Bumps: Curve Fitting and Uncertainty Analysis

Release 0.9.0

Paul Kienzle

Sep 21, 2022
# Contents

## 1 Getting Started
- 1.1 Installing the application ............................................. 1
- 1.2 Server installation ..................................................... 6
- 1.3 Contributing Changes ............................................... 9
- 1.4 License ................................................................... 9
- 1.5 Credits ................................................................... 11

## 2 Tutorial
- 2.1 Simple functions ...................................................... 13
- 2.2 Peak Fitting .......................................................... 24
- 2.3 Test functions ........................................................ 24
- 2.4 Check the entropy calculator ...................................... 28
- 2.5 Bayesian Experimental Design ................................. 30
- 2.6 Calling fit from scripts .............................................. 31
- 2.7 Inequality constraints .............................................. 32

## 3 User’s Guide
- 3.1 Using Bumps .......................................................... 36
- 3.2 Data Representation ............................................... 37
- 3.3 Experiment ............................................................. 37
- 3.4 Parameters ............................................................ 40
- 3.5 Fitting ................................................................. 42
- 3.6 Optimizer Selection ................................................ 52
- 3.7 Bumps Options ...................................................... 69
- 3.8 Calculating Entropy ................................................ 77

## 4 Reference: bumps
- 4.1 bounds - Parameter constraints ................................. 79
- 4.2 bspline - B-Spline interpolation library ....................... 89
- 4.3 cheby - Freeform - Chebyshev .................................. 90
- 4.4 cli - Command line interface ..................................... 91
- 4.5 curve - Model a fit function ...................................... 93
- 4.6 data - Data handling utilities .................................... 95
- 4.7 errplot - Plot sample profile uncertainty .................... 96
- 4.8 fitproblem - Interface between models and fitters ........ 97
- 4.9 fitservice - Remote job plugin for fit jobs .................. 104
- 4.10 fitters - Wrappers for various optimization algorithms 104
4.11 formatnum - Format numbers and uncertainties ........................................... 111
4.12 history - Optimizer evaluation trace ......................................................... 112
4.13 initpop - Population initialization strategies ............................................... 114
4.14 lsqerror - Least squares error analysis .................................................... 115
4.15 mapper - Parallel processing implementations ............................................ 118
4.16 monitor - Monitor fit progress ................................................................. 119
4.17 mono - Freeform - Monotonic Spline ....................................................... 120
4.18 names - External interface .................................................................... 121
4.19 options - Command line options processor .............................................. 121
4.20 parameter - Optimization parameter definition ......................................... 124
4.21 partemp - Parallel tempering optimizer .................................................... 146
4.22 pdfwrapper - Model a probability density function .................................... 147
4.23 plotutil - Plotting utilities ................................................................. 149
4.24 plugin - Domain branding ................................................................. 151
4.25 pmath - Parametric versions of standard functions .................................... 152
4.26 pymcfit - Wrapper for pyMC models ....................................................... 154
4.27 quasinewton - BFGS quasi-newton optimizer ........................................... 155
4.28 random_lines - Random lines and particle swarm optimizers .................. 156
4.29 simplex - Nelder-Mead simplex optimizer (amoeba) ................................ 158
4.30 util - Miscellaneous functions ................................................................ 159
4.31 wsolve - Weighted linear and polynomial solver with uncertainty ............ 161

5 Reference: bumps.dream ................................................................. 165
  5.1 acr - A C Rencher normal outlier test ...................................................... 165
  5.2 bounds - Bounds handling ................................................................... 166
  5.3 core - DREAM core .............................................................................. 168
  5.4 corrplot - Correlation plots ................................................................... 170
  5.5 crossover - Adaptive crossover support ................................................. 171
  5.6 diffev - Differential evolution MCMC stepper ........................................ 173
  5.7 entropy - Entropy calculation .............................................................. 173
  5.8 exppow - Exponential power density parameter calculator ...................... 176
  5.9 formatnum - Format values and uncertainties nicely for printing ............ 176
  5.10 gelman - R-statistic convergence test ..................................................... 177
  5.11 geweke - Geweke convergence test ....................................................... 177
  5.12 initpop - Population initialization routines .............................................. 178
  5.13 ksmirnov - Kolmogorov-Smirnov test for MCMC convergence .............. 178
  5.14 mahal - Mahalanobis distance calculator ............................................... 179
  5.15 metropolis - MCMC step acceptance test ............................................... 179
  5.16 model - MCMC model types .............................................................. 179
  5.17 outliers - Chain outlier tests ............................................................... 183
  5.18 state - Sampling history for MCMC ...................................................... 183
  5.19 stats - Statistics helper functions .......................................................... 188
  5.20 tile - Split a rectangle into n panes ...................................................... 189
  5.21 util - Miscellaneous utilities ............................................................... 189
  5.22 varplot - Plot histograms for individual parameters ............................... 189
  5.23 views - MCMC plotting methods ............................................................ 190

Python Module Index .................................................. 193

Index ............................................................................ 195
Bumps is a set of routines for curve fitting and uncertainty analysis from a Bayesian perspective. In addition to traditional optimizers which search for the best minimum they can find in the search space, Bumps provides uncertainty analysis which explores all viable minima and finds confidence intervals on the parameters based on uncertainty in the measured values. Bumps has been used for systems of up to 100 parameters with tight constraints on the parameters. Full uncertainty analysis requires hundreds of thousands of function evaluations, which is only feasible for cheap functions, systems with many processors, or lots of patience.

Bumps includes several traditional local optimizers such as Nelder-Mead simplex, BFGS and differential evolution. Bumps uncertainty analysis uses Markov chain Monte Carlo to explore the parameter space. Although it was created for curve fitting problems, Bumps can explore any probability density function, such as those defined by PyMC. In particular, the bumps uncertainty analysis works well with correlated parameters.

Bumps can be used as a library within your own applications, or as a framework for fitting, complete with a graphical user interface to manage your models.

1.1 Installing the application

- Building from source
  - Windows
  - Linux
  - OS/X
- Fast Stepper for DREAM on MPI
- Building Documentation
- Windows Installer

Bumps 0.9.0 is provided as a Windows installer or as source:
The Windows installer walks through the steps of setting the program up to run on your machine and provides the sample data to be used in the tutorial.

### 1.1.1 Building from source

Before building bumps, you will need to set up your python environment. We depend on many external packages. The versions listed below are a snapshot of a configuration that we are using. The program may work with older versions of the package, and we will try to keep it compatible with the latest versions.

Our base scientific python environment contains:

- python 2.7 (also tested on 2.6 and 3.5)
- matplotlib 1.4.3
- numpy 1.9.2
- scipy 0.14.0
- wxPython 3.0.0.0
- setuptools 20.1.1

To run tests we use:

- nose 1.3.0

To build the HTML documentation we use:

- sphinx 1.3.1
- docutils 0.12
- jinja2 2.8

The PDF documentation requires a working LaTeX installation.

You can install directly from PyPI using pip:

```
pip install bumps
```

If this fails, then follow the instructions to install from the source archive directly. Platform specific details for setting up your environment are given below.

### Windows

There are a number of python environments for windows, including:

- Anaconda
- Canopy
- Python(X,Y)
- WinPython
You can also build your environment from the individually distributed python packages.

You may want a C compiler to speed up parts of bumps. Microsoft Visual C++ for Python 2.7 is one option. Once it is installed, you will need to enable the compiler using vcvarsall 64.

Alternatively, your python environment may supply the MinGW C/C++ compiler, but fail to set it as the default compiler. To do so you will need to create distutils configuration file in the python lib directory (usually C:\Python27\Lib\distutils\distutils.cfg) with the following content:

```
[build]
compiler=mingw32
```

Next start a Windows command prompt in the directory containing the source. This will be a command like the following:

```
cd "C:\Documents and Settings\<username>\My Documents\bumps-src"
```

Now type the command to build and install:

```
python setup.py install
python test.py
```

Now change to your data directory:

```
cd "C:\Documents and Settings\<username>\My Documents\data"
```

To run the program use:

```
python -m bumps.cli -h
```

**Linux**

Many linux distributions will provide the base required packages. You will need to refer to your distribution documentation for details.

On Ubuntu you can use:

```
sudo apt-get install python-matplotlib python-scipy python-nose python-sphinx
```

From a terminal, change to the directory containing the bumps source and type:

```
python setup.py build
python test.py
sudo python setup.py install
```

This should install the application somewhere on your path.

To run the program use:

```
bumps -h
```

**OS/X**

Building a useful python environment on OS/X is somewhat involved, and frequently evolving so this document will likely be out of date. We’ve had success using the Anaconda 64-bit python 2.7 environment from Continuum Analytics, which provides the required packages, but other distributions should work as well.

1.1. Installing the application

3
You will need to install XCode from the app store, and set the preferences to install the command line tools so that a C compiler is available (look in the Downloads tab of the preferences window). If any of your models require fortran, you can download gfortran binaries from r.research.att.com/tools (scroll down to the Apple Xcode gcc-42 add-ons). This sets up the basic development environment.

From a terminal, change to the directory containing the source and type:

```bash
conda create -n bumps numpy scipy matplotlib nose sphinx wxpython
source activate bumps
python setup.py install
cd ..
```

# Optional: allow bumps to run from outside the bumps environment
```bash
mkdir ~/bin # create user terminal app directory if it doesn't already exist
ln -s `python -c "import sys;print sys.prefix"`/bin/bumps ~/bin
```

To run the program, start a new Terminal shell and type:

```
bumps -h
```

### 1.1.2 Fast Stepper for DREAM on MPI

When running DREAM on larger clusters, we found a significant slowdown as the number of processes increased. This is due to Amdahl's law, where the run time speedup is limited by the slowest serial portion of the code. In our case, the DE stepper and the bounds check. Compiling this in C with OpenMP allows us to scale to hundreds of nodes until the stepper again becomes a bottleneck.

To use the compiled DE stepper and bounds checks use:

```bash
(cd bumps/dream && cc compiled.c -I ../../Random123/include/ -O2 -fopenmp -shared -lm -o _compiled.so -fPIC)
```

Note: clang doesn’t support OpenMP, so on OS/X use:

```bash
(cd bumps/dream && cc compiled.c -I ../../Random123/include/ -O2 -shared -lm -o _compiled.so -fPIC)
```

This only works when _compiled.so is in the bumps/dream directory. If running from a pip installed version, you will need to fetch the bumps repository:

```
$ git clone https://github.com/bumps/bumps.git
$ cd bumps
```

Compile as above, then find the bumps install path using the following:

```
$ python -c "import bumps.dream; print(bumps.dream.__file__)"
#dream/path/__init__.py
```

Copy the compiled module to the install (substituting #dream/path above):

```
$ cp bumps/dream/_compiled.so #dream/path
```

There is no provision for using _compiled.so in a frozen application.

Run with no more than 64 OMP threads. If the number of processors is more than 64, then use:

```
OMP_NUM_THREADS=64 ./run.py ...
```
I don’t know how OMP_NUM_THREADS behaves if it is larger than the number of processors.

### 1.1.3 Building Documentation

Building the package documentation requires a working Sphinx installation and a working LaTex installation. Your latex distribution should include the following packages:

- multirrow, titlesec, framed, threeparttable, wrapfig, collection-fontsrecommended

You can then build the documentation as follows:

```
(cd doc \&\& make clean html pdf)
```

Windows users please note that this only works with a unix-like environment such as *gitbash, msys* or *cygwin*. There is a skeleton `make.bat` in the directory that will work using the `cmd` console, but it doesn’t yet build PDF files.

You can see the result of the doc build by pointing your browser to:

- `bumps/doc/_build/html/index.html`
- `bumps/doc/_build/latex/Bumps.pdf`

ReStructured text format does not have a nice syntax for superscripts and subscripts. Units such as g·cm⁻³ are entered using macros such as `|g/cm^3|` to hide the details. The complete list of macros is available in `doc/sphinx/rst_prolog`

In addition to macros for units, we also define cdot, angstrom and degrees unicode characters here. The corresponding latex symbols are defined in `doc/sphinx/conf.py`.

There is a bug in older sphinx versions (1.0.7 as of this writing) in which latex tables cannot be created. You can fix this by changing:

```python
self.body.append(self.table.colspec)
```

to:

```python
self.body.append(self.table.colspec.lower())
```

in `site-packages/sphinx/writers/latex.py`. This may have been fixed in newer versions.

### 1.1.4 Windows Installer

To build a windows standalone executable with py2exe you may first need to create an empty file named `C:\Python27\Lib\numpy\distutils\tests\__init__.py`. Without this file, py2exe raises an error when it is searching for the parts of the numpy package. This may be fixed on recent versions of numpy. Next, update the `__version__` tag in `bumps/__init__.py` to mark it as your own.

Now you can build the standalone executable using:

```
python setup_py2exe
```

This creates a dist subdirectory in the source tree containing everything needed to run the application including python and all required packages.

To build the Windows installer, you will need two more downloads:

- Inno Setup 5.3.10 QuickStart Pack

### 1.1. Installing the application
The C++ redistributable package is needed for programs compiled with the Microsoft Visual C++ compiler, including the standard build of the Python interpreter for Windows. It is available as vcredist_x86.exe from the Microsoft Download Center. Be careful to select the version that corresponds to the one used to build the Python interpreter — different versions can have the same name. For the Python 2.7 standard build, the file is 1.7 Mb and is dated 11/29/2007. We have a copy (vcredist_x86.exe) on our website for your convenience. Save it to the C:\Python27 directory so the installer script can find it.

Inno Setup creates the installer executable. When installing Inno Setup, be sure to choose the ‘Install Inno Setup Preprocessor’ option.

With all the pieces in place, you can run through all steps of the build and install by changing to the top level python directory and typing:

```
python master_builder.py
```

This creates the redistributable installer bumps-<version>-win32.exe for Windows one level up in the directory tree. In addition, source archives in zip and tar.gz format are produced as well as text files listing the contents of the installer and the archives.

### 1.2 Server installation

**Warning:** The remote fitting feature is not actively maintained and will likely not work.

- **Job Controller**
- **Cluster**
- **Security**

Bumps jobs can be submitted to a remote batch queue for processing. This allows users to share large clusters for faster processing of the data. The queue consists of several components.

- **job controller**
  - http service layer which allows users to submit jobs and view results
- **queue**
  - cluster management layer which distributes jobs to the working nodes
- **worker**
  - process monitor which runs a job on the working nodes
- **mapper**
  - mechanism for evaluating \( R(x_i) \) for different \( x_i \) on separate CPUs

If you are setting up a local cluster for performing Bumps analysis then you will need to read this section, otherwise you can continue to the next section.

Assuming that the bumps server is installed as user ‘bumps’ in a virtualenv of ~/bumpserve, MPLCONFIGDIR is set to ~/bumpserve/.matplotlib, and bumpworkd has been configured, you can start with the following profile:

```
TODO: fill in some details on bumps server
```
1.2.1 Job Controller

`extra/jobqueue` is an independent package within bumps. It implements an http API for interacting with jobs.

