## Contents

1 Installation .............................................. 2  
   1.1 Operating System ........................................ 2  
   1.2 Python ................................................... 2  
   1.3 PyBNF .................................................. 3  
   1.4 Installation of External Simulators ....................... 3  

2 Quick Start ............................................. 5  
   2.1 Verify installation with simple examples ................ 5  
   2.2 Set up your own fitting job .............................. 6  

3 Configuring a Fitting Job ............................... 8  
   3.1 The Configuration File ................................. 8  
   3.2 Model Files ............................................ 8  
   3.3 Experimental Data Files ............................... 9  
   3.4 Property (BPSL) files ................................. 10  

4 Configuration Keys ..................................... 14  
   4.1 Required Keys .......................................... 14  
   4.2 Other Path Keys ....................................... 15  
   4.3 Parameter and Model Specification ...................... 16  
   4.4 Simulation Actions ................................... 17  
   4.5 Parallel Computing .................................... 18  
   4.6 General Options ...................................... 19  
   4.7 Algorithm-specific Options ............................ 22  

5 Fitting Algorithms ...................................... 30  
   5.1 Summary of Available Algorithms ....................... 30  
   5.2 General implementation features for all algorithms .... 30  
   5.3 Differential Evolution ................................ 31  
   5.4 Scatter Search ......................................... 33  
   5.5 Particle Swarm ......................................... 34  
   5.6 Metropolis-Hastings MCMC ............................. 35  
   5.7 Simulated Annealing ................................... 36  
   5.8 Parallel Tempering ..................................... 36  
   5.9 Adaptive MCMC ........................................ 37  
   5.10 Simplex ................................................. 38
6 Advanced Features
   6.1 Model Checking .......................................................... 40
   6.2 Bootstrapping ............................................................... 40
   6.3 Custom Postprocessing .................................................. 41

7 Algorithm Development ....................................................... 43
   7.1 Implementation ........................................................... 43
   7.2 Adding configuration options ......................................... 44
   7.3 Pull requests ............................................................... 44

8 Running on a cluster .......................................................... 45
   8.1 SLURM ................................................................. 45
   8.2 TORQUE/PBS ........................................................... 46
   8.3 Manual configuration with node names ............................... 46
   8.4 Manual configuration with Dask ..................................... 46
   8.5 (Optional) Logging configuration for remote machines  ...... 47

9 Examples ............................................................................. 48
   9.1 List of examples included in PyBNF ................................. 48
   9.2 Index of examples by attribute ........................................ 51

10 Troubleshooting ................................................................. 53
   10.1 Failed simulations ......................................................... 53
   10.2 Timed out simulations .................................................... 54
   10.3 Unexpected behavior when generating SBML files in COPASI .................................................. 54
   10.4 Too many open files ...................................................... 55
   10.5 Too many threads ......................................................... 55
   10.6 Jobs still running after PyBNF stops ............................... 56
   10.7 PyBNF has encountered a Fatal error ............................. 56
   10.8 An unknown error occurred ......................................... 56
   10.9 Other issues ............................................................... 56

11 PyBNF Module Reference ..................................................... 57
   11.1 PyBNF Module References ............................................. 57

Bibliography ........................................................................... 58
PyBioNetFit (PyBNF) is a general-purpose program for parameterizing biological models specified using the BioNet-Gen rule-based modeling language (BNGL) or the Systems Biology Markup Language (SBML). PyBioNetFit offers a suite of parallelized metaheuristic algorithms (differential evolution, particle swarm optimization, scatter search) for parameter optimization. In addition to model parameterization, PyBNF supports uncertainty quantification by bootstrapping or Bayesian approaches, and model checking. PyBNF includes the Biological Property Specification Language (BPSL) for defining qualitative data for use in parameterization or checking. It runs on most Linux and macOS workstations as well on computing clusters.

To get started using PyBNF, follow the installation instructions, then look at the examples in the Quick Start.
CHAPTER 1

Installation

1.1 Operating System

PyBNF can be installed on any recent Linux or macOS operating system.

PyBNF can also be installed on Windows, but functionality on Windows has been less extensively tested (in particular, Windows clusters and multicore workstations have not been tested).

1.2 Python

PyBNF requires an installation of Python version 3.5 or higher.

1.2.1 Linux and Mac

Python 3 comes built-in on many new Linux and Mac operating systems. To check if you have Python 3, run the command `python3`. This will start Python and print the version number, or will give an error if you don’t have Python 3.

Also confirm that your Python 3 has the `pip` package manager, which is used to install PyBNF. Run the command `python3 -m pip`. This will give a help message if you have pip, or an error if not.

If you are missing `python3` or `pip`, an easy way to get them is by installing the Anaconda Python distribution for Python v3.5 or higher. Instructions for installing on various platforms can be found on the Anaconda website.

1.2.2 Windows

Windows does not come with built-in Python, so it must be installed separately. Additionally, if BioNetGen will be used, Perl installation is required in the same environment as the python installation (i.e., the commands `python` and `perl` must both work on the same command line).
Our recommended configuration consists of installing Strawberry Perl and Anaconda Python 3. The Windows distribution of Anaconda includes the application “Anaconda Prompt”, which provides a command line. This is the command line that you should use whenever this documentation refers to the command line or terminal. After installing both Anaconda and Strawberry Perl, a system restart may be required for Anaconda Prompt to find the Perl installation.

For troubleshooting, or more advanced configuration, note that the requirement is to have both Python 3 and Perl on the current path. The current path can be checked with the command `echo %PATH%` and set (temporarily) with the command `set PATH=[newpath]`, where [newpath] is a semicolon-delimited list of directories to search.

1.3 PyBNF

1.3.1 Installing from PyPI

Simply type the following in a terminal:

```
python3 -m pip install pybnf
```

Windows users running Anaconda Python 3 from “Anaconda Prompt” should instead type only `pip install pybnf`.

The above command will use your current version of Python 3 to install the most recent version of PyBNF released on the Python Package Index, along with all required dependencies.

Depending on your Python configuration, the above command may require root access and install PyBNF for all users on the computer. If you don’t want to do this, you may add the flag `--user` to the end of the command, to install without root access for only the current user.

Advanced Python users may consider installing PyBNF in a `virtualenv` (which also does not require root access) to avoid conflicts between PyBNF’s dependencies and other uses of Python on the computer.

1.3.2 Installing from source

To use bleeding edge PyBNF, the source code may be found on GitHub at [https://github.com/lanl/PyBNF](https://github.com/lanl/PyBNF). To use, simply download or clone the repository and run the following command in the repository’s root directory:

```
python3 -m pip install -e .
```

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

1.4 Installation of External Simulators

1.4.1 BioNetGen

PyBNF is designed to work with simulators present in the BioNetGen software suite, version 2.3, available for download from the BioNetGen website. Note that for Linux distributions other than Ubuntu, the pre-built binary is unreliable, and it is necessary to rebuild BioNetGen from source. For Windows, Perl must be installed separately, as described above. 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.

On Windows systems, the equivalent commands are set BNGPATH=C:\path\to\bng2 to set on the current command line, and setx BNGPATH C:\path\to\bng2 to permanently set for all future command lines (but not the current one).

