PyBNF Documentation

Release

Eshan Mitra, Ryan Suderman

Jul 02, 2018

CONTENTS:

PyBNF is a tool for parameter fitting of dynamical systems biology models specified using the BioNetGen rule-based modeling language (BNGL) or the Systems Biology Markup Language (SBML). It currently runs on most Linux and macOS workstations as well as distributed computing systems using the SLURM queueing manager.

CHAPTER

ONE

INSTALLATION

1.1 Python

PyBNF requires an installation of Python version 3.5 or higher. This should come built-in with most new Linux and Mac operating systems. However, we recommend installing the Anaconda Python distribution for Python v3.5 or higher. Installing Anaconda facilitates managing and installing Python packages as well as maintaining multiple Python environments. Instructions for installing on various platforms can be found on the Anaconda website.

1.2 PyBNF

The `pip` package manager is included with Anaconda and should be used to install PyBNF from the command line.

It is also possible to install `pip` alone without Anaconda. If you choose to do this, be sure your `pip` is associated with Python 3 (on some systems, this command is `pip3`).

1.2.1 Installing from PyPI

Note: This option is not yet available

Simply type the following in a terminal:

```
pip install pybnf
```

The above command will install the most recent version of PyBNF released on the Python Package Index, along with all required dependencies.

1.2.2 Installing from source

To use bleeding edge PyBNF, the source code may be found on GitHub at https://github.com/NAU-BioNetFit/PyBNF . To use, simply download or clone the repository and run the following command in the repository's root directory:

```
pip install -e .
```

This also allows developers to modify the source code while still having access to the commmand line functionality anywhere in the filesystem.

1.2.3 (Optional) Logging configuration for remote machines

By default, PyBNF logs to the file `bnf_timestamp.log` to maintain a record of important events in the application. When running PyBNF on a cluster, some of the logs may be written while on a node distinct from the main thread. If these logs are desired, the user must configure the scheduler to retrieve these logs.

Upon installation of PyBNF, the dependencies `dask` and `distributed` should be installed. Installing them will create a `.dask/` folder in the home directory with a single file: `config.yaml`. Open this file to find a `logging:` block containing information for how distributed outputs logs. Add the following line to the file, appropriately indented:

```
pybnf.algorithms.job:  info
```

where `info` can be any string corresponding to a Python logging level (e.g. `info`, `debug`, `warning`)

1.3 Installation of External Simulators

1.3.1 BioNetGen

PyBNF is designed to work with simulators present in the BioNetGen software suite, version 2.3. The current BioNetGen distribution includes support for both network-based simulations and network-free simulations. PyBNF will need to know the location of BioNetGen – specifically the location of the script `BNG2.pl` within the BioNetGen installation. This path can be included in the PyBNF configuration file with the *bng_command* key. A convenient alternative is to set the environment variable `BNGPATH` to the BioNetGen directory using the following command:

```
export BNGPATH=/path/to/bng2
```

where `/path/to/bng2` is the path to the folder containing `BNG2.pl`, not including the "BNG2.pl" file name. This setting may be made permanent as of your next login, by copying above command into the file `.bash_profile` in your home directory.

1.3.2 SBML

PyBNF runs simulations of SBML models using libroadrunner, which is installed automatically through `pip` as part of PyBNF installation.

To work with SBML files, it is useful to install software such as COPASI that is capable of reading and writing models in SBML format.

CHAPTER

TWO

QUICK START

2.1 Verify installation with simple examples

examples/demo contains two simple example configurations to verify that PyBNF and associated simulators are installed and working correctly. The model files consist of simple polynomial functions, and the entire fitting run should complete in under a minute.

To run the examples, use the following commands from the PyBNF root directory

For a simple job using BioNetGen: `pybnf -c examples/demo/demo_bng.conf`

For a simple job using SBML: `pybnf -c examples/demo/demo_xml.conf`

The examples will print progress to the terminal as the fitting proceeds, and the results will be saved in the directory PyBNF-output/demo/, located one level up from the PyBNF root directory (this output directory can be changed by editing `demo_bng.conf` and `demo_xml.conf`).

In PyBNF-output/demo/Results, the file sorted_params.txt contains the parameter sets tested during the fitting run. Open this file and verify that the best-fit parameter set (first line of the file) is close to the ground truth value of v1__FREE=0.5, v2__FREE=1.5, v3__FREE=3.0.

After verifying that PyBNF is installed correctly, it should be possible to run any of the other examples in the examples/ directory. For more information about these examples and the features they include, see *Examples*

2.1.1 On a SLURM cluster

To run the examples on a cluster with the Slurm resource manager, start by allocating 2 nodes for your job:

```
salloc -N 2
```

Log in to your allocated nodes (depending on your cluster, this may happen automatically without this command):

```
slogin
```

Then run pybnf as on a single machine, but use the `-t` flag to indicate that you are on a cluster:

```
pybnf -c examples/demo/demo_bng.conf -t slurm
```

```
pybnf -c examples/demo/demo_xml.conf -t slurm
```

To close your Slurm session after completing the jobs, run the command `exit` twice (once to log out of the node, and a second time to relinquish the job allocation)

2.2 Set up your own fitting job

In this Quick Start, we will assume your fitting run consists of a single BNGL file and a single experimental data set. For more advanced use cases, see the complete section on *Configuring a Fitting Job*.

Start by creating a new folder to contain your modified BNGL file, data file, configuration file, and results.

2.2.1 Modify your BNGL file

In your bngl file, replace each value you want PyBNF to fit with a name ending in `__FREE`

For example, if you want to fit var1, var2, and var3 in the following parameters block:

```
begin parameters

    var1 1
    var2 3
    var3 7

end parameters
```

Modify the BNGL code to:

```
begin parameters

    var1 var1__FREE
    var2 var2__FREE
    var3 var3__FREE

end parameters
```

In addition, edit your `simulate` command to include the `suffix` argument. For example:

```
simulate(method=>"ode",t_end=>60,suffix=>"data1")
```

2.2.2 Make your data file

Create a text file with the extension ".exp" and the same name as the suffix you defined above, for example, `data1.exp`.

The first line of this file should be a header, and the remaining lines should contain data in whitespace-delimited format. Your header should start with "#", followed by "time", followed by the names of observables in your BNGL file. Enter your data points on the subsequent lines, for example:

```
# time Obs1 Obs2
5      1.7  1e5
10     3.7  1.5e5
60     4.2  5e5
```

2.2.3 Make your configuration file

We'll run the fitting job using the differential evolution algorithm. Create the config file `my_config.conf` with the following contents:

```
model=model.bngl: data1.exp
output_dir=output/
bng_command=/path/to/bng2/BNG2.pl

objfunc=sos
fit_type=de
population_size=20
max_iterations=30

uniform_var=var1__FREE 1 10
uniform_var=var2__FREE 1 10
uniform_var=var3__FREE 1 10
```

Replace `model.bngl` and `data1.exp` with the names of your .bngl and .exp files. Replace `/path/to/bng2/BNG2.pl` with the full path to the file BNG2.pl on your computer (or delete the line if you have the BNGPATH enviorment variable set). Replace the variable names `var1__FREE` etc. with the names of the free parameters in your bngl file, and replace the corresponding numbers `1 10` with the minimum and maximum bounds for each parameter.

This config file will run the differential evolution algorithm on a population of 20 individuals for 30 iterations (600 simulations total), and evaluate the best fits using a sum-of-squares objective function. Adjust these settings as is suited for your model.

Once you have your config file edited as needed, run PyBNF from the folder containing all of your files:

```
pybnf -c my_config.conf
```

Congratulations, you've just completed your first PyBNF fitting job!

CHAPTER

THREE

CONFIGURING A FITTING JOB

3.1 The Configuration File

The configuration file is a plain text file with the extension ".conf" that specifies all of the information that PyBNF needs to perform the fitting: the location of the model and data files, and the details of the fitting algorithm to be run.

Several examples of .conf files are included in the examples/ folder.

Each line of a conf file has the general format config_key=value, which assigns the configuration key "config_key" to the value "value".

The available configuration keys to be specified are detailed in *Configuration Keys*.

3.2 Model Files

3.2.1 BioNetGen

BioNetGen models are specified in plain text files written in BioNetGen language (BNGL). Documentation for BNGL can be found at http://www.csb.pitt.edu/Faculty/Faeder/?page_id=409.

Two small modifications of a BioNetGen-compatible BNGL file are necessary to use the file with PyBNF

1. Replace each value to be fit with a name that ends in the string "__FREE".

For example, if the parameters block in our original file was the following:

```
begin parameters

    v1 17
    v2 42
    v3 37
    NA 6.02e23

end parameters
```

the revised version for PyBNF should look like:

```
begin parameters

    v1 v1__FREE
    v2 v2__FREE
    v3 v3__FREE
    NA 6.02e23
```

```
end parameters
```

We have replaced each fixed parameter value in the original file with a "FREE" parameter to be fit. Parameters that we do not want to fit (such as the physical constant NA) are left as is.

2. Use the "suffix" argument to create a correspondence between your simulation command and your experimental data file.

For example, if your simulation call `simulate({method=>"ode"})` generates data to be fit using the data file data1.exp, you should edit your call to `simulate({method=>"ode", suffix=>"data1"})`.

3.2.2 SBML

SBML files can be used with PyBNF as is, with no modifications required. PyBNF will match parameter names given in the configuration file, with the IDs of parameters or species in the SBML file. If the name of a species is given, PyBNF fits for the initial concentration of that species.

PyBNF assumes that any parameters and species that are not named in the config file are not meant to be fit - such values are held constant at the value specified in the SBML file.

To avoid mistakes in configuration, you may optionally append "__FREE" to the names of parameters to be fit, as with BioNetGen models. PyBNF will raise an error if it finds a parameter ending in "__FREE" in the SBML that is not specified in the configuration file.

Caution: If you are using COPASI to export SBML files, renaming a parameter is not straightforward. Typically, renaming a parameter only changes its `name` field, but PyBNF reads the `id` field.

Note that SBML files do not contain information about what time course or parameter scan simulations should be run on the model. Therefore, when using SBML files, it is required to specify this information in the configuration file with the `time_course` and `param_scan` keys.

3.3 Experimental Data Files

Experimental data file are plain text files with the extension ".exp" that contain whitespace-delimited tables of data to be used for fitting.

The first line of the .exp file is the header. It should contain the character #, followed by the names of each column. The first column name should be the name of the independent variable (e.g. "time" for a time course simulation). The rest of the column names should match the names of observables in a BNGL file, or species in an SBML file. The following lines should contain data, with numbers separated by whitespace. Use "nan" to indicate missing data. Here is a simple example of an exp file. In this case, the corresponding BNGL file should contain observables named X and Y:

```
#    time    X    Y
    0    5    1e4
    5    7    1.5e4
    10    9    4e4
    15    nan    6.5e4
    20    15    1.1e5
```

If your are fitting with the chi-squared objective function, you also need to provide a standard deviation for each experimental data point. To do so, include a column in the .exp file with "_SD" appended to the variable name. For example:

```
#    time     X     Y          X_SD     Y_SD
    0    5     1e4          1     2e2
    5    7     1.5e4     1.2     2e2
    10     9     4e4          1.4     4e2
    15     nan     6.5e4     nan     5e2
    20     15     1.1e5     0.9     5e2
```

3.4 Constraint files

Constraint files are plain text files with the extension ".con" that contain inequality constraints to be imposed on the outputs of the model. Such constraints can be used to formalize qualitative data known about the biological system of interest.

Each line of the .con file should contain constraint declaration consisting of three parts: an inequality to be satisfied, an enforcement condition that specifies when in the simulation time course the constraint is applied, and a weight that controls the penalty to add to the objective function if the constraint is not satisfied. Specifically, if a constraint of the form $A < B$ with weight w is violated, then the value added to the objective function is $w * (A - B)$.

The weight may be omitted and defaults to 1. The inequality and enforcement clauses are required

3.4.1 Inequality

The inequality can consist of any relationship (<, >, <=, or >=) between two observables, or between one observable and a constant. For example `A < 5` , or `A >= B`. Note that < and <= are equivalent unless the `min` keyword is used (see *Weight*).

3.4.2 Enforcement

Four keywords are available to specify when the inequality is enforced.

- `always` - Enforce the inequality at all time points during the simulation.

  `A < 5 always`

- `once` - Require that the inequality be true at at least one time point during the simulation. `A < 5 once`
- `at` - Enforce the inequality at one specific time point. This could be a constant time point:

  `A < 5 at 6` or equivalently, `A < 5 at time=6`

  It is also possible to specify the time point in terms of another observable.

  `A < 5 at B=6` - Enforce the inequality at the first time point such that B=6 (more exactly, the first time such that B crosses the value of 6 between two consecutive time steps)

  Using similar syntax, we can specify that the constraint is enforced at every time B=6, not just the first, using the `everytime` keyword

  `A < 5 at B=6 everytime`

  The `first` keyword says that the constraint should only (this is the default behavior, so this keyword is optional)

  `A<5 at B=6 first`

If the specified condition (B=6 in the example) is never met, then the constraint is not applied. It is often useful to add a second constraint to ensure that an "at" constraint is enforced. In this example, assuming the initial value of B is below 6, we could add the constraint `B>=6 once`

- `between` - Enforce the inequality at all times between the two specified time points. The time points may be specified in the same format as with the at keyword above, and should be separated by a comma.

  `A < 5 between 7, B=6` would enforce the inequality from time=7 to the first time after time=7 such that B=6.

  If the first condition (time=7 in the example) is never met, then the constraint is never enforced. If the second condition (B=6 in the example) is never met, then the constraint is enforced from the start time until the end of the simulation.