It is implemented as a WSGI python application using Flask

Here is our WSGI setup for apache for our reflectometry modeling service:

```xml
<VirtualHost *:80>
  ServerAdmin pkienzle@nist.gov
  ServerName www.reflectometry.org
  ServerAlias reflectometry.org
  ErrorLog logs/bumps-error_log
  CustomLog logs/bumps-access_log common

  WSGIDaemonProcess bumps_serve user=pkienzle group=refl threads=3
  WSGIScriptAlias /queue /home/pkienzle/bumps/www/jobqueue.wsgi

  <Directory "/home/pkienzle/bumps/www">
    WSGIProcessGroup bumps_serve
    WSGIApplicationGroup %{GLOBAL}
    Order deny,allow
    Allow from all
  </Directory>

  DocumentRoot /var/www/bumps
  <Directory "/var/www/bumps/">
    AllowOverride All
  </Directory>
</VirtualHost>
```

There is a choice of two different queuing systems to configure. If your environment supports a traditional batch queue you can use it to manage cluster resources. New jobs are added to the queue, and when they are complete, they leave their results in the job results directory. Currently only slurm is supported, but supporting torque as well would only require a few changes.

You can also set up a central dispatcher. In that case, you will have remote clusters pull jobs from the server when they are available, and post the results to the job results directory when they are complete. The remote cluster may be set up with its own queuing system such as slurm, only taking a few jobs at a time from the dispatcher so that other clusters can share the load.

1.2.2 Cluster

If you are using the dispatcher queuing system, you will need to set up a work daemon on your cluster to pull jobs from the queue. This requires adding bumpworkd to your OS initialization scripts.

1.2.3 Security

Because the jobqueue can run without authentication we need to be especially concerned about the security of our system. Techniques such as AppArmor or virtual machines with memory mapped file systems provide a relatively safe environment to support anonymous computing.

To successfully set up AppArmor, there are a few operations you need.

Each protected application needs a profile, usually stored in /etc/apparmor.d/path.to.application. With the reflenv virtual environment in the reflectometry user, the following profile would be appropriate for the worker daemon:
This gives read/execute access to python and its C extensions, and read access to everything else in the bumps virtual environment.

The rw access to .bumpserve is potentially problematic. Hostile models can interfere with each other if they are running at the same time. In particular, they could inject html into the returned data set which can effectively steal authentication credentials from other users through cross site scripting attacks, and so would not be appropriate on an authenticated service. Restricting individual models to their own job directory at .bumpserve/worker/jobid/** would reduce this risk, but this author does not know how to do so without elevating bumpworkd privileges to root.

Once the profile is in place, restart the apparmor.d daemon to enable it:

```
sudo service apparmor restart
```

You can debug the profile by running a trace while the program runs unrestricted. To start the trace, use:

```
sudo genprof /path/to/application
```

Switch to another window then run:

```
/path/to/app
```

When your application is complete, return to the genprof window and hit ‘S’ to scan /var/log/syslog for file and network access. Follow the prompts to update the profile. The documentation on AppArmor on Ubuntu and AppArmor on SUSE is very helpful here.

To reload a profile after running the trace, use:

```
sudo apparmor_parser -r /etc/apparmor.d/path.to.application
```

To delete a profile that you no longer need:

```
sudo rm /etc/apparmor.d/path.to.application
sudo service apparmor restart
```

Similar profiles could be created for the job server, and indeed, any web service you have on your machine to reduce the risk that bugs in your code can be used to compromise your security, but this is less critical since your code is not running in general running with arbitrary user defined functions.
1.3 Contributing Changes

The best way to contribute to the Bumps package is to work from a copy of the source tree in the revision control system.

The bumps project is hosted on github at:

http://github.com/bumps

You can obtain a copy via git using:

```shell
git clone https://github.com/bumps/bumps.git
cd bumps
python setup.py develop
```

By using the `develop` keyword on setup.py, changes to the files in the package are immediately available without the need to run setup each time you change the code.

Track updates to the original package using:

```shell
git pull
```

If you find you need to modify the package, please update the documentation and add tests for your changes. We use doctests on all of our examples to help keep the documentation synchronized with the code. More thorough tests are found in the test directory. Using the the nose test package, you can run both sets of tests:

```shell
pip install nose
python2.5 tests.py
python2.6 tests.py
```

When all the tests run, generate a patch and send it to pkienzle@nist.gov:

```shell
git diff > patch
```

Windows user can use TortoiseGit package which provides similar operations.

Instead of sending patches, you can set up a github account and create your own bumps fork. This allows you to develop code at your leisure with the safety of source control, and issue pull requests when your code is ready to merge with the main repository.

Please make sure that the documentation is up to date, and can be properly processed by the sphinx documentation system. See `_docbuild` for details.

1.4 License

Bumps is in the public domain.

Code in individual files has copyright and license set by the authors. Only free and open source software is used in this package.

1.4.1 Bumps GUI

Copyright (C) 2006-2011, University of Maryland

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to
use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

1.4.2 DREAM

Copyright (c) 2008, Los Alamos National Security, LLC All rights reserved.

Copyright 2008. Los Alamos National Security, LLC. This software was produced under U.S. Government contract DE-AC52-06NA25396 for Los Alamos National Laboratory (LANL), which is operated by Los Alamos National Security, LLC for the U.S. Department of Energy. The U.S. Government has rights to use, reproduce, and distribute this software.

NEITHER THE GOVERNMENT NOR LOS ALAMOS NATIONAL SECURITY, LLC MAKES ANY WARRANTY, EXPRESS OR IMPLIED, OR ASSUMES ANY LIABILITY FOR THE USE OF THIS SOFTWARE. If software is modified to produce derivative works, such modified software should be clearly marked, so as not to confuse it with the version available from LANL.

Additionally, redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of Los Alamos National Security, LLC, Los Alamos National Laboratory, LANL the U.S. Government, nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY LOS ALAMOS NATIONAL SECURITY, LLC AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL LOS ALAMOS NATIONAL SECURITY, LLC OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

1.4.3 numdifftools

Copyright (c) 2014, Per A. Brodtkorb, John D’Errico All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
• Redistributions of source code must retain the above copyright notice, this list of conditions and the following
disclaimer.

• Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the follow-
ing disclaimer in the documentation and/or other materials provided with the distribution.

• Neither the name of the {organization} nor the names of its contributors may be used to endorse or promote
products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY
EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT
SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, IN-
CIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED
TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSI-
NESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CON-
TRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY
WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAM-
AGE.

1.5 Credits

Bumps package was developed under DANSE project and is maintained by its user community.

Please cite:


We are grateful for the existence of many fine open source packages such as NumPy and Python without which this
package would be much more difficult to write.
This tutorial will describe walk through the steps of setting up a model with Python scripting. Scripting allows the user to create complex models with many constraints relatively easily.

2.1 Simple functions

Bumps allows fits with varying levels of complexity. Simple fits accept a function \( f(x; p) \) and data \( x, y, \sigma_y \), where vector \( y \) is the value measured in conditions \( x \), and \( \sigma_y \) is the \( 1 - \sigma \) uncertainty in the measurement. Bumps also provides a simple wrapper for poisson data taken from counting statistics, with function \( f(x; p) \) and data \( x, y \). sim.py is a simulation of data from a poisson process, showing maximum likelihood, expected value and variance.

The ode2 example shows how to fit a system of coupled differential equations where multiple values are tracked at each time step.

2.1.1 Fitting a curve

Fitting a curve to a data set and getting uncertainties on the parameters was the main reason that bumps was created, so it should be very easy to do. Let’s see if it is.

First let’s import the standard names:

```python
from bumps.names import *
```

Next we need some data. The \( x \) values represent the independent variable, and the \( y \) values represent the value measured for condition \( x \). In this case \( x \) is 1-D, but it could be a sequence of tuples instead. We also need the uncertainty on each measurement if we want to get a meaningful uncertainty on the fitted parameters.

```python
x = [1, 2, 3, 4, 5, 6]
y = [2.1, 4.0, 6.3, 8.03, 9.6, 11.9]
dy = [0.05, 0.05, 0.2, 0.05, 0.2, 0.2]
```

Instead of using lists we could have loaded the data from a three-column text file using:
data = np.loadtxt("data.txt").T
x, y, dy = data[0, :], data[1, :], data[2, :]

The variations are endless — cleaning the data so that it is in a fit state to model is often the hardest part in the analysis.

We now define the function we want to fit. The first argument to the function names the independent variable, and the remaining arguments are the fittable parameters. The parameter arguments can use a bare name, or they can use name=value to indicate the default value for each parameter. Our function defines a straight like of slope \( m \) with intercept \( b \) defaulting to 0.

```python
def line(x, m, b=0):
    return m*x + b
```

We can build a curve fitting object from our function and our data. This assumes that the measurement uncertainty is normally distributed, with a 1-\( \sigma \) confidence interval \( dy \) for each point. We specify initial values for \( m \) and \( b \) when we define the model, and then constrain the fit to \( m \in [0, 4] \) and \( b \in [-5, 5] \) with the parameter range method.

```python
M = Curve(line, x, y, dy, m=2, b=2)
M.m.range(0, 4)
M.b.range(-5, 5)
```

Every model file ends with a problem definition including a list of all models and datasets which are to be fitted.

```python
problem = FitProblem(M)
```

The complete model file `curve.py` looks as follows:

```python
from bumps.names import *
x = [1, 2, 3, 4, 5, 6]
y = [2.1, 4.0, 6.3, 8.03, 9.6, 11.9]
dy = [0.05, 0.05, 0.2, 0.05, 0.2, 0.2]
def line(x, m, b=0):
    return m*x + b
M = Curve(line, x, y, dy, m=2, b=2)
M.m.range(0, 4)
M.b.range(-5, 5)
problem = FitProblem(M)
```

We can now load and run the fit:

```
$ bumps.py curve.py --fit=newton --steps=100 --store=T1
```

The `--fit=newton` option says to use the quasi-newton optimizer for not more than 100 steps. The `--store=T1` option says to store the initial model, the fit results and any monitoring information in the directory T1.

As the fit progresses, we are shown an iteration number and a cost value. The cost value is approximately the normalized \( \chi^2_N \). The value in parentheses is like the uncertainty in \( \chi^2_N \), in that a 1-\( \sigma \) change in parameter values should increase \( \chi^2_N \) by that amount.

Here is the resulting fit:

All is well: Normalized \( \chi^2_N \) is close to 1 and the line goes nicely through the data.
2.1. Simple functions
2.1.2 Fitting Poisson data

Data from poisson processes, such as the number of counts per unit time or counts per unit area, do not have the same pattern of uncertainties as data from gaussian processes. Poisson data consists of natural numbers occurring at some underlying rate. The fitting process checks if the number of counts observed is consistent with the proposed rate for each point in the dataset, much like the fitting process for gaussian data checks if the observed value is consistent with the proposed value within the measurement uncertainty.

Using `bumps.curve.PoissonCurve` instead of `bumps.curve.Curve`, we can fit a set of counts at conditions $x$ using a function $f(x, p1, p2, \ldots)$ to propose rates for the various $x$ values given the parameters, yielding parameter values $p1, p2, \ldots$ that are most consistent with the counts at $x$. When measuring poisson processes, the underlying rate is not known, so the measurement variance, which is a property of the rate, is not associated with the data but instead associated with the theory function which predicts the rates. This is opposite from what we have with gaussian data, in which the uncertainty is associated with the measurement device, and explains why the call to PoissonCurve only accepts $x$ and counts, not $x$, $y$, and $dy$.

One property of the Poisson distribution is that it is well approximated by a gaussian distribution for values above about 10. It will never be perfect match since numbers from a poisson distribution can never be negative, whereas gaussian numbers can always be negative, albeit with vanishingly small probability some of the time. Below 10, there are various ways you can approximate the poisson distribution with a gaussian. This example explores some of the options.

In particular, the handling of zero counts can be problematic when treating the measurement as gaussian. You cannot simply drop the points with zero counts. Once you've done various reduction steps, the resulting non-zero value for the uncertainty will carry meaning. The longer you count, the smaller the uncertainty should be, once you've normalized for counting time or monitor. Being off by a factor of 2 on the residuals is much better than being off by a factor of infinity using uncertainty = zero, and better than dropping the point altogether.

There are a few things you can do with zero counts without being completely arbitrary:

1) $\lambda = (k + 1) \pm \sqrt{k + 1}$ for all $k$
2) $\lambda = (k + 1/2) \pm \sqrt{k + 1/4}$ for all $k$
3) $\lambda = k \pm \sqrt{k + 1}$ for all $k$
4) $\lambda = k \pm \sqrt{k}$ for $k > 0$, $1/2 \pm 1/2$ for $k = 0$
5) $\lambda = k \pm \sqrt{k}$ for $k > 0$, $0 \pm 1$ for $k = 0$

See the notes from the CDF Statistics Committee for details at https://www-cdf.fnal.gov/physics/statistics/notes/pois_eb.txt.

Of these, option 5 works slightly better for fitting, giving the best estimate of the background.

The ideal case is to have your model produce an expected number of counts on the detector. It is then trivial to compute the probability of seeing the observed counts from the expected counts and fit the parameters using PoissonCurve. Unfortunately, this means incorporating all instrumental effects when modelling the measurement rather than correcting for instrumental effects in a data reduction program, and using a common sample model independent of instrument.

Setting $\lambda = k$ is good since that is the maximum likelihood value for $\lambda$ given observed $k$, but this breaks down at $k = 0$, giving zero uncertainty regardless of how long we measured.

Since the Poisson distribution is slightly skew, a good estimate is $\lambda = k + 1$ (option 1 above). This follows from the formula for the expected value of a distribution:

$$E[x] = \int_{-\infty}^{\infty} xP(x)dx$$

For the poisson distribution, this is:

$$E[\lambda] = \int_{0}^{\infty} \lambda^k e^{-\lambda} \frac{d\lambda}{k!}$$
Running some simulations, we can see that \( \hat{\lambda} = (k + 1) \pm \sqrt{k + 1} \) (see sim.py). This is the best fit RMS value to the distribution of possible \( \lambda \) values that could give rise to the observed \( k \).

The current practice is to use \( \hat{\lambda} = k \pm \sqrt{k} \). Convincing the world to accept \( \lambda = k + 1 \) would be challenging since the expected value is not the most likely value. As a compromise, one can use \( 0 \pm 1 \) for zero counts, and \( k \pm \sqrt{k} \) for other values. This provides a reasonable estimate for the uncertainty on zero counts, which after normalization becomes smaller for longer counting times or higher incident flux.

Another option is to choose the center and bounds so that the uncertainty covers \( 1 - \sigma \) from the distribution (68%). A simple approximation which does this is \( (n + 1/2) \pm \sqrt{n + 1/4} \). Again, hard to convince the world to do, so one could compromise and choose \( 1/2 \pm 1/2 \) for \( k = 0 \) and \( k \pm \sqrt{k} \) otherwise.

What follows is a model which allows us to fit a simulated peak using these various definitions of \( \lambda \) and see which version best recovers the true parameters which generated the peak.

```python
from bumps.names import *

Define the peak shape. We are using a simple gaussian with center, width, scale and background.

def peak(x, scale, center, width, background):
    return scale*np.exp(-0.5*(x-center)**2/width**2) + background
```

Generate simulated peak data with poisson noise. When running the fit, you can choose various values for the peak intensity. We are using a large number of points so that the peak is highly constrained by the data, and the returned parameters are consistent from run to run. Real data is likely not so heavily sampled.

```python
x = np.linspace(5, 20, 345)
y = np.random.poisson(peak(x, 3, 12, 1.5, 1))
```

Define the various conditions. These can be selected on the command line by listing the condition name after the model file. Note that bumps will make any option not preceded by "-" available to the model file as elements of `sys.argv`. `sys.argv[0]` is the model file itself.