1.4.2 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 2

Quick Start

2.1 Verify installation with simple examples

Example jobs configured for PyBNF are available in the examples folder of the PyBNF GitHub repository. If you installed PyBNF with pip, you can download this examples folder separately from GitHub or the following direct link.

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 examples/demo directory

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

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

The examples will print progress to the terminal as the fitting proceeds, and the results will be saved in the directory examples/demo/output (this output directory can be changed by editing demo_bng.conf and demo_xml.conf).

In examples/demo/output/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=1.0, 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 demo_bng.conf -t slurm
pybnf -c 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 bnlg 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:

```bnlg
begin parameters
  var1 1
  var2 3
  var3 7
end parameters
```

Modify the BNGL code to:

```bnlg
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:

```bash
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:
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

objcfunc=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 environment 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!
### 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](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:

```bash
begin parameters
  v1 17
  v2 42
  v3 37
  NA 6.02e23
end parameters
```

Replace "v1", "v2", and "v3" with "v1__FREE", "v2__FREE", and "v3__FREE", respectively:

```bash
begin parameters
  v1__FREE 17
  v2__FREE 42
  v3__FREE 37
  NA__FREE 6.02e23
end parameters
```
the revised version for PyBNF should look like:

```plaintext
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 # (optional, to match the output format of BioNetGen), 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 or functions in a BNGL file, or species in an SBML file (in this section, we refer to all of these options as “observables”). 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:

<table>
<thead>
<tr>
<th>#</th>
<th>time</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>1e4</td>
<td></td>
</tr>
</tbody>
</table>
If you 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:

<table>
<thead>
<tr>
<th>time</th>
<th>X</th>
<th>Y</th>
<th>X_SD</th>
<th>Y_SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>1e4</td>
<td>1</td>
<td>2e2</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>1.5e4</td>
<td>1.2</td>
<td>2e2</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>4e4</td>
<td>1.4</td>
<td>4e2</td>
</tr>
<tr>
<td>15</td>
<td>nan</td>
<td>6.5e4</td>
<td>nan</td>
<td>5e2</td>
</tr>
<tr>
<td>20</td>
<td>15</td>
<td>1.1e5</td>
<td>0.9</td>
<td>5e2</td>
</tr>
</tbody>
</table>

### 3.4 Property (BPSL) files

Property files are plain text files with the extension “.prop” that define qualitative system properties. In PyBNF, properties are expressed as 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. The syntax for writing .prop files, described in this section, is called the Biological Property Specification Language (BPSL).

Each line of the .prop 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 clause that specifies how the constraint should be incorporated into the objective function.

Two methods of incorporating constraints are supported. A static penalty model can be used by providing a Weight clause. In this case, if a constraint of the form \( A < B \) with weight \( w \) is violated, then the value added to the objective function is \( w \cdot (A - B) \). Alternatively, a likelihood-based model can be used by providing a Confidence clause. In this case, the contribution is the negative log probability of constraint satisfaction. The likelihood-based model should be used when statistically rigorous results are important, such as when performing Bayesian uncertainty quantification. It is not recommended to mix between the static penalty and likelihood models within the same fitting problem.

If neither a weight nor a confidence clause is provided, a static penalty model is assumed with a weight of 1.

#### 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 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:

\[ A < 5 \text{ at 6} \] or equivalently, \[ A < 5 \text{ at time=6} \]

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

\[ A < 5 \text{ at } B=6 \] - 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

\[ A < 5 \text{ at } B=6 \text{ everytime} \]

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

\[ A<5 \text{ at } B=6 \text{ first} \]

The *before* keyword moves the enforcement the last time point before the condition is met

\[ A<5 \text{ at } B=6 \text{ before} \]

- Enforce the inequality at the last time point before \( B=6 \).

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 \( B>=6 \text{ once} \)

It is also possible to write inequalities with two *at* keywords to compare observables at two different values of the independent variable:

\[ A < 5 \text{ at } B \text{ at } C=6 \] - Compares the value of \( A \) at time 5 to the value of \( B \) at the first time point such that \( C=6 \).

• *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.

\[ A < 5 \text{ between } 7, \ B=6 \] 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.

• *once between* - Require that the inequality be satisfied at at least one point within the specified time range. The syntax is the same as for a *between* constraint.

\[ A < 5 \text{ once between } 7, \ B=6 \] would require that \( A<7 \) at some point between time=7 and the first time after time=7 such that \( B=6 \).

The above definitions assume that time is the independent variable, but note that the same keywords can be used for parameter scans with a different independent variable.

### 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:

\[ A < 5 \text{ at 6 weight 2} \] - 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.

### 3.4.4 Confidence

A confidence clause can be provided instead of a weight clause to use a likelihood-based model to incorporate the constraint into the objective function. The clause consists of the confidence keyword, followed by a number, followed by the tolerance keyword, followed by a number.

Under this model, the inequality is rewritten in the form $g < 0$ for a function $g$. For example, in the constraint $A < B$ at time=5, $g = A(5) − B(5)$, and in the constraint $A > 5$ always, $g = 5 − \min(A)$ The tolerance represents the standard deviation of the quantity $g$, which is assumed to have a Gaussian distribution. The confidence represents the probability that the constraint should be satisfied by the model. With probability 1-confidence, there is assumed to be a model discrepancy, such that whether the constraint is satisfied is unrelated to the model or its parameters.

The value added to the objective function given confidence $conf$, tolerance $tol$, and $g$ as defined above is $-\log(conf + (1 − 2conf)\cdot cdf(g, tol, 0))$, where $cdf(\mu, \sigma, 0)$ is the cumulative distribution function of a Gaussian distribution with mean $\mu$ and standard deviation $\sigma$, evaluated at 0.

If tolerance is omitted, it is assumed to be 0, resulting in a step function.

The following examples illustrate the use of confidence clauses:

- **A < 5 at time = 4 confidence 0.98 tolerance 1** - The term added to the objective function would be $-\log(0.01 + 0.98 \cdot cdf(A(4) − 5, 1, 0))$

- **A > 5 always confidence 0.98** - Tolerance is assumed to be 0. The term added to the objective function would be $-\log(0.99)$ if $\min(A) > 5$ or $-\log(0.01)$ otherwise.

The keywords $pmin$ and $pmax$ may be used in place of confidence to specify different minimum and maximum probabilities of the constraint. In this case, the term added to the objective function is $-\log(p_{min} + (p_{max} − p_{min})\cdot cdf(g, tol, 0))$. For example

- **A < 5 at time = 4 $pmin$ 0.01 $pmax$ 0.98** - The term added to the objective function would be $-\log(0.01 + 0.97 \cdot cdf(A(4) − 5, 1, 0))$

### 3.4.5 Constraints involving multiple models

By default, observables in property files are assumed to come from the model that the .prop file is mapped to, and the simulation suffix matching the .prop file’s name (the same convention as for .exp files). However, it is possible to use “dot notation” to refer to observables in other simulations, as in the following example.