3.4.3 Weight

The weight clause consists of the `weight` keyword followed by a number. This number is multiplied by the extent of constraint violation to give the value to be added to the objective function. For example:

```
A < 5 at 6 weight 2
```

If the inequality A < 5 is not satisfied at time 6, then a penalty of 2*(A-5) is added to the objective function.

The `min` keyword indicates the minimum possible penalty to apply if the constraint is violated. This minimum is still multiplied by the constraint weight.

```
A < 5 at 6 weight 2 min 4
```

If the inequality A < 5 is not satisfied at time 6, the penalty is $2 * \max((A-5), 4)$. Since we used the strict < operator, the minimum penalty of 8 is applied even if A=5 at time 6.

In some unusual cases, it is desirable to use a different observable for calculating penalties than the one used in the inequality. For example, the variable in the inequality might be a discrete variable, and it would be desirable to calculate the penalty with a corresponding continuous variable. This substitution may be made using the `altpenalty` keyword in the weight clause, followed by the new inequality to use for calculating the penalty.

```
A < 5 at B=3 weight 10 altpenalty A2<4 min 1
```

This constraint would check if A<5 when B reaches 3. If A >= 5 at that time, it instead calculates the penalty based on the inequality A2<4 with a weight of 10: $10 * \max(0, A2 - 4)$. If the initial inequality is violated but the penalty inequality is satisfied, then the penalty is equal to the weight times the min value (10*1 in the example), or zero if no min was declared.

CHAPTER

FOUR

CONFIGURATION KEYS

The following sections give all possible configuration keys that may be used in your .conf file to configure your fitting run.

4.1 Paths

`model = path/to/model1.bngl : path/to/data1.exp, path/to/model2.xml : path/to/data2.con`
Specifies the mapping between model files (.bngl or .xml) and .exp files (.exp or .con). If no experimental files are associated with a model write `none` instead of a file path. This key is required.

`bng_command = path/to/BNG2.pl` Path to BNG2.pl, including the BNG2.pl file name. This key is required if your fitting includes any .bngl files, unless the BioNetGen path is specified with the BNGPATH env variable. Default: Uses the BNGPATH env variable

`output_dir = dirname` Directory where we should save the output. Default: bnf_out

`mutant = basemodel name statement1 statement2: data1name.exp, data2name.exp`
Declares a model that does not have its own model file, but instead is defined based on another model `basemodel`. `name` is the name of the mutant model; this name is appended to the suffixes of the base model. I.e, if the base model has data files data1.exp and data2.exp, a corresponding mutant model with the name "m1" should use the files data1m1.exp and data2m1.exp. `statement1`, `statement2`, etc. specify how to change `basemodel` to make the mutant model. The statments have the format [variable][operator][value] ; for example `a__FREE=0` or `b__FREE*2`. Supported operators are `=`, `+`, `-`, `*`, `/`. Default: None

4.2 Required Algorithm Options

`fit_type = str` Which fitting algorithm to use. Options: `de` - *Differential Evolution*, `ade` - *Asynchronous Differential Evolution*, ss - *Scatter Search*, `pso` - *Particle Swarm Optimization*, `bmc` - *Bayesian Markov chain Monte Carlo*, `sim` - *Simplex* local search, `sa` - *Simulated Annealing*, `pt` - *Parallel tempering*. Default: de

`objfunc = str` Which *objective function* to use. Options: `chi_sq` - Chi Squared, `sos` - Sum of squares, `norm_sos` - Sum of squares, normalized by the value at each point, `ave_norm_sos` - Sum of squares, normalized by the average value of the variable. Default: chi_sq

`population_size = int` How many parameter sets to maintain in the algorithm's population. See algorithm descriptions for more information. This key is required.

`max_iterations = int` Maximum number of iterations. This key is required.

4.3 Parameter Specification

uniform_var = name__FREE min max A uniformly distributed variable with bounds [min, max]

normal_var = name__FREE mu sigma A normal-distributed variable in regular space with mean mu, std dev sigma. A box constraint to keep ≥ 0 is also assumed.

loguniform_var = name__FREE min max A log-uniform distributed variable with bounds [min, max]. Bounds should be in regular space, eg [0.01, 100]

lognormal_var = name__FREE mu sigma A log-normal distributed variable with mean mu, std dev sigma. mu and sigma should be given in log base 10 space.

The following keys are to be used only with the *simplex* algorithm. Simplex should not use any of the other parameter specifications. If you are using another algorithm with the flag refine, you still should not use these, you must set step size with simplex_step or simplex_log_step.

var = name__FREE init step A variable that starts at init with an initial step size step. step is optional; defaults to simplex_step

logvar = name__FREE init step A variable that starts at init with an initial step size step, that moves in log space. init and step should be given in log base 10 space. step is optional; defaults to simplex_log_step.

4.4 Parallel Computing

parallel_count = int For a local (non-cluster) fitting run, how many jobs to run in parallel. Default: Use all available cores.

cluster_type = str Type of cluster used for running the fit. This key may be omitted, and instead specified on the command line with the -t flag. Currently suports slurm or none. Will support torque and pbs in the future. Default: None (local fitting run).

scheduler_node = str Manually set node used for creating the distributed Client – takes a string identifying a machine on a network. If running on a cluster with SLURM, it is recommended to use *automatic configuration* with the flag -t slurm instead of using this key. Default: None

worker_nodes = str1 str2 str3 Manually set nodes used for computation - takes one or more strings separated by whitespace identifying machines on a network. If running on a cluster with SLURM, it is recommended to use *automatic configuration* with the flag -t slurm instead of using this key. Default: None

4.5 General Options

4.5.1 Output Options

delete_old_files = int If 1, delete simulation folders immediately after they complete. If 2, delete both old simulation folders and old sorted_params.txt result files. If 0, do not delete any files (warning, could consume a large amount of disk space). Default: 1

num_to_output = int The maximum number of PSets to write when writing the trajectory. Default: 5000

output_every = int Write the Trajectory to file every x iterations. Default: 20

verbosity = int Specifies the amount of information output to the terminal. 0 - Quiet; user prompts and errors only. 1 - Normal; Warnings and concise progress updates. 2 - Verbose; Information and detailed progress updates. Default: 1

4.5.2 Algorithm Options

bootstrap = int If assigned a positive value, estimate confidence intervals through a bootstrapping procedure. The assigned integer is the number of bootstrap replicates to perform. Default: 0 (no bootstrapping)

bootstrap_max_obj = float The maximum value of a fitting run's objective function to be considered valid in the bootstrapping procedure. If a fit ends with a larger objective value, it is discarded. Default: None

constraint_scale = float Scale all weights in all constraint files by this multiplicative factor. For convenience only: The same thing could be achieved by editing constraint files, but this option is useful for tuning the relative contributions of quantitative and qualitative data. Default: 1 (no scaling)

ind_var_rounding = int If 1, make sure every exp row is used by rounding it to the nearest available value of the independent variable in the simulation data. (Be careful with this! Usually, it is better to set up your simulation so that all experimental points are hit exactly) Default: 0

initialization = str How to initialize parameters. `rand` - initialize params randomly according to the distributions. `lh` - For `random_vars` and `loguniform_vars`, initialize with a latin hypercube distribution, to more uniformly cover the search space.

local_objective_eval = int If 1, evaluate the objective function locally, instead of parallelizing this calculation on the workers. This option is automatically enabled when using the `smoothing` feature. Default: 0 (unless smoothing is enabled)

min_objective = float Stop fitting if an objective function lower than this value is reached. Default: None; always run for the maximum iterations

normalization = type;normalization = type : d1.exp, d2.exp;normalization = type: (d1.exp Indicates that simulation data must be normalized in order to compare with exp files. Choices for `type` are: `init` - normalize to the initial value, `peak` - normalize to the maximum value, `zero` - normalize such that each column has a mean of 0 and a standard deviation of 1, `unit` - Scales data so that the range of values is between (min-init)/(max-init) and 1 (if the maximum value is 0 (i.e. max == init), then the data is scaled by the minimum value after subtracting the initial value so that the range of values is between 0 and -1). If only the type is specified, the normalization is applied to all exp files. If one or more exp files included, it applies to only those exp files. Additionally, you may enclose an exp file in parentheses, and specify which columns of that exp file get normalized, as in `(data1.exp: 1,3-5)` or `(data1.exp: var1,var2)` Multiple lines with this key can be used. Default: No normalization

refine = int If 1, after fitting is completed, refine the best fit parameter set by a local search with the simplex algorithm. Default: 0

smoothing = int Number of replicate runs to average together for each parameter set (useful for stochastic simulations). Default: 1

wall_time_gen = int Maximum time (in seconds) to wait to generate the network for a BNGL model. Will cause the program to exit if exceeded. Default: 3600

wall_time_sim = int Maximum time (in seconds) to wait for a simulation to finish. Exceeding this results in an infinite objective function value. Caution: For SBML models, using this option has an overhead cost, so don't use it unless needed. Default: 3600

4.6 Algorithm-specific Options

4.6.1 Simplex

These settings for the *simplex* algorithm may also be used when running other algorithms with `refine = 1`.

simplex_step = float In initialization, we perturb each parameter by this step size. If you specify a step size for a specific variable via `var` or `logvar`, it overrides this setting. Default: 1

simplex_log_step = float Equivalent of `simplex_step`, for variables that move in log space. Default: `simplex_step`

simplex_reflection = float When we reflect a point through the centroid, what is the ratio of dilation on the other side? Default: 1.0

simplex_expansion = float If the reflected point was the global minimum, how far do we keep moving in that direction? (as a ratio to the initial distance to centroid) Default: 1.0

simplex_contraction = float If the reflected point was not an improvement, we retry at what distance from the centroid? (as a ratio of the initial distance to centroid) Default: 0.5

simplex_shrink = float If a whole iteration was unproductive, shrink the simplex by setting simplex point $s[i]$ to $x * s[0] + (1 - x) * s[i]$, where x is the value of this key and $s[0]$ is the best point in the simplex. Default: 0.5

simplex_max_iterations = int If specified, overrides the `max_iterations` setting. Useful if you are using the `refine` flag and want `max_iterations` to refer to your main algorithm.

simplex_stop_tol = float Stop the algorithm if all parameters have converged to within this value (specifically, if all reflections in an iteration move the parameter by less than this value) Default: 0 (don't use this criterion)

4.6.2 Differential Evolution

PyBNF offers two versions of *differential evoltution*: synchronous differential evolution (`fit_type = de`) and asynchronous differential evolution (`fit_type = ade`). Both versions may be configured with the follwing keys.

mutation_rate = float When generating a new individual, mutate each parameter with this probability. Default: 0.5

mutation_factor = float When mutating a parameter x, change it by mutation_factor*(PS1[x] - PS2[x]) where PS1 and PS2 are random other PSets in the population. Default: 1.0

stop_tolerance = float Stop the run if within the current popluation $max(objective)/min(objective) < 1+e$, where e = this value. This criterion triggers when the entire population has converged to roughly the same objective. Default: 0.002

de_strategy = str Specifies how new parameter sets are chosen. Options are: `rand1`, `rand2`, `best1`, `best2`, `all1`, `all2`. The parameter set we mutate is: 'rand' - a random one, 'best' - the one with the lowest objective value, 'all' - the one we are proposing to replace (so all psets are mutated once per iteration). The amount of mutation is based on: '1' - 1 pair of other parameter sets $(p_1 - p_2)$, '2' - 2 pairs of other parameter sets $(p1 - p2 + p3 - p4)$. Default: rand1

The following options are only available with `fit_type = de`, and serve to make the algorithm more asynchronous. If used, these options enable *island-based* differential evolution, which is asynchronous in that each island can independently proceed to the next iteration.

islands = int Number of separate populations to evolve. Default: 1

migrate_every = int After this number of generations, migrate some individuals between islands. Default: 20 (but Inf if `islands = 1`)

num_to_migrate = int How many individuals to migrate off of each island during migration. Default: 3

4.6.3 Scatter Search

init_size = int Number of PSets to test to generate the initial population. Default: 10 * number of variables

local_min_limit = int If a point is stuck for this many iterations without improvement, it is assumed to be a local min and replaced with a random parameter set. Default: 5

reserve_size = int Scatter Search maintains a latin-hypercube-distributed "reserve" of parameter sets. When it needs to pick a random new parameter set, it takes one from the reserve, so it's not similar to a previous random choice. The initial size of the reserve is this value. If the reserve becomes empty, we revert to truly random pset choices. Default: max_iterations

4.6.4 Particle Swarm

cognitive = float Acceleration toward a particle's own best fit

social = float Acceleration toward the global best fit

particle_weight = float Inertia weight of particle. A value less than 1 can be thought of as friction that contiuously decelerates the particle. Default: 1

v_stop = float Stop the algorithm if the speeds of all parameters in all particles are less than this value. Default: 0 (don't use this criterion)

A variant of particle swarm that adaptively changes the `particle_weight` over the course of the fitting run is configured with the following parameters. See the *algorithm documentation* for more information.

particle_weight_final The final particle weight after the adaptive changing. Default: the value of `particle_weight`, effectively disabling this feature.

adaptive_n_max After this many "unproductive" iterations, we have moved halfway from the initial weight to the final weight. Default: 30

adaptive_n_stop Afer this many "unproductive" iterations, stop the fitting run. Default: Inf

adaptive_abs_tol Parameter for checking if an iteration was "unproductive" Default: 0

adaptive_rel_tol Parameter for checking if an iteration was "unproductive" Default: 0

4.6.5 Bayesian Algorithms (bmc, pt, sa)

In the family of Bayesian algoritms with Metropolis sampling, PyBNF includes *MCMC* (`fit_type = bmc`), *Parallel Tempering* (`fit_type = pt`), *Simulated Annealing* (`fit_type = sa`). These algorithms have many configuration keys in common, as described below.