The options correspond to the five options listed above, with an additional option “poisson” which is used to select PoissonCurve rather than Curve in the fit.

```python
cond = sys.argv[1] if len(sys.argv) > 1 else "pearson"
if cond == "poisson": # option 0: use PoissonCurve rather than Curve to fit
    pass
elif cond == "expected": # option 1: L = (y+1) +/- sqrt(y+1)
    y += 1
dy = np.sqrt(y)
elif cond == "pearson": # option 2: L = (y + 0.5) +/- sqrt(y + 1/4)
    dy = np.sqrt(y+0.25)
    y = y + 0.5
elif cond == "expected_mle": # option 3: L = y +/- sqrt(y+1)
    dy = np.sqrt(y+1)
elif cond == "pearson_zero": # option 4: L = y +/- sqrt(y); L[0] = 0.5 +/- 0.5
    dy = np.sqrt(y)
    y = np.asarray(y, 'd')
y[0] = 0.5
dy[0] = 0.5
elif cond == "expected_zero": # option 5: L = y +/- sqrt(y); L[0] = 0 +/- 1
    dy = np.sqrt(y)
    dy[0] = 1.0
```

(continues on next page)
else:
    raise RuntimeError("Need to select uncertainty: pearson, pearson_zero, expected, 
→expected_zero, expected_mle, poisson")

Build the fitter, and set the range on the fit parameters.

```python
if cond == "poisson":
    M = PoissonCurve(peak, x, y, scale=1, center=2, width=2, background=0)
else:
    M = Curve(peak, x, y, dy, scale=1, center=2, width=2, background=0)
dx = max(x) - min(x)
M.scale.range(0, max(y)*1.5)
M.center.range(min(x)-0.2*dx, max(x)+0.2*dx)
M.width.range(0, 0.7*dx)
M.background.range(0, max(y))
```

Set the fit problem as usual.

```python
problem = FitProblem(M)
```

We can now load and run the fit. Be sure to substitute COND for one of the conditions defined above:

```
$ bumps.py poisson.py --fit=dream --burn=600 --store=/tmp/T1 COND
```

Comparing the results for the various conditions, we can see that all methods yield a good fit to the underlying center, scale and width. It is only the background that causes problems. Using poisson statistics for the fit gives the proper background estimate, and using the traditional method of \( \lambda = k \pm \sqrt{k} \) for \( k > 0 \), and \( 0 \pm 1 \) for \( k = 1 \) gives the best gaussian approximation.

### Table 1: Fit results

<table>
<thead>
<tr>
<th>#</th>
<th>method</th>
<th>background</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>poisson</td>
<td>1.0</td>
</tr>
<tr>
<td>1</td>
<td>expected</td>
<td>1.55</td>
</tr>
<tr>
<td>2</td>
<td>pearson</td>
<td>0.16</td>
</tr>
<tr>
<td>3</td>
<td>expected_mle</td>
<td>0.55</td>
</tr>
<tr>
<td>4</td>
<td>pearson_zero</td>
<td>0.34</td>
</tr>
<tr>
<td>5</td>
<td>expected_zero</td>
<td>0.75</td>
</tr>
</tbody>
</table>

#### 2.1.3 Poisson simulation

For the poisson background estimation problem, `poisson.py`, we explore different options for estimating the rate parameter \( \lambda \) from an observed number of counts. This program uses a Monte Carlo method to generate the true probability distribution \( P(\lambda) \) of the observed number of counts \( k \) coming from an underlying rate \( \lambda \). We do this by running a Poisson generator to draw thousands of samples of \( k \) from each of a range of values \( \lambda \). By counting the number of times \( k \) occurs in each \( \lambda \) bin, and normalizing by the bin size and by the total number of times that \( k \) occurs across all bins, the resulting vector is a histogram of the \( \lambda \) probability distribution.

With this histogram we can compute the expected value as:

\[
\hat{\lambda} = \int_0^\infty \lambda P(\lambda|k) d\lambda
\]

and the variance as:

\[
d\hat{\lambda}^2 = \int_0^\infty (\lambda - \hat{\lambda})^2 P(\lambda|k) d\lambda
\]
Generate a bunch of samples from different underlying rate parameters \( L \) in the range 0 to 20

\[
P = \text{np.random.poisson}(L) \\
L = \text{linspace}(0, 20, 1000) \\
X = P(L, \text{size}=(10000, \text{len}(L)))
\]

Generate the distributions

\[
P = \text{dict}((k, \text{sum}(X == k, \text{axis}=0)/\text{sum}(X == k)) \text{ for } k \text{ in range(4)})
\]

Show the expected value of \( L \) for each observed value \( k \)

```
print("Expected value of L for a given observed k")
for k, Pi in sorted(P.items()):
    print(k, sum(L*Pi))
```

Show the variance. Note that we are using \( \hat{\lambda} = k + 1 \) as observed from the expected value table. This is not strictly correct since we have lost a degree of freedom by using \( \hat{\lambda} \) estimated from the data, but good enough for an approximate value of the variance.

```
print("Variance of L for a given observed k")
for k, Pi in sorted(P.items()):
    print(k, sum((L-(k+1))**2*Pi))
```

Plot the distribution of \( \lambda \) that give rise to each observed value \( k \).

```
for k, Pi in sorted(P.items()):
    plot(L, Pi/(L[1]-L[0]), label="k=%d"%k)
xlabel(r'$\lambda$')
ylabel(r'$P(\lambda|k)$')
xticks([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
axis([0, 10, 0, 0.5])
title('Probability of underlying rate $\lambda$ for different observed $k$')
legend()
grid(True)
show()
```

Output:

```
Expected value of L for a given observed k
0 0.989473184121
1 2.00279003084
2 2.99802515025
3 3.9990621889
Variance of L for a given observed k
0 0.998074244206
1 2.00796671097
2 2.99095589399
3 3.99952301552
```
Fig. 1: The figure clearly shows that the maximum likelihood value for $\lambda$ is equal to the observed counts $k$. Because the histogram is skew right, the expected value is a little larger, with an estimated value of $k + 1$, as seen from the output.
2.1.4 Fitting an ODE

Bumps can fit black-box functions, such as odeint from scipy.

The following example is adapted from:
https://people.duke.edu/~ccc14/sta-663/CalibratingODEs.html.

Instructor: Cliburn Chan cliburn.chan@duke.edu
Instructor: Janice McCarthy janice.mccarthy@duke.edu

```python
from bumps.names import *
import numpy as np
from scipy.integrate import odeint

Define the ODE

```python
def g(t, x0, a, b):
    """
    Solution to the ODE x'(t) = f(t,x,k) with initial condition x(0) = x0
    """
    return odeint(dfdt, x0, t, args=(a, b)).flatten()

def dfdt(x, t, a, b):
    """Receptor synthesis-internalization model."
    return a - b*x
```

Simulate some data.

Note that the function `bumps.util.push_seed()` is to set the random number generator to a known state so that this function will create the same data every time the simulation is run. If not, then you wouldn’t be able to resume a fit since each time you resumed you would be fitting different data.

```python
def simulate():
    from bumps.util import push_seed
    
    # Fake some data
    a = 2.0
    b = 0.5
    x0 = 10.0
    t = np.linspace(0, 10, 10)
    dy = 0.2*np.ones_like(t)
    with push_seed(1):
        y = g(t, x0, a, b) + dy*np.random.normal(size=t.shape)
        #print(a, b, x0, t, dt, gt)
    return t, y, dy
```

Define the fit problem.

In this case `bumps.curve.Curve` is initialized with `plot_x` as a vector of length 1000. This is so that a smooth curve is drawn between the ten data points that were simulated in the fit.

```python
M = Curve(g, t, y, dy, x0=1., a=1., b=1.,
          plot_x=np.linspace(t[0], t[-1], 1000))
```

(continues on next page)
M.x0.range(0, 100)
M.a.range(0, 10)
M.b.range(0, 10)

problem = FitProblem(M)

### 2.1.5 Fitting a multi-valued function

Like the ODE fit function, but this example fits a set of coupled ODEs. In this case, there are multiple values reported at each time step, two of which are measured and fitted.

From SciPy cookbook coupled spring mass example:


```python
from bumps.names import *
from scipy.integrate import odeint
```

Use ODEINT to solve the differential equations defined by the vector field

```python
def vectorfield(w, t, p):
    """
    Defines the differential equations for the coupled spring-mass system.

    Arguments:
    w : vector of the state variables:
        w = [x1,y1,x2,y2]
    t : time
    p : vector of the parameters:
        p = [m1,m2,k1,k2,L1,L2,b1,b2]
    """
    x1, y1, x2, y2 = w
    m1, m2, k1, k2, L1, L2, b1, b2 = p

    # Create f = (x1',y1',x2',y2'):
    f = [y1,
         (-b1 * y1 - k1 * (x1 - L1) + k2 * (x2 - x1 - L2)) / m1,
         y2,
         (-b2 * y2 - k2 * (x2 - x1 - L2)) / m2]
    return f
```

ODE solver parameters

```python
abserr = 1.0e-8
relerr = 1.0e-6
```

Curve function with all parameters exposed so that bumps knows their names. Only tracking x1, x2 with our measurements and not y1, y2, so returning components 0 and 2 of the `vectorfield` result. The multi-valued y values are stacked into an array whose first axis matches t. This is needed so that the plotter can sort out the different lines.

```python
def f(t, x1, y1, x2, y2, m1, m2, k1, k2, L1, L2, b1, b2):
    """Pack up the parameters and initial conditions:"
    p = [m1, m2, k1, k2, L1, L2, b1, b2]
    w0 = [x1, y1, x2, y2]
    return f(t, x1, y1, x2, y2, m1, m2, k1, k2, L1, L2, b1, b2)
```

(continues on next page)
# Call the ODE solver.
wsol = odeint(vectorfield, w0, t, args=(p,),
atol=abserr, rtol=relerr)
return np.vstack((wsol[:, 0], wsol[:, 2]))

Simulation parameter values

# Masses
m1 = 1.0
m2 = 1.5

# Spring constants
k1 = 8.0
k2 = 40.0

# Natural lengths
L1 = 0.5
L2 = 1.0

# Friction coefficients
b1 = 0.8
b2 = 0.5

Initial conditions

# x1 and x2 are the initial displacements; y1 and y2 are the initial velocities
x1 = 0.5
y1 = 0.0
x2 = 2.25
y2 = 0.0

Simulate data

def simulate():
    from bumps.util import push_seed

    # Create the time samples for the output of the ODE solver.
    # These are the times that the data is sampled, not the times at
    # which to evaluate the ode solver.
    t = np.linspace(0, 10, 100)

    # Pack up the parameters and initial conditions:
    p = [m1, m2, k1, k2, L1, L2, b1, b2]
    w0 = [x1, y1, x2, y2]
    ft = f(t, *(w0 + p))

    noise = 0.1*np.ones_like(ft)
    with push_seed(l):
        data = ft + noise*np.random.randn(*ft.shape)
    return t, data, noise

t, y, dy = simulate()

Initial values for most parameters are known from system configuration. We are not including the spring constants or the friction coefficients since these will be fitted to the measured position over time. labels allow you to set the labels for the x-axis and y-axis and the legend for the two data lines on the plot.

2.1. Simple functions
M = Curve(f, t, y, dy, m1=m1, m2=m2, L1=L1, L2=L2, x1=x1, y1=y1, x2=x2, y2=y2,
   labels=['time', 'value', 'x1', 'x2'], plot_x=np.linspace(0, 10, 1000))

Fitted parameters

Only fitting spring constants and friction coefficients since these are not immediately measurable. If we wanted to be fancy, we could set the prior on position and mass according to the uncertainty in our initial configuration and allow them to vary slightly.

```python
# Masses: Allow mass estimate to be off by +/- 2% (1-sigma) *untested*
#M.m1.dev(0.02*m1)
#M.m2.dev(0.02*m2)

# Spring constants
M.k1.range(0, 100)
M.k2.range(0, 100)

# Natural lengths
#M.L1.range(0, 10)
#M.L2.range(0, 10)

# Friction coefficients
M.b1.range(0, 1)
M.b2.range(0, 1)

# Initial conditions
# x1 and x2 are the initial displacements; y1 and y2 are the initial velocities
#M.x1.range(0, 10)
#M.x2.range(0, 10)
#M.y1.range(0, 10)
#M.y2.range(0, 10)
```

```python
problem = FitProblem(M)
```

### 2.2 Peak Fitting

This example shows how to develop multipart models using bumps parameters. The data format is 2D, so the usual 1D x-y plots are not sufficient, and a special plot method is needed to display the data.

### 2.3 Test functions

Test a variety of more difficult problems to see how well DREAM can recover the correct probability definition.

#### 2.3.1 Anticorrelation demo

Model with strong correlations between the fitted parameters.

We use $a \times x = y + N(0,1)$ made complicated by defining $a=p1+p2$.

The expected distribution for $p1$ and $p2$ will be uniform, with $p2 = a-p1$ in each sample. Because this distribution is inherently unbounded, artificial bounds are required on at least one of the parameters for finite duration simulations.
The expected distribution for $p1+p2$ can be determined from the linear model $y = a*x$. This is reported along with the values estimated from MCMC.

```python
from bumps.names import *

Anticorrelated function

```python
def fn(x, a, b):
    return (a+b)*x
```

Fake data

```python
sigma = 1
x = np.linspace(-1., 1, 40)
dy = sigma*np.ones_like(x)
y = fn(x,5,5) + np.random.randn(*x.shape)*dy
```

Wrap it in a curve fitter

```python
M = Curve(fn, x, y, dy, a=(-20,20), b=(-20,20))
```

Alternative representation, fitting $a$ and $S=a+b$, and setting $b=S-a$.

```python
S = Parameter((-20,20), name="sum")
M.b = S-M.a
```

```python
problem = FitProblem(M)
```

### 2.3.2 Boundary check

Check probability at boundaries.

In this case we define the probability density function (PDF) directly in an n-dimensional uniform box.

Ideally, the correlation plots and variable distributions will be uniform.

```python
from bumps.names import *

Adjust domain from 1e-150 to 1e+150 and you will see that DREAM is equally adept at filling the box.

```python
domain = 1
```

Uniform cost function.

```python
def box(x):
    ""
    A flat top mesa with a square border in [-1, 1].
    ""
    return 0 if np.all(np.abs(x) <= domain) else np.inf

def ramp(x):
    ""
    A ramp in the first parameter, all other parameters uniform over [-1, 1].
    ""
    p = abs(x[0])/domain
    return -log(p) if np.all(np.abs(x) <= domain) else np.inf
```

(continues on next page)
def cone(x):
    """
    An inverted cone with peak probability at the rim of radius 1.
    """
    r = np.sqrt(sum(xk**2 for xk in x[:2]))
    r = np.sqrt(sum(xk**2 for xk in x))
    return -log(r) if r <= domain else np.inf

def diamond(x):
    """
    A flat top mesa with a diamond border.
    """
    return 0 if np.sum(np.abs(x)) <= domain else np.inf

def sawtooth(x):
    """
    A symmetric sawtooth of frequency 1, phase 0, so f(0)=1, f(1/2)=0.
    """
    p = [2*abs(xk/domain%1 - 1/2) for xk in x]
    return -sum(np.log(pk) for pk in p)

def triangle_constraints():
    """
    The triangle below y=x.
    """
    a, b = M.a.value, M.b.value
    return 0 if a < b else 1e6 + (b-a)**2

def box_constraints():
    """
    A square over [-1/2, 1/2].
    """
    a, b = M.a.value, M.b.value
    return 0 if abs(a) <= domain/2 and abs(b) <= domain/2 else np.inf

def circle_constraints():
    """
    A circle of radius 1.
    """
    a, b = M.a.value, M.b.value
    r = np.sqrt(a**2 + b**2)
    return 0 if r <= domain*2/3 else np.inf

def ring_constraints():
    """
    A ring of inner radius 2/3.
    """
    a, b = M.a.value, M.b.value
    r = np.sqrt(a**2 + b**2)
    return 0 if domain*2/3 <= r <= domain else 1e6 + (r/domain - 1)**2

def sawtooth_constraints():
    """
    Sets one peak at the edge of the domain and another in the middle. Use
    this to investigate whether rejection outside the domain leads to
    distortion of the density at the boundary of the domain. You will need
Wrap it in a PDF object which turns an arbitrary probability density into a fitting function. Give it a valid initial value, and set the bounds to a unit cube with one corner at the origin.

