns a title to the file.

The following messages (first argument msg) are understood and the following arguments can be provided in combination with the message:

- *`pypet.pypetconstants.PREPARE_MERGE`* ('PREPARE_MERGE'):
  Called to prepare a trajectory for merging, see also 'MERGE' below.

  Will also be called if merging cannot happen within the same hdf5 file. Stores already enlarged parameters and updates meta information.
  param stuff_to_store Trajectory that is about to be extended by another one
  param changed_parameters List containing all parameters that were enlarged due to merging
  param old_length Old length of trajectory before merge
- *`pypet.pypetconstants.MERGE`* ('MERGE')
  Note that before merging within HDF5 file, the storage service will be called with msg='PREPARE_MERGE' before, see above.

  Raises a ValueError if the two trajectories are not stored within the very same hdf5 file. Then the current trajectory needs to perform the merge slowly item by item.

  Merges two trajectories, parameters are:
  param stuff_to_store The trajectory data is merged into
  param other_trajectory_name Name of the other trajectory
  param rename_dict Dictionary containing the old result and derived parameter names in the other trajectory and their new names in the current trajectory.

param move_nodes Whether to move the nodes from the other to the current trajectory

param delete_trajectory Whether to delete the other trajectory after merging.

- *pypet.pypetconstants.BACKUP* ('BACKUP')

  param stuff_to_store Trajectory to be backed up

  param backup_filename Name of file where to store the backup. If None the backup file will be in the same folder as your hdf5 file and named 'backup_XXXXX.hdf5' where 'XXXXX' is the name of your current trajectory.

- *pypet.pypetconstants.TRAJECTORY* ('TRAJECTORY')

  Stores the whole trajectory

  param stuff_to_store The trajectory to be stored

  param only_init If you just want to initialise the store. If yes, only meta information about the trajectory is stored and none of the nodes/leaves within the trajectory.

  param store_data How to store data, the following settings are understood:

  pypet.pypetconstants.STORE_NOTHING: (0)

  Nothing is stored

  pypet.pypetconstants.STORE_DATA_SKIPPING: (1)

  Data of not already stored nodes is stored

  pypet.pypetconstants.STORE_DATA: (2)

  Data of all nodes is stored. However, existing data on disk is left untouched.

  pypet.pypetconstants.OVERWRITE_DATA: (3)

  Data of all nodes is stored and data on disk is overwritten. May lead to fragmentation of the HDF5 file. The user is adviced to recompress the file manually later on.

- *pypet.pypetconstants.SINGLE_RUN* ('SINGLE_RUN')

  param stuff_to_store The trajectory

  param store_data How to store data see above

  param store_final If final meta info should be stored

- *pypet.pypetconstants.LEAF*

  Stores a parameter or result

  Note that everything that is supported by the storage service and that is stored to disk will be perfectly recovered. For instance, you store a tuple of numpy 32 bit integers, you will get a tuple of numpy 32 bit integers after loading independent of the platform!

  param stuff_to_sore Result or parameter to store

  In order to determine what to store, the function '_store' of the parameter or result is called. This function returns a dictionary with name keys and data to store as values. In order to determine how to store the data, the storage flags are considered, see below.

  The function '_store' has to return a dictionary containing values only from the following objects:

  - python natives (int, long, str, bool, float, complex),
  - numpy natives, arrays and matrices of type np.int8-64, np.uint8-64, np.float32-64, np.complex, np.str
  - python lists and tuples of the previous types (python natives + numpy natives and arrays) Lists and tuples are not allowed to be nested and must be homogeneous, i.e. only contain data of one particular type. Only integers, or only

floats, etc.
- python dictionaries of the previous types (not nested!), data can be heterogeneous, keys must be strings. For example, one key-value-pair of string and int and one key-value pair of string and float, and so on.
- pandas DataFrames
- *ObjectTable*

The keys from the '_store' dictionaries determine how the data will be named in the hdf5 file.

param store_data How to store the data, see above for a descitpion.

param store_flags Flags describing how to store data.

`ARRAY` ('ARRAY')
: Store stuff as array

`CARRAY` ('CARRAY')
: Store stuff as carray

`TABLE` ('TABLE')
: Store stuff as pytable

`DICT` ('DICT')
: Store stuff as pytable but reconstructs it later as dictionary on loading

`FRAME` ('FRAME')
: Store stuff as pandas data frame

Storage flags can also be provided by the parameters and results themselves if they implement a function '_store_flags' that returns a dictionary with the names of the data to store as keys and the flags as values.