```bash
fit.conf:
model = model1.bngl : wt.exp
model = model2.bngl : mut.prop
```

mut.prop:

### 3.4. Property (BPSL) files

12
$A < wt.A$ always

In this example, the constraint would check that the value of $A$ in the simulation of model2 with suffix “mut” is less than the value of $A$ in the simulation of model1 with suffix “wt”. In this way, it is possible to write constraints involving the outputs of multiple models.

To use this feature, all simulation suffixes must be unique across all models. In addition all observables used in a single constraint must have the same independent variable with the same step size.
The following sections give all possible configuration keys that may be used in your .conf file to configure your fitting run. Each line of the .conf file sets the value of a configuration key with the general syntax:

\[
\text{key = value}
\]

### 4.1 Required Keys

**model** Specifies the mapping between model files (.bngl or .xml) and data files (.exp or .prop). Model paths and files are followed by a ‘:’ and then a comma-delimited list of experimental data files or property files corresponding to the model files. If no experimental files are associated with a model write `none` instead of a file path.

Examples:

- `model = path/to/model1.bngl : path/to/data1.exp`
- `model = path/to/model2.xml : path/to/data2.prop, path/to/data2.exp`
- `model = path/to/model3.xml : none`

**fit_type** The choice of fitting algorithm. Options:

- `de` - *Differential Evolution*
- `ade` - *Asynchronous Differential Evolution*
- `ss` - *Scatter Search*
- `pso` - *Particle Swarm Optimization*
- `mh` - *Metropolis-Hastings MCMC (Not recommended)*
- `sim` - *Simplex local search*
- `sa` - *Simulated Annealing (Not recommended)*
- `pt` - *Parallel tempering (Not recommended)*
• am - Adaptive MCMC
• check - Run *model checking* instead of fitting

Example:
• fit_type = de

**objfunc** Which *objective function* to use.

• chi_sq - Chi squared
• chi_sq_dynamic - Chi squared with sigma as a free parameter (Requires sigma__FREE in the model and the configuration file)
• neg_bin - Negative Binomial (Requires neg_bin_r set to a number in the configuration file i.e neg_bin_r = 2, Default = 24)
• neg_bin_dynamic - Negative Binomial with r as a free parameter (Requires r__FREE in the model and the configuration file)
• kl - Kullback-Leibler
• sos - Sum of squares
• sod - Sum of differences
• 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

Example:
• objfunc = chi_sq

**population_size** The number parameter sets to maintain in a single iteration of the algorithm. See algorithm descriptions for more information.

Example:
• population_size = 50

**max_iterations** Maximum number of iterations

Example:
• max_iterations = 200

### 4.2 Other Path Keys

**bng_command** 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 environmental variable

Example:
• bng_command = path/to/BNG2.pl

**output_dir** Directory where we should save the output.

Default: “pybnf_output”

Example:
4.3 Parameter and Model Specification

mutant  Declares a model that does not have its own model file, but instead is defined based on another model (the “base model”), changing only a small number of parameter values. The first word of the declaration gives the name of the base model (not including the path or .bngl/.xml extension). The second word is the name of the mutant model; this name is appended to the suffixes of the base model. That is, 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`. Following the name of the mutant model is a series of statements that specify how to change `basemodel` to make the mutant model. The statements have the format `[variable][operator][value]`; for example `a__FREE=0` or `b__FREE*2`. Supported operators are `=, +, -, *, /`.

Default: None

Example:

Elsewhere in your .conf file, you have specified `model1`:

```conf
• model = path/to/model1.bngl : data1.exp
```

Then you can use this key as follows:

```conf
• mutant = model1 no_a a__FREE=0 : data1no_a.exp, data2no_a.exp
• mutant = model1 extra_ab a__FREE*2 b__FREE*2 : data1extra_ab.exp
```

uniform_var  A bounded uniformly distributed variable defined by a 3-tuple corresponding to the variable name, minimum value, and maximum value. If the tag `U` is added to the end, the bounds are enforced only during initialization, not during fitting.

Examples:

```conf
• uniform_var = k__FREE 10 20
• uniform_var = k__FREE 10 20 U
```

normal_var  A normally distributed variable defined by a 3-tuple: the name, mean value, and standard deviation. The distribution is truncated at 0 to prevent negative values.

Example:

```conf
• normal_var = d__FREE 10 1
```

loguniform_var  A variable distributed uniformly in logarithmic space. The value syntax is identical to the uniform_var syntax.

Examples:

```conf
• loguniform_var = p__FREE 0.001 100
• loguniform_var = p__FREE 0.001 100 U
```

lognormal_var  A variable normally distributed in logarithmic space. The value syntax is a 3-tuple specifying the variable name, the base 10 logarithm of the mean, and the base 10 logarithm of the standard deviation.

Example:

```conf
• lognormal_var = l__FREE 1 0.1
```
The following two keys (\texttt{var} and \texttt{logvar}) 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 \texttt{refine}, you must set the simplex algorithm’s parameters with \texttt{simplex\_step} or \texttt{simplex\_log\_step}.

**\texttt{var}** The starting point for a free parameter. It is defined by a 3-tuple, corresponding to the variable’s name, its initial value and an initial step size (optional). If not specified, the initial step size defaults to the value specified by the simplex-specific parameter \texttt{simplex\_step} (see \texttt{simplex})

Examples:
- \texttt{var = k\_FREE 10}
- \texttt{var = d\_FREE 2 0.05}

**\texttt{logvar}** Syntax and semantics are identical to the \texttt{var} key above, but the initial value and initial step should be specified in base 10 logarithmic space.

Example:
- \texttt{logvar = k\_FREE -3 1}

### 4.4 Simulation Actions

These keys specify what simulations should be performed with the models. For SBML models, simulation actions are required. For BNGL models, the same information can be specified in the actions block of the BNGL file, so use of these keys is optional.

**\texttt{time\_course}** Run a time course simulation on the model. Specify a comma-delimited list of key:value pairs, with the following possible keys:

- \texttt{time}: The simulation time. Required.
- \texttt{suffix}: The suffix of the data file to save. You should map the model to a .exp file of the same name. Default: time\_course
- \texttt{step}: The simulation time step. Default: 1
- \texttt{model}: The name of the model to run (not including the path or .bngl/.xml extension). Default: All models in the fitting run.
- \texttt{subdivisions}: Only for use with \texttt{sbml\_integrator=euler}, specifies the number of internal Euler steps to perform between each output step specified by \texttt{step}. Default: 1
- \texttt{method} The simulation method to use. Default is \texttt{ode}. Options are:
  - \texttt{ode}: Numerical integration of differential equations
  - \texttt{ssa}: Stochastic simulation algorithm (BioNetGen’s “ssa” algorithm for BNGL models; Gillespie’s direct method for SBML models)
  - \texttt{pla}: Partitioned-leaping algorithm (BNGL models only)
  - \texttt{nf}: Network-free simulation with NFsim (BNGL models only)

Example:
- \texttt{time\_course = time:60, model:model1, suffix:data1}

**\texttt{param\_scan}** Run a parameter scan on the model. Specify a comma-delimited list of key:value pairs, with the following possible keys:

- \texttt{param}: Name of the parameter to scan. Required.
- \texttt{min}: Minimum value of the parameter. Required
• **max**: Maximum value of the parameter. Required.
• **step**: Change in the parameter value between consecutive simulations in the scan. Required.
• **time**: The simulation time. Required.
• **suffix**: The suffix of the data file to save. You should map the model to a .exp file of the same name. Default: param_scan
• **logspace**: If 1, take step to be in log (base 10) space, and scan the parameter in log (base 10) space. Default: 0
• **model**: The name of the model to run (not including the path or .bngl/.xml extension). Default: All models in the fitting run.
• **subdivisions**: Only for use with sbml_integrator=euler, specifies the number of internal Euler steps to perform for each simulation. Default: 1000
• **method**: The simulation method to use. Options are the same as in time_course. Default: ode

**Example:**
```
param_scan = param:x, min:1, max:1000, step:0.5, logspace:1, time:60, model:model1, suffix:datal
```

### 4.5 Parallel Computing

**parallel_count** The number of jobs to run in parallel. This may be set for both local and cluster fitting runs. For cluster runs, this number is divided by the number of available nodes (and rounded up) to determine the number of parallel jobs per node.

Default: Use all available cores. On a cluster, the number of available cores per node is determined by running multiprocessing.cpu_count() from the scheduler node.

**Example:**
```
parallel_count = 7
```

**cluster_type** 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 supports slurm or none.

Default: None (local fitting run).

**Example:**
```
cluster_type = slurm
```

**parallelize_models** For fitting jobs that include multiple models, run those models on different cores, utilizing a total of this number of cores per parameter set evaluation. Should not be set higher than the total number of models. Using this option incurs additional communication overhead, and causes the objective function to be evaluated locally, not in parallel. Therefore, only certain types of problems will benefit from this option.

Default: 1

**Example:**
```
parallelize_models = 3
```

**scheduler_file** Provide a scheduler file to link PyBNF to a Dask scheduler already created outside of PyBNF. See Manual configuration with Dask for more information. This option may also be specified on the command line with the -s flag.

Default: None
Example:

- scheduler_file = cluster.json

**scheduler_node** 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

Example:

- scheduler_node = cn180

**simulation_dir** Optional setting for a different directory where we should save (or temporarily store) simulation output. Usually not necessary to set separately from `output_dir`. However, if you are running on a cluster with a Lustre filesystem, you may want to set this to a different disk to avoid excessive reads and writes to the Lustre disk.

Default: Use the same directory as `output_dir`.

Example:

- simulation_dir = /scratch/sim_output

**worker_nodes** 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

Example:

- worker_nodes = cn102 cn104 cn10511

### 4.6 General Options

#### 4.6.1 Output Options

**delete_old_files** Takes an integer for a value. 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

Example:

- delete_old_files = 2

**num_to_output** The maximum number of parameter sets to output when writing the trajectory to file. The parameter sets are ordered by their corresponding objective function value to ensure the best fits are outputted.

Default: 5000

Example:

- num_to_output = 100000

**output_every** The number of iterations in between consecutive events writing the trajectory to file.

Default: 20

Example:
• output_every = 1000

**save_best_data** If 1, run an extra simulation at the end of fitting using the best-fit parameters, and save the best-fit .gdat and .scan files to the Results directory.

Default: 0

Example:

• save_best_data = 1

**verbosity** An integer value that 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

Example:

• verbosity = 0

### 4.6.2 Algorithm Options

**bootstrap** 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)

Example:

• bootstrap = 10

**bootstrap_max_obj** 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

Example:

• bootstrap_max_obj = 1.5

**constraint_scale** Scale all weights in all .prop files by this multiplicative factor. For convenience only - The same thing could be achieved by editing .prop files, but this option is useful for tuning the relative contributions of quantitative and qualitative data.

Default: 1 (no scaling)

Example:

• constraint_scale = 1.5

**ind_var_rounding** 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

Example:

• ind_var_rounding = 1

**initialization** 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.

Default: `lh`

Example:

```
initialization = rand
```

**local_objective_eval** If 1, evaluate the objective function locally, instead of parallelizing this calculation on the workers. This option is automatically enabled when using the `smoothing` or `parallelize_models` feature.

Default: 0 (unless smoothing is enabled)

Example:

```
local_objective_eval = 1
```

**min_objective** Stop fitting if an objective function lower than this value is reached.

Default: None; always run for the maximum iterations

Example:

```
min_objective = 0.01
```

**normalization** Indicates that simulation data must be normalized in order to compare with exp files. Specify one of the following types of normalization:

• `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 the type is followed by a ':' and a comma-delimited list of exp files, 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

Examples:

```
normalization = init

normalization = init:  data1.exp, data2.exp

normalization = init:  (data1.exp:  1,3-5), (data2.exp:  var1,var2)
```

**postprocess** Used to specify a custom Python script for postprocessing simulation results before evaluating the objective function. Specify the path to the Python script, followed by a list of all of the simulation suffixes for which that postprocessing script should be applied. For how to set up a postprocessing script, see *Custom Postprocessing*.

Default: No postprocessing

Example:

```
postprocess = path/to/script.py suff1 suff2
```
refine  If 1, after fitting is completed, refine the best fit parameter set by a local search with the simplex algorithm.

  Default: 0
  Example:
  • refine = 1

sbml_integrator  Which integrator to use for SBML models. Options are cvode, rk4, gillespie, or euler, and are described in the libroadrunner documentation. If your time_course or param_scan key specifies method: ssa, then gillespie is used for that action, overriding this setting.

  Default: cvode
  Example:
  • sbml_integrator = rk4

smoothing  Number of replicate runs to average together for each parameter set (useful for stochastic simulations).

  Default: 1
  Example:
  • smoothing = 2

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

  Default: 3600
  Example:
  • wall_time_gen = 600

wall_time_sim  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 only use it when needed.

  Default: 3600 for BNGL models; No limit for SMBL models
  Example:
  • wall_time_sim = 600

4.7 Algorithm-specific Options

4.7.1 Simplex

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

simplex_step  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
  Example:
  • simplex_step = 0.5

simplex_log_step  Equivalent of simplex_step, for variables that move in log space.

  Default: Value of simplex_step
  Example:
- `simplex_log_step` = 0.5

**simplex_reflection**  When we reflect a point through the centroid, what is the ratio of dilation on the other side?
- Default: 1.0
- Example:
  - `simplex_reflection` = 0.5

**simplex_expansion**  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
- Example:
  - `simplex_expansion` = 0.5

**simplex_contraction**  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
- Example:
  - `simplex_contraction` = 0.3

**simplex_shrink**  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
- Example:
  - `simplex_shrink` = 0.3

**simplex_max_iterations**  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.
- Example:
  - `simplex_max_iterations` = 20

**simplex_stop_tol**  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)
- Example:
  - `simplex_stop_tol` = 0.01

### 4.7.2 Differential Evolution

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

**mutation_rate**  When generating a new individual, mutate each parameter with this probability.
- Default: 0.5
- Example:
  - `mutation_rate` = 0.7
mutation_factor  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

Example:

- mutation_factor = 0.7

stop_tolerance  Stop the run if within the current population, \( max \_objective/min \_objective < 1 + \epsilon \), where \( \epsilon \) is the value of this key. This criterion triggers when the entire population has converged to roughly the same objective function value.

Default: 0.002

Example:

- stop_tolerance = 0.001

de_strategy  Specifies how new parameter sets are chosen. The following options are available:

- rand1
- rand2
- best1
- best2
- all1
- all2

The first part of the string determines which parameter set we mutate:

- 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 second part of the string specifies how we calculate the amount by which to mutate each parameter:

- 1 - Use 1 pair of other parameter sets: \((p_1 - p_2)\)
- 2 - Use 2 pairs of other parameter sets: \((p_1 - p_2 + p_3 - p_4)\).

Default: rand1

Example:

- de_strategy = rand2

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  Number of separate populations to evolve.

Default: 1

Example:

- islands = 2

migrate_every  After this number of generations, migrate some individuals between islands.

Default: 20 (but Infinity if islands = 1)

Example:
migrate_every = 10

**num_to_migrate**  How many individuals to migrate off of each island during migration.

Default: 3
Example:

• num_to_migrate = 5

### 4.7.3 Scatter Search

**init_size**  Number of parameter sets to test to generate the initial population.

Default: 10 \* number of parameters
Example:

• init_size = 100

**local_min_limit**  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
Example:

• local_min_limit = 10

**reserve_size**  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: Value of max_iterations
Example:

• reserve_size = 100

### 4.7.4 Particle Swarm

**cognitive**  Acceleration toward a particle’s own best fit

Default: 1.5
Example:

• cognitive = 1.7

**social**  Acceleration toward the global best fit

Default: 1.5
Example:

• social = 1.7

**particle_weight**  Inertia weight of particle. A value less than 1 can be thought of as friction that continuously decelerates the particle.

Default: 0.7
Example:

• particle_weight = 0.9
v_stop  Stop the algorithm if the speeds of all parameters in all particles are less than this value.

   Default: 0 (don’t use this criterion)
   Example:
      • v_stop = 0.01

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 weight changing.

   Default: the value of particle_weight, effectively disabling this feature.
   Example:
      • particle_weight_final = 0.5

adaptive_n_max  After this many “unproductive” iterations, we have moved halfway from the initial weight to the final weight.

   Default: 30
   Example:
      • adaptive_n_max = 20

adaptive_n_stop  After this many “unproductive” iterations, stop the fitting run.

   Default: Inf
   Example:
      • adaptive_n_stop = 50

adaptive_abs_tol  Parameter for checking if an iteration was “unproductive”

   Default: 0
   Example:
      • adaptive_abs_tol = 0.01

adaptive_rel_tol  Parameter for checking if an iteration was “unproductive”

   Default: 0
   Example:
      • adaptive_rel_tol = 0.01

4.7.5 Bayesian Algorithms (mh, pt, sa)

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

For all Bayesian algorithms

step_size  When proposing a Monte Carlo step, the step in n-dimensional parameter space has this length.

   Default: 0.2
   Example:
- `step_size = 0.5`

**beta** 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 `mh`, 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 contribute 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.

Default: 1

Examples:
- `beta = 0.9`
- `beta = 0.7 0.8 0.9 1`

**For all Bayesian algorithms except sa**

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

Example:
- `sample_every = 20`

**burn_in** Don't sample for this many iterations at the start, to let the system equilibrate.

Default: 10000

Example:
- `burn_in = 1000`

**output_hist_every** Every `x` samples (i.e every `x*sample_every` iterations), save a histogram file for each parameter, 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

Example:
- `output_hist_every = 10`

**hist_bins** Number of bins used when writing the histogram files.

Default: 10

Example:
- `hist_bins = 20`

**credible_intervals** Specify one or more numbers here. For each `n`, the algorithm will save a file giving bounds for each parameter such that in `n%` of the samples, the parameter lies within the bounds.

Default: 68 95

Examples:
- `credible_intervals = 95`
- `credible_intervals = 20 68 95`
For Simulated Annealing

**beta_max**  Stop the algorithm if all replicates reach this beta (1/temperature) value.
   Default: Infinity (don’t use this stop criterion)
   Example:
   • beta_max = 1.5

**cooling**  Each time a move to a higher energy state is accepted, increase beta (1/temperature) by this value.
   Default: 0.01
   Example:
   • cooling = 0.001

For Parallel Tempering

**exchange_every**  Every x iterations, perform replica exchange, swapping replicas that are adjacent in temperature with a statistically correct probability
   Default: 20
   Example:
   • exchange_every = 10

**reps_per_beta**  How many identical replicas to run at each temperature. Must be a divisor of population_size.
   Default: 1
   Example:
   • reps_per_beta = 5

**beta_range**  As an alternative to setting beta, the range of values of beta to use. Specify the minimum value, followed by the maximum value. The replicates will use population_size/reps_per_beta geometrically spaced beta values within this range. Only the replicas at the max beta value will be sampled. For the true probability distribution, the maximum value should be 1.
   Default: None (betas are set with the beta key)
   Example:
   • beta_range = 0.5 1

For Adaptive MCMC

**stablizingCov**  Stabilize the covariant matrix of the proposal.
   Default: 0.001
   Example:
   • stablizingCov = 0.1

**adaptive**  The number of iterations that the simulation will spend collecting data to observe the data for calculation of the differential matrix.
   Default: 10000
   Example:
• adaptive = 50000

**output_noise_trajectory (Only for use with neg_bin and neg_bin_dynamic functions)** Calculate and add the negative binomial noise to the specified observables or functions then save the output of the user defined observable or function from the simulation output to a .txt file.

Default: None (multiple values can be defined separated by a comma)

Note: output_trajectory and output_noise_trajectory can both be declared in the same configuration file but may result in slower performance

Example:

- output_noise_trajectory = ObservableA
- output_noise_trajectory = ObservableA, ObservableB, FunctionA

**output_trajectory** Save the output of the user defined observable or function from the simulation output to a .txt file.

Default: None (multiple values can be defined separated by a comma)

Example:

- output_trajectory = ObservableA
- output_trajectory = ObservableA, ObservableB, FunctionA

**continue_run** When set to 1 the chains began at the MAP parameters, calculated covariance matrix, and diffusivity from the previous chain.

Default: 0

Example:

- continue_run = 1

**calculate_covari** Calculate the covariance matrix of a defined segment of the previous run

Default: None

Example:

- calculate_covari = 1 50000

**starting_params** Seed the run from a defined set of starting parameters listed in the same order they are defined with a space separating each value in the order they are listed as free parameters in the configuration file

Default: None

Example:

- starting_params = 5.5 2 3
5.1 Summary of Available Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Class</th>
<th>Parallelization</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Differential Evolution</td>
<td>Population-based</td>
<td>Synchronous or Asynchronous</td>
<td>General-purpose parameter fitting</td>
</tr>
<tr>
<td>Scatter Search</td>
<td>Population-based</td>
<td>Synchronous</td>
<td>General-purpose parameter fitting, especially difficult problems with high dimensions or many local minima</td>
</tr>
<tr>
<td>Particle Swarm</td>
<td>Population-based</td>
<td>Asynchronous</td>
<td>Fitting models with high variability in runtime</td>
</tr>
<tr>
<td>Metropolis-Hastings MCMC</td>
<td>Metropolis sampling</td>
<td>Independent Markov Chains</td>
<td>Finding probability distributions of parameters (deprecated)</td>
</tr>
<tr>
<td>Simulated Annealing</td>
<td>Metropolis sampling</td>
<td>Independent Markov Chains</td>
<td>Problem-specific applications (deprecated)</td>
</tr>
<tr>
<td>Parallel Tempering</td>
<td>Metropolis sampling</td>
<td>Synchronized Markov Chains</td>
<td>Finding probability distributions in challenging probability landscapes (deprecated)</td>
</tr>
<tr>
<td>Simplex</td>
<td>Local search</td>
<td>Synchronous</td>
<td>Local optimization, or refinement of a result from another algorithm.</td>
</tr>
<tr>
<td>Adaptive MCMC</td>
<td>Metropolis sampling</td>
<td>Independent Markov Chains</td>
<td>Finding probability distributions in challenging probability landscapes</td>
</tr>
</tbody>
</table>