For all Bayesian algorithms

step_size = float When proposing a Monte Carlo step, the step in n-dimensional parameter space has this length. Default: 0.2

beta = int ; beta = b1 b2 b3 Sets the initial beta (1/temperature). A smaller beta corresponds to a more broad exploration of parameter space. If a single value is provided, that beta is used for all replicates. If multiple values are provided, an equal number of replicates uses each value.

For `mcmc`, should be set to 1 (the default) to get the true probability distribution.

For `pt`, should specify multiple values: the number of values should equal `population_size`/`reps_per_beta`. Or you may instead use the `beta_range` key. Only the

largest beta value in the list will constribute to statistical samples, and to get the true probability distribution, this maximum value should be 1.

For `sa`, should typically be set to a single, small value which will increase over the course of the fitting run.

For all Bayesian algorithms except `sa`

`sample_every = int` Every x iterations, save the current PSet into the sampled population. Default: 100

`burn_in = int` Don't sample for this many iterations at the start, to let the system equilibrate. Default: 10000

`output_hist_every = int` Every x samples (i.e every x*sample_every iterations), save a historgram file for each variable, and the credible interval files, based on what has been sampled so far. Regardless, we also output these files at the end of the run. Default: 100

`hist_bins = int` Number of bins used when writing the histogram files. Default: 10

`credible_intervals = n1 n2 n3` Specify one or more numbers here. For each n, the algorithm will save a file giving bounds for each variable such that in n% of the samples the variable lies within the bounds. Default: 68 95

For Simulated Annealing

`beta_max = float` Stop the algorithm if all replicates reach this beta (1/temperature) value. Default: Inf (don't use this stop criterion)

`cooling = float` Each time a move to a higher energy state is accepted, increase beta (1/temperature) by this value. Default: 0.01

For Parallel Tempering

`exchange_every = int` Every x iterations, perform replica exchange, swapping replicas that are adjacent in temperature with a statistically correct probability

`reps_per_beta = int` How many identical replicas to run at each temperature. Must be a divisor of population_size

`beta_range=min max` As an alternative to setting `beta`, the range of values of beta to use. The replicates will use population_size/reps_per_beta evenly spaced beta values within this range. Only the replicas at the max beta value will be sampled. For the true probability distribution, max should be 1.

CHAPTER

FIVE

FITTING ALGORITHMS

5.1 Summary of Available Algorithms

Algorithm	Class	Parallelization	Applications
Differential Evolution	Population-based	Synchronous or Asynchronous	General-purpose parameter fitting; Optimal use of a very large number of processors (if Asynchronous)
Scatter Search	Population-based	Synchronous	Difficult problems with high dimensions or many local minima
Particle Swarm	Population-based	Asynchronous	Problem-specific applications
Markov Chain Monte Carlo	Metropolis sampling	Independent Markov Chains	Finding probability distributions of parameters
Simulated Annealing	Metropolis sampling	Independent Markov Chains	Problem-specific applications
Parallel Tempering	Metropolis sampling	Synchronized Markov Chains	Finding probability distributions in challenging probablity landscapes
Simplex	Local search	Synchronous	Local optimization, or refinement of a result from another algorithm.

5.2 General implementation features for all algorithms

All algorithms in PyBNF keep track of a list of parameter sets (a "population"), and over the course of the simulation, submit new parameter sets to run on the simulator. Algorithms periodically output the file `sorted_params.txt` containing the best parameter sets found so far, and the corresponding objective function values.

5.2.1 Initialization

The initial population of parameter sets is generated based on the keys specified for each free parameter: `uniform_var`, `loguniform_var`, `normal_var` or `lognormal_var`. The value of the parameter in each new random parameter set is drawn from the specified probability distribution.

The `latin_hypercube` option for initialization is enabled by default. This option only affects initialization of `uniform_vars` and `loguniform_vars`. When enabled, instead of drawing an independent random value for each starting parameter set, the starting parameter sets are generated with Latin hypercube sampling, which ensures a roughly even distribution of the parameter sets throughout the search space.

5.2.2 Objective functions

All algorithms use an objective function to evaluate the quality of fit for each parameter set. The objective function is set with th

- Chi squared (`obj_func = chi_sq`): $f(y,a) = \sum_i \frac{(y_i-a_i)^2}{\sigma_i^2}$. σ_i is the standard deviation of point y_i, and must be specified in the *exp file*.
- Sum of squares (`obj_func = sos`): $f(y,a) = \sum_i (y_i - a_i)^2$
- Normalized sum of squares (`obj_func = norm_sos`): $f(y,a) = \sum_i \frac{(y_i-a_i)^2}{y_i}$
- Average-normalized sum of squares (`obj_func = ave_norm_sos`): $f(y,a) = \sum_i \frac{(y_i-a_i)^2}{\bar{y}}$, where $\bar{y}$ is the average of the entire data column y.

If you include any *constraints* in your fit, the constraints add extra terms to the objective function.

5.2.3 Changing parameter values

All algorithms perform changes to parameter values as the fitting proceeds. The way these changes are calculated depends on the type of parameter.

`loguniform_vars` and `lognormal_vars` are moved in logarithmic space (base 10) throughout the entire fitting run.

`uniform_vars` and `loguniform_vars` avoid moving outside the defined initialization range. If a move is attempted that would take the parameter outside the bounds, the parameter value is reflected over the boundary, back within bounds. This feature can be disabled by appending `U` to the end of the variable definition (e.g. `uniform_var = x__FREE 10 30 U`)

5.3 Differential Evolution

5.3.1 Algorithm

A population of individuals (points in parameter space) are iteratively evaluated with an objective function. Parent individuals from the current iteration are selected to form new individuals in the next iteration. The new individual's parameters are derived by combining parameters from the parents. New individuals are accepted into the population if they have an objective value lower than that of a member of the current population.

5.3.2 Parallelization

Three versions of differential evolution are available: All run in parallel, but they differ in their level of synchronicity.

Asynchronous differential evolution (`fit_type = ade`) never allows processors to sit idle. One new simulation is started every time a simulation completes. This version is the best choice when a large number of processors are available.

Synchronous differential evolution (`fit_type = de`) consists of discrete iterations. In each iteration, n simulations are run in parallel, but all must complete before moving on to the next iteration.

Island-based differential evolution *[Penas2015]* is partially asynchronous algorithm. To use this version, set `fit_type = de` and set a value greater than 1 for the `islands` key. In this version, the current population consists of m islands. Each island is able to move on to the next iteration even if other islands are still in progress. If m is set to the number of available processors, then processors will never sit idle. Note however that this might still underperform compared to the synchronous algorithm run on the same number of processors.

5.3.3 Implementation details

We maintain a list of `population_size` current parameter sets, and in each iteration, `population_size` new parameter sets are proposed. The method to propose a new parameter set is specified by the config key `de_strategy`. The default setting `rand1` works best for most problems, and runs as follows: We choose 3 random parameter sets p1, p2, and p3 in the current population. For each free parameter P, the new parameter set is assigned the value p1[P] + `mutation_factor` * (p2[P]-p3[P]) with probability `mutation_rate`, or p1[P] with probability 1 - `mutation_rate`. The new parameter set replaces the parameter set with the same index in the current population if it has a lower objective value.

With `de_strategy` of `best1` or `best2`, we force the above p1 to be the parameter set with the lowest objective value. With `de_strategy` of `all1` or `all2`, we force p1 to be the parameter set at the same index we are proposing to replace. The `best` strategy results in fast convergence to what is likely only a local optimum. The `all` strategy converges more slowly, and prevents the entire population from converging to the same value. However, there is still a risk of each member of the population becoming stuck in its own local minimum. For the `de_strategy``s ending in ``2`, we instead choose a total of 5 parameter sets, p1 through p5, and set the new parameter value as p1[P] + `mutation_factor` * (p2[P]-p3[P] + p4[P]-p5[P])

Asynchronous version

The asynchronous version of the algorithm is identical to the sychronous algorithm, except that whenever a simulation completes, a new parameter set is immediately proposed based on the current population. Therefore, the random parameter sets p1, p2, and p3 might come from different iteration numbers.

Island-based version

In the island-based version of the algorithm *[Penas2015]*, the population is divided into `num_islands` islands, which each follow the above update procedure independently. Every `migrate_every` iterations, a migration step occurs in which `num_to_migrate` individuals from each island are transferred randomly to others (according to a random permutation of the islands, keeping the number of individuals on each island constant). The migration step does not require synchronization of the islands; it is performed when the last island reaches the appropriate iteration number, regardless of whether other islands are already further along.

5.3.4 Applications

In our experience, differential evolution tends to be the best general-purpose algorithm, and we suggest it as a starting point for a new fitting problem if you are unsure which algorithm to choose.

The asynchronous version becomes advantageous over the other available algorithms when many processors are available (>100), and when the runtime per simulation can vary greatly depending on the parameter set (such as in some SSA and NFSim runs). In these cases, the asynchronicity of the particle swarm allows you to take full advantage of all available processors at all times.

5.4 Scatter Search

5.4.1 Algorithm

Scatter Search *[Glover2000]* functions similarly to differential evolution, but maintains a smaller current population than the number of available processors. In each iteration, every possible pair of individuals are combined to propose a new individual.

5.4.2 Parallelization

In a scatter search run of population size n, each iteration requires n*(n-1) independent simulations that can all be run in parallel. Scatter search requires synchronization at the end of each iteration, waiting for all simulations to complete before moving to the next iteration.

5.4.3 Implementation details

The PyBNF implementation follows the outline presented in the introduction of *[Penas2017]* and uses the recombination method described in *[Egea2009]*.

We maintain a reference set of `population_size` individuals, recommended to be a small number (~ 12-18). Each newly proposed parameter set is based on a "parent" parameter set and a "helper" parameter set, both from the current reference set. In each iteration, we consider all possible parent-helper combinations, for a total of n*(n-1) parameter sets. The new parameter set depends on the rank of the parent and helper (call them p_i and h_i) when the reference set is sorted from best to worst.

Then we apply a series of formulas to choose the next parameter value.

Let α = -1 if $h_i > p_i$ or 1 if $p_i < h_i$, let $\beta = (|h_i - p_i| - 1)/(n-2)$, let $d = \text{helper}[P] - \text{parent}[P]$ for some parameter P.

Then the in the new parameter set, $P = \text{parent}[P] + \text{rand\_uniform}(-d * (1 + \alpha * \beta), d * (1 - \alpha * \beta))$

Intuitively what we do here is perturb P on the order of d (which acts as a measure of the variability of P in the population). If the parent is better than the helper, we keep P closer to the parent, and if the helper is better, we shift it closer to the helper.

The proposed new parameter set is accepted if it achieves a lower objective value than its parent.

If a parent goes `local_min_limit` iterations without being replaced by a new parameter set, it is assumed to be stuck in a local minimum, and is replaced with a new random parameter set. The random parameter set is drawn from a "reserve queue", which is initialized at the start of the fitting run to contain `reserve_size` Latin hypercube distributed samples. The reserve queue ensures that each time we take a new random parameter set, we are sampling a part of parameter space that we have not sampled previously.

5.4.4 Applications

We find scatter search is also a good general-purpose fitting algorithm. It performs especially well on fitting problems that are difficult due to a search space that is high dimensional or contains many local minima.

5.5 Particle Swarm

5.5.1 Algorithm

In particle swarm optimization, each parameter set is represented by a particle moving through parameter space at some velocity. The acceleration of each particle is set in a way that moves it toward more favorable areas of parameter space: the acceleration has contributions pointing toward both the best parameter set seen so far by the individual particle, and the global best parameter set seen by any particle in the population.

5.5.2 Parallelization

Particle swarm optimization in PyBNF is an asynchronous, parallel algorithm. As soon as one simulation completes, that particle can calculate its next parameter set and begin a new simulation. Processors will never remain idle.

5.5.3 Implementation details

The PyBNF implementation is based on the description in *[Moraes2015]*. Each particle keeps track of its current position, velocity, and the best parameter set it has seen during the run.

After each simulation completes, the velocity of the particle is updated according to the formula $v_{i+1} = w * v_i + c_1 * u_1 * (x_i - x_{\text{min}}) + c_2 * u_2 * (x_i - x_{\text{globalmin}})$. The constants in the above formula may be set with config keys: w is `particle_weight`, c_1 is `cognitive`, and c_2 is `social`. x_i is the current particle position, v_i is the current velocity, v_{i+1} is the updated velocity, x_{min} is the best parameter set this particle has seen, and $x_{\text{globalmin}}$ is the best parameter set any particle has seen. u_1 and u_2 are uniform random numbers in the range [0,1]. Following the velocity update, the position of the particle is updated by adding its current velocity.

We apply a special treatment if a `uniform_var` or `loguniform_var` moves outside of the specified box constraints. As with other algorithms, the particle position is reflected back inside the boundaries. In addition, the component of the velocity corresponding to the parameter that moved out of bounds is set to zero, to prevent the particle from immediately crossing the same boundary again. An optional feature (discussed in *[Moraes2015]*) allows the particle weight *w* to vary over the course of the simulation. In the original algorithm descirption, *w* was called "inertia weight", but when *w* takes a value less than 1, it can be thought of as friction - a force that decelerates particles regardless of the objective function evaluations. The idea is to reduce *w* (increase friction) over the course of the fitting run, to make the particles come to a stop at a good objective value by the end of the run.