If no storage flags are provided, they are automatically inferred from the data. See `pypet.HDF5StorageService.TYPE_FLAG_MAPPING` for the mapping from type to flag.

param overwrite Can be used if parts of a leaf should be replaced. Either a list of HDF5 names or *True* if this should account for all.

- *pypet.pypetconstants.DELETE* ('DELETE')

  Removes an item from disk. Empty group nodes, results and non-explored parameters can be removed.

  param stuff_to_store The item to be removed.

  param delete_only Potential list of parts of a leaf node that should be deleted.

  param remove_from_item If *delete_only* is used, whether deleted nodes should also be erased from the leaf nodes themseleves.

  param recursive If you want to delete a group node you can recursively delete all its children.

- *pypet.pypetconstants.GROUP* ('GROUP')

  param stuff_to_store The group to store

  param store_data How to store data

  param recursive To recursively load everything below.

  param max_depth Maximum depth in case of recursion. *None* for no limit.

- *pypet.pypetconstants.TREE*

  Stores a single node or a full subtree

  param stuff_to_store Node to store

  param store_data How to store data

  param recursive Whether to store recursively the whole sub-tree

  param max_depth Maximum depth in case of recursion. *None* for no limit.

- *pypet.pypetconstants.DELETE_LINK*

  Deletes a link from hard drive

param name The full colon separated name of the link

- *pypet.pypetconstants.LIST*
  Stores several items at once
  param stuff_to_store Iterable whose items are to be stored. Iterable must contain tuples, for example *[(msg1,item1,arg1,kwargs1),(msg2,item2,arg2,kwargs2),...]*
- *pypet.pypetconstants.ACCESS_DATA*
  Requests and manipulates data within the storage. Storage must be open.
  param stuff_to_storeA colon separated name to the data path
  param item_name The name of the data item to interact with
  param requestA functional request in form of a string
  param args Positional arguments passed to the reques
  param kwargs Keyword arguments passed to the request
- *pypet.pypetconstants.OPEN_FILE*
  Opens the HDF5 file and keeps it open
  param stuff_to_store None
- *pypet.pypetconstants.CLOSE_FILE*
  Closes an HDF5 file that was kept open, must be open before.
  param stuff_to_store None
- *pypet.pypetconstants.FLUSH*
  Flushes an open file, must be open before.
  param stuff_to_store None

RaisesNoSuchServiceError if message or data is not understood

item
alias of str

3.10.2 The Multiprocessing Wrappers

class pypet.storageservice.**LockWrapper**(*storage_service*, *lock=None*)
For multiprocessing in *WRAP_MODE_LOCK* mode, augments a storage service with a lock.

The lock is acquired before storage or loading and released afterwards.

is_open
Normally the file is opened and closed after each insertion.

However, the storage service may provide the option to keep the store open and signals this via this property.

load(**args*, ***kwargs*)
Acquires a lock before loading and releases it afterwards.

multiproc_safe
Usually storage services are not supposed to be multiprocessing safe

store(**args*, ***kwargs*)
Acquires a lock before storage and releases it afterwards.

class pypet.storageservice.**QueueStorageServiceSender**(*storage_queue=None*)
For multiprocessing with *WRAP_MODE_QUEUE*, replaces the original storage service.

All storage requests are send over a queue to the process running the *QueueStorageServiceWriter*.

Does not support loading of data!

send_done()
Signals the writer that it can stop listening to the queue

store(**args*, ***kwargs*)
Puts data to store on queue.

Note that the queue will no longer be pickled if the Sender is pickled.

class pypet.storageservice.**QueueStorageServiceWriter**(*storage_service*, *storage_queue*)

Wrapper class that listens to the queue and stores queue items via the storage service.

3.10.3 Empty Storage Service for Debugging

class pypet.storageservice.**LazyStorageService**(**args*, ***kwargs*)

This lazy guy does nothing! Only for debugging purposes.

Ignores all storage and loading requests and simply executes *pass* instead.

load(**args*, ***kwargs*)

Nope, I won't care, dude!

store(**args*, ***kwargs*)

Do whatever you want, I won't store anything!