```
#M = PDF(lambda a, b: box([a, b]))
M = PDF(lambda a, b: diamond([a, b]))
#M = PDF(lambda a, b: ramp([a, b]))
#M = PDF(lambda a, b: cone([a, b]))
#M = PDF(lambda a, b: sawtooth([a, b]))
constraints = None
#constraints = triangle_constraints
constraints = box_constraints
#constraints = circle_constraints
#constraints = ring_constraints
#constraints = sawtooth_constraints
M.a.range(-2*scale, 2*scale)
M.b.range(-2*scale, 2*scale)
# Make the PDF a fit problem that bumps can process.
problem = FitProblem(M, constraints=constraints)
```

### 2.3.3 Cross-shaped anti-correlation

Example model with strong correlations between the fitted parameters.

In this case we define the probability density function (PDF) directly as an ‘X’ pattern, with width sigma.

Ideally, the a-b correlation plot will show the ‘X’ completely filled within the bounds.

```
from bumps.names import *
```

Adjust scale from 1e-150 to 1e+150 and you will see that DREAM is equally adept at filling the cross. However, if sigma gets too small relative to scale the fit will get stuck on one of the arms, and if sigma gets too large, then the whole space will be filled and the x will not form.

```
scale = 10
sigma = 0.1*scale
#sigma = 0.001*scale  # Too small
#sigma = 10*scale    # Too large
```

Simple gaussian cost function based on the distance to the closest ridge $x=y$ or $x=-y$.

```
def fn(a, b):
    return 0.5*min(abs(a+b), abs(a-b))**2/sigma**2 + 1
```

Wrap it in a PDF object which turns an arbitrary probability density into a fitting function. Give it an initial value away from the cross.
M = PDF(fn, a=3*scale, b=1.2*scale)

Set the range of values to include the cross. You can skip the center of the cross by setting b.range to (1,3), and for reasonable values of sigma both arms will still be covered. Extend the range too far (e.g., a.range(-3000,3000), b.range(-1000,3000)), and like a value of sigma that is too small, only one arm of the cross will be filled.

M.a.range(-3*scale,3*scale)
M.b.range(-1*scale,3*scale)

Make the PDF a fit problem that bumps can process.

problem = FitProblem(M)

### 2.4 Check the entropy calculator

A single measure for a multivariate distribution is the entropy

By comparing the entropy of the prior distribution (usually a box uniform distribution with entropy \( \sum_{i=1}^{n} \log(w_i) \) where \( w_i \) is the range on parameter \( i \) and \( n \) is the number of parameters, but maybe lower if explicit priors are given for any of the parameters based on information from other sources) to the entropy computed from the posterior, you can estimate the number of bits of information from the fit to the data.

Note that bumps calculates the entropy expected from the closest multivariate normal distribution (MVN) as well as directly from the samples. The sample derived entropy has more variability, particularly in high dimensions.

Many of the probability distributions in scipy.stats include a method to compute the entropy of the distribution. We can use these to test the values from bumps against known good values.

```python
import numpy as np
from math import log
from scipy.stats import distributions, multivariate_normal
from bumps.names import *
from bumps.dream.entropy import Box, MultivariateT, Joint

Create the distribution using the name and parameters from the command line. Provide some handy help if the no distribution is given.

TODO: create version of dirichlet that we can sample from. For dirichlet, need to enforce \( x_k \) in \([0,1]\) and \( \sum(x) = 1 \). By reducing the number of parameters by 1 and setting

```
Set the fitting problem using the direct PDF method. In this case, bumps is not being used to fit data, but instead to explore the probability distribution directly through the negative log likelihood function. The only argument to this function is the parameter value x, which becomes the fitting parameter. This model file will not work for multivariate distributions.

```python
def D_nllf(x):
    return -D.logpdf(x)
dim = getattr(D, 'dim', 1)
if dim == 1:
    M = PDF(D_nllf, x=0.9)
```

(continues on next page)
```python
M.x.range(-inf, inf)
else:
    M = VectorPDF(D_nllf, np.ones(dim))
    for k in range(dim):
        getattr(M, 'p'+str(k)).range(-inf, inf)

if dist_name == "mvskewn":
    for k in range(dim):
        getattr(M, 'p'+str(k)).value = D.distributions[k].mean()

problem = FitProblem(M)
```

Before fitting, print the expected entropy from the fit.

```python
entropy = D.entropy()
print("*** Expected entropy: %.4f bits %.4f nats"%(entropy/log(2), entropy))
```

To exercise the entropy calculator, try fitting some non-normal distributions:

```plaintext
t 84 # close to normal
t 4 # high kurtosis
uniform -5 100 # high entropy
cauky 0 1 # undefined variance
expon 0.1 0.2 # asymmetric, narrow
beta 0.5 0.5 # 'antimodal' u-shaped pdf
beta 2 5 # skewed
mvn 1,1,1 1,2,3 # 3-D multivariate normal at (1,2,3)
mvt 4 1,1,1,1,1 # 5-D multivariate t-distribution with df=4 at origin
mvnu 1,1,1,1,1 # 5-D unit uniform distribution centered at origin
mvcauchy 1,1,1 # 3-D multivariate Cauchy distribution at origin
mvskewn 5 1,1,1 # 3-D multivariate skew normal with alpha=5 at origin
```

Ideally, the entropy estimated by bumps will match the predicted entropy when using --fit=dream. This is not the case for beta 0.5 0.5. For the other distributions, the estimated entropy is within uncertainty of actual value, but the uncertainty is a bit high.

The other fitters, which use the curvature at the peak to estimate the entropy, do not work reliably when the fit is not normal. Try the same distributions with --fit=amoeba to see this.

## 2.5 Bayesian Experimental Design

Perform a tradeoff comparison between point density and counting time when measuring a peak in a poisson process.

Usage:

```bash
bumps peak.py N --entropy --store=/tmp/T1 --fit=dream
```

The parameter N is the number of data points to use within the range.

```python
from bumps.names import *
from numpy import exp, sqrt, pi, inf

# Define the peak shape as a gaussian plus background
def peak(x, scale, center, width, background):
    return scale*exp(-0.5*(x-center)**2/width**2)/sqrt(2*pi*width**2) + background
```
# Get the number of points from the command line
if len(sys.argv) == 2:
    npoints = int(sys.argv[1])
else:
    raise ValueError("Expected number of points n in the fit")

# set a constant number of counts, equally divided between points
x = np.linspace(5, 20, npoints)
scale = 10000/npoints

# Build the model, along with the valid fitting range. there is no data yet,
# so y is None
M = PoissonCurve(peak, x, y=None, scale=scale, center=15, width=1.5, background=1)
M.scale.range(0, inf)
dx = max(x)-min(x)
M.center.range(min(x) - 0.2*dx, max(x) + 0.2*dx)
M.width.range(0, 0.7*dx)
M.background.range(0, inf)

# Make a fake dataset from the given x spacing
M.simulate_data()

problem = FitProblem(M)

Running this problem for a few values of the number of points is showing that adding points and reducing counting
time per point is better able to recover the peak parameters.

## 2.6 Calling fit from scripts

Revisiting our curve fit example, let’s call the optimizer directly from the script.

Setting up the problem remains the same:

```python
from __future__ import print_function
from bumps.names import *

x = [1, 2, 3, 4, 5, 6]
y = [2.1, 4.0, 6.3, 8.03, 9.6, 11.9]
dy = [0.05, 0.05, 0.2, 0.05, 0.2, 0.2]

def line(x, m, b=0):
    return m*x + b

M = Curve(line, x, y, dy, m=2, b=2)
M.m.range(0, 4)
M.b.range(-5, 5)
problem = FitProblem(M)
```

With the problem defined, we can now call the fitter. The following uses the minimalist fit interface defined in bumps,
which takes a problem definition and returns a results object with x, dx attributes for the best value and the estimated
uncertainty. The ‘dream’ fitter will additionally return the dream state, which allows for more detailed uncertainty
analysis.
from bumps.fitters import fit
from bumps.formatnum import format_uncertainty

# Allow choice of fitter from the command line
method = 'amoeba' if len(sys.argv) < 2 else sys.argv[1]

print("initial chisq", problem.chisq_str())
result = fit(problem, method=method, xtol=1e-6, ftol=1e-8)
print("final chisq", problem.chisq_str())
for k, v, dv in zip(problem.labels(), result.x, result.dx):
    print(k, ":", format_uncertainty(v, dv))

\section*{2.7 Inequality constraints}

The usual pattern for constraints within bumps is to set the value for one parameter to be some function of the other
parameters. This does not allow constraints of the form $a < b$ for parameters $a$ and parameter $b$.

Instead, along with the fit problem definition, you can supply your own penalty constraints function which adds an
artificial value to the probability function for points outside the feasible region. The ideal constraints function will
incorporate the distance from the boundary of the feasible region so that if the fitter is started outside forces the fit
back into the feasible region.

The `soft_limit` value can be used in conjunction with the penalty to avoid evaluating the function outside the feasible
region. For example, the function $\log(a - b)$ is only defined for $a > b$, so setting a constraint such as $10^6 + (a - b)^2$
for $a \leq b$ and 0 along with a soft limit of $10^6$ will keep the function defined everywhere. With the penalty value
sufficiently large, the probability of any evaluation in the infeasible region will be negligible, and will not skew the
posterior distribution statistics.

Define the model as usual

```python
from bumps.names import *

def line(x, m, b):
    return m*x + b

x = [1, 2, 3, 4, 5, 6]
y = [2.1, 4.0, 6.3, 8.03, 9.6, 11.9]
dy = [0.05, 0.05, 0.2, 0.05, 0.2, 0.2]
M = Curve(line, x, y, dy, m=2, b=0)
M.m.range(0, 4)
M.b.range(0, 5)
```

Define the constraints as a function which takes no parameters and returns a floating point value. Note the value $1e6$
in the penalty condition: this is the soft limit value which we will use to avoid evaluating the curve in the infeasible
region.

```python
def constraints():
    m, b = M.m.value, M.b.value
    return 0 if m < b else 1e6 + (m-b)**6
```

Attach the constraints to the problem. Give the soft limit value that is used for the constraints. Without the soft limit,
the fit would stall since we started it at a deep local minimum near the true solution without constraints.
The constraint relies on the ability for python to access the parameters from the module. Furthermore, the parameters still “boxed”, and so you need to reference the value attribute to get the parameter value at the time the constraint is evaluated. Not an elegant solution, but it works. Eventually we will add constraint expressions such as $M.m < M.b$ or $M.m + M.b < 10$ using the same infrastructure as equality constraints.
Bumps is designed to determine the ideal model parameters for a given set of measurements, and provide uncertainty on the parameter values. This is an inverse problem, where measured data can be predicted from theory, but theory cannot be directly inferred from measured data. This means that bumps must search through parameter space, calling the theory function many times to find the parameter values that are most consistent with the data.

Unlike traditional Levenburg-Marquardt fitting programs, Bumps does not require normally distributed measurement uncertainty. If a measurement comes from counting statistics, for example, you can define your model with poisson probability rather than gaussian probability. Parameter values can have constraints. For example, if the size of a sample is known to within 5%, the size parameter in the model can set to a gaussian distribution with a standard deviation of 5%. Simple bounds are also supported. Parameter expressions allow you to set the value of a parameter based on other parameters, which allows simultaneous fitting of multiple datasets to different models without having to define a specialized fit function.

Bumps includes Markov chain Monte Carlo (MCMC) methods to compute the joint distribution of parameter probabilities. These methods require hundreds of thousand function calls to explore the search space, so for moderately complex problems, you need to run in parallel. Bumps can fully utilize multiple cores on one computer, or through MPI, it runs on supercomputing clusters.

```plaintext
# Data handling has been removed so that we can ship a pure python package. In addition to inverse problem solving, bumps has acquired code for theory building and data handling. For example, many problems have measurements in which the instrument resolution plays a role, and the theory function must be convolved with a data dependent resolution function.

Using Bumps

Model scripts associate a sample description with data and fitting options to define the system you wish to refine.

Data Representation

Data management is the responsibility of the modeller. Bumps provides a generic data loader `bumps.data` with a key-value header section followed by columns of numeric data, but it is up to the model script to compute the theory along with any resolution effects and compare that with the data. The `bumps.curve.Curve` class associates a theory function with measurements with Gaussian uncertainty, and `bumps.curve.PoissonCurve` does the same for measurements following Poisson statistics.
Parameters

The adjustable values in each component of the system are defined by Parameter objects. When you set the range on a parameter, the system will be able to automatically adjust the value in order to find the best match between theory and data.

Fitting

One or more experiments can be combined into a FitProblem. This is then given to one of the many fitters, such as DEFit, which adjust the fitting parameters, trying to find the best fit. See Optimizer Selection for a description of available optimizers and Bumps Options for a description of the bumps options. Entropy can be calculated when the fit is complete. See Calculating Entropy.

3.1 Using Bumps

The first step in using Bumps is to define a fit file. This is python code defining the function, the fitting parameters and any data that is being fitted.

A fit file usually starts with an import statement:

```python
from bumps.names import *
```

This imports names from bumps.names and makes the available to the model definition.

Next the fit file should load the data with something like np.loadtxt which loads columnar ASCII data into an array. This data feeds into a Fitness function for a particular model that gives the probability of seeing the data for a given set of model parameters. These model functions can be quite complex, involving not only the calculation of the theory function, but also simulating instrumental resolution and background signal.

The fitness function will have Parameter objects defining the fittable parameters. Usually the model is initialized without any fitted parameters, allowing the user to set a range on each parameter that needs to be fitted. Although it is a little tedious to set up, keeping the fitted ranges separate from the model definition works better in the fitting process, which usually involves multiple iterations with different configurations. It is convenient to be able to turn on and off fitting for individual parameter with a simple comment character (‘#’) at the start of the line.

Every fit file ends with a FitProblem definition:

```python
problem = FitProblem(model)
```

In fact, this is the only requirement of the fit file. The Bumps engine loads the fit file, retrieves the problem symbol and feeds it to one of the fitters. Some fit files do not even use FitProblem to define problem, or use Parameter objects for the fitted parameters, so long as problem implements the BaseFitProblem interface, which provides getp to get the existing parameter vector, setp to set a new parameter vector, bounds to return the parameter bounds, and nllf to to compute the negative log likelihood function. The remaining methods are optional.

Note that the pattern of importing all names from a file using from bumps.names import *, while convenient for simple scripts, can make the code more difficult to understand later, and can lead to unexpected results when moving code around to other files. The alternative pattern to use is:

```python
import bumps.names as bmp
...
problem = bmp.FitProblem(model)
```

This documents to the reader unfamiliar with your code (such as you, dear reader, when looking at your model files two years from now) exactly where the name comes from.