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 the objfunc key. The following options are available. Note that \( y_i \) are the experimental data points and \( a_i \) are the simulated data points. The summation is over all experimental data points.

- Chi squared (objfunc = chi_sq): \( f(y, a) = \sum_i \frac{(y_i - a_i)^2}{\sigma_i^2} \), where \( \sigma_i \) is the standard deviation of point \( y_i \), which must be specified in the exp file.
- Sum of squares (objfunc = sos): \( f(y, a) = \sum_i (y_i - a_i)^2 \)
- Sum of differences (objfunc = sod): \( f(y, a) = \sum_i |y_i - a_i| \)
- Normalized sum of squares (objfunc = norm_sos): \( f(y, a) = \sum_i \frac{(y_i - a_i)^2}{y_i^2} \)
- Average-normalized sum of squares (objfunc = ave_norm_sos): \( f(y, a) = \sum_i \frac{(y_i - a_i)^2}{\bar{y}^2} \), 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 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 synchronous 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 a good general-purpose algorithm.
The asynchronous version has similar advantages to Particle Swarm.

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 (~ 9-18). 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 $\alpha = -1$ if $h_i > p_i$ or $1$ if $p_i < h_i$, let $\beta = (|h_i - p_i| - 1)/(n - 2)$, let $d = helper[P] - parent[P]$ for some parameter P.

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

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 $v_{i+1} = w \cdot v_i + c_1 \cdot u_1 \cdot (x_i - x_{\text{min}}) + c_2 \cdot u_2 \cdot (x_i - x_{\text{globalmin}})$. 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, $x_{\text{min}}$ 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 description, $w$ 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 $w = particle_weight + (particle_weight_final - particle_weight) \cdot N / (N + adaptive_n_max)$. 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

Particle swarm optimization tends to be the best-performing algorithm for problems that benefit from asynchronicity, namely, models for which the runtime per simulation can vary greatly depending on the parameter set. Models simulated by SSA or NFsim often fall into this category. In these cases, synchronous algorithms would cause some cores...
to remain idle while slow-running simulations complete on other cores, whereas asynchronous algorithms like particle swarm allow you to use all cores at all times.

### 5.6 Metropolis-Hastings MCMC

#### 5.6.1 Algorithm

Metropolis-Hastings Markov chain Monte Carlo (MCMC) is a Bayesian sampling method. The free parameters are taken to be random variables with unknown probability distributions, and the algorithm samples points in parameter space are sampled with a frequency proportional to the probability of the parameter set given the data. The result is a probability distribution over parameter space that expresses the probability 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 this algorithm in PyBNF, it is assumed that the objective function is a **likelihood** function, that is, the objective function value gives the negative log probability of the data given the parameter set. There is a solid mathematical basis for making this assumption when the **chi-squared objective function** is used. It is not recommended to use this algorithm with different objective function, or an objective function that includes qualitative data.

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

Metropolis-Hastings 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 negative log 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 $\Delta F$ is the change in the posterior, and $\beta$ represents the inverse “temperature” at which the Metropolis sampling occurs. To generate the
true posterior distribution, $\beta$ should be set to 1. The sampled distribution becomes more broad with smaller $\beta$ and more narrow with a larger $\beta$.

### 5.6.4 Applications

Metropolis-Hastings 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 simultaneously, 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 $\beta$ 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 (MCMC). 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 Metropolis-Hastings MCMC.

Like Metropolis-Hastings, it is recommended to only use parallel tempering with the **chi-squared objective function**.

### 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 [Gupta2018a]. Markov chains are run by the same method as in Metropolis-Hastings, except that the value of $\beta$ 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 $\beta = 1/\text{Temperature}$, and $\Delta F$ 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 $\beta$) 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 $\beta$s 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 conventional Metropolis-Hastings MCMC, the goal of parallel tempering is to provide a distribution of possible parameter values rather than a single point estimate.

Compared to Metropolis-Hastings MCMC, 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 Metropolis-Hastings 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 Adaptive MCMC

#### 5.9.1 Algorithm

Adaptive Markov chain Monte Carlo (MCMC) is a Bayesian sampling method. The Bayesian method is described in further detail under the Metropolis-Hasting definition. When running this algorithm in PyBNF, it is assumed that the objective function is a likelihood function, that is, the objective function value gives the negative log probability of the data given the parameter set. It is recommended that the Chi-squared or negative binomial or their dynamic versions are used with this algorithm. The output when using the adaptive MCMC in PyBNF consists of the posterior from each chain and a file containing parameters from all chains saved in the ‘/Results/AMCMC/Run’ folder.
5.9.2 Parallelization

Like Metropolis-Hastings, Adaptive MCMC is not an inherently parallel algorithm. Like the Metropolis-Hastings algorithm PyBNF supports running several independent Markov chains using the populations_size key. Note as stated in the Metropolis-Hastings description 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.

5.9.3 Implementation details

The implementation algorithm can be found in Andrieu and Thoms, Stat Comput 18: 343–373 (2008). The algorithm uses a random walk like MCMC during the training period. While in the training period data is collected to determine the covariance of the posterior. Once the training phase is completed real time on the fly calculation of the diffusivity and covariance is performed for the remaining iterations. Note: as stated in the Metropolis-Hastings description 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.

5.10 Simplex

5.10.1 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.

5.10.2 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.

5.10.3 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 $d \times \text{simplex_reflect}$ 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 $d \times \text{simplex_expansion}$ away from the hyperplane ($P_{21}$).
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 $d \times \text{simplex_contraction}$ 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 $\text{simplex_contract} \times P_0 + (1 - \text{simplex_contract}) \times P$, where $P_0$ is the best point in the simplex.

### 5.10.4 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.
CHAPTER 6

Advanced Features

6.1 Model Checking

PyBNF includes a model checking utility that evaluates how well an already parameterized model agrees with the given experimental data. To use this feature, set the fit_type config key to check. PyBNF will run a single simulation on a single core, and output the objective function value to the terminal. For problems containing constraints (.prop files), PyBNF will also output the total number of constraints that are satisfied. Finally, for each input property (.prop) file, PyBNF will output a text file named <.prop file name>_constraint_eval.txt that itemizes the penalties to the constraints: each line of the text file gives the penalty associated with the corresponding line of the .prop file.

Note that for model checking, input models should not contain any free parameters tagged with __FREE; all parameters should already be defined.

6.2 Bootstrapping

Bootstrapping is a method of uncertainty quantification in which fitting is repeated several times with random subsets of the data. PyBNF can be configured to perform bootstrapping by setting the bootstrap config key to a value equal to the number of bootstrap replicates.

After the initial fitting run completes, PyBNF will repeat the fitting run the specified number of times. For each of these bootstrap replicates, a different random sample of the experimental data is used. For an exp file with \( n \) data points, the random sample consists of \( n \) points sampled with replacement, such that some points are used multiple times and others are unused.

Note that the random sampling is performed at the level of exp files. For example, if you have data1.exp with 20 data points and data2.exp with 5 data points, each random sample will contain 20 points from data1.exp and 5 points from data2.exp. However, if data1.exp contains 10 points for observable A and 10 points for observable B, the random sample might contain an unequal number of A points and B points. Also note that property files are not sampled in bootstrapping; all bootstrap replicates enforce all constraints.

If the config key bootstrap_max_obj is set, then each bootstrap replicate must achieve the specified objective value. If a bootstrap replicate completes fitting with a larger objective value, then the replicate is discarded and a new replicate is run.
PyBNF will output additional files describing the bootstrap results. Each bootstrap replicate will have its own Simulations and Results folders. The Results folder will contain extra files of the form `<suffix>_weights_<replicate>.txt` that indicate which random sample of the data was used for this bootstrap replicate. The main Results folder will contain the file `bootstrapped_parameter_sets.txt`, which contains the best-fit parameter set from each bootstrap replicate, and can be used to calculate confidence intervals for each parameter.

### 6.3 Custom Postprocessing

PyBNF provides an interface for custom Python scripts to postprocess simulation results. For example, you might want to perform curve fitting on one of your simulation outputs, or normalize your simulation data by an advanced method not offered in the PyBNF code base. Postprocessing scripts are configured with the `postprocess` key which specifies the path to the custom script, and a list of suffixes of which simulations should be fed to the script for postprocessing.

Your custom script should be a Python file that defines the function `postprocess(simdata)`. The function's argument `simdata` is a PyBNF `Data` object containing the simulation data. The function should return a modified `Data` object.

The `Data` object is essentially a container for an array containing data from a .gdat or .scan file. The syntax `simdata['A']` can be used to access and modify the column of data corresponding to observable `A`. The field `simdata.data` contains the array itself, which can be accessed and modified like a normal numpy array. Each column of the array gives the value of one variable over the course of the time course or parameter scan, with column 0 corresponding to the independent variable. The mapping of observable names to column indices is stored in the dictionary `simdata.cols`, and the reverse mapping from column indices to observable names is stored in `simdata.headers`. For example:

```python
def postprocess(simdata):
    simdata.data # a 2D numpy array containing the data
    simdata['A'] # a 1D numpy array containing the output for observable A
    simdata.cols['A'] # The column number corresponding to the observable A
    simdata.headers[3] # The observable name corresponding to column 3
    simdata.data[3,0] = 42. # At the 3rd data point, set the independent variable to a value of 42.
```

It is also possible to create and return an entirely new `Data` object, replacing the original one for evaluation by the objective function. Import the `data` module with `from pybnf import data`, and then use the constructor `data.Data()`. Be sure to set the `data`, `cols`, and `headers` fields in your new object.

The following example postprocessing script could be used to normalize the observable `A` such that it has a mean of 0, but retains its original standard deviation:

```python
import numpy as np
def postprocess(simdata):
    a_data = simdata['A']
    a_data -= np.mean(a_data)
    simdata['A'] = a_data

    return simdata
```
6.3.1 Debugging scripts

PyBNF will import and execute your script during runtime. If something goes wrong, PyBNF will report an unknown error occurred, or a simulation failed with an unknown error. Tracebacks for any errors will be saved in the PyBNF log file.
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)*.

### 7.1 Implementation

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

```python
class NewAlgorithm(Algorithm):
    def __init__(self, config, **kwargs):
        super(NewAlgorithm, self).__init__(config)
        # Other setup that may involve additional arguments

    def my_custom_function(self):
        # User defined support function
        ...
```

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 is called at the start of the fitting run. It must return a list of PSet instances, as the first batch of parameter sets to be evaluated. The Algorithm superclass functions `random_pset` and `random_latin_hypercube_psets` may be useful:

```python
def start_run(self):
    return self.random_latin_hypercube_psets(self.population_size)
```

The `got_result` method is called each time an evaluation of a PSet is completed on a worker. It 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). For example:
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
    else:
        return []  # Waiting for synchronization

Four additional support methods in the Algorithm superclass may optionally be overridden, depending on the details of the new algorithm, such that the new algorithm is compatible with all features of PyBNF.

- `add_iterations(self,n)` is required to support adding extra iterations with the \(-r\) flag. This method should add \(n\) iterations to the algorithm’s maximum iteration count. The superclass implementation simply adds \(n\) to the attribute `self.max_iterations`. You should override the method if your algorithm tracks iteration count in a different way.

- `reset(self, bootstrap)` is required to support bootstrapping. This method should call the superclass method, and then reset the state of the algorithm so that another fitting replicate can be run.

- `get_backup_every(self)` helps choose when to save a backup of the algorithm. This method should return an integer telling after how many individual simulations we should back up the algorithm. The superclass implementation uses a formula that should work in most cases, but you can override this depending on details of your algorithm.

- `cleanup(self)` is used to clean up after an error. This method is called just before PyBNF exits due to an error or keyboard interrupt, and may be used to save any useful files before exiting.

### 7.2 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.

### 7.3 Pull requests

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

Installation of PyBNF on a cluster has the same requirements as installation on a workstation, namely Python 3 with the pip package manager. This is available on most clusters, but may require loading a module to access. In Slurm, you can view the available modules with the command `module avail`, and load the appropriate one with `module load [modulename]`. Once Python 3 and pip are loaded, the same installation instructions apply as for a standard installation. Assistance from the cluster administrators may be helpful if any cluster-specific issues arise during installation.

### 8.1 SLURM

The user may run PyBNF interactively or as a batch job using the `salloc` or `sbatch` commands respectively.

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.

#### 8.1.1 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 the appropriate Python environment

Initiate a PyBNF fitting run, including the flag `-t slurm`
8.1.2 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.

8.1.3 Troubleshooting: SSH access to nodes

The above instructions assume that PyBNF can access all allocated nodes via SSH. For some clusters, additional configuration is necessary to enable SSH access: use `ssh_keygen` (documented in many places, such as here, or here for instructions specific to PyBNF’s Dask scheduler) to set up SSH keys.

To confirm that SSH keys are set up correctly, make sure that you are able to SSH into all allocated nodes without needing to enter a password.

If SSH access is not possible on your cluster, you will have to use Manual configuration with Dask.

8.2 TORQUE/PBS

Not yet implemented. Please refer to Manual configuration below

8.3 Manual configuration with node names

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.

8.4 Manual configuration with Dask

PyBNF uses Dask.distributed to manage cluster computing. In most cases, it is not necessary for the user to interact directly with Dask. However, if PyBNF’s automatic Dask setup is unsatisfactory, then the instructions in this section may be helpful to set up Dask manually.

In the automatic PyBNF setup, the command `dask-ssh` is run on one of the available nodes (which becomes the scheduler node), with all available nodes as arguments (which become the worker nodes). `dask-ssh` is run with `--nthreads 1` and `--nprocs` equal to the number of available cores per node. The default number of available processes per core is the value returned by `multiprocessing.cpu_count()`: this default can be overridden by specifying the `parallel_count` key equal to the total number of processes over all nodes. This entire automatic setup with `dask-ssh` can be overridden as described below. If overriding the automatic setup, it is recommended to keep `nthreads` equal to 1 for SBML models because the SBML simulator is not thread safe.
For manual configuration, you will need to run the series of commands described below. All of these commands must remain running during the entire PyBNF run. Utilities such as `nohup` or `screen` are helpful for keeping multiple commands running at once.

To begin, run the command `dask-scheduler` on the node you want to use as the scheduler. Pass the argument `--scheduler-file` to create a JSON-encoded text file containing connection information. For example:

```
dask-scheduler --scheduler-file cluster.json
```

On each node you want to use as a worker, run the command `dask-worker`. Pass the scheduler file, and also specify the number of processes and threads per process to use on that worker. For example:

```
dask-worker --scheduler-file cluster.json --nprocs 32 --nthreads 1
```

Finally, run PyBNF, and pass PyBNF the scheduler file using the `-s` command line argument or the `scheduler_file` configuration key:

```
pybnf -c fit.conf -s cluster.json
```

For additional `dask-scheduler` and `dask-worker` options, refer to the Dask.distributed documentation.

### 8.5 (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`)
The PyBNF GitHub repository 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 / property 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, cd into the examples/demo directory and run `pybnf -c demo_bng.conf`. Results will be saved in a directory called “output” inside the examples/demo 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*

Even more examples are available on RuleHub.

### 9.1 List of examples included in PyBNF

#### 9.1.1 constraint_advanced

A demonstration of the features of PyBNF’s Biological Property Specification Language (BPSL). The model consists of simple fourth-order polynomial functions. The property files contain constraints of various forms, showcasing the available capabilities for constraint handling in PyBNF. All properties are consistent with a known ground truth for the model, so it should be possible to fit with a very low final objective value.

#### 9.1.2 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].
9.1.3 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 qualitative and quantitative data. This is the same problem described in Fig. 2 of [Mitra2018].

9.1.4 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 Metropolis Hastings 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: Metropolis Hastings, 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.

9.1.5 demo

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

9.1.6 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 advantage of asynchronicity in PSO and ADE).

9.1.7 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]).

9.1.8 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]).
9.1.9 fceri_gamma

A benchmark rule-based model of IgE-FcεRI signaling, originally published in [Faeder2003] and adapted in [Sneddon2011]. The BNGL file was provided 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.

9.1.10 igf1r

A model if IGF1R interaction with IGF, originally published and fit with BioNetFit 1 in [Erickson2019]. 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 [Erickson2019].

9.1.11 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.

9.1.12 receptor

A simple ligand-receptor model fit using synthetic data.

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

9.1.13 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]).

9.1.14 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.

9.1.15 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.

9.1.16 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 property 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 libRoad-Runner/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.

9.2 Index of examples by attribute

9.2.1 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, raf_sbml
- Difficult (Recommended on a cluster only): tcr, tlbr, yeast_cell_cycle

9.2.2 Examples by source

- Novel fits described in the PyBNF paper: 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

9.2.3 Examples by data/model types

- Property (.prop) data files: constraint_demo, constraint_raf, constraint_advanced, yeast_cell_cycle
- SBML models: raf_sbml, yeast_cell_cycle
- Multiple data files: degranulation
- Mutant models: yeast_cell_cycle
9.2.4 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*
10.1 Failed simulations

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

10.1.1 Check the simulation logs

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.

By default, PyBNF saves logs of roughly the first 10 failed simulations encountered. If PyBNF is run with the -d flag, logs from all failed simulations will be saved. If the fit was run with delete_old_files=0 in the config file, all logs can be found in the appropriate folders in the Simulations/ directory.

10.1.2 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.

### 10.1.3 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.

It may be possible to run such simulations with a different SBML integrator, set with the sbml_integrator key.

#### 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.

### 10.2 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.

### 10.3 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 \texttt{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 \textit{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.

10.4 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.

\textbf{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.

\textbf{Remedies:} You can check the limit of open files per program on the command line: \texttt{ulimit -n} gives you the “soft” limit, and \texttt{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 \texttt{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 \texttt{num_parallel} or \texttt{population_size} settings.

10.5 Too many threads

This error can come up in parallelized runs in which simulations are very fast. Similar to the \textit{Too many open files} error, it occurs when PyBNF exceeds the number of threads allowed by the system for a single user.

You can check the thread limit on the command line with \texttt{ulimit -u}. Many operating systems have this limit very high (over 100,000), but if yours has it set on the order of 4096, it could cause this error.

We recommend having an administrator with root access increase your default thread limit on the machine. Edit the file \texttt{/etc/security/limits.conf} and add the lines:

\begin{verbatim}
username soft nproc 100000
username hard nproc 100000
\end{verbatim}

where \texttt{username} is your user name, and \texttt{100000} is the new thread limit (use any reasonably large value). Restart the system for the changes to take effect.

We do not recommend increasing the thread limit via the command line as in \textit{Too many open files}: This change would only affect the current terminal, so although PyBNF could keep running, the rest of your system would become unresponsive after the original limit was exceeded.

10.4. Too many open files
10.6 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.

10.7 PyBNF has encountered a Fatel error

This error occurs when the scheduler loses connection with the cluster. The simulation data is generally backed up and the simulation can be resumed from the point it exited using the -r flag ‘pybnf -c .conf -r’.

10.8 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/lanl/PyBNF/), it will be helpful for us if you include the debug log file with your bug report.

10.9 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/lanl/PyBNF/.
CHAPTER 11

PyBNF Module Reference

Detailed documentation of the PyBNF code base for advanced users

11.1 PyBNF Module References

11.1.1 PyBNF entry point (pybnf.pybnf)
11.1.2 PyBNF algorithms (pybnf.algorithms)
11.1.3 PyBNF cluster setup (pybnf.cluster)
11.1.4 PyBNF configuration (pybnf.config)
11.1.5 PyBNF constraint specification (pybnf.constraint)
11.1.6 PyBNF data container (pybnf.data)
11.1.7 PyBNF objective functions (pybnf.objective)
11.1.8 PyBNF configuration parsing (pybnf.parse)
11.1.9 PyBNF printing functions (pybnf.printing)
11.1.10 PyBNF model and parameter containers (pybnf.pset)
Bibliography


---

**Bibliography**