3.11 Brian Parameters, Results and Monitors

Module containing results and parameters that can be used to store BRIAN data.

Parameters handling BRIAN data are instantiated by the *BrianParameter* class for any BRIAN Quantity.

The *BrianResult* can store BRIAN Quantities and the *BrianMonitorResult* extracts data from BRIAN Monitors.

All these can be combined with the experimental framework in *pypet.brian.network* to allow fast setup of large scale BRIAN experiments.

3.11.1 BrianParameter

class pypet.brian.parameter.**BrianParameter**(*full_name*, *data=None*, *comment=''*, *storage_mode='FLOAT'*)

A Parameter class that supports BRIAN Quantities.

Note that only scalar BRIAN quantities are supported, lists, tuples or dictionaries of BRIAN quantities cannot be handled.

There are two storage modes, that can be either passed to constructor or changed via *v_storage_mode*:

- *FLOAT_MODE*: ('FLOAT')

  The value is stored as a float and the unit as a sting.

  i.e. *12 mV* is stored as *12.0* and *'1.0 * mV'*
- *STRING_MODE*: ('STRING')

  The value and unit are stored combined together as a string.

  i.e. *12 mV* is stored as *'12.0 * mV'*

Supports data for the standard *Parameter*, too.

FLOAT_MODE = 'FLOAT'

Float storage mode

IDENTIFIER = '__brn__'

Identification string stored into column title of hdf5 table

STRING_MODE = 'STRING'

String storage mode

f_supports(*data*)

Simply checks if data is supported

v_storage_mode

There are two storage modes:

- *FLOAT_MODE*: ('FLOAT')
  The value is stored as a float and the unit as a sting.

  i.e. *12 mV* is stored as *12.0* and *'1.0 * mV'*
- *STRING_MODE*: ('STRING')
  The value and unit are stored combined together as a string.

  i.e. *12 mV* is stored as *'12.0 * mV'*

3.11.2 BrianDurationParameter

class pypet.brian.parameter.**BrianDurationParameter**(*full_name*, *data=None*, *order=0*, *comment=''*, *storage_mode='FLOAT'*)

Special BRIAN parameter to specify orders and durations of subruns.

The *NetworkRunner* extracts the individual subruns for a given network from such duration parameters. The order of execution is defined by the property *v_order*. The exact values do not matter only the rank ordering.

A Duration Parameter should be in time units (ms or s, for instance).

DEPRECATED: Please use a normal `BrianParameter` instead and add the property *order* to it's *Annotations*. No longer use:

```
>>> subrun = BrianDurationParameter('mysubrun', 10 * s, order=42)
```

But use:

```
>>> subrun = BrianParameter('mysubrun', 10 * s)
>>> subrun.v_annotations.order=42
```

3.11.3 BrianResult

class pypet.brian.parameter.**BrianResult**(*full_name*, **args*, ***kwargs*)

A result class that can handle BRIAN quantities.

Note that only scalar BRIAN quantities are supported, lists, tuples or dictionaries of BRIAN quantities cannot be handled.

Supports also all data supported by the standard *Result*.

Storage mode works as for *BrianParameter*.

FLOAT_MODE = 'FLOAT'
Float storage mode

IDENTIFIER = '__brn__'
Identifier String to label brian data

STRING_MODE = 'STRING'
String storage mode

f_set_single(*name*, *item*)
Sets a single data item of the result.

Raises TypeError if the type of the outer data structure is not understood. Note that the type check is shallow. For example, if the data item is a list, the individual list elements are NOT checked whether their types are appropriate.

Parameters
- **name** – The name of the data item
- **item** – The data item

RaisesTypeError

Example usage:

```
>>> res.f_set_single('answer', 42)
>>> res.f_get('answer')
42
```

v_storage_mode

There are two storage modes:

- *FLOAT_MODE*: ('FLOAT')

  The value is stored as a float and the unit as a sting,

  i.e. *12 mV* is stored as *12.0* and *'1.0 * mV'*

- *STRING_MODE*: ('STRING')

  The value and unit are stored combined together as a string,

  i.e. *12 mV* is stored as *'12.0 * mV'*

3.11.4 BrianMonitorResult

class pypet.brian.parameter.**BrianMonitorResult**(*full_name*, **args*, ***kwargs*)

A Result class that supports brian monitors.