When using the adaptive friction feature, *w* starts at `particle_weight`, and approaches `particle_weight_final` by the end of the simulation. The value of *w* changes based on how many iterations we deem "unproductive" according to the following criterion: An iteration is unproductive if the global best objective function obj_min changes by less than `adaptive_abs_tol` + `adaptive_rel_tol` * obj_min, where `adaptive_abs_tol` and `adaptive_rel_tol` can be set in the config. Then, we keep track of N, the total number of unproductive iterations so far. At each iteration we set *w* = `particle_weight` + (`particle_weight_final` - `particle_weight`) * N / (N + `adaptive_n_max`). As can be seen in the above formula, the config key `adaptive_n_max` sets the number of unproductive iterations it takes to reach halfway between `particle_weight` and `particle_weight_final`.

5.5.4 Applications

We have not found any problems for which particle swarm optimization is better than the other available algorithms, but provide the functionality with the hope that it proves useful for some specific problems.

Like asynchronous differential evolution, the algorithm is strongest in cases where many processors (>100) are available because the asynchronicity allows it to take advantage of all processors at all times.

5.6 Markov Chain Monte Carlo

5.6.1 Algorithm

Markov chain Monte Carlo is a Bayesian method in which points in parameter space are sampled with a frequency proportional to the probability that the parameter set is correct given the data. The result is a probability distribution over parameter space that expresses the likelihood of each possible parameter set. With this algorithm, we obtain not just a point estimate of the best fit, but a means to quantify the uncertainty in each parameter value.

When running Markov chain Monte Carlo, PyBNF outputs additional files containing this probability distribution information. The files in `Results/Histograms/` give histograms of the marginal probability distributions for each free parameter. The files `credible##.txt` (e.g., `credible95.txt`) use the marginal histogram for each parameter to calculate a *credible interval* - an interval in which the parameter value is expected to fall with the specified probability (e.g. 95%). Finally, `samples.txt` contains all parameter sets sampled over the course of the fitting run, allowing the user to perform further custom analysis on the sampled probability distribution.

5.6.2 Parallelization

Markov chain Monte Carlo is not an inherently parallel algorithm. In the Markov chain, we need to know the current state before proposing the next one. However, PyBNF supports running several independent Markov chains by specifying the number of chains with the `population_size` key. All samples from all parallel chains are pooled to obtain a better estimate of the final posterior probability distribution.

Note that each chain must independently go through the burn-in period, but after the burn-in, your rate of sampling will be improved proportional to the number of parallel chains in your run.

5.6.3 Implementation details

Our implementation is described in *[Kozer2013]*. We start at a random point in parameter space, and make a step of size `step_size` to move to a new location in parameter space. We take the value of the objective function to be the probability of the data given the parameter set (the *likelihood* in Bayesian statistics). We assume a prior distribution based on the parameter definitions in the config file – a uniform, loguniform, normal, or lognormal distribution, depending on the config key used. Note: If a uniform or loguniform prior is used, the prior does not affect the result other than to confine the distribution within the specified range. If a normal or lognormal prior is used, the prior does affect the probability of accepting each proposed move, and therefore the choice of prior affects the final sampled probability distribution.

The Bayesian *posterior* distribution – the probability of the parameters given the data – is given by the product of the above likelihood and prior. We use the value of the posterior to determine whether to accept the proposed move.

Moves are accepted according to the Metropolis criterion. If a move increases the value of the posterior, it is always accepted. If it decreases the value of the posterior, it is accepted with probability $e^{-\beta \Delta F}$, where ΔF is the change in the posterior, and β represents the inverse "temperature" at which the Metropolis sampling occurs. To generate the true posterior distribution, β should be set to 1. The sampled distribution becomes more broad with smaller β and more narrow with a larger β.

5.6.4 Applications

Markov chain Monte Carlo is the simplest method available in PyBNF to generate a probability distribution in parameter space.

5.7 Simulated Annealing

5.7.1 Algorithm

Simulated annealing is another Markov chain-based algorithm, but our goal is not to find a full probability distribution, just find the optimal parameter set. To do so, we start the Markov chain at a high temperature, where unfavorable moves are accepted frequently, and gradually reduce the temperature over the course of the simulation. The idea is that we will explore parameter space broadly at the start of the fitting run, and become more confined to the optimal region of parameter space as the run proceeds.

5.7.2 Parallelization

Simulated annealing is not an inherently parallel algorithm. The trajectory is a Markov chain in which we need to know the current state before proposing the next one. However, PyBNF supports running several independent simulated annealing chains in parallel. By running many chains simulatenously, we have a better chance that one of the chains achieves a good final fit.

5.7.3 Implementation details

The Markov chain is implemented in the same way as described above for the Markov chain Monte Carlo algorithm, incorporating both the objective function value and the prior distribution to calculate the posterior probability density.

The difference is in the Metropolis criterion for acceptance of a proposed move. Here, a move that decreases the value of the posterior is accepted with probability $e^{-\beta \Delta F}$, where β decreases over the course of the fitting run.

5.7.4 Applications

We have not found any problems for which simulated annealing is better than the other available algorithms, but provide the functionality with the hope that it proves useful for some specific problems.

5.8 Parallel Tempering

5.8.1 Algorithm

Parallel tempering is a more sophisticated version of Markov chain Monte Carlo. We run several Markov chains in parallel at different temperatures. At specified iterations during the run, there is an opportunity to exchange replicates between the different temperatures. Only the samples recorded at the lowest temperature count towards our final probability distribution, but the presence of the higher temperature replicates makes it easier to escape local minima and explore the full parameter space.

When running parallel tempering, PyBNF outputs files containing probability distribution information, the same as with Markov chain Monte Carlo.

5.8.2 Parallelization

The replicates are run in parallel. Synchronization is required at every iteration in which we attempt replica exchange.

5.8.3 Implementation details

The PyBNF implementation is based on the description in *[Gupta2018]*. Markov chains are run by the same method as in Markov chain Monte Carlo, except that the value of β in the acceptance probability $e^{-\beta \Delta F}$ varies between replicas.

Every `exchange_every` iterations, we attempt replica exchange. We propose moves that consist of swapping two replicas between adjacent temperatures. Moves are accepted with probability $\min(1, e^{\Delta\beta \Delta F})$ where $\Delta\beta$ is the change in β = 1/Temperature, and ΔF is the difference in the objective values of the replicas. In other words, moves that transfer a lower-objective replica to a lower temperature (higher β) are always accepted, and those that transfer a higher-objective replica to a lower temperature are accepted with a Metropolis-like probability based on the extent of objective difference.

The list of βs used is customizable with the `beta` or `beta_range` key. The number of replicas per temperature is also customizable. To maintain detailed balance, it is required that each temperature contains the same number of replicas.

5.8.4 Applications

Like ordinary Markov chain Monte Carlo, the goal of parallel tempering is to provide a distribution of possible parameter values rather than a single point estimate.

Compared to ordinary Markov chain Monte Carlo, parallel tempering offers a trade-off: Parallel tempering generates fewer samples per unit CPU time (because most of the processors run higher temperature simulations that don't sample the distribution of interest), but traverses parameter space more efficiently, making each sample more valuable. The decision between parallel tempering and Markov chain Monte Carlo therefore depends on the nature of your parameter space: parallel tempering is expected to perform better when the space is complex, with many local minima that make it challenging to explore.

5.9 Simplex

5.9.1 Algorithm

Simplex is a local search algorithm that operates solely on objective evaluations at single points (i.e. it does not require calculation of gradients). The algorithm maintains a set on N+1 points in N-dimensional parameter space, which are thought of as defining an N-dimensional solid called a *simplex*. Individual points may be reflected through the lower-dimensional solid defined by the other N points, to obtain a local improvement in objective function value. The simplex algorithm has been nicknamed the "amoeba" algorithm because the simplex crawls through parameter space similar to an amoeba, extending protrusions in favorable directions.

5.9.2 Parallelization

The PyBNF Simplex implementation is parallel and synchronous. Synchronization is required at the end of every iteration. Parallelization is achieved by simultaneously evaluating a subset of the N+1 points in the simplex. Therefore, this parallelization can take advantage of at most N+1 processors, where N is the number of free parameters.

5.9.3 Implementation details

PyBNF implements the parallelized Simplex algorithm described in *[Lee2007]*.

The initial simplex consists of N+1 points chosen deterministically based on the specified step size (set with the `simplex_step` and `simplex_log_step` keys, or for individual parameters with the `var` and `log_var` keys). One point of the simplex is the specified starting point for the search. The other N points are obtained by adding the step size to one parameter, and leaving the other N-1 parameters at the starting values.

Each iteration, we operate on the k worst points in the simplex, where k is the number of available processors (`parallel_count`). For each point P, we consider the hyperplane defined by the other N points in the simplex (blue line). Let d be the distance from P to the hyperplane. We evaluate point P_1 obtained by reflecting P through the hyperplane, to a distance of d * `simplex_reflect` on the other side. Depending on the resulting objective value, we try another point in the second phase of the iteration. Three cases are possible.

1. The new point is better than the current global minimum: We try a second point continuing in the same direction for a distance of d * `simplex_expansion` away from the hyperplane ($P_{2,1}$).

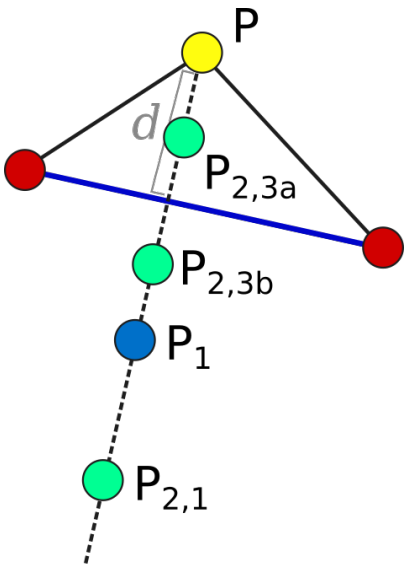

Fig. 5.1: Illustration of the simplex algorithm, modifying point P on a 3-point simplex in 2 dimensions

2. The new point is worse than the global minimum, but better than the next worst point in the simplex: We don't try a second point.
3. The new point is worse than the next worst point in the simplex: We try a second point moving closer to the hyperplane. If P was better than P_1, we try a point a distance of d * `simplex_contraction` from the hyperplane in the direction of P ($P_{2,3a}$). If P_1 was better than P, we instead try the same distance from the hyperplane in the direction of P_1 ($P_{2,3b}$).

In all cases, P in the simplex is set to the best choice among P, P_1, or whichever second point we tried.

If in a given iteration, all k points resulted in Case 3 and did not update to $P_{2,3a}$ or $P_{2,3b}$, the iteration did not effectively change the state of the simplex. Then, we contract the simplex towards the best point: We set each point P to `simplex_contract` * P0 + (1 - `simplex_contract`) * P, where P0 is the best point in the simplex.

5.9.4 Applications

Local optimization with the simplex algorithm is useful for improving on an already known good solution. In PyBNF, the most common application is to apply the simplex algorithm to the best-fit result obtained from one of the other algorithms. You can automatically refine your final result with the simplex algorithm by setting the `refine` key to 1, and setting simplex config keys in addition to the config for your main algorithm.

It is also possible to run the Simplex algorithm on its own, using a custom starting point. In this case, you should use the `var` and `log_var` keys to specify your known starting point.

5.10 References

CHAPTER

SIX

ALGORITHM DEVELOPMENT

PyBNF was designed with extensibility in mind. As a result, all of the algorithms implemented here subclass the Algorithm class found in *PyBNF algorithms (pybnf.algorithms)*.

6.1 Implementation

A new algorithm can be written by creating a class that subclasses the Algorithm class:

```
class NewAlgorithm(Algorithm):
    def __init__(self, config, **kwargs):
        super(NewAlgorithm, self).__init__(config)
        # Other setup that may involve additional arguments
        self.current_iter_results = []
        self.ready_for_next_iter = False

    def generate_random_psets(self):
        # User defined support function that
        # generates a list of PSet instances
        ...
```

The new algorithm requires defining three methods, with the first being the `__init__` constructor method. This method will likely take a Configuration object as its first argument. The other two required methods that must be implemented are the `start_run` and `got_result` methods.

The `start_run` method will return a list of PSet instances that correspond to the first batch of parameter set evaluations.:

```
def start_run(self):
    return self.generate_random_psets()
```

The `got_result` method takes a Result instance as an argument and returns either a list of new PSet instances for another round of parameter set evaluations, or the string "STOP" to terminate the fitting run. Note that an empty list is valid if the algorithm requires synchronization (and thus must wait for all jobs in the current iteration to finish).:

```
def got_result(self, res):
    if self.satisfies_stop_condition(res):
        return "STOP"  # Terminates algorithm

    self.current_iter_results.append(res)
    if self.ready_for_next_iter:  # Synchronization check
        new_psets = []
        for r in self.current_iter_results:
            new_psets.append(self.generate_new_pset(r))
```

```
        return new_psets

    return []  # Waiting for synchronization
```

6.2 Adding configuration options

If the new algorithm requires user configuration via the configuration file, new options may be added to the `pybnf.parse` module. The configuration parser uses the `pyparsing` module and new grammars for parsing individual lines may be added to the `pybnf.parse.parse` function based on the key text. Default values for parameters may be added to the `Configuration` object via its `default_config` method in the `pybnf.config` module if desired. Other supporting configuration methods should also be added to the `Configuration` object if necessary.