The Tutorial walks through the process for several different data sets.
3.2 Data Representation

Data is x,y,dy. Anything more complicated you will need to define yourself.

3.3 Experiment

- Simple experiments
- Likelihood functions
- Complex models
- Linear models
- Foreign models
- External constraints

It is the responsibility of the user to define their own experiment structure. The usual definition will describe the sample of interest, the instrument configuration, and the measured data, and will provide a theory function which computes the expected data given the sample and instrument parameters. The theory function frequently has a physics component for computing the ideal data given the sample and an instrument effects component which computes the expected data from the ideal data. Together, sample, instrument, and theory function define the fitting model which needs to match the data.

The curve fitting problem can be expressed as:

\[
P(\text{model} \mid \text{data}) = \frac{P(\text{data} \mid \text{model})P(\text{model})}{P(\text{data})}
\]

That is, the probability of seeing a particular set of model parameter values given the observed data depends on the probability of seeing the measured data given a proposed set of parameter values scaled by the probability of those parameter values and the probability of that data being measured. The experiment definition must return the negative log likelihood as computed using the expression on the right. Bumps will explore the space of the sample and instrument parameters in the model, returning the maximum likelihood and confidence intervals on the parameters.

There is a strong relationship between the usual $\chi^2$ optimization problem and the maximum likelihood problem. Given Gaussian uncertainty for data measurements, we find that data $y_i$ measured with uncertainty $\sigma_i$ will be observed for sample parameters $p$ when the instrument is at position $x_i$ with probability

\[
P(y_i \mid f(x_i; p)) = \frac{1}{\sqrt{2\pi\sigma_i^2}} \exp\left(-\frac{(y_i - f(x_i; p))^2}{2\sigma_i^2}\right)
\]

The negative log likelihood of observing all points in the data set for the given set of sample parameters is

\[
-\log \prod_i P(y_i \mid f(x_i; p)) = \frac{1}{2} \sum_i \frac{(y_i - f(x_i; p))^2}{\sigma_i^2} - \frac{1}{2} \sum \log 2\pi\sigma_i^2 = \frac{1}{2} \chi^2 + C
\]

Note that this is the unnormalized $\chi^2$, whose expected value is the number of degrees of freedom in the model, not the reduced $\chi^2_R$ whose expected value is 1. The Bumps fitting process is not sensitive to the constant $C$ and it can be safely ignored.

Casting the problem as a log likelihood problem rather than $\chi^2$ provides several advantages. We can support a richer set of measurement techniques whose uncertainties do not follow a Gaussian distribution. For example, if we have a Poisson process with a low count rate, the likelihood function will be asymmetric, and a gaussian fit will tend to over-estimate the rate. Furthermore, we can properly handle background rates since we can easily compute the probability...
of seeing the observed number of counts given the proposed signal plus background rate. Gaussian modeling can lead to negative rates for signal or background, which is fundamentally wrong. See Simple functions for a demonstration of this effect.

We can systematically incorporate prior information into our models, such as uncertainty in instrument configuration. For example, if our sample angle control motor position follows a Gaussian distribution with a target position of \(3^\circ\) and an uncertainty of \(0.2^\circ\), we can set

\[
-\log P(\text{model}) = \frac{-1}{2} \left( \frac{(\theta - 3)^2}{0.2^2} \right)
\]

ignoring the scaling constant as before, and add this to \(\frac{1}{2} \chi^2\) to get log of the product of the uncertainties. Similarly, if we know that our sample should have a thickness of \(100 \pm 3.5 \text{ Å}\) based on how we constructed the sample, we can incorporate this information into our model in the same way.

### 3.3.1 Simple experiments

The simplest experiment is defined by a python function which takes a list of instrument configuration and arguments defining the parameters. For example, to fit a line you would need:

```python
def line(x, m, b):
    return m*x + b
```

Assuming the data was in a 3 column ascii file with x, y and uncertainty, you would turn this into a bumps model file using:

```python
# 3 column data file with x, y and uncertainty
x,y,dy = numpy.loadtxt('line.txt').T
M = Curve(line, x, y, dy)
```

Using the magic of python introspection, Curve is able to determine the names of the fittable parameters from the arguments to the function. These are converted to Parameter objects, the basis of the Bumps modeling system. For each parameter, we can set bounds or values:

```python
M.m.range(0,1)  # limit slope between 0 and 45 degrees
M.b.value = 1   # the intercept is set to 1.
```

We could even set a parameter to a probability distribution, using Parameter.dev for Gaussian distributions or setting parameter.bounds to Distribution for other distributions.

Bumps includes code for polynomial interpolation including B-splines, monotonic splines, and chebyshev polynomials.

For counts data, PoissonCurve is also available.

### 3.3.2 Likelihood functions

If you are already have the negative log likelihood function and you don’t need to manage data, you can use it with PDF:

```python
x,y,dy = numpy.loadtxt('line.txt').T
def nllf(m, b):
    return numpy.sum(((y - (m*x + b))/dy)**2)
M = PDF(nllf)
```

You can use \(M.m\) and \(M.b\) to the parameter ranges as usual, then return the model as a fitting problem:
3.3.3 Complex models

More sophisticated models, with routines for data handling and specialized plotting should define the Fitness interface. The Peak Fitting example sets up a problem for fitting multiple peaks plus a background against a 2-D data set.

Models are parameterized using Parameter objects, which identify the fitted parameters in the model, and the bounds over which they may vary. The fitness object must provide a set of fitting parameters to the fit problem using the parameters method. Usually this returns a dictionary, with the key corresponding to the attribute name for the parameter and the value corresponding to a parameter object. This allows the user of the model to guess that parameter “p1” for example can be referenced using model.p1. If the model consists of parts, the parameters for each part must be returned. The usual approach is to define a parameters method for each part and build up the dictionary when needed (the parameters function is only called at the start of the fit, so it does not need to be efficient). This allows the user to guess that parameter “p1” of part “a” can be referenced using model.a.p1. A set of related parameters, p1, p2, ... can be placed in a list and referenced using, e.g., model.a.p[1].

The fitness constructor should accept keyword arguments for each parameter giving reasonable defaults for the initial value. The parameter attribute should be created using Parameter.default. This method allows the user to set an initial parameter value when the model is defined, or set the value to be another parameter in the fitting problem, or to a parameter expression. The name given to the default method should include the name of the model. That way when the same type of model is used for different data sets, the two sets of parameters can be distinguished. Ideally the model name would be based on the data set name so that you can more easily figure out which parameter goes with which data.

During an analysis, the optimizer will ask to evaluate a series of points in parameter space. Once the parameters have been set, the update method will be called, if there is one. This method should clear any cached results from the last fit point. Next the nllf method will be called to compute the negative log likelihood of observing the data given the current values of the parameters. This is usually just \( \sum (y_i - f(x_i))^2/(2\sigma_i^2) \) for data measured with Gaussian uncertainty, but any probability distribution can be used.

For the Levenberg-Marquardt optimizer, the residuals method will be called instead of nllf. If residuals are unavailable, then the L-M method cannot be used.

The numpoints method is used to report fitting progress. With Gaussian measurement uncertainty, the nllf return value is \( \chi^2/2 \), which has an expected value of the number of degrees of freedom in the fit. Since this is an awkward number, the normalized chi-square, \( \chi^2_N = \chi^2/\text{DoF} = -2 \ln(P)/(n-p) \), is shown instead, where \( -\ln P \) is the nllf value, \( n \) is the of points and \( p \) is the number of fitted parameters. \( \chi^2_N \) has a value near 1 for a good fit. The same calculation is used for non-gaussian distributions even though nllf is not returning sum squared residuals.

The save and plot methods will be called at the end of the fit. The save method should save the model for the current point. This may include things such as the calculated scattering curve and the real space model for scattering inverse problems, or it may be a save of the model parameters in a format that can be loaded by other programs. The plot method should use the current matplotlib figure to draw the model, data, theory and residuals.

The resynth_data method is used for an alternative monte carlo error analysis where random data sets are generated from the measured value and the uncertainty then fitted. The resulting fitted parameters can be processed much like the MCMC datasets, yielding a different estimate on the uncertainties in the parameters. The restore_data method restores the data to the originally measured values. These methods are optional, and only used if the alternative error analysis is requested.
3.3.4 Linear models

Linear problems with normally distributed measurement error can be solved directly. Bumps provides `bumps.wsolve.wsolve()`, which weights values according to the uncertainty. The corresponding `bumps.wsolve.wpolyfit()` function fits polynomials with measurement uncertainty.

3.3.5 Foreign models

If your modeling environment already contains a sophisticated parameter handling system (e.g. Sympy or PyMC) you may want to tie into the Bumps system at a higher level. In this case you will need to define a class which implements the `FitProblem` interface. This has been done already for `PyMCPProblem` and interested parties are directed therein for a working example.

3.3.6 External constraints

3.4 Parameters

- **Free Variables**

Bumps fitting is centered on `Parameter` objects. Parameters define the search space, the uncertainty analysis and even the user interface. Constraints within and between models are implemented through parameters. Prior probabilities are defined by for parameters.

Model classes for Bumps should make it easy to define the initial value of fitting parameters and tie parameters together. When creating a model, you should be able specify `parameter=value` for each of the model parameters. Later, you should be able to reference the parameter within the model using `M.parameter`. Parameters can also be tied together by assigning the same `Parameter` object to two different parameters. For example, a hollow cylinder can be created using:

```python
solvent = Parameter("solvent", value=1.2)
shell = Parameter("shell", value=4.5)
M = CoreShellCylinder(core=solvent, shell=shell, solvent=solvent,
radius=95, thickness=10, length=100)
```

The model parameter can also be a derived value that is the result of a parameter expression. For example, the following creates a cylinder whose length is twice the radius:

```python
radius = Parameter("radius", value=3)
M = Cylinder(radius=radius, length=2*radius)
```

Any time you ask for `M.length.value` it will compute the result as `2*radius.value` and return that.

You can also tie parameters together after the fact. For example, you can create the constrained cylinder using:

```python
M = Cylinder(radius=3, length=6)
M.length = 2*M.radius
```

The advantage of this method is that you can easily comment out the constraint when exploring the model space, and fit `length` and `radius` freely.

Once you have defined your models and constraints you can set up you fitting parameters. There are several parameter methods which are helpful:
• *range* forces the parameter to lie within a fixed range. The parameter value can take on any value within the range with equal probability, and has zero probability outside the range.

• *pm* is a convenient way to set up a range based on the initial value of parameter. For example, *M.thickness.pm(10)* will allow the thickness parameter to vary by plus or minus 10. You can do asymmetric ranges by calling *pm* with plus and minus values, such as *M.thickness.pm(-3,2)*. The actual range gets set to a *nice_range* that includes the bounds.

• *pmp* is like *pm* but the range is specified as a percent. For example, to let thickness vary by 10%, use *M.thickness.pmp(10)*. Again, a *nice_range* is used.

• *dev* sets up a parameter whose prior probability is not equal across its range, but instead follows a normal distribution. If for example, you have measure the thickness to be 32.1 ± 0.6 by some other technique, you can use this information to constrain your model by initializing *thickness* to 32.1 and setting *M.thickness.dev(0.6)* as a fitting constraint. The *dev* method also accepts absolute limits, creating a truncated normal distribution. You can set the central value *mu* as well, but you probably want to do this in the model initialization so that you are free to turn fitting of the parameter on and off by commenting out the *dev* line.

• *soft_range* is a combination of *range* and *dev* in that the parameter has equal probability within [*low,* high*] but Gaussian probability of width *std* as it strays outside of the range.

• *pdf* is like *dev* but works with any continuous scipy.stats distribution.

All these methods set the *bounds* attribute on the parameter in one way or another. See *bumps.bounds* for details. Technically, setting the parameter to *dev*, *soft_range* or *pdf* is equivalent to creating a probability distribution model with a single data point and *Fitness.nllf* equal to the negative log likelihood of seeing the parameter value in the distribution. This *PDF* model would be fit simultaneously with your target model with the parameter shared between them. The result is statistically sound (it is just more prior information), and conveniently, it does not affect the number of degrees of freedom in the fit.

When defining new model classes, use the static method *Parameter.default()* to initialize the parameter. This will accept the input argument passed in by the user and depending on its type, either create a new parameter slot and set its initial value, or link the slot to another parameter.

### 3.4.1 Free Variables

When fitting multiple datasets, you will undoubtedly have models with many shared parameters, and some parameters that differ between the models. Common patterns include:

- different measurements may use the same material but different contrast agents,
- they may use the same contrast agent but different materials,
- the same material and contrast, but different sizes, or
- a cross product with several materials and several sizes.

Often with complex models the parameter of interest is buried within the model structure. One approach is to clone the models using a deep copy of the entire structure, then tie together parameters for the bits that are changing. This proves to be confusing and difficult for new python programmers, so instead *FitProblem* was extended to support *FreeVariables*. The FreeVariables class allows you to use the same model structure with different data sets, but have some parameters that vary between the models. Each varying parameter is a slot, and FreeVariables keeps an array of parameters (actually a *ParameterSet*) to fill that slot, one for each model.

To define the free variables, you need the names of the different models, a parameter slot to hold the values, and a list of the different parameter values for each model. You then define the free variables as follows:

```python
free = FreeVariables(names=["model1", "model2", ...],
                      p1=model.p1, p2=model.p2, ...
)```

(continues on next page)
The slots can be referenced by name, with the underlying parameters referenced by variable number. In the above, `free.p1[1]` refers to the parameter p1 when fitting data2. You can also refer to the slots by name, such as `free.p1[data2.name]`. The parameters in the slots have the usual properties of parameters, such as values and fit ranges. Setting the fit range makes the parameter a fitted parameter, and the fit will give the uncertainty on each parameter independently. Parameters can be copied, so that a pair of models can share the same value.

The following examples shows a neutron scattering problems with two datasets, one measured with light water and the other measured with heavy water, you can share the same material object, but use the light water scattering factors in the first and the heavy water scattering factors in the second. The problem would be composed as follows:

```python
material = SLD('silicon', rho=2.07)
solvent = SLD('solvent')  # unspecified rho
model = Sphere(radius=10, material=material, solvent=solvent)
M1 = ScatteringFitness(model, hydrogenated_data)
M2 = ScatteringFitness(model, deuterated_data)
free = FreeVariables(names=['hydrogenated', 'deuterated'], solvent=solvent.sld)
free.solvent.values = [-0.561, 6.402]
model.radius.range(1, 35)
problem = FitProblem([M1, M2], freevars=free)
```

In this particular example, the solvent is fixed for each measurement, and the sphere radius is allowed to vary between 1 and 35. Since the radius is not a free variable, the fitted radius will be chosen such that it minimizes the combined fitness of both models. In a more complicated situation, we may not know either the sphere radius or the solvent densities, but still the radius is shared between the two models. In this case we could set:

```python
fv.solvent.range(-1, 7)
```

and the SLD of the solvent would be fitted independently in the two data sets. Notice that we did not refer to the individual model index when setting the range. This is a convenience—range, pm and pmp can be set on the entire set as above, or individually using, e.g.,

```python
fv.solvent[0].range(-1, 0)
fv.solvent[1].range(6, 7)
```

### 3.5 Fitting

- **Quick Fit**
- **Uncertainty Analysis**
- **Using the posterior distribution**
- **Publication Graphics**
- **Tough Problems**
- **Command Line**

Obtaining a good fit depends foremost on having the correct model to fit.
For example, if you are modeling a curve with spline, you will overfit the data if you have too many spline points, or underfit it if you do not have enough. If the underlying data is ultimately an exponential, then the spline order required to model it will require many more parameters than the corresponding exponential.

Even with the correct model, there are systematic errors to address (see Data Representation). A distorted sample can lead to broader resolution than expected for the measurement technique, and you will need to adjust your resolution function. Imprecise instrument control will lead to uncertainty in the position of the sample, and corresponding changes to the measured values. For high precision experiments, your models will need to incorporate these instrument effects so that the uncertainty in instrument configuration can be properly accounted for in the uncertainty in the fitted parameter values.

### 3.5.1 Quick Fit

While generating an appropriate model, you will want to perform a number of quick fits. The Nelder-Mead Simplex works well for this. You will want to run enough iterations --steps=1000 so the algorithm has a chance to converge. Restarting a number of times --starts=10 gives a reasonably thorough search of the fit space. Once the fit converges, additional starts are very quick. From the graphical user interface, using --starts=1 and clicking the fit button to improve the fit as needed works pretty well. From the command line interface, the command line will be something like:

```
bumps --fit=amoeba --steps=1000 --starts=20 --parallel model.py --store=T1
```

Here, the results are kept in a directory --store=T1 relative to the current directory, with files containing the current model in model.py, the fit result in model.par and a plots in model-*.png. The parallel option indicates that multiple cores should be used on the cpu when running the fit.

The fit may be able to be improved by using the current best fit value as the starting point for a new fit:

```
bumps --fit=amoeba --steps=1000 --starts=20 --parallel model.py --store=T1 --pars=T1/model.par
```

If the fit is well behaved, and a numerical derivative exists, then switching to Quasi-Newton BFGS is useful, in that it will very rapidly converge to a nearby local minimum.