Subclasses *Result*, NOT *BrianResult*. The storage mode here works slightly different than in *BrianResult* and *BrianParameter*, see below.

Monitor attributes are extracted and added as results with the attribute names. Note the original monitors are NOT stored, only their attribute/property values are kept.

Add monitor on *__init__* via *monitor=* or via *f_set(monitor=brian_monitor)*

IMPORTANT: You can only use 1 result per monitor. Otherwise a 'TypeError' is thrown.

Example:

```
>>> brian_result = BrianMonitorResult('example.brian_test_test.mymonitor',
                                      monitor=SpikeMonitor(...),
                                      storage_mode='TABLE',
                                      comment='Im a SpikeMonitor Example!')
>>> brian_result.nspikes
1337
```

There are two storage modes in case you use the SpikeMonitor and StateSpikeMonitor:

- *TABLE_MODE*: ('TABLE')

  Default, information is stored into a single table where one column contains the neuron index, another the spiketimes and following columns contain variable values (in case of the StateSpikeMonitor) This is a very compact storage form.

- ARRAY_MODE: ('ARRAY')

  For each neuron there will be a new hdf5 array, i.e. if you have many neurons your result node will have many entries. Note that this mode does sort everything according to the neurons but reading and writing of data might take muuuuuch longer compared to the other mode.

Following monitors are supported and the following values are extraced:

- SpikeCounter
  - –count

    Array of spike counts for each neuron
  - –nspikes

    Number of recorded spikes
  - –source

    Name of source recorded from as string.
- VanRossumMetric
  - –tau

    Time constant of kernel.

- –tau_unit
  'second'
- –distance
  A square symmetric matrix containing the distances
- –N
  Number of neurons.
- –source

•PopulationSpikeCounter

- –delay
  Recording delay
- –nspikes
- –source

•StateSpikeMonitor

- –delay
- –nspikes
- –source
- –varnames
  Names of recorded variables as tuple of strings.
- –spiketimes_unit
  'second'
- –variablename_unit
  Unit of recorded variable as a string. 'variablename' is mapped to the name of a recorded variable. For instance, if you recorded the membrane potential named 'vm' you would get a field named 'vm_unit'.

If you use v_storage_mode = *TABLE_MODE*

- –spikes
  pandas DataFrame containing in the columns:

  'neuron': neuron indices

  'spiketimes': times of spiking

  'variablename': values of the recorded variables

If you use v_storage_mode = *ARRAY_MODE*

- –spiketimes_XXX
  spiketimes of neuron 'XXX' for each neuron you recorded from. The number of digits used to represent and format the neuron index are chosen automatically.
- –variablename_XXX
  Value of the recorded variable at spiketimes for neuron *XXX*
- –format_string
  String used to format the neuron index, for example *'%03d'*.

•PopulationRateMonitor

- –times
  The times of the bins.
- –times_unit
  'second'
- –rate
  An array of rates in Hz
- –rate_unit
  'Hz'
- –source
- –bin
  The duration of a bin (in second).
- –delay

•ISIHistogramMonitor:

- –source
- –delay
- –nspikes
- –bins

The bins array passed at initialisation of the monitor.

–count

An array of length *len(bins)* counting how many ISIs were in each bin.

•SpikeMonitor

–delay

–nspikes

–source

–spiketimes_unit

'second'

If you use v_storage_mode = *TABLE_MODE*

–spikes

pandas DataFrame containing in the columns:

'neuron': neuron indices

'spiketimes': times of spiking

If you use v_storage_mode = *ARRAY_MODE*

–spiketimes_XXX

spiketimes of neuron 'XXX' for each neuron you recorded from. The number of digits used to represent and format the neuron index are chosen automatically.

–format_string

String used to format the neuron index, for example *'%03d'*.

•StateMonitor

–source

–record

What to record. Can be 'True' to record from all neurons. A single integer value or a list of integers.

–when

When recordings were made, for a list of potential values see BRIAN.

–timestep

Integer defining the clock timestep a recording was made.

–times

Array of recording times

–times_unit

'second'

–mean

Mean value of the state variable for every neuron in the group. Only extracted if mean values are calculated by BRIAN. Note that for newer versions of BRIAN, means and variances are no longer extracted if *record* is NOT set to *False*.