6.3 Pull requests

To have new algorithms added into the PyBNF software suite, submit a pull request to the master branch at https://github.com/NAU-BioNetFit/PyBNF

CHAPTER

SEVEN

RUNNING ON A CLUSTER

PyBNF is designed to run on computing clusters that utilize a shared network filesystem. PyBNF comes with built-in support for clusters running Slurm. It may also be manually configured to run on clusters with other managers (Torque, PBS, etc.).

The Dask.distributed package, which is installed as a dependency of PyBNF, has a scheduler that we use for handling simulations in distributed computing environments (clusters).

While users can likely install PyBNF using `pip`'s `--user` flag, assistance from the cluster administrators may be helpful

7.1 SLURM

The user may run PyBNF interactively or as a batch job using the `salloc` or `sbatch` commands respectively. Note that the user must have set up their Python environment prior to running PyBNF on a cluster.

To tell PyBNF to use Slurm, pass "slurm" with the `-t` flag, i.e. `pybnf -t slurm`. It is also possible to instead specify the `cluster_type` key in the config file.

7.1.1 Interactive (quickstart)

Execute the `salloc -Nx` command where *x* is an integer denoting the number of nodes the user wishes to allocate

Log in to one of the nodes with the command `slogin`

Load appropriate Python environment

Initiate a PyBNF fitting run, including the flag `-t slurm`

7.1.2 Batch

Write a shell script specifying the desired nodes and their properties according to SLURM specifications. Be sure that your script includes loading the appropriate Python environment if this step is required for your cluster, and that your call to pybnf includes the flag `-t slurm`. For an example shell script, see examples/tcr/tcr_batch.sh.

Submit the batch job to the queueing system using the command `sbatch script.sh` where `script.sh` is the name of the shell script.

7.2 TORQUE/PBS

Not yet implemented. Please refer to Manual configuration below

7.3 Manual configuration

It is possible to run PyBNF on any cluster regardless of resource manager by simply telling PyBNF the names of the nodes it should run on.

Use manager-specific commands to allocate some number of nodes for your job, and find the names of those nodes. For example, in Torque: `qsub -I <options>` followed by `qstat -u <username>`.

Then set the keys `scheduler_node` and `worker_nodes` in your PyBNF config file. `scheduler_node` should be the name of one of the nodes allocated for your job, and `worker_nodes` should be the space-delimited names of all of your nodes (including the one set as `scheduler_node`).

PyBNF will then run this fitting job on the specified cluster nodes.

CHAPTER

EIGHT

EXAMPLES

PyBNF contains 17 example fitting jobs in the examples/ directory.

Each example directory contains all files required to run the example: the config file, model file(s), and data / constraint file(s). The config file paths are specified such that the examples should be run from the root PyBNF directory, i.e., to run the "demo" example, run `pybnf -c examples/demo/demo_bng.conf` from the root PyBNF directory. Results will be saved in a directory called "PyBNF-output" in the level above the root PyBNF directory. Examples with BioNetGen assume that you have set the `BNGPATH` environmental variable to point to your BioNetGen installation; if not, you should add the `bng_command` key to the config file to specify the location of your BioNetGen.

The examples are described below. For an index of which examples demonstrate which PyBNF features, refer to *Index of examples by attribute*

8.1 List of examples included in PyBNF

8.1.1 constraint_advanced

A demonstration of the features of the PyBNF constraint language. The model consists of simple fourth-order polynomial functions. The constraint files consist of constraints of various forms, showcasing the available capabilities for constraint handling in PyBNF. All constraints are consistent with a known ground truth for the model, so it should be possible to fit with a very low final objective value.

8.1.2 constraint_demo

A simple demonstration of a constrained fitting problem, in which we fit a parabola and a line to both quantitative and qualitative data. This is the same problem described in Fig. 1 of *[Mitra2018]*.

8.1.3 constraint_raf

A small, biologically relevant fitting problem that includes both constraints and quantitative data. The model describes the process by which Raf dimerizes and binds inhibitor. In certain parameter regimes, it is possible for the inhibitor to counterintuitively cause Raf activation, by promoting dimerization and increasing the concentration of the highly active species RIR. Two equilibrium constants, K3 and K5, are assumed to be unknown, and are fit using synthetic qualtitative and quantitative data. This is the same problem described in Fig. 2 of *[Mitra2018]*.

8.1.4 degranulation

A model that relates the initial events of IgE-FcεRI signaling to the degranulation response. The model is fit to experimental data from a microfluidic device that was used to measure mast cell degranulation in response to time courses of alternating stimulatory and non-stimulatory inputs. The data and model were originally published in *[Harmon2017]*.

In the original study, the model was analyzed by Bayesian MCMC to acquire probability distributions for each parameter. We provide config files to repeat this analysis in PyBNF, using both of our algorithms that calculate probability distributions: MCMC, and Parallel tempering. In both cases, the results from PyBNF are expected to match the results shown in Fig. S10 of *[Harmon2017]*. A large number of samples is required to obtain an acceptable distribution, so we recommend running on a cluster or powerful multi-core workstation. An example batch file to submit the job to a SLURM cluster is provided. For best performance, the config key `population_size` should be set to the number of available cores. .. Note: DREAM also provided, but it gives the wrong distribution.

8.1.5 demo

Fit a simple parabola implemented in either BNGL or SBML. Useful to validate that PyBNF and associated simulators are installed correctly.

8.1.6 egfr_bleaching

A model of EGFR fit using novel experimental data, presented for the first time with PyBNF. Ryan is going to explain this one.

8.1.7 egfr_benchmark

A benchmark rule-based model of EGFR signaling, originally published in *[Blinov2006]* and considered in *[Gupta2018]*. To create an example fitting problem, we generated synthetic data based on the published ground truth, and try to recover the ground truth parameters by fitting.

We used this benchmark problem to test and showcase all of the fitting algorithms available in PyBNF. The folder contains one config file for each of the available PyBNF algorithms. All config files are set the same number of total simulations are run (note that this comparison does not take into account the avantage of asynchronicity in PSO and ADE).

8.1.8 egfr_nf

A model of EGFR signaling described in *[Kozer2013]*. Simulations are performed in NFsim.

This problem was considered as example2 in the original BioNetFit (*[Thomas2016]*).

8.1.9 egfr_ode

A model of EGFR signaling described in *[Kozer2013]*. Simulations are performed by numerical integration of ODEs in BioNetGen.

This problem was considered as example1 in the original BioNetFit (*[Thomas2016]*).

8.1.10 fceri_gamma

A benchmark rule-based model of IgE-FcεRI signaling, published in *[Gupta2018]*. To create an example fitting problem, we generated synthetic data based on the published ground truth, and try to recover the ground truth parameters by fitting.

8.1.11 igf1r

A model if IGF1R interation with IGF, orignally published and fit with BioNetFit 1 in *[Erickson2018]*. We provide the config and data files to solve the same fitting problem as in the original study.

The original study also performed bootstrapping to assess parameter uncertainty. We provide the config igf1r_boot.conf to perform the same analysis in PyBNF. The results are expected to match the bootstrapping figure in *[Erickson2018]*.

8.1.12 raf_sbml

A SBML model of MST2 and Raf1 crosstalk described in *[Romano2014]* and published on BioModels Database. We include this problem as an example of fitting a typical SBML model found on BioModels Database. We generated synthetic data using the ground truth parameters in the published model, and try to recover the ground truth by fitting.

Fitting every free parameter in the model (63 parameters) is computationally difficult, recommended only on a cluster. To try out fitting with a smaller scale of computation, we also provide the config raf_sbml_simple.conf, in which only a subset of the parameters are free, and the remaining parameters are fixed at the published values.

8.1.13 receptor

A simple ligand-receptor model fit using synthetic data.

This problem was considered as example5 in the original BioNetFit (*[Thomas2016]*).

8.1.14 receptor_nf

A simple ligand-receptor model fit using synthetic data, simulated in NFsim.

This problem was considered as example6 in the original BioNetFit (*[Thomas2016]*).

8.1.15 tcr

A model of T cell receptor signaling, originally published in *[Chylek2014]*. This problem was considered as example4 in the original BioNetFit (*[Thomas2016]*).

This is a computationally expensive model run in NFsim, with each individual simulation taking tens of minutes to complete. We recommend only attempting to run this on a cluster. An example batch file to submit the job to a SLURM cluster is provided.

8.1.16 tlbr

A model trivalent ligand, bivalent receptor system. The model is described in *[Monine2010]* and fit to data in *[Posner2007]*. The problem was considered as example3 in the original BioNetFit (*[Thomas2016]*).

The model is run in NFSim, and can grow computationally expensive in parameter regimes that result in the formation of large aggregates. An example batch file to submit the job to a SLURM cluster is provided.

8.1.17 yeast_cell_cycle

A detailed model for cell cycle control in yeast, described and fit in *[Oguz2013]* using a binary objective function. The model was refit in *[Mitra2018]* with an objective function that combined qualitative and quantitative data, as a demonstration of incorporating constraints into fitting. We provide config, data, and constraint files to reproduce the fit of *[Mitra2018]*.

This is the most difficult example provided in PyBNF. Due to the huge size of parameter space (150 parameters), we require many iterations of fitting to expect a good result. Although each simulation is fast, each objective evaluation requires a total of 120 simulations of different mutant yeast strains, which take a total of ~ 30 seconds on the libRoadRunner/CVODE simulator. Replicating the fit under the same specifications used in *[Mitra2018]* is expected to take several weeks on a cluster or powerful workstation.

The config file may be inspected as an example of how to use the `mutant` keyword to consider "mutant" models that differ only slightly from another model used in fitting. In this problem, each yeast mutant considered is declared using the `mutant` keyword to change a few parameters compared to the base model. By doing so, we avoid having to maintain 120 separate, nearly identical .xml files.

8.2 Index of examples by attribute

8.2.1 Examples by complexity

- Trivial (for validating installation): *demo*, *constraint_demo*
- Easy (Can run on a personal computer): *receptor*, *receptor_nf* *constraint_raf*, *fceri_gamma*, *egfr_benchmark*
- Moderate: *degranulation*, *igf1r*, *egfr_ode*, *egfr_nf*, *egfr_bleaching*, *raf_sbml*
- Difficult (Recommended on a cluster only): *tcr*, *tlbr*, *yeast_cell_cycle*

8.2.2 Examples by source

- Novel fits described in the PyBNF paper: *egfr_bleaching*, *yeast_cell_cycle*
- Examples from BioNetFit 1: *egfr_ode*, *egfr_nf*, *tlbr*, *tcr*, *receptor*, *receptor_nf*
- Published applications of BioNetFit 1: *degranulation*, *igf1r*
- Synthetic data with known ground truth: *constraint_raf*, *fceri_gamma*, *egfr_benchmark*, *raf_sbml*

8.2.3 Examples by data/model types

- Constraint (.con) data files: *constraint_demo*, *constraint_raf*, *constraint_advanced*, *yeast_cell_cycle*
- SBML models: *raf_sbml*. *yeast_cell_cycle*
- Multiple data files: *degranulation*
- Multiple model files: *egfr_bleaching*
- Mutant models: *yeast_cell_cycle*

8.2.4 Examples by PyBNF feature

- Comparison of all available algorithms: *egfr_benchmark*
- Bootstrapping: *igf1r*
- Calculating Bayesian posterior: *degranulation*
- Advanced constraint configuration: *constraint_advanced*
- Submitting jobs to a cluster: *tlbr*, *tcr*, *degranulation*

8.3 References

CHAPTER

NINE

TROUBLESHOOTING

9.1 Failed simulations

If most or all of your simulations are failing (and generate messages like "Job init0 failed"), troubleshooting is necessary at the level of the simulator (BioNetGen or libRoadRunner).

9.1.1 Check the simulation logs

Rerun the fit with the debugging flag `-d`. Failed simulations will send their logs (generally stdout and stderr) to a `FailedSimLogs` folder in the specified output directory. These logs should usually contain more information about why the simulator failed to run.

If the fit was run with `delete_old_files=0` in the config file, the logs can instead be found in the appropriate folder in the `Simulations/` directory.

9.1.2 For BNGL simulations

Confirm that the BioNetGen path is set

Confirm that PyBNF is looking for BioNetGen in the right place: it will use the `bng_command` specified in your config file if present, and otherwise will use your `BNGPATH` environmental variable (we recommend this second option). To check that `BNGPATH` is set correctly, run `$BNGPATH/BNG2.pl`; you should see a help message including your BioNetGen version number. If not, *try setting BNGPATH again*.

Confirm that the model runs in BioNetGen

If the simulation logs are not sufficient to diagnose the problem, you may want to check whether you can run BioNetGen on the PyBNF-generated model files by hand. Run the fit with the config key `delete_old_files=0`, and refer to the subdirectory of the `Simulations` folder corresponding to a job that failed. Try running BioNetGen on that .bngl file and check for errors; also examine the .bngl file and confirm that PyBNF did not introduce any errors to the model.

If your model is not running in BioNetGen, the best place to find help is the documentation and troubleshooting for BioNetGen, at http://bionetgen.org

Known BioNetGen issues:

- If you are using a Linux distribution other than Ubuntu, it may be necessary to compile BioNetGen from source rather than installing the pre-built binary. Specifically, on CentOS, the binary appears to work at first glance, but fails to parse models containing functions.

9.1.3 For SBML simulations

Confirm accuracy of SBML

If the SBML file was generated in COPASI, refer to *Unexpected behavior when generating SBML files in COPASI*.