–var

Unbiased estimated of variances of state variable for each neuron. Only extracted if variance values are calculated by BRIAN.

–values

A 2D array of the values of all recorded neurons, each row is a single neuron's value

–unit

The unit of the values as a string

–varname

Name of recorded variable

•MultiStateMonitor

As above but instead of values and unit, the result contains 'varname_values' and 'varname_unit', where 'varname' is the name of the recorded variable.

ARRAY_MODE = 'ARRAY'

Array storage mode, not recommended if you have many neurons!

TABLE_MODE = 'TABLE'

Table storage mode for SpikeMonitor and StateSpikeMonitor

f_set_single(*name*, *item*)
To add a monitor use *f_set_single('monitor', brian_monitor)*.

Otherwise *f_set_single* works similar to *f_set_single()*.

v_monitor_type
The type of the stored monitor. Each MonitorResult can only manage a single Monitor.

v_storage_mode
The storage mode for SpikeMonitor and StateSpikeMonitor

There are two storage modes:

- *TABLE_MODE*: ('TABLE')
  Default, information is stored into a single table where the first column is the neuron index, second column is the spike time following columns contain variable values (in case of the StateSpikeMonitor) This is a very compact storage form.
- *ARRAY_MODE*: ('ARRAY')
  For each neuron there will be a new hdf5 array, i.e. if you have many neurons your result node will have many entries. Note that this mode does sort everything according to the neurons but reading and writing of data might take muuuuuch longer compared to the other mode.

3.12 Brian Network Framework

Module for easy compartmental implementation of a BRIAN network.

Build parts of a network via subclassing *NetworkComponent* and *NetworkAnalyser* for recording and statistical analysis.

Specify a *NetworkRunner* (subclassing optionally) that handles the execution of your experiment in different subruns. Subruns can be defined as *BrianParameter* instances in a particular trajectory group. You must add to every parameter's *Annotations* the attribute *order*. This order must be an integer specifying the index or order the subrun should about to be executed in.

The creation and management of a BRIAN network is handled by the *NetworkManager* (no need for subclassing). Pass your components, analyser and your runner to the manager.

Pass the *run_network()* function together with a *NetworkManager* to your main environment function *f_run()* to start a simulation and parallel parameter exploration. Be aware that in case of a *pre-built* network, successful parameter exploration requires parallel processing (see *NetworkManager*).

3.12.1 Quicklinks

These function can directly be called or used by the user.

run_network	Top-level simulation function, pass this together with a NetworkManager to the er
NetworkManager.add_parameters	Adds parameters for a network simulation.
NetworkManager.pre_run_network	Starts a network run before the individual run.
NetworkManager.pre_build	Pre-builds network components.

The private functions of the runner and the manager are also listed below to allow fast browsing of the source code.

3.12.2 Functions that can be implemented by a Subclass

These functions can be implemented in the subclasses:

NetworkComponent.build	Builds network objects at the beginning of each individual experimental
NetworkComponent.add_to_network	Can add network objects before a specific *subrun*.
NetworkComponent.remove_from_network	Can remove network objects before a specific *subrun*.
NetworkComponent.pre_build	Builds network objects before the actual experimental runs.
NetworkAnalyser.analyse	Can perform statistical analysis on a given network.

I would suggest in case one subclasses *NetworkRunner* to implement its *add_parameters()* method (inherited from *NetworkComponent*) in order to add *BrianDurationParameter* instances to *traj.parameters.simulation.durations* or *traj.parameters.simulation.pre_durations* to define the length and order of individual subruns.

For a description of the structure and different phases of an individual simulation run see *run_network()*.

3.12.3 Top-Level run_network Function

pypet.brian.network.**run_network**(*traj*, *network_manager*)

Top-level simulation function, pass this together with a NetworkManager to the environment.

DEPRECATED: Please pass *network_manager.run_network* to the environment's *f_run* function

Parameters

- **traj** – Trajectory container
- **network_manager** – *NetworkManager* instance
  - Creates a BRIAN network.
  - Manages all *NetworkComponent* instances, all *NetworkAnalyser* and a single *NetworkRunner*.

3.12.4 NetworkManager

class pypet.brian.network.**NetworkManager**(*network_runner*, *component_list*, *analyser_list=()*, *force_single_core=False*, *network_constructor=None*)

Manages a BRIAN network experiment and creates the network.

An experiment consists of

Parameters

- **network_runner** – A *NetworkRunner*

  Special component that handles the execution of several subruns. A NetworkRunner can be subclassed to implement the *add_parameters()* method to add *BrianParameter* instances defining the order and duration of subruns.
- **component_list** – List of `NetworkComponents` instances to create and manage individual parts of a network. They are build and added to the network in the order defined in the list.

  NetworkComponent always needs to be sublcassed and defines only an abstract interface. For instance, one could create her or his own subclass called NeuronGroupComponent that creates NeuronGroups, Whereas a SynapseComponent creates Synapses between the before built NeuronGroups. Accordingly, the SynapseComponent instance is listed after the NeuronGroupComponent.
- **analyser_list** – List of `Analyser` instances for recording and statistical evaluation of a BRIAN network. They should be used to add monitors to a network and to do further processing of the monitor data.

This division allows to create compartmental network models where one can easily replace parts of a network simulation.

Parameters

- **force_single_core** – In case you *pre_build()* or even `pre_run()` a network, you usually cannot use single core processing. The problem with single core processing is that iterative exploration of the parameter space alters the

network on every iteration and the network cannot be reset to the initial conditions holding before the very first experimental run. This is an inherent problem of BRIAN. The only way to overcome this problem is multiprocessing and copying (either by pickling or by forking) the whole BRIAN environment.

If you are not bothered by not starting every experimental run with the very same network, you can set *force_single_core=True*. The NetworkManager will do iterative single processing and ignore the ongoing modification of the network throughout all runs.

In case *multiproc=True* for your environment, the setting of *force_single_core* is irrelevant and has no effect.

- **network_constructor** – If you have a custom network constructor apart from the Brian one, pass it here.

_run_network(*traj*)

Starts a single run carried out by a NetworkRunner.

Called from the public function `run_network()`.

Parameters **traj** – Trajectory container

add_parameters(*traj*)

Adds parameters for a network simulation.

Calls *add_parameters()* for all components, analyser, and the network runner (in this order).

Parameters **traj** – Trajectory container

build(*traj*)

Pre-builds network components.

Calls *build()* for all components, analysers and the network runner.

build does not need to be called by the user. If *~pypet.brian.network.run_network* is passed to an *Environment* with this Network manager, *build* is automatically called for each individual experimental run.

Parameters **traj** – Trajectory container

pre_build(*traj*)

Pre-builds network components.

Calls *pre_build()* for all components, analysers, and the network runner.

pre_build is not automatically called but either needs to be executed manually by the user, either calling it directly or by using `pre_run()`.

This function does not create a BRIAN network, but only it's components.

Parameters **traj** – Trajectory container

pre_run_network(*traj*)

Starts a network run before the individual run.

Useful if a network needs an initial run that can be shared by all individual experimental runs during parameter exploration.

Needs to be called by the user. If *pre_run_network* is started by the user, *pre_build()* will be automatically called from this function.

This function will create a new BRIAN network which is run by the *NetworkRunner* and it's *execute_network_pre_run()*.

To see how a network run is structured also take a look at `run_network()`.

Parameters **traj** – Trajectory container

run_network(*traj*)

Top-level simulation function, pass this to the environment

Performs an individual network run during parameter exploration.

run_network does not need to be called by the user. If the top-level ~*pypet.brian.network.run_network* method (not this one of the NetworkManager) is passed to an *Environment* with this NetworkManager, *run_network* and *build()* are automatically called for each individual experimental run.

This function will create a new BRIAN network in case one was not pre-run. The execution of the network run is carried out by the *NetworkRunner* and it's *execute_network_run()* (also take a look at this function's documentation to see the structure of a network run).

Parameters **traj** – Trajectory container

3.12.5 NetworkRunner

class pypet.brian.network.**NetworkRunner**(*report='text'*, *report_period=None*, *durations_group_name='simulation.durations'*, *pre_durations_group_name='simulation.pre_durations'*)

Specific NetworkComponent to carry out the running of a BRIAN network experiment.

A NetworRunner only handles the execution of a network simulation, the BRIAN network is created by a *NetworkManager*.