CVODE errors

For SBML models, if your logs in `FailedSimLogs/` include errors from CVODE such as "CV_ERR_FAILURE: Error test failures occurred too many times during one internal time step" or "CV_TOO_MUCH_WORK: The solver took mxstep internal steps but could not reach tout", it means that CVODE, the ODE integrator used in libRoadRunner to simulate SBML models, decided that the model was too difficult to simulate and gave up. This might happen when the solution to the ODE system is not sufficiently smooth.

Unfortunately, we do not know of a good workaround for this error.

Resource not available

We have seen this error message come up and cause all simulations to fail when some especially badly behaved SBML process is still running from a previous fitting run (see *Jobs still running after PyBNF stops*). Killing all of the offending processes typically resolves this error.

9.2 Timed out simulations

PyBNF enforces a maximum run time for simulations, with a default value of 1 hour. If you find a large number of your simulations are timing out, increase this value using the config key `wall_time_sim`.

A time limit is also enforced for network generation in BNGL models. The default value is 1 hour, and this can be modified with the `wall_time_gen` key.

9.3 Unexpected behavior when generating SBML files in COPASI

While COPASI is a useful tool for generating SBML files, it is important to note that some settings in COPASI do not get converted into SBML. This can lead to unexpected model behavior in PyBNF.

To help confirm that your model is running as expected, you can set `delete_old_files=0` in your config file, which causes the model output as it was simulated by libRoadRunner in PyBNF to be saved in the `Simulations/` directory.

The following are known issues in translating from COPASI to SBML / libRoadRunner:

- Writing formulas in terms of derivatives of species is possible in COPASI, but does not export to SBML.
- If you rename a parameter or species in COPASI (some time after its creation), the parameter / species is not renamed in the exported SBML, likely causing a PyBNF error about a name not being found. To effectively rename a parameter or species, do a find/replace for `id="oldname"` in the SBML file itself, or delete the object in COPASI and create a new one.
- Defining an "Initial expression" for the concentration of a species is supported in COPASI, but does not export to SBML.

- Events are handled differently if the trigger is true at time 0. COPASI provides options for behavior, with the default being that the event does not fire. These options do not export to SBML. In libRoadRunner and PyBNF, the only option is that the event *continues to fire as long as the trigger remains true*. Note that this is different behavior than for events triggered at time > 0, which will only fire once.

9.4 Too many open files

Some highly parallelized runs may encounter the error "Too many open files". This error occurs when PyBNF exceeds the number of open files allowed by the system for a single program. When this error comes up, it prevents PyBNF from saving results and backups of the run, and may also interfere with its ability to run simulations.

Source of the bug: Each time that PyBNF submits a job, it uses 2 file handles to keep track of the connection between the scheduler and the worker. These file handles are closed eventually, but remain open for a short time after a job completes. If you have a fast running simulation, you might get ~ 5 iterations' worth of these handles left open at the same time. If that many handles exceeds your system limit, you will encounter this bug.

Remedies: You can check the limit of open files per program on the command line: `ulimit -n` gives you the "soft" limit, and `ulimit -Hn` gives you the "hard" limit. The soft limit is what is actually enforced. You can increase the soft limit up to the hard limit with, for example `ulimit -n 4096` if your hard limit is 4096 (this only affects the current terminal, so do it in the same terminal where you will run PyBNF). This might give you enough file handles to avoid the bug. If not, the hard limit can be increased with root access to the machine.

If you are unable to increase the open file handle limit, then you will have to reduce the number of parallel jobs submitted in PyBNF by adjusting the `num_parallel` or `population_size` settings.

9.5 Jobs still running after PyBNF stops

Ordinarily, PyBNF kills simulation jobs that run longer than the time limit. However, if PyBNF itself exits (terminated by the user, or finished a fitting run with jobs still pending), then it is no longer able to enforce the time limit on any jobs that are still running. Any such jobs will continue until they finish or are killed.

If the undead jobs become problematic, it is possible to kill them manually. Use the command `top` to see if you have any such jobs: the processes will have the name `run_network`, `NFsim`, or `python`, depending on which simulator you are using. Note the PID of the offending process(es), and then run `kill <PID>` on the appropriate PIDs. It is also possible to kill all of the jobs at once by running `killall run_network`, `killall NFsim`, or `killall python`, provided that you have no running processes of the same name that you want to keep.

9.6 An unknown error occurred

If you get this message, you found an error that we did not catch during development. Sorry. It might be an unusual, user-generated situation that we didn't think of but is fixable on your end, or could be a bug in the PyBNF source code.

Refer to the log file to try to diagnose the problem - it will contain the Python traceback of the error that was thrown, which sometimes contains enough information to identify what happened.

Rerun the fit with the debugging `-d` flag to generate a more detailed log file (with a "debug" tag).

If you would like to report the bug to the developers (https://github.com/NAU-BioNetFit/PyBNF/), it will be helpful for us if you include the debug log file with your bug report.

9.7 Other issues

If you encounter a bug that is not documented here, or have a request for a new feature, please contact the developers at https://github.com/NAU-BioNetFit/PyBNF/ .

CHAPTER

TEN

PYBNF MODULE REFERENCE

Detailed documentation of the PyBNF code base for advanced users

10.1 PyBNF Module References

10.1.1 PyBNF entry point (pybnf.pybnf)

The entry point for the PyBNF application containing the main function and version

pybnf.pybnf.**main**()
The main function for running a fitting job

10.1.2 PyBNF algorithms (pybnf.algorithms)

Contains the Algorithm class and subclasses as well as support classes and functions for running simulations

class pybnf.algorithms.**Algorithm**(*config*)
A superclass containing the structures common to all metaheuristic and MCMC-based algorithms defined in this software suite

__init__(*config*)
Instantiates an Algorithm with a Configuration object. Also initializes a Trajectory instance to track the fitting progress, and performs various additional configuration that is consistent for all algorithms

Parameters **config** (*Configuration*) – The fitting configuration

add_iterations(*n*)
Adds n additional iterations to the algorithm. May be overridden in subclasses that don't use self.max_iterations to track the iteration count

add_to_trajectory(*res*)
Adds the information from a Result to the Trajectory instance

backup(*pending_psets=()*)
Create a backup of this algorithm object that can be reloaded later to resume the run

Parameters **pending_psets** – Iterable of PSets that are currently submitted as jobs, and will need to get re-submitted

when resuming the algorithm :return:

cleanup()
Called before the program exits due to an exception. :return:

get_backup_every()
Returns a number telling after how many individual simulation returns should we back up the algorithm. Makes a good guess, but could be overridden in a subclass

got_result(*res*)
Called by the scheduler when a simulation is completed, with the pset that was run, and the resulting simulation data

Parameters **res** (Result) – result from the completed simulation

Returns List of PSet(s) to be run next or 'STOP' string.

make_job(*params*)
Creates a new Job using the specified params, and additional specifications that are already saved in the Algorithm object If smoothing is turned on, makes n identical Jobs and a JobGroup

Parameters **params** (PSet) –

Returns list of Jobs (of length equal to smoothing setting)

output_results(*name=''*, *no_move=False*)
Tells the Trajectory to output a log file now with the current best fits.

This should be called periodically by each Algorithm subclass, and is called by the Algorithm class at the end of the simulation. :return: :param name: Custom string to add to the saved filename. If omitted, we just use a running counter of the number of times we've outputted. :param no_move: If True, overrides the config setting delete_old_files=2, and does not move the result to overwrite sorted_params.txt :type name: str

random_latin_hypercube_psets(*n*)
Generates n random PSets with a latin hypercube distribution More specifically, the uniform_var and loguniform_var variables follow the latin hypercube distribution, while lognorm are randomized normally.

Parameters **n** – Number of psets to generate

Returns

random_pset()
Generates a random PSet based on the distributions and bounds for each parameter specified in the configuration

Returns

reset(*bootstrap*)
Resets the Algorithm, keeping loaded variables and models

Parameters **bootstrap** (*int or None*) – The bootstrap number (None if not bootstrapping)

Returns

run(*log_prefix*, *scheduler_node=None*, *resume=None*, *debug=False*)
Main loop for executing the algorithm

static should_pickle(*k*)
Checks to see if key 'k' should be included in pickling. Currently allows all entries in instance dictionary except for 'trajectory'

Parameters **k** –

Returns

start_run()
Called by the scheduler at the start of a fitting run. Must return a list of PSets that the scheduler should run.

Algorithm subclasses optionally may set the .name field of the PSet objects to give a meaningful unique identifier such as 'gen0ind42'. If so, they MUST BE UNIQUE, as this determines the folder name. Uniqueness will not be checked elsewhere.

Returns list of PSets

class pybnf.algorithms.**AsynchronousDifferentialEvolution**(*config*)
Implements a simple asynchronous differential evolution algorithm.

Contains no islands or migrations. Instead, each time a PSet finishes, proposes a new PSet at the same index using the standard DE formula and whatever the current population happens to be at the time.

__init__(*config*)
Initializes algorithm based on the config object.

got_result(*res*)
Called when a simulation run finishes

Parameters **res** – Result object

Returns

class pybnf.algorithms.**BasicBayesMCMCAlgorithm**(*config*, *sa=False*)
Implements a Bayesian Markov chain Monte Carlo simulation.

This is essentially a non-parallel algorithm, but here, we run n instances in parallel, and pool all results. This will give you a best fit (which is maybe not great), but more importantly, generates an extra result file that gives the probability distribution of each variable. This distribution depends on the prior, which is specified according to the variable initialization rules.

With sa=True, this instead acts as a simulated annealing algorithm with n indepdendent chains.

choose_new_pset(*oldpset*)
Helper function to perturb the old PSet, generating a new proposed PSet If the new PSet fails automatically because it violates box constraints, returns None.

Parameters **oldpset** (PSet) – The PSet to be changed

Returns the new PSet

cleanup()
Called when quitting due to error. Save the histograms in addition to the usual algorithm cleanup

got_result(*res*)
Called by the scheduler when a simulation is completed, with the pset that was run, and the resulting simulation data

Parameters **res** (Result) – PSet that was run in this simulation

Returns List of PSet(s) to be run next.

replica_exchange()
Performs replica exchange for parallel tempering. Then proposes n new parameter sets to resume all chains after the exchange. :return: List of n PSets to run

should_sample(*index*)
Checks whether this replica index is one that gets sampled. For mcmc, always True. For pt, must be a replica at the max beta

start_run()
Called by the scheduler at the start of a fitting run. Must return a list of PSets that the scheduler should run.

Returns list of PSets

try_to_choose_new_pset(*index*)
Helper function Advances the iteration number, and tries to choose a new parameter set for chain index i If that fails (e.g. due to a box constraint), keeps advancing iteration number and trying again.

If it hits an iteration where it has to stop and wait (a replica exchange iteration or the end), returns None Otherwise returns the new PSet.

Parameters **index** –

Returns

class pybnf.algorithms.**BayesianAlgorithm**(*config*)
Superclass for Bayesian MCMC algorithms

cleanup()
Called when quitting due to error. Save the histograms in addition to the usual algorithm cleanup

ln_prior(*pset*)
Returns the value of the prior distribution for the given parameter set

Parameters **pset** (PSet) –

Returns float value of ln times the prior distribution

load_priors()
Builds the data structures for the priors, based on the variables specified in the config.

sample_pset(*pset*, *ln_prob*)
Adds this pset to the set of sampled psets for the final distribution. :param pset: :type pset: PSet :param ln_prob - The probability of this PSet to record in the samples file. :type ln_prob: float

update_histograms(*file_ext*)
Updates the files that contain histogram points for each variable :param file_ext: String to append to the save file names :type file_ext: str :return:

class pybnf.algorithms.**DifferentialEvolution**(*config*)
Implements the parallelized, island-based differential evolution algorithm described in Penas et al 2015.

In some cases, I had to make my own decisions for specifics I couldn't find in the original paper. Namely: At each migration, a user-defined number of individuals are migrated from each island. For each individual, a random index is chosen; the same index for all islands. A random permutation is used to redistribute individuals with that index to different islands.

Each island performs its migration individually, on the first callback when all islands are ready for that migration. It receives individuals from the migration iteration, regardless of what the current iteration is. This can sometimes lead to wasted effort. For example, suppose migration is set to occur at iteration 40, but island 1 has reached iteration 42 by the time all islands reach 40. Individual j on island 1 after iteration 42 gets replaced with individual j on island X after iteration 40. Some other island Y receives individual j on island 1 after iteration 40.

__init__(*config*)
Initializes algorithm based on the config object.

The following config keys specify algorithm parameters. For move information, see config_documentation.txt population_size num_islands max_iterations mutation_rate mutation_factor migrate_every num_to_migrate

got_result(*res*)
Called when a simulation run finishes

This is not thread safe - the Scheduler must ensure only one process at a time enters this function. (or, I should rewrite this function to make it thread safe)

Parameters **res** – Result object

Returns

class pybnf.algorithms.**Job**(*models*, *params*, *job_id*, *output_dir*, *timeout*, *calc_future*, *norm_settings*, *delete_folder=False*)
Container for information necessary to perform a single evaluation in the fitting algorithm

__init__(*models*, *params*, *job_id*, *output_dir*, *timeout*, *calc_future*, *norm_settings*, *delete_folder=False*)
Instantiates a Job

Parameters

- **models** (*list of Model instances*) – The models to evaluate
- **params** (PSet) – The parameter set with which to evaluate the model
- **job_id** (*str*) – Job identification; also the folder name that the job gets saved to

:param output_dir path to the directory where I should create my simulation folder :type output_dir: str :param calc_future: Future for an ObjectiveCalculator containing the objective function and experimental data, which we can use to calculate the objective value. :type calc_future: Future :param norm_settings: Config value for 'normalization': a string representing the normalization type, a dict mapping exp files to normalization type, or None :type norm_settings: Union[str, dict, NoneType] :param delete_folder: If True, delete the folder and files created after the simulation runs :type delete_folder: bool

run_simulation(*debug=False*, *failed_logs_dir=''*)
Runs the simulation and reads in the result

class pybnf.algorithms.**JobGroup**(*job_id*, *subjob_ids*)
Represents a group of jobs that are identical replicates to be averaged together for smoothing

__init__(*job_id*, *subjob_ids*)

Parameters

- **job_id** – The name of the Job this group is representing
- **subjob_ids** – A list of the ids of the identical replicate Jobs.

average_results()
To be called after all results are in for this group. Averages the results and returns a new Result object containing the averages

Returns New Result object with the job_id of this JobGroup and the averaged Data as the sim-data

job_finished(*res*)
Called when one job in this group has finished :param res: Result object for the completed job :return: Boolean, whether everything in this job group has finished

class pybnf.algorithms.**ParticleSwarm**(*config*)
Implements particle swarm optimization.