Can potentially be subclassed to allow the adding of parameters via *add_parameters()*. These parameters should specify an experimental run with a :class:~pypet.brian.parameter.BrianParameter' to define the order and duration of network subruns. For the actual experimental runs, all subruns must be stored in a particular trajectory group. By default this *traj.parameters.simulation.durations*. For a pre-run the default is *traj.parameters.simulation.pre_durations*. These default group names can be changed at runner initialisation (see below).

The network runner will look in the *v_annotations* property of each parameter in the specified trajectory group. It searches for the entry *order* to determine the order of subruns.

Parameters

- **report** – How simulation progress should be reported, see also the parameters of *run(...)* in a BRIAN network and the magic run method.
- **report_period** – How often progress is reported. If not specified 10 seconds is chosen.
- **durations_group_name** – Name where to look for *BrianParameter* instances in the trajectory which specify the order and durations of subruns.
- **pre_durations_group_name** – As above, but for pre running a network.

Moreover, in your subclass you can log messages with the private attribute *_logger* which is initialised in *_set_logger()*.

_execute_network_run(*traj*, *network*, *network_dict*, *component_list*, *analyser_list*, *pre_run=False*)

Generic *execute_network_run* function, handles experimental runs as well as pre-runs.

See also ***execute_network_run()*** **and** *execute_network_pre_run()*.

_extract_subruns(*traj*, *pre_run=False*)

Extracts subruns from the trajectory.

Parameters

- **traj** – Trajectory container
- **pre_run** – Boolean whether current run is regular or a pre-run

Raises RuntimeError if orders are duplicates or even missing

execute_network_pre_run(*traj*, *network*, *network_dict*, *component_list*, *analyser_list*)

Runs a network before the actual experiment.

Called by a *NetworkManager*. Similar to `run_network()`.

Subruns and their durations are extracted from the trajectory. All *BrianParameter* instances found under *traj.parameters.simulation.pre_durations* (default, you can change the name of the group where to search for durations at runner initialisation). The order is determined from the *v_annotations.order* attributes. There must be at least one subrun in the trajectory, otherwise an AttributeError is thrown. If two subruns equal in their order property a RuntimeError is thrown.

Parameters

- **traj** – Trajectory container
- **network** – BRIAN network
- **network_dict** – Dictionary of items shared among all components
- **component_list** – List of *NetworkComponent* objects
- **analyser_list** – List of *NetworkAnalyser* objects

execute_network_run(*traj*, *network*, *network_dict*, *component_list*, *analyser_list*)

Runs a network in an experimental run.

Called by a *NetworkManager*.

A network run is divided into several subruns which are defined as *BrianParameter* instances.

These subruns are extracted from the trajectory. All *BrianParameter* instances found under *traj.parameters.simulation.durations* (default, you can change the name of the group where to search for durations at runner initialisation). The order is determined from the *v_annotations.order* attributes. An error is thrown if no orders attribute can be found or if two parameters have the same order.

There must be at least one subrun in the trajectory, otherwise an AttributeError is thrown. If two subruns equal in their order property a RuntimeError is thrown.

For every subrun the following steps are executed:

1. Calling *add_to_network()* for every every *NetworkComponent* in the order as they were passed to the *NetworkManager*.
2. Calling *add_to_network()* for every every *NetworkAnalyser* in the order as they were passed to the *NetworkManager*.
3. Calling *add_to_network()* of the NetworkRunner itself (usually the network runner should not add or remove anything from the network, but this step is executed for completeness).
4. Running the BRIAN network for the duration of the current subrun by calling the network's *run* function.
5. Calling *analyse()* for every every *NetworkAnalyser* in the order as they were passed to the *NetworkManager*.
6. Calling *remove_from_network()* of the NetworkRunner itself (usually the network runner should not add or remove anything from the network, but this step is executed for completeness).
7. Calling *remove_from_network()* for every every *NetworkAnalyser* in the order as they were passed to the *NetworkManager*
8. Calling *remove_from_network()* for every every *NetworkComponent* in the order as they were passed to the *NetworkManager*.

These 8 steps are repeated for every subrun in the *subrun_list*. The *subrun_list* passed to all *add_to_network*, *analyse* and *remove_from_network* methods can be modified within these functions to potentially alter the order of execution or even erase or add upcoming subruns if necessary.

For example, a NetworkAnalyser checks for epileptic pathological activity and cancels all coming subruns in case of undesired network dynamics.

Parameters

- **traj** – Trajectory container
- **network** – BRIAN network
- **network_dict** – Dictionary of items shared among all components
- **component_list** – List of *NetworkComponent* objects
- **analyser_list** – List of *NetworkAnalyser* objects

3.12.6 NetworkComponent

class pypet.brian.network.**NetworkComponent**

Abstract class to define a component of a BRIAN network.

Can be subclassed to define the construction of NeuronGroups or Connections, for instance.

add_parameters(*traj*)

Adds parameters to *traj*.

Function called from the *NetworkManager* to define and add parameters to the trajectory container.

add_to_network(*traj*, *network*, *current_subrun*, *subrun_list*, *network_dict*)
Can add network objects before a specific *subrun*.

Called by a *NetworkRunner* before a the given *subrun*.

Potentially one wants to add some BRIAN objects later to the network than at the very beginning of an experimental run. For example, a monitor might be added at the second subrun after an initial phase that is not supposed to be recorded.

Parameters

- **traj** – Trajectoy container
- **network** – BRIAN network where elements could be added via *add(...)*.
- **current_subrun** – *BrianParameter* specifying the very next subrun to be simulated.
- **subrun_list** – List of *BrianParameter* objects that are to be run after the current subrun.
- **network_dict** – Dictionary of items shared by all components.

build(*traj*, *brian_list*, *network_dict*)
Builds network objects at the beginning of each individual experimental run.

Function called from the *NetworkManager* at the beginning of every experimental run,

Parameters

- **traj** – Trajectory container
- **brian_list** – Add BRIAN network objects like NeuronGroups or Connections to this list. These objects will be automatically added at the instantiation of the network in case the network was not pre-run via *Network(*brian_list)* (see the BRIAN network class).
- **network_dict** – Add any item to this dictionary that should be shared or accessed by all your components and which are not part of the trajectory container. It is recommended to also put all items from the *brian_list* into the dictionary for completeness.

For convenience I recommend documenting the implementation of *build* and *pre-build* and so on in the subclass like the following. Use statements like *Adds* for items that are added to the list and the dict and statements like *Expects* for what is needed to be part of the *network_dict* in order to build the current component.

brian_list:

Adds:

4 Connections, between all types of neurons (e->e, e->i, i->e, i->i)

network_dict:

Expects:

'neurons_i': Inhibitory neuron group

'neurons_e': Excitatory neuron group

Adds:

'connections': List of 4 Connections,between all types of neurons (e->e, e->i, i->e, i->i)

pre_build(*traj*, *brian_list*, *network_dict*)
Builds network objects before the actual experimental runs.

Function called from the *NetworkManager* if components can be built before the actual experimental runs or in case the network is pre-run.

Parameters are the same as for the *build()* method.

remove_from_network(*traj*, *network*, *current_subrun*, *subrun_list*, *network_dict*)
Can remove network objects before a specific *subrun*.

Called by a *NetworkRunner* after a given *subrun* and shortly after analysis (see *NetworkAnalyser*).

Parameters

- **traj** – Trajectoy container
- **network** – BRIAN network where elements could be removed via *remove(...)*.
- **current_subrun** – *BrianParameter* specifying the current subrun that was executed shortly before.
- **subrun_list** – List of *BrianParameter* objects that are to be run after the current subrun.
- **network_dict** – Dictionary of items shared by all components.

3.12.7 NetworkAnalyser

class pypet.brian.network.**NetworkAnalyser**

Specific NetworkComponent that analysis a network experiment.

Can be subclassed to create components for statistical analysis of a network and network monitors.

analyse(*traj*, *network*, *current_subrun*, *subrun_list*, *network_dict*)

Can perform statistical analysis on a given network.

Called by a *NetworkRunner* directly after a given *subrun*.

Parameters

- **traj** – Trajectoy container
- **network** – BRIAN network
- **current_subrun** – *BrianParameter* specifying the current subrun that was executed shortly before.
- **subrun_list** – List of *BrianParameter* objects that are to be run after the current subrun. Can be deleted or added to change the actual course of the experiment.
- **network_dict** – Dictionary of items shared by all components.

Contact and License

4.1 Contact

Robert Meyer

robert.meyer (at) ni.tu-berlin.de

Marchstr. 23

TU-Berlin, MAR 5.046

D-10587 Berlin

4.2 License

p

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W