The implementation roughly follows Moraes et al 2015, although is reorganized to better suit PyBNF's format. Note the global convergence criterion discussed in that paper is not used (would require too long a computation), and instead uses ????

__init__(*config*)
Initial configuration of particle swarm optimizer :param conf_dict: The fitting configuration :type conf_dict: Configuration

The config should contain the following definitions:

population_size - Number of particles in the swarm max_iterations - Maximum number of iterations. More precisely, the max number of simulations run is this times the population size. cognitive - Acceleration toward the particle's own best social - Acceleration toward the global best particle_weight - Inertia weight of the particle (default 1)

The following config parameters relate to the complicated method presented is Moraes et al for adjusting the inertia weight as you go. These are optional, and this feature will be disabled (by setting particle_weight_final = particle_weight) if these are not included. It remains to be seen whether this method is at all useful for our applications.

particle_weight_final - Inertia weight at the end of the simulation adaptive_n_max - Controls how quickly we approach wf - After nmax "unproductive" iterations, we are halfway from w0 to wf adaptive_n_stop - nd the entire run if we have had this many "unproductive" iterations (should be more than adaptive_n_max) adaptive_abs_tol - Tolerance for determining if an iteration was "unproductive". A run is unproductive if the change in global_best is less than absolute_tol + relative_tol * global_best adaptive_rel_tol - Tolerance 2 for determining if an iteration was "unproductive" (see above)

got_result(*res*)
Updates particle velocity and position after a simulation completes.

Parameters **res** – Result object containing the run PSet and the resulting Data.

Returns

start_run()
Start the run by initializing n particles at random positions and velocities :return:

class pybnf.algorithms.**Result**(*paramset*, *simdata*, *name*)
Container for the results of a single evaluation in the fitting algorithm

__init__(*paramset*, *simdata*, *name*)
Instantiates a Result

Parameters

- **paramset** (PSet) – The parameters corresponding to this evaluation
- **simdata** – The simulation results corresponding to this evaluation, as a nested dictionary structure.

Top-level keys are model names and values are dictionaries whose keys are action suffixes and values are Data instances :type simdata: dict Returns a :param log: The stdout + stderr of the simulations :type log: list of str

normalize(*settings*)
Normalizes the Data object in this result, according to settings :param settings: Config value for 'normalization': a string representing the normalization type, a dict mapping exp files to normalization type, or None :return:

class pybnf.algorithms.**ScatterSearch**(*config*)
Implements ScatterSearch as described in the introduction of Penas et al 2017 (but not the fancy parallelized version from that paper). Uses the individual combination method described in Egea et al 2009

get_backup_every()
Overrides base method because Scatter Search runs n*(n-1) PSets per iteration.

got_result(*res*)
Called when a simulation run finishes

Parameters **res** –

:type res Result :return:

class pybnf.algorithms.**SimplexAlgorithm**(*config*, *refine=False*)
Implements a parallelized version of the Simplex local search algorithm, as described in Lee and Wiswall 2007, Computational Economics

a_plus_b_times_c_minus_d(*a*, *b*, *c*, *d*, *v*)
Performs the calculation a + b*(c-d), where a, c, and d are assumed to be in log space if v is in log space, and the final result respects the box constraints on v.

Parameters

- **a** –
- **b** –
- **c** –
- **d** –
- **v** (FreeParameter) –

Returns

ab_plus_cd(*a*, *b*, *c*, *d*, *v*)
Performs the calculation ab + cd where b and d are assumed to be in log space if v is in log space, and the final result respects the box constraints on v :param a: :param b: :param c: :param d: :param v: :type v: FreeParameter :return:

get_sums()
Simplex helper function Returns a dict mapping parameter name p to the sum of the parameter value over the entire current simplex :return: dict

pybnf.algorithms.**exp10**(*n*)
Raise 10 to the power of a possibly user-defined value, and raise a helpful error if it overflows :param n: A float :return: 10.** n

pybnf.algorithms.**latin_hypercube**(*nsamples*, *ndims*)
Latin hypercube sampling.

Returns a nsamples by ndims array, with entries in the range [0,1] You'll have to rescale them to your actual param ranges.

pybnf.algorithms.**run_job**(*j*, *debug=False*, *failed_logs_dir=''*)
Runs the Job j. This function is passed to Dask instead of j.run_simulation because if you pass j.run_simulation, Dask leaks memory associated with j.

10.1.3 PyBNF cluster setup (pybnf.cluster)

Functions for managing dask cluster setup and teardown on distributed computing systems

pybnf.cluster.**get_scheduler**(*config*)

Parameters **config** (*pybnf.config.Configuration*) – PyBNF configuration

Returns scheduler node, string composed of all available nodes

pybnf.cluster.**setup_cluster**(*node_string*, *out_dir*)
Sets up a Dask cluster using the *dask-ssh* convenience script

Parameters

- **node_string** – A string composed of a list of compute nodes
- **out_dir** – A directory for

Returns subprocess.Popen

pybnf.cluster.**teardown_cluster**(*dsp*)

Terminates the process running the *dask-ssh* script after completion of fitting run

Parameters **dsp** – subprocess.Popen

Returns

10.1.4 PyBNF configuration (pybnf.config)

Classes and methods for configuring the fitting run

10.1.5 PyBNF constraint specification (pybnf.constraint)

Classes for defining various constraints that can be applied to the fitting run. Used when incorporating qualitative data into the fit

class pybnf.constraint.**Constraint**(*quant1*, *sign*, *quant2*, *base_model*, *base_suffix*, *weight*, *alt-penalty=None*, *minpenalty=0.0*)

Abstract class representing an optimization constraint with a penalty for violating the constraint

find_keys(*sim_data_dict*)

Function to be called on the first evaluation of the penalty. Read through the data dictionary and figure out what keys I need to use to access each relevant variable.

This needs to be done kind of by brute force, because we were never told the model file to use, and it's even possible the input is poorly defined by referring to a suffix that exists in multiple models. :param sim_data_dict: :return:

get_key(*q*, *sim_data_dict*)

Converts a string from a constraint file ('Observable' or 'suffix.observable') into a tuple of the 3 keys you would need to index into the sim_data_dict to get the right column of data.

Parameters **q** –

Returns

get_penalty(*sim_data_dict*, *imin*, *imax*, *once=False*, *require_length=None*)

Helper function for calculating the penalty, that can be called from the subclasses. Enforces the constraint for the entire interval unless the once option is set.

Parameters

- **sim_data_dict** – The dictionary of data objects
- **imin** – First index at which to check the constraint
- **imax** – Last index at which to check the constraint (exclusive)
- **once** – If true, enforce that the constraint holds once at some point during the time interval
- **require_length** – If set to an integer, raise an error if the length of the selected data column(s) is not

equal to that value. (Used to check that "at" and "between" constraints are not encountering an unsupported case) :return:

index(*sim_data_dict*, *keys*)
Shortcut function for applying all 3 indices to the data object :return:

penalty(*sim_data_dict*)
penalty function for violating the constraint. Returns 0 if constraint is satisfied, or a positive value if the constraint is violated. Implementation depends on the type of constraint.

Parameters **sim_data_dict** (*dict*) – Dictionary of the form {modelname: {suffix1: Data1}} containing the simulated data objects

class pybnf.constraint.**ConstraintSet**(*base_model*, *base_suffix*)
Represents the set of all constraints provided in one con file

load_constraint_file(*filename*, *scale=1.0*)
Parse the constraint file filename and load them all into my constraint list :param filename: Path of constraint file :param scale: Factor by which we multiply all constraint weights

total_penalty(*sim_data_dict*)
Evaluate the sim_data_dict against all constraints, and return the total penalty

Parameters **sim_data_dict** – Dictionary of the form {modelname: {suffix1: Data1}} containing the simulated data objects

Returns

10.1.6 PyBNF data container (pybnf.data)

Class with methods to manage experimental and simulation data

class pybnf.data.**Data**(*file_name=None*, *arr=None*, *named_arr=None*)
Top level class for managing data

static average(*datas*)
Calculates the average of several data objects. The input Data objects should have the same column labels and independent variable values (NOT CURRENTLY CHECKED)

Parameters **datas** – Iterable of Data objects of identical size to be averaged

Returns Data object

gen_bootstrap_weights()
Generates a integer weight for each point in the set of dependent variables. Equivalent to sampling with replacement. Weights are used when calculating the objective function for bootstrapped data. Used for experimental data sets

Returns

get_row(*col_header*, *value*)
Returns the (first) data row in which field col_header is equal to value. This should typically be used for col_header as the independent variable.

Parameters **col_header** – Data column name

:type col_header str :param value: :type value: str :return: 1D numpy array consisting of the requested row

load_data(*file_name*, *sep='\\s+'*)
Loads column data from a text file

Parameters

- **file_name** (*str*) – Name of data file
- **sep** (*str*) – String that separates columns

Returns None

load_rr_header(*header*)
Loads the header from a RoadRunner NamedArray :param header: The colnames attribute of a RoadRunner NamedArray (a list of str)

normalize(*method*)
Normalize the data according to the specified method: 'init', 'peak', 'unit', or 'zero' The method could also be a list of ordered pairs [('init', [columns]), ('peak', [columns])], where columns is a list of integers or column labels

Updates the data array in this object, returns none.

normalize_to_init(*idx=0*, *cols='all'*)
Normalizes all data columns (except the independent variable) to the initial value in their respective columns

Updates the data array in this object, returns none.

Parameters

- **idx** (*int*) – Index of independent variable
- **cols** – List of column indices to normalize, or 'all' for all columns but independent variable

normalize_to_peak(*idx=0*, *cols='all'*)
Normalizes all data columns (except the independent variable) to the peak value in their respective columns

Updates the data array in this object, returns none.

Parameters

- **idx** (*int*) – Index of independent variable
- **cols** – List of column indices to normalize, or 'all' for all columns but independent variable

Returns Normalized Numpy array (including independent variable column)

normalize_to_unit_scale(*idx=0*, *cols='all'*)
Scales data so that the range of values is between (min-init)/(max-init) and 1. If the maximum value is 0 (i.e. max == init), then the data is scaled by the minimum value after subtracting the initial value so that the range of values is between 0 and -1

Parameters

- **idx** (*int*) – Index of independent variable
- **cols** – List of column indices to normalize, or 'all' for all columns but independent variable

Type list or str

Returns

normalize_to_zero(*idx=0*, *bc=True*, *cols='all'*)
Normalizes data so that each column's mean is 0

Updates the data array in this object, returns none.

Parameters

- **idx** (*int*) – Index of independent variable
- **bc** (*bool*) – If True, the standard deviation is normalized by 1/(N-1). If False, by 1/N.
- **cols** – List of column indices to normalize, or 'all' for all columns but independent variable

10.1.7 PyBNF objective functions (pybnf.objective)

Classes defining various objective functions used for evaluating points in parameter space

class pybnf.objective.**AveNormSumOfSquaresObjective**(*ind_var_rounding=0*)
Sum of squares where each point is normalized by the average value of that variable, ((y-y')/ybar)^2

class pybnf.objective.**NormSumOfSquaresObjective**(*ind_var_rounding=0*)
Sum of squares where each point is normalized by the y value at that point, ((y-y')/y)^2

class pybnf.objective.**ObjectiveCalculator**(*objective*, *exp_data_dict*, *constraints*)
Wrapper for all of the objects needed for the workers to calculate the objective function value. Contains the objective function, exp_data_dict, and constraint tuple

evaluate_objective(*sim_data_dict*, *show_warnings=True*)
Evaluate the objective using the input simulation data and the info contained in this object :param sim_data_dict: Dictionary of the form {modelname: {suffix1: Data1}} containing the simulated data objects :param show_warnings: If True, print warnings about unused data :type show_warnings: bool :return:

class pybnf.objective.**ObjectiveFunction**
Abstract class representing an objective function Subclasses customize how the objective value is calculated from the quantitative exp data The base class includes all the support we need for constraints.

evaluate(*sim_data*, *exp_data*, *show_warnings=True*)

Parameters

- **sim_data** (Data) – A Data object containing simulated data
- **exp_data** (Data) – A Data object containing experimental data
- **show_warnings** (*bool*) – If True, print warnings about unused data

Returns float, value of the objective function, with a lower value indicating a better fit.

evaluate_multiple(*sim_data_dict*, *exp_data_dict*, *constraints=()*, *show_warnings=True*)
Compute the value of the objective function on several data sets, and return the total. Optionally may pass an iterable of ConstraintSets whose penalties will be added to the total

Parameters

- **sim_data_dict** (*dict*) – Dictionary of the form {modelname: {suffix1: Data1}} containing the simulated data objects
- **exp_data_dict** (*dict*) – Dictionary of the form {modelname: {suffix1: Data1}} containing experimental Data objects
- **constraints** – Iterable of ConstraintSet objects containing the constraints that we should evaluate using

the simulated data :type constraints: Iterable of ConstraintSet :param show_warnings: If True, print warnings about unused data :type show_warnings: bool :return:

class pybnf.objective.**SummationObjective**(*ind_var_rounding=0*)

Represents a type of objective function in which we perform some kind of summation over all available experimental data points. Currently, this describes all objective functions in PyBNF.

eval_point(*sim_data*, *exp_data*, *sim_row*, *exp_row*, *col_name*)

Calculate the objective function for a single point in the data

This evaluation is what differentiates the different objective functions.

Parameters

- **sim_data** – The simulation Data object
- **exp_data** – The experimental Data object
- **sim_row** – The row number to look at in sim_data
- **exp_row** – The row number to look at in exp_data
- **col_name** – The column name to look at (same for the sim_data and the exp_data)

Returns

evaluate(*sim_data*, *exp_data*, *show_warnings=True*)

Parameters

- **sim_data** (Data) – A Data object containing simulated data
- **exp_data** (Data) – A Data object containing experimental data
- **show_warnings** (*bool*) – If True, print warnings about unused data

Returns float, value of the objective function, with a lower value indicating a better fit.

10.1.8 PyBNF configuration parsing (pybnf.parse)

Grammar and methods for parsing the configuration file

pybnf.parse.**parse_normalization_def**(*s*)

Parse the complicated normalization grammar If the grammar is specified incorrectly, it will end up calling something invalid the normalization type or the exp file, and this error will be caught later.

Parameters **s** – The string following the equals sign in the normalization key

Returns What to write in the config dictionary: A string, or a dictionary {expfile: string} or

{expfile: (string, index_list)} or {expfile: (string, name_list)}

10.1.9 PyBNF printing functions (pybnf.printing)

Contains printing commands that respect the application-wide verbosity setting.

exception pybnf.printing.**PybnfError**(*log_message*, *user_message=None*)

Represents a user-generated error for which we can provide an informative message to the user about what went wrong with the input before quitting.

pybnf.printing.**print0**(*s*)

Print the statement at any verbosity level

pybnf.printing.**print1**(*s*)

Print the statement only if the verbosity level is at least 1

pybnf.printing.**print2**(*s*)
Print the statement only if the verbosity level is 2

10.1.10 PyBNF model and parameter containers (pybnf.pset)

Classes for storing models, parameter sets, and the fitting trajectory

class pybnf.pset.**Action**
Represents a simulation action performed within a model

class pybnf.pset.**BNGLModel**(*bngl_file*, *pset=None*)
Class representing a BNGL model

copy_with_param_set(*pset*)
Returns a copy of this model containing the specified parameter set.

Parameters **pset** (PSet) – A PSet object containing the parameters for the new instance

Returns BNGLModel

execute(*folder*, *filename*, *timeout*, *with_mutants=True*)

Parameters **folder** – Folder in which to do all the file creation

Returns Data object

get_suffixes()
Return a list of valid data suffixes to use in this model, including all combinations of action suffix + mutation name

model_text(*gen_only=False*)
Returns the text of a runnable BNGL file, which includes the contents of the original BNGL file, and also values assigned to each __FREE parameter, as determined by this model's PSet

Returns str

save(*file_prefix*, *gen_only=False*, *pset=None*)
Saves a runnable BNGL file of the model, including definitions of the __FREE parameter values that are defined by this model's pset, to the specified location.

Parameters

- **file_prefix** – str, path where the file should be saved
- **gen_only** – bool, output model with only generate_network action if True

save_all(*file_prefix*)
Saves BNGL files of the original model and all mutants :param file_prefix:

exception pybnf.pset.**FailedSimulationError**
Raised when a simulation fails that was not a result of a subprocess.run() call (currently only use with SbmlModelNoTimeout)

class pybnf.pset.**FreeParameter**(*name*, *type*, *p1*, *p2*, *value=None*, *bounded=True*)
Class representing a free parameter in a model

add(*summand*, *reflect=True*)
Adds a value to the existing value and returns a new FreeParameter instance. Since free parameters can exist in regular or logarithmic space, the value to add is expected to already be transformed to the appropriate space

Parameters **summand** – Value to add

Returns

add_rand(*lb*, *ub*, *reflect=True*)
Like FreeParameter.add but instead adds a uniformly distributed random value according to the bounds provided

Parameters

- **lb** –
- **ub** –

Returns

diff(*other*)
Calculates the difference between two FreeParameter instances. Both instances must occupy the same space (log or regular) and if they are both in log space, the difference will be calculated based on their logarithms. :param other: A FreeParameter from which the difference will be calculated :return:

sample_value()
Samples a value for this parameter based on its defined initial distribution

Returns new FreeParameter instance or None

set_value(*new_value*, *reflect=True*)
Creates a copy of the parameter with the given value

Parameters

- **new_value** (*float*) – A numeric value assigned to the FreeParameter
- **reflect** (*bool*) – Determines whether to reflect the parameter value if it is outside of the defined bounds

Returns FreeParameter

class pybnf.pset.**Model**
An abstract class representing an executable model

add_mutant(*mut_set*)
Add a mutant to run along with this model :param mut_set: MutationSet that should be applied to this mutant :type mut_set: MutationSet :return:

copy_with_param_set(*pset*)
Returns a copy of the model with a new parameter set

Parameters **pset** (PSet) – A new parameter set

Returns Model

execute(*folder*, *filename*, *timeout*)
Executes the model, working in folder/filename, with a max runtime of timeout. Loads the resulting data, and returns a dictionary mapping suffixes to data objects. For model types without a notion of suffixes, the dictionary will contain one key mapping to one Data object

Parameters

- **folder** – The folder to save to, eg 'Simulations/init22'
- **filename** – The name of the model file to create, not including the extension, eg 'init22'
- **timeout** – Maximum runtime in seconds

Returns dict of Data

get_suffixes()
Return a list of valid data suffixes to use in this model, including all combinations of action suffix + mutation name

save(*file_prefix*, ***kwargs*)
Saves the model to file

Returns

class pybnf.pset.**MutationSet**(*mutations=()*, *suffix=''*)
A set of mutations that represents a mutant model

class pybnf.pset.**PSet**(*fps*)
Class representing a parameter set

get_param(*name*)
Gets the full FreeParameter based on its name

Parameters **name** –

Returns

keys()
Returns a list of the parameter keys :return: list

keys_to_string()
Returns the keys (parameter names) in a tab-separated str in alphabetical order

Returns str

values_to_string()
Returns the parameter values in a tab-separated str, in alphabetical order according to the parameter name :return: str

class pybnf.pset.**Trajectory**(*max_output*)
Tracks the various PSet instances and the corresponding objective function values

add(*pset*, *obj*, *name*, *append_file=None*, *first=False*)
Adds a PSet to the fitting trajectory

Parameters

- **pset** – A particular point in parameter space
- **obj** – The objective function value upon executing the model at this point in parameter space

Raises Exception

best_fit()
Finds the best fit parameter set

Returns PSet

best_fit_name()
Finds the name of the best fit parameter set (which is also the folder where that result is stored)

Returns str

best_score()
Returns the best objective value in this trajectory :return: float

static **load_trajectory**(*filename*, *variables*, *max_output*)
Loads a Trajectory from file given Algorithm.variables information

write_to_file(*filename*)
Writes the Trajectory to a specified file

Parameters **filename** – File to store Trajectory

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[Egea2009] Egea, J. A.; Balsa-Canto, E.; García, M.-S. G.; Banga, J. R. Dynamic Optimization of Nonlinear Processes with an Enhanced Scatter Search Method. Ind. Eng. Chem. Res. 2009, 48 (9), 4388–4401.

[Glover2000] Glover, F.; Laguna, M.; Martí, R. Fundamentals of Scatter Search and Path Relinking. Control Cybern. 2000, 29 (3), 652–684.

[Gupta2018] Gupta, S.; Hogg, J. S.; Lee, R. E. C.; Faeder, J. R. Evaluation of Parallel Tempering to Accelerate Markov Chain Monte Carlo Methods for Parameter Estimation in Systems Biology. arXiv 2018, 1801.09831.

[Kozer2013] Kozer, N.; Barua, D.; Orchard, S.; Nice, E. C.; Burgess, A. W.; Hlavacek, W. S.; Clayton, A. H. A. Exploring Higher-Order EGFR Oligomerisation and Phosphorylation—a Combined Experimental and Theoretical Approach. Mol. BioSyst. Mol. BioSyst 2013, 9 (9), 1849–1863.

[Lee2007] Lee, D.; Wiswall, M. A Parallel Implementation of the Simplex Function Minimization Routine. Comput. Econ. 2007, 30 (2), 171–187.

[Moraes2015] Moraes, A. O. S.; Mitre, J. F.; Lage, P. L. C.; Secchi, A. R. A Robust Parallel Algorithm of the Particle Swarm Optimization Method for Large Dimensional Engineering Problems. Appl. Math. Model. 2015, 39 (14), 4223–4241.

[Penas2015] Penas, D. R.; González, P.; Egea, J. A.; Banga, J. R.; Doallo, R. Parallel Metaheuristics in Computational Biology: An Asynchronous Cooperative Enhanced Scatter Search Method. Procedia Comput. Sci. 2015, 51 (1), 630–639.

[Penas2017] Penas, D. R.; González, P.; Egea, J. A.; Doallo, R.; Banga, J. R. Parameter Estimation in Large-Scale Systems Biology Models: A Parallel and Self-Adaptive Cooperative Strategy. BMC Bioinformatics 2017, 18 (1), 52.

[Vrugt2016] Vrugt, J. Markov chain Monte Carlo simulation using the DREAM software package: Theory, concepts, and MATLAB implementation. Environmental Modelling and Software 2016, 75, 273-316.

[Blinov2006] Blinov, M. L.; Faeder, J. R.; Goldstein, B.; Hlavacek, W. S. A Network Model of Early Events in Epidermal Growth Factor Receptor Signaling That Accounts for Combinatorial Complexity. BioSystems 2006, 83 (2–3 SPEC. ISS.), 136–151.

[Chylek2014] Chylek, L. A.; Akimov, V.; Dengjel, J.; Rigbolt, K. T. G.; Hu, B.; Hlavacek, W. S.; Blagoev, B. Phosphorylation Site Dynamics of Early T-Cell Receptor Signaling. PLoS One 2014, 9 (8), e104240.

[Erickson2018] Erickson, K.; et. al. Under review.

[Gupta2018] Gupta, A.; Mendes, P. An Overview of Network-Based and -Free Approaches for Stochastic Simulation of Biochemical Systems. Computation 2018, 6 (1), 9.

[Harmon2017] Harmon, B.; Chylek, L. A.; Liu, Y.; Mitra, E. D.; Mahajan, A.; Saada, E. A.; Schudel, B. R.; Holowka, D. A.; Baird, B. A.; Wilson, B. S.; et al. Timescale Separation of Positive and Negative Signaling Creates History-Dependent Responses to IgE Receptor Stimulation. Sci. Rep. 2017, 7 (1), 15586.

[Kozer2013] Kozer, N.; Barua, D.; Orchard, S.; Nice, E. C.; Burgess, A. W.; Hlavacek, W. S.; Clayton, A. H. A. Exploring Higher-Order EGFR Oligomerisation and Phosphorylation—a Combined Experimental and Theoretical Approach. Mol. BioSyst. Mol. BioSyst 2013, 9 (9), 1849–1863.

[Mitra2018] Mitra, E. D.; Dias, R.; Posner, R. G.; Hlavacek, W. S. Using Both Qualitative and Quantitative Data in Parameter Identification for Systems Biology Models. Under review.

[Monine2010] Monine, M. I.; Posner, R. G.; Savage, P. B.; Faeder, J. R.; Hlavacek, W. S. Modeling Multivalent Ligand-Receptor Interactions with Steric Constraints on Configurations of Cell-Surface Receptor Aggregates. Biophys. J. 2010, 98 (1), 48–56.

[Oguz2013] Oguz, C.; Laomettachit, T.; Chen, K. C.; Watson, L. T.; Baumann, W. T.; Tyson, J. J. Optimization and Model Reduction in the High Dimensional Parameter Space of a Budding Yeast Cell Cycle Model. BMC Syst. Biol. 2013, 7 (1), 53.

[Posner2007] Posner, R. G.; Geng, D.; Haymore, S.; Bogert, J.; Pecht, I.; Licht, A.; Savage, P. B. Trivalent Antigens for Degranulation of Mast Cells. Org. Lett. 2007, 9 (18), 3551–3554.

[Romano2014] Romano, D.; Nguyen, L. K.; Matallanas, D.; Halasz, M.; Doherty, C.; Kholodenko, B. N.; Kolch, W. Protein Interaction Switches Coordinate Raf-1 and MST2/Hippo Signalling. Nat. Cell Biol. 2014, 16 (7), 673–684.

[Thomas2016] Thomas, B. R.; Chylek, L. A.; Colvin, J.; Sirimulla, S.; Clayton, A. H. A.; Hlavacek, W. S.; Posner, R. G. BioNetFit: A Fitting Tool Compatible with BioNetGen, NFsim and Distributed Computing Environments. Bioinformatics 2016, 32 (5), 798–800.

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

I

J

K

L

M

N

O

P

R

S

T

U

V

W