```
bumps --fit=newton model.py --pars=T1/model.par --store=T1
```

Differential Evolution is an alternative to Nelder-Mead Simplex, perhaps a little more likely to find the global minimum but somewhat slower. This is a population based algorithms in which several points from the current population are selected, and based on the position and value, a new point is generated. The population is specified as a multiplier on the number of parameters in the model, so for example an 8 parameter model with DE’s default population --pop=10 would create 80 points each generation. This algorithms can be called from the command line as follows:

```
bumps --fit=de --steps=3000 --parallel model.py --store=T1
```

Some fitters save the complete state of the fitter on termination so that the fit can be resumed. Use --resume=path/to/previous/store to resume. The resumed fit also needs a --store=path/to/store, which could be the same as the resume path if you want to update it, or it could be a completely new path.

See Optimizer Selection for a description of the available optimizers, and Bumps Options for a description of all the bumps options.

### 3.5.2 Uncertainty Analysis

More important than the optimal value of the parameters is an estimate of the uncertainty in those values. The best fit is an accident of the measurement; perform the measurement again and you will get a different optimum. Given the
uncertainty in the measurement, there is a joint distribution of parameter values that are consistent with the measurement. For example, when fitting a line, the choice of slope will affect the range of intercepts that fit the data. The goal of uncertainty analysis is to determine this distribution and summarize it for the reader.

By casting our problem as the likelihood of seeing the data given the model, we not only give ourselves the ability to incorporate prior information into the fit systematically, but we also give ourselves a strong foundation for assessing the uncertainty of the parameters.

There are multiple ways to perform the analysis:

1. **Bayesian inference.** Given the probability on the parameters and the probability that the measured data will be seen with those parameters, infer the probability of the parameters given the measured data. This is the primary method in Bumps and will be discussed at length below.

2. **Sensitivity analysis.** Given the best fit parameter values, look at the curvature around that point as a normal distribution with covariance computed from the Hessian matrix. Further, pretend that there is no interaction between the parameters (that is they are uncorrelated and independent), and report the uncertainty as the square root of the diagonal. This is the default method for most optimizers in Bumps.

3. **Uncertainty contour.** Assuming the measurement data is independent and normally distributed, a given increase in $\chi^2$ above the minimum corresponds to 1-$\sigma$ confidence interval. By following this contour you can find the set of all points $\xi$ such that $\chi^2(\xi) = \chi^2(x) + C$ where $x$ is the point of maximum likelihood. Look in Numerical Recipes chapter on nonlinear least squares for a more complete discussion. Bumps does not include algorithms for this kind of analysis.

4. **Forward Monte Carlo.** Bumps has the option \texttt{--resynth} to perform a forward Monte Carlo estimate of the maximum likelihood. That is, you can use the measurement uncertainty to “rerun” the experiment, synthesizing a new dataset with the same uncertainty but slightly different values, then find the new maximum likelihood. After $n$ runs you will be able to estimate the uncertainty in the best fit parameters. This method can be applied with any of the optimizers.

5. **Repeated measurement.** A direct way to estimate the parameter uncertainty is to repeat the experiment many times and look at the distribution of best fit results. This is the classic approach which you need to follow if you don’t know anything about the uncertainty in your measurement processes (other than the assumption of independence between measurements). You can use this during experimental design, simulating the experiment in different conditions to figure out the best strategy to retrieve the quantity of interest. For example, to plan a reflectometry experiment you want to know if it would be better to measure with a pair of contrast agents, or to spend twice as long on a single contrast. The result gives the expected uncertainty in the parameters before the measurement is ever performed. You might call this model driven forward Monte Carlo as opposed to the data driven forward MC listed above.

Bayesian inference is performed using \texttt{DREAM}. This is a Markov chain Monte Carlo (MCMC) method with a differential evolution step generator. Like simulated annealing, the MCMC explores the space using a random walk, always accepting a better point, but sometimes accepting a worse point depending on how much worse it is.

DREAM can be started with a variety of initial populations. The random population \texttt{--init=random} distributes the initial points using a uniform distribution across the space of the parameters. Latin hypersquares \texttt{--init=lhs} improves on random by making sure that there is on value for each subrange of every variable. The covariance population \texttt{--init=cov} selects points from the uncertainty ellipse computed from the derivative at the initial point. This method will fail if the fitting parameters are highly correlated and the covariance matrix is singular. The $\epsilon$-ball population \texttt{--init=eps} starts DREAM from a tiny region near the initial point and lets it expand from there. It can be useful to start with an epsilon ball from the previous best point when DREAM fails to converge using a more diverse initial population.

The Markov chain will take time to converge on a stable population. This burn in time needs to be specified at the start of the analysis. After burn, DREAM will collect all points visited for $N$ iterations of the algorithm. If the burn time was long enough, the resulting points can be used to estimate uncertainty on parameters.

A common command line for running DREAM is:
Bayesian uncertainty analysis is described in the GUM Supplement 1.[8] and is a valid technique for reporting parameter uncertainties in NIST publications. Given sufficient burn time, points in the search space will be visited with probability proportional to the goodness of fit. The file T1/model.err contains a table showing for each parameter the mean(std), median and best values, and the 68% and 95% credible intervals. The mean and standard deviation are computed from all the samples in the returned distribution. These statistics are not robust: if the Markov process has not yet converged, then outliers will significantly distort the reported values. Standard deviation is reported in compact notation, with the two digits in parentheses representing uncertainty in the last two digits of the mean. Thus, for example, 24.9(28) is 24.9 ± 2.8. Median is the best value in the distribution. Best is the best value ever seen. The 68% and 95% intervals are the shortest intervals that contain 68% and 95% of the points respectively. In order to report 2 digits of precision on the 95% interval, approximately 1000000 samples drawn from the distribution are required, or steps = 1000000/(#parameters #pop). The 68% interval will require fewer draws, though how many has not yet been determined.

Histogramming the set of points visited will gives a picture of the probability density function for each parameter. This histogram is generated automatically and saved in T1/model-var.png. The histogram range represents the 95% credible interval, and the shaded region represents the 68% credible interval. The green line shows the highest probability observed given that the parameter value is restricted to that bin of the histogram. With enough samples, this will correspond to the maximum likelihood value of the function given that one parameter is restricted to that bin. In practice, the analysis has converged when the green line follows the general shape of the histogram.
The correlation plots show that the parameters are not uniquely determined from the data. For example, the thickness of lamellae 3 and 4 are strongly anti-correlated, yielding a 95% CI of about 1 nm for each compared to the bulk nafion thickness CI of 0.2 nm. Summing lamellae thickness in the sampled points, we see the overall lamellae thickness has a CI of about 0.3 nm. The correlation plot is saved in T1/model-corr.png.

To assure ourselves that the uncertainties produced by DREAM do indeed correspond to the underlying uncertainty in the model, we perform a Monte Carlo forward uncertainty analysis by selecting 50 samples from the computed posterior distribution, computing the corresponding theory function and calculating the normalized residuals. Assuming that our measurement uncertainties are approximately normally distributed, approximately 68% of the normalized residuals should be within +/- 1 of the residual for the best model, and 98% should be within +/- 2. Note that our best fit does not capture all the details of the data, and the underlying systematic bias is not included in the uncertainty.
estimates.

Plotting the profiles generated from the above sampling method, aligning them such that the cross correlation with the best profile is maximized, we see that the precise details of the lamellae are uncertain but the total thickness of the lamellae structure is well determined. Bayesian analysis can also be used to determine relative likelihood of different number of layers, but we have not yet performed this analysis. This plot is stored in T1/model-errors.png.

The trace plot, T1/model-trace.png, shows the mixing properties of the first fitting parameter. If the Markov process is well behaved, the trace plot will show a lot of mixing. If it is ill behaved, and each chain is stuck in its own separate local minimum, then distinct lines will be visible in this plot.

The convergence plot, T1/model-logp.png, shows the log likelihood values for each member of the population. When the Markov process has converged, this plot will be flat with no distinct lines visible. If it shows a general upward sweep, then the burn time was not sufficient, and the analysis should be restarted. The ability to continue to burn from the current population is not yet implemented.

Just because all the plots are well behaved does not mean that the Markov process has converged on the best result. It is practically impossible to rule out a deep minimum with a narrow acceptance region in an otherwise unpromising part of the search space.

In order to assess the DREAM algorithm for suitability for our problem space we did a number of tests. Given that our fit surface is multimodal, we need to know that the uncertainty analysis can return multiple modes. Because the fit problems may also be ill-conditioned, with strong correlations or anti-correlations between some parameters, the uncertainty analysis needs to be able to correctly indicate that the correlations exist. Simple Metropolis-Hastings sampling does not work well in these conditions, but we found that DREAM is able to handle them. We are still affected by the curse of dimensionality. For correlated parameters in high dimensional spaces, even DREAM has difficulty taking steps which lead to improved likelihood. For example, we can recover an eight point spline with generous ranges on its 14 free parameters close to 100% of the time, but a 10 point spline is rarely recovered.

3.5.3 Using the posterior distribution

You can load the DREAM output population and perform uncertainty analysis operations after the fact. To run an interactive bumps session use the following:

```
bumps -i
```

First you need to import some functions:

```python
import os
import matplotlib.pyplot as plt
from bumps.dream.state import load_state
from bumps.dream.views import plot_corrmatrix
from bumps.dream.stats import var_stats, format_vars
from bumps.dream.varplot import plot_vars
```

Then you need to reload the MCMC chains:

```python
store = "/tmp/t1"  # path to the --store="/tmp/t1" directory
modelname = "model"  # model file name without .py extension

# Reload the MCMC data
basename = os.path.join(store, modelname)
state = load_state(basename)
state.mark_outliers()  # ignore outlier chains

# Attach the labels from the .par file:
```

(continues on next page)
with open(basename+".par") as fid:
    state.labels = [" ".join(line.strip().split()[:-1]) for line in fid]

Now you can plot the data:

```
state.show()  # Create the standard plots
```

You can choose to plot only some of the variables:

```
# Select the data to plot (the 3rd and the last two in this case):
draw = state.draw(vars=[2, -2, -1])

# Histograms
stats = var_stats(draw)  # Compute statistics such as the 90% interval
print(format_vars(stats))
plt.figure()
plot_vars(draw, stats)

# Correlation plots
plt.figure()
plot_cormatrix(draw)
```

You can restrict those variables to a certain range. For example, to restrict the third parameter to [0.8, 1.0] and the last to [0.2, 0.4]:

```
from bumps.dream import views
selection={2: (0.8,1.0), -1:(0.2,0.4),...}
draw = state.draw(vars=[2, -2, -1], selection=selection)
```

You can add create derived variables using a function to generate the new variable from some combination of existing variables. For example, to add the first two variables together to create the derived variable “x+y” use:

```
state.derive_vars(lambda p: p[0]+p[1], labels=["x+y"])  
```

You can generate multiple derived parameters at a time with a function that returns a sequence:

```
state.derive_vars(lambda p: (p[0]*p[1],p[0]-p[1]), labels=["x*y","x-y"])  
```

These new parameters will show up in the plots:

```
state.show()
```

Here is an example from a fit to bovine serum albumin with a two layer model. The parameter of interest ($\Gamma$) is derived from the SLD $\rho$ and thickness $t$ of the constituent layers using $\Gamma = 0.06955(\rho_1t_1 + \rho_2t_2)$. Using intermediate values for $\rho_1t_1$ and $\rho_2t_2$ to show the difference between gaussian error propagation and full correlation analysis, the derived parameters as set up as follows:

```
from bumps.dream.state import load_state
state = load_state("1000ppm_Pb4.9 NRW_OM_2layer model")
state.labels = ["r1", "t1", "r2", "t2"]
state.derive_vars(lambda p: (p[0]*p[1],p[2]*p[3],0.06955*(p[0]*p[1]+p[2]*p[3])),
    labels=["r1t1","r2t2","G"])
state.show()
```

This gives the following output:
Parameter | mean     | median   | best     | 68% interval | 95% interval |
---|---------|----------|----------|--------------|--------------|
1 | r1  | 0.3321(98) | 0.3322 | 0.3327 | [0.322 0.342] | [0.312 0.351] |
2 | t1  | 50.37(89) | 50.381  | 50.286 | [49.47 51.21] | [48.49 52.21] |
3 | r2  | 1.199(22) | 1.1976  | 1.1980 | [1.177 1.224] | [1.158 1.242] |
4 | t2  | 24.90(80) | 24.922  | 24.901 | [24.06 25.76] | [23.37 26.44] |
5 | r1t1| 16.73(58) | 16.712  | 16.729 | [16.16 17.30] | [15.61 17.86] |
6 | r2t2| 29.84(48) | 29.863  | 29.832 | [29.36 30.33] | [28.87 30.78] |
7 | G   | 3.239(27) | 3.238   | 3.238  | [3.21 3.27]  | [3.19 3.29]  |

Using simple gaussian propagation of errors (from the wonderfully convenient uncertainties package) can compare the computed uncertainties:

```python
from uncertainties import ufloat as U
C = 0.06955
r1t1 = U(0.3321, 0.0098) * U(50.37, 0.89)
r2t2 = U(1.199, 0.022) * U(24.90, 0.80)
G = C*(r1t1 + r2t2)
print("r1*t1 =", r1t1)
print("r2*t2 =", r2t2)
print("G =", C*(r1t1 + r2t2))
```

which produces:

```
r1*t1 = 16.7 ± 0.6  # same as forward MC
r2*t2 = 29.9 ± 1.1  # compared to 29.8 ± 0.5 from forward MC
G = 3.24 ± 0.09    # compared to 3.24 ± 0.03 from forward MC
```

That is, the gaussian approximation assuming uncorrelated uncertainties is 3x larger than the forward Monte Carlo approximation from the joint distribution of the fitted parameters. Much of the reduction comes from the strong negative correlation between \( \rho_2 \) and \( t_2 \), with the remainder coming from the negative correlation between the products \( \rho_1 t_1 \) and \( \rho_2 t_2 \).

You can see this in the correlation plots, with \( r_2:t_2 \) having a very narrow diagonal (hence strong correlation) and \( r_1:t_1:r_2\times t_2 \) having a somewhat wider diagonal (hence weaker correlation).

The plotting code is somewhat complicated, and matplotlib doesn’t have a good way of changing plots interactively. If you are running directly from the source tree, you can modify the dream plotting libraries as you need for a one-off plot, then replot the graph:

```
# ... change the plotting code in dream.views/dream.corrplot
reload(dream.views)
```

(continues on next page)
Be sure to restore the original versions when you are done. If the change is so good that everyone should use it, be sure to feed it back to the community via the bumps source control system at github.

### 3.5.4 Publication Graphics

The matplotlib package is capable of producing publication quality graphics for your models and fit results, but it requires you to write scripts to get the control that you need. These scripts can be run from the Bumps application by first loading the model and the fit results then accessing their data directly to produce the plots that you need.

The model file (call it `plot.py`) will start with the following:

```python
import sys
from bumps.cli import load_problem, load_best
model, store = sys.argv[1:3]
problem = load_problem([model])
load_best(problem, os.path.join(store, model[:-3]+".par"))
chisq = problem.chisq
print("chisq", chisq)
```

Assuming your model script is in `model.py` and you have run a fit with `--store=X5`, you can run this file using:

```
$ bumps plot.py model.py X5
```

Now `model.py` is loaded and the best fit parameters are set.

To produce plots, you will need access to the data and the theory. This can be complex depending on how many models you are fitting and how many datasets there are per model. For single experiment models defined by `FitProblem`, your original experiment object is referenced by `problem.fitness`. For simultaneous refinement defined by `FitProblem` with multiple `Fitness` objects, use `problem.models[k].fitness` to access the experiment for model `k`. Your experiment object should provide methods for retrieving the data and plotting data vs. theory.

How does this work in practice? Consider the reflectivity modeling problem where we have a simple model such as nickel film on a silicon substrate. We measure the specular reflectivity as various angles and try to recover the film thickness. We want to make sure that our model fits the data within the uncertainty of our measurements, and we want some graphical representation of the uncertainty in our film of interest. The refl1d package provides tools for generating the sample profile uncertainty plots. We access the experiment information as follows:

```python
experiment = problem.fitness
z, rho, irho = experiment.smooth_profile(dz=0.2)
# ... insert profile plotting code here ...
QR = experiment.reflectivity()
for p, th in self.parts(QR):
    # ... insert reflectivity plotting code here ...
```

Next we can reload the the error sample data from the DREAM MCMC sequence:

```python
import dream.state
from bumps.errplot import calc_errors_from_state, align_profiles
```
state = load_state(os.path.join(store, model[:-3]))
state.mark_outliers()
# ... insert correlation plots, etc. here ...
profiles, slabs, Q, residuals = calc_errors_from_state(problem, state)
aligned_profiles = align_profiles(profiles, slabs, 2.5)
# ... insert profile and residuals uncertainty plots here ...

The function `bumps.errplot.calc_errors_from_state()` calls the calc_errors function defined by the reflectivity model. The return value is arbitrary, but should be suitable for the show_errors function defined by the reflectivity model.

Putting the pieces together, here is a skeleton for a specialized plotting script:

```python
import sys
import pylab
from bumps.dream.state import load_state
from bumps.cli import load_problem, load_best
from bumps.errplot import calc_errors_from_state
from refl1d.align import align_profiles

model, store = sys.argv[1:3]

problem = load_problem([model])
load_best(problem, os.path.join(store, model[:-3]+".par"))

chisq = problem.chisq
experiment = problem.fitness
z, rho, irho = experiment.smooth_profile(dz=0.2)
# ... insert profile plotting code here ...
QR = experiment.reflectivity()
for p, th in self.parts(QR):
    # ... insert reflectivity plotting code here ...

# Loading errors is expensive; may not want to do so all the time.
if 1:
    # Loading errors is expensive; may not want to do so all the time.
    state = load_state(os.path.join(store, model[:-3]))
    state.mark_outliers()
    # ... insert correlation plots, etc. here ...
    profiles, slabs, Q, residuals = calc_errors_from_state(problem, state)
    aligned_profiles = align_profiles(profiles, slabs, 2.5)
    # ... insert profile and residuals uncertainty plots here ...

pylab.show()
raise Exception() # We are just plotting; don't run the model
```

### 3.5.5 Tough Problems

**Note:** DREAM is currently our most robust fitting algorithm. We are exploring other algorithms such as parallel tempering, but they are not currently competitive with DREAM.

With the toughest fits, for example freeform models with arbitrary control points, DREAM only succeeds if the model is small or the control points are constrained. We have developed a parallel tempering (fit=pt) extension to DREAM. Whereas DREAM runs with a constant temperature, $T = 1$, parallel tempering runs with multiple temperatures...
concurrently. The high temperature points are able to walk up steep hills in the search space, possibly crossing over into a neighbouring valley. The low temperature points aggressively seek the nearest local minimum, rejecting any proposed point that is worse than the current. Differential evolution helps adapt the steps to the shape of the search space, increasing the chances that the random step will be a step in the right direction. The current implementation uses a fixed set of temperatures defaulting to \(--T\text{min}=0.1\) through \(--T\text{max}=10\) in \(--nT=25\) steps; future versions should adapt the temperature based on the fitting problem.

Parallel tempering is run like dream, but with optional temperature controls:

```
bumps --fit=dream --burn=1000 --samples=1e5 --init=cov --parallel --pars=T1/model.par model.py --store=T2
```

Parallel tempering does not yet generate the uncertainty plots provided by DREAM. The state is retained along the temperature for each point, but the code to generate histograms from points weighted by inverse temperature has not yet been written.

Parallel tempering performance has been disappointing. In theory it should be more robust than DREAM, but in practice, we are using a restricted version of differential evolution with the population defined by the current chain rather than a set of chains running in parallel. When the Markov chain has converged these populations should be equivalent, but apparently this optimization interferes with convergence. Time permitting, we will improve this algorithm and look for other ways to improve upon the robustness of DREAM.

### 3.5.6 Command Line

The GUI version of Bumps is slower because it frequently updates the graphs showing the best current fit.

Run multiple models overnight, starting one after the last is complete by creating a batch file (e.g., run.bat) with one line per model. Append the parameter –batch to the end of the command lines so the program doesn’t stop to show interactive graphs:

```
bumps model.py ... --parallel --batch
```

You can view the fitted results in the GUI the next morning using:

```
bumps --edit model.py --pars=T1/model.par
```

### 3.6 Optimizer Selection

Bumps has a number of different optimizers available, each with its own control parameters:

- *Levenberg-Marquardt*
- *Nelder-Mead Simplex*
- *DREAM*
- *Differential Evolution*
- *Quasi-Newton BFGS*
- *Random Lines* [experimental]
- *Particle Swarm* [experimental]
- *Parallel Tempering* [experimental]
In general there is a trade-off between convergence rate and robustness, with the fastest algorithms most likely to find a local minimum rather than a global minimum. The gradient descent algorithms \textit{(Levenberg-Marquardt, Quasi-Newton BFGS)} tend to be fast but they will find local minima only, while the population algorithms \textit{(DREAM, Differential Evolution)} are more robust and likely slower. \textit{Nelder-Mead Simplex} is somewhere between, with a small population keeping the search local but more robust than the gradient descent algorithms.

Each algorithm has its own set of control parameters for adjusting the search process and the stopping conditions. The same option may mean slightly different things to different optimizers. The Bumps package provides a dialog box for selecting the optimizer and its options when running the fit wx application. This only includes the common options for the most useful optimizers. For full control, the fit will need to be run from the command line interface or through a python script.

For parameter uncertainty, most algorithms use the covariance matrix at the optimum to estimate an uncertainty ellipse. This is okay for a preliminary analysis, but only works reliably for weakly correlated parameters. For full uncertainty analysis, \textit{DREAM} uses a random walk to explore the parameter space near the minimum, showing pair-wise correlations amongst the parameter values. In order for \textit{DREAM} to return the correct uncertainty, the function to be optimized should be a conditional probability density, with $nllf$ as the negative log likelihood function of seeing point $x$ in the parameter space. Other functions can be fitted, but uncertainty estimates will be meaningless.

Most algorithms have been adapted to run in parallel at least to some degree. The implementation is not heavily tuned, either in terms of minimizing the overhead per function evaluation or for distributing the problem across multiple processors. If the theory function is implemented in parallel, then the optimizer should be run in serial. Mixed mode is also possible when running on a cluster with a multi-threaded theory function. In this case, only one theory function will be evaluated on each cluster node, but the optimizer will distribute the parameters values to the cluster nodes in parallel. Do not run serial algorithms \textit{(Levenberg-Marquardt, Quasi-Newton BFGS)} on a cluster.

We have included a number of optimizers in Bumps that did not perform particularly well on our problem sets. However, they may be perfect for your problem, so we have left them in the package for you to explore. They are not available in the GUI selection.
3.6.1 Levenberg-Marquardt

The Levenberg-Marquardt\textsuperscript{1,2} algorithm has been the standard method for non-linear data fitting. As a gradient descent trust region method, it starts at the initial value of the function and steps in the direction of the derivative until it reaches the minimum. Set up as an explicit minimization of the sum of square differences between theory and model, it uses a numerical approximation of the Jacobian matrix to set the step direction and an adaptive algorithm to set the size of the trust region.

\textsuperscript{1} Levenberg, K. *Quarterly Journal of Applied Mathematics* 1944, II (2), 164–168.
When to use

Use this method when you have a reasonable fit near the minimum, and you want to get the best possible value. This can then be used as the starting point for uncertainty analysis using DREAM. This method requires that the problem definition includes a residuals method, but this should always be true when fitting data.

When modeling the results of an experiment, the best fit value is an accident of the measurement. Redo the same measurement, and the slightly different values you measure will lead to a different best fit. The important quantity to report is the credible interval covering 68% (1-\(\sigma\)) or 95% (2-\(\sigma\)) of the range of parameter values that are somewhat consistent with the data.

This method uses lmfit from scipy, and does not run in parallel.

Options

Steps is the number of gradient steps to take. Each step requires a calculation of the Jacobian matrix to determine the direction. This needs \(2mn\) function evaluations, where \(n\) is the number of parameters and each function is evaluated and \(m\) data points (assuming center point formula for finite difference estimate of the derivative). The resulting linear equation is then solved, but for small \(n\) and expensive function evaluation this overhead can be ignored. Use --steps=n from the command line.

\(f(x)\) tolerance and \(x\) tolerance are used to determine when the fit has reached the point where no significant improvement is expected. If the function value does not improve significantly within the step, or the step is too short, then the fit will terminate. Use --ftol=v and --xtol=v from the command line.

From the command line, --starts=n will automatically restart the algorithm after it has converged so that a slightly better value can be found. If --keep_best is included then restart will use a value near the minimum, otherwise it will restart the fit from a random point in the parameter space.

Use --fit=lm to select the Levenberg-Marquardt fitter from the command line.

Notes

v0.8.2 Changed from scipy.leastsq to mpfit for better bounds handling. Use --fit=scipy.leastsq to restore the previous behaviour.
3.6.2 Nelder-Mead Simplex

The Nelder-Mead\textsuperscript{3} downhill simplex algorithm is a robust optimizer which does not require the function to be continuous or differentiable. It uses the relative values of the function at the corners of a simplex (an n-dimensional triangle) to decide which points of the simplex to update. It will take the worst value and try moving it inward or outward, or reflect it through the centroid of the remaining values stopping if it finds a better value. If none of these values are better, then it will shrink the simplex and start again. The name amoeba comes from the book \textit{Numerical Recipes}\textsuperscript{4} wherein they describe the search as acting like an amoeba, squeezing through narrow valleys as it makes its way down

\textsuperscript{3} Nelder, J. A.; Mead, R. \textit{The Computer Journal} 1965, 7 (4), 308–313. DOI: 10.1093/comjnl/7.4.308

to the minimum.

**When to use**

Use this method as a first fit to your model. If your fitting function is well behaved with few local minima this will give a quick estimate of the model, and help you decide if the model needs to be refined. If your function is poorly behaved, you will need to select a good initial value before fitting, or use a more robust method such as *Differential Evolution* or *DREAM*.

The uncertainty reported comes from a numerical derivative estimate at the minimum.

This method requires a series of function updates, and does not benefit much from running in parallel.

**Options**

*Steps* is the simplex update iterations to perform. Most updates require one or two function evaluations, but shrinking the simplex evaluates every value in the simplex. Use `--steps=n` from the command line.

*Starts* tells the optimizer to restart a given number of times. Each time it restarts it uses a random starting point. Use `--starts=n` from the command line.

*Simplex radius* is the initial size of the simplex, as a portion of the bounds defining the parameter space. If a parameter is unbounded, then the radius will be treated as a portion of the parameter value. Use `--radius=n` from the command line.

*x tolerance* and *f(x) tolerance* are used to determine when the fit has reached the point where no significant improvement is expected. If the simplex is tiny (that is, the corners are close to each other) and flat (that is, the values at the corners are close to each other), then the fit will terminate. Use `--xtol=v` and `--ftol=v` from the command line.

From the command line, use `--keep_best` so that restarts are centered on a value near the minimum rather than restarting from a random point within the parameter bounds.

Use `--fit=amoeba` to select the Nelder-Mead simplex fitter from the command line.
References

3.6.3 Quasi-Newton BFGS

Broyden-Fletcher-Goldfarb-Shanno\(^5\) is a gradient descent method which uses the gradient to determine the step direction and an approximation of the Hessian matrix to estimate the curvature and guess a step size. The step is further refined with a one-dimensional search in the direction of the gradient.

When to use

Like *Levenberg-Marquardt*, this method converges quickly to the minimum. It does not assume that the problem is in the form of a sum of squares and does not require a *residuals* method.

The $n$ partial derivatives are computed in parallel.

**Options**

*Steps* is the number of gradient steps to take. Each step requires a calculation of the Jacobian matrix to determine the direction. This needs $2mn$ function evaluations, where $n$ is the number of parameters and each function is evaluated and $m$ data points (assuming center point formula for finite difference estimate of the derivative). The resulting linear equation is then solved, but for small $n$ and expensive function evaluation this overhead can be ignored. Use `--steps=n` from the command line.

*Starts* tells the optimizer to restart a given number of times. Each time it restarts it uses a random starting point. Use `--starts=n` from the command line.

*f(x) tolerance* and *x tolerance* are used to determine when the fit has reached the point where no significant improvement is expected. If the function is small or the step is too short then the fit will terminate. Use `--ftol=v` and `--xtol=v` from the command line.

From the command line, `--keep_best` uses a value near the previous minimum when restarting instead of using a random value within the parameter bounds.

Use `--fit=newton` to select BFGS from the commandline.
Differential evolution is a population based algorithm which uses differences between points as a guide to selecting new points. For each member of the population a pair of points is chosen at random, and a difference vector is computed. This vector is scaled, and a random subset of its components are added to the current point based on crossover ratio. This new point is evaluated, and if its value is lower than the current point, it replaces it in the population. There are many variations available within DE that have not been exposed in Bumps. Interested users can modify `bumps.fitters.DEFit` and experiment with different crossover and mutation algorithms, and perhaps add them as command line options.

---

Differential evolution is a robust directed search strategy. Early in the search, when the population is disperse, the difference vectors are large and the search remains broad. As the search progresses, more of the population goes into the valleys and eventually all the points end up in local minima. Now the differences between random pairs will often be small and the search will become more localized.

The population is initialized according to the prior probability distribution for each parameter. That is, if the parameter is bounded, it will use a uniform random number generate within the bounds. If it is unbounded, it will use a uniform value in \([0,1]\). If the parameter corresponds to the result of a previous measurement with mean \(\mu\) and standard deviation \(\sigma\), then the initial values will be pulled from a gaussian random number generator.

**When to use**

Convergence with differential evolution will be slower, but more robust.

Each update will evaluate \(k\) points in parallel, where \(k\) is the size of the population.

**Options**

*Steps* is the number of iterations. Each step updates each member of the population. The population size scales with the number of fitted parameters. Use `--steps=n` from the command line.

*Population* determines the size of the population. The number of individuals, \(k\), is equal to the number of fitted parameters times the population scale factor. Use `--pop=k` from the command line.

*Crossover ratio* determines what proportion of the dimensions to update at each step. Smaller values will likely lead to slower convergence, but more robust results. Values must be between 0 and 1. Use `--CR=v` from the command line.

*Scale* determines how much to scale each difference vector before adding it to the candidate point. The selected mutation algorithm chooses a scale factor uniformly in \([0, F]\). Use `--F=v` from the command line.

*\(f(x)\) tolerance* and *\(x\) tolerance* are used to determine when the fit has reached the point where no significant improvement is expected. If the population is flat (that is, the minimum and maximum values are within tolerance) and tiny (that is, all the points are close to each other) then the fit will terminate. Use `ftol=v` and `xtol=v` from the command line.

Use `--fit=de` to select differential evolution from the commandline.
3.6.5 DREAM

DREAM\textsuperscript{7} is a population based algorithm like differential evolution, but instead of only keeping individuals which improve each generation, it will sometimes keep individuals which get worse. Although it is not fast and does not give the very best value for the function, we have found it to be a robust fitting engine which will give a good value given enough time.

The progress of each individual in the population from generation to generation can considered a Markov chain, whose transition probability is equal to the probability of taking the step times the probability that it keeps the step based on the difference in value between the points. By including a purely random stepper with some probability, the detailed

balance condition is preserved, and the Markov chain converges onto the underlying equilibrium distribution. If the theory function represents the conditional probability of selecting each point in the parameter space, then the resulting chain is a random draw from the posterior distribution.

This means that the DREAM algorithm can be used to determine the parameter uncertainties. Unlike the hessian estimate at the minimum that is used to report uncertainties from the other fitters, the resulting uncertainty need not be Gaussian. Indeed, the resulting distribution can even be multi-modal. Fits to measured data using theory functions that have symmetric solutions have shown all equivalent solutions with approximately equal probability.

When to use

Use DREAM when you need a robust fitting algorithm. It takes longer but it does an excellent job of exploring different minima and getting close to the global optimum.

Use DREAM when you want a detailed analysis of the parameter uncertainty.

Like differential evolution, DREAM will evaluate $k$ points in parallel, where $k$ is the size of the population.

Options

*Samples* is the number of points to be drawn from the Markov chain. To estimate the 68% interval to two digits of precision, at least $1e5$ (or 100,000) samples are needed. For the 95% interval, $1e6$ (or 1,000,000) samples are needed. The default $1e4$ samples gives a rough approximation of the uncertainty relatively quickly. Use `--samples=n` from the command line.

*Burn-in steps* is the number of iterations to required for the Markov chain to converge to the equilibrium distribution. If the fit ends early, the tail of the burn will be saved to the start of the steps. Use `--burn=n` from the command line.

*Population* determines the size of the population. The number of individuals, $k$, is equal to the number of fitted parameters times the population scale factor. Use `--pop=k` from the command line.

*Initializer* determines how the population will be initialized. The options are as follows:

- **eps** (epsilon ball), in which the entire initial population is chosen at random from within a tiny hypersphere centered about the initial point
- **lhs** (latin hypersquare), which chops the bounds within each dimension in $k$ equal sized chunks where $k$ is the size of the population and makes sure that each parameter has at least one value within each chunk across the population.
- **cov** (covariance matrix), in which the uncertainty is estimated using the covariance matrix at the initial point, and points are selected at random from the corresponding gaussian ellipsoid
- **random** (uniform random), in which the points are selected at random within the bounds of the parameters

Use `--init=type` from the command line.

*Thinning* is the amount of thinning to use when collecting the population. If the fit is somewhat stuck, with most steps not improving the fit, then you will need to thin the population to get proper statistics. Use `--thin=k` from the command line.

*Convergence* gives a cutoff value $\alpha$ for determining when the Markov chain has converged. The default is `--alpha=0.00` for no convergence tests. Various tests are used, such as comparing the distribution of points in the first part of the chain to the last part and looking for trends in the log-likelihood values. You may need to use smaller $\alpha$ for shorter sequences (samples over variables times population) since the test statistics will have higher variance. Convergence is tested every $n$ steps.

*Outliers* is the test to use to check for outlier chains. Default is `--outliers=none` for no outlier test. Options are `iqr`, which uses the inter-quartile range on the likelihoods, `grubbs`, which uses a t-test on the likelihoods, and `mahal`
which looks at the distance from the best chain in parameter space. Outlier removal occurs every $2n$ steps where $n$ is $\#\text{samples}/(\#\text{pars} \times \#\text{pop})$, or when the convergence test indicates the chains are stable. Outliers are replaced by non-outlier chains at random. These new chains need at least $n$ steps to mix before being used. If the MCMC exploration stops due to time, some of the chains may not be properly mixed.

*Burn-in trim* is used to clear spurious samples from the Markov chains. If --trim=true then bumps finds the “burn point” after which the chains appear to have converged. Samples before this point are ignored when computed statistics and making plots. The trimmed samples are still written to the MCMC output files so they will be available when the fit is resumed.

*Calculate entropy*, if true, computes the entropy for the fit. This is an estimate of the amount of information in the data. Use --entropy=method from the command line, where method is one of llf (default), gmm, mvn or wnn. See below for details.

*Steps*, if not zero, determines the number of iterations to use for drawing samples after burn in. Each iteration updates the full population, which is $(\text{population} \times \text{number of fitted parameters})$ points. This option is available for compatibility; it is more useful to set the number of samples directly. Use --steps=n from the command line.

Use --fit=dream to select DREAM from the commandline. Consider using --parallel and --checkpoint as well. When running in a batch queue, add --batch and use --mpi rather than --parallel.

**Output**

DREAM produces a number of different outputs, and there are a number of things to check before using its reported uncertainty values. The main goal of selecting --burn=n is to wait long enough to reach the equilibrium distribution.

For each parameter in the fit, DREAM finds the mean, median and best value, as well as the 68% and 95% credible intervals. The mean value is defined as $\int_{x} x P(x) \, dx$, which is just the expected value of the probability distribution for the parameter. The median value is the 50% point in the probability distribution, and the best value is the maximum likelihood value seen in the random walk. The credible intervals are the central intervals which capture 68% and 95% of the parameter values respectively. You need approximately 100,000 samples to get two digits of precision on the 68% interval, and 1,000,000 samples for the 95% interval.

<table>
<thead>
<tr>
<th>#</th>
<th>Parameter</th>
<th>mean</th>
<th>median</th>
<th>best</th>
<th>[68% interval]</th>
<th>[95% interval]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>M1.background</td>
<td>0.059925(41)</td>
<td>0.059924</td>
<td>0.059922</td>
<td>[0.05988 0.05997]</td>
<td>[0.05985 0.06000]</td>
</tr>
<tr>
<td>2</td>
<td>M1.radius</td>
<td>2345.3(15)</td>
<td>2345.234</td>
<td>2345.174</td>
<td>[2343.83 2346.74]</td>
<td>[2342.36 2348.29]</td>
</tr>
<tr>
<td>3</td>
<td>M1.radius.width</td>
<td>0.00775(41)</td>
<td>0.00774</td>
<td>0.00777</td>
<td>[0.0074 0.0081]</td>
<td>[0.0070 0.0086]</td>
</tr>
<tr>
<td>4</td>
<td>M1.scale</td>
<td>0.21722(20)</td>
<td>0.217218</td>
<td>0.217244</td>
<td>[0.21702 0.21743]</td>
<td>[0.21681 0.21761]</td>
</tr>
</tbody>
</table>

The *Convergence* plot shows the range of $\chi^2$ values in the population for each iteration. The band shows the 68% of values around the median, and the solid line shows the minimum value. If the distribution has reached equilibrium, then convergence graph should be roughly flat, with little change in the minimum value throughout the graph. If there is no convergence, then the remaining plots don’t mean much.

The *Correlations* plot shows cross correlation between each pair of parameters. If the parameters are completely uncorrelated then the boxes should contain circles. Diagonals indicate strong correlation. Square blocks indicate that the fit is not sensitive to one of the parameters. The range plotted on the correlation plot is determined by the 95% interval of the data. The individual correlation plots are too small to show the range of values for the parameters. These can instead be read from the *Uncertainty* plot for each parameter, which covers the same range of values and indicates 68% and 95% intervals. If there are some chains that are wandering around away from the minimum, then the plot will look fuzzy, and not have a nice blob in the center. If a correlation plot has multiple blobs, then there are

---

Fig. 1: This DREAM fit is incomplete, as can be seen on all four plots. The Convergence plot is still decreasing, Parameter Trace plot does not show random mixing of Markov chain values, the Correlations plots are fuzzy and mostly empty, the Uncertainty plot shows black histograms (indicating that there are a few stray values far away from the best) and green maximum likelihood spikes not matching the histogram (indicating that the region around the best value has not been adequately explored).
Fig. 2: This DREAM fit completed successfully. The *Convergence* plot is flat, the *Parameter Trace* plot is flat and messy, the *Correlations* plots show nice blobs (and a bit of correlation between the *M1.radius* parameter and the *M1.radius.width* parameter), and the uncertainty plots show a narrow range of -log(P) values in the mostly brown histograms and a good match to the green constrained maximum likelihood line.
multiple minima in your problem space, usually because there are symmetries in the problem definition. For example, a model fitting $x + a^2$ will have identical solutions for $\pm a$.

The Uncertainty plot shows histograms for each fitted parameter generated from the values for that parameter across all chains. Within each histogram bar the values are sorted and displayed as a gradient from black to copper, with black values having the lowest $\chi^2$ and copper values having the highest. The resulting histogram should be dark brown, with a black hump in the center and light brown tips. If there are large lumps of light brown, or excessive black then its likely that the optimizer did not converge. The green line over the histogram shows the best value seen within each histogram bin (the maximum likelihood given $p_k \equiv x$). With enough samples and proper convergence, it should roughly follow the outline of the histogram. The yellow band in the center of the plot represents the 68% interval for the data. The histogram cuts off at 95%. These values along with the median are shown as labels along the x axis. The green asterisk represents the best value, the green $E$ the mean value and the vertical green line the median value. If the fit is not sensitive to a parameter, or if two parameters are strongly correlated, the parameter histogram will show a box rather than a hump. Spiky shapes (either in the histogram or the maximum likelihood line) indicate lack of convergence or maybe not enough steps. A chopped histograms indicates that the range for that parameter is too small.

The Parameter Trace plot is diagnostic for models which have poor mixing. In this cases no matter how the parameter values are changing, they are landing on much worse values for the $\chi^2$. This can happen if the problem is highly constrained with many tight and twisty values.

The Data and Theory plot should show theory and data lining up pretty well, with the theory overlaying about 2/3 of the error bars on the data (1-σ = 68%). The Residuals plot shows the difference between theory and data divided by uncertainty. The residuals should be 2/3 within [-1, 1], They should not show any structure, such as humps where the theory misses the data for long stretches. This indicates some feature missing from the model, or a lack of convergence to the best model.

If entropy is requested, then Bumps will show the total number of bits of information in the fit, where entropy is defined as:

\[
S = \sum_k \log_2 p(\theta_k).
\]

Since we already have a sample from the posterior distribution $p(\Theta)$ the Monte Carlo integral should be $S \approx \sum_k \log_2 p(\theta_k)$. However, we do not know $p(\theta_k)$, especially when we are integrating over nuisance parameters and only computing entropy for the parameters of interest. There are numerous methods in the literature for performing this calculation, and we have implemented the following:

- **gmm** fits the MCMC sample to a Gaussian mixture model (GMM) and then estimates the entropy of the GMM through Monte Carlo integration.
- **llf** finds the average ratio between the unnormalized negative log likelihood (NLLF) and a kernel density estimate (sklearn KernelDensity with default options), then estimates the entropy from the normalized likelihood through Monte Carlo integration.\(^8\) This technique will not work for marginal likelihood estimates.
- **mvn** fits the MCMC sample to a multivariate Gaussian and returns the entropy of that Gaussian. This is fast and accurate when the sample is well behaved (i.e., the uncertainty distribution is approximately Gaussian).
- **wnn** estimates entropy from nearest-neighbour distances in the sample.\(^10\)

Using entropy and simulation we hope to be able to make experiment planning decisions in a way that maximizes information, by estimating whether it is better to measure more precisely or to measure different but related values and fit them with shared parameters.

\(^8\) Kramer, A.; Hasenauer, J.; Allgower, F.; Radde, N. In 2010 IEEE International Conference on Control Applications (CCA) 2010; pp 493–498. DOI: 10.1109/CCA.2010.5611198


3.6. Optimizer Selection
References

3.6.6 Particle Swarm

Inspired by bird flocking behaviour, the particle swarm\(^{11}\) algorithm is a population-based method which updates an individual according to its momentum and a force toward the current best fit parameter values. We did not explore variations of this algorithm in any detail.

When to use

Particle swarm performed well enough in our low dimensional test problems, but made little progress when more fit parameters were added.

The population updates can run in parallel, but the tiny population size limits the amount of parallelism.

Options

\(--\text{steps}=n\) is the number of iterations. Each step updates each member of the population. The population size scales with the number of fitted parameters.

\(--\text{pop}=k\) determines the size of the population. The number of individuals, \(k\), is equal to the number of fitted parameters times the population scale factor. The default scale factor is 1.

Use \(--\text{fit}=ps\) to select particle swarm from the commandline.

Add a few more lines

References

3.6.7 Random Lines

Most of the population based algorithms ignore the value of the function when choosing the points in the next iteration. Random lines\(^{12}\) is a new style of algorithm which fits a quadratic model to a selection from the population, and uses that model to propose a new point in the next generation of the population. The hope is that the method will inherit the robustness of the population based algorithms as well as the rapid convergence of the newton descent algorithms.

When to use

Random lines works very well for some of our test problems, showing rapid convergence to the optimum, but on other problems it makes very little progress.

The population updates can run in parallel.

Options

\(--\text{steps}=n\) is the number of iterations. Each step updates each member of the population. The population size scales with the number of fitted parameters.

\(--\text{pop}=k\) determines the size of the population. The number of individuals, \(k\), is equal to the number of fitted parameters times the population scale factor. The default scale factor is 0.5.


--CR=v is the crossover ratio, determining what proportion of the dimensions to update at each step. Values must be between 0 and 1.

--starts=n tells the optimizer to restart a given number of times. Each time it restarts it uses a random starting point.

--keep_best uses a value near the previous minimum when restarting instead of using a random value within the parameter bounds. This option is not available in the options dialog.

Use --fit=rl to select random lines from the commandline.

References

3.6.8 Parallel Tempering

Parallel tempering\textsuperscript{13} is an MCMC algorithm for uncertainty analysis. This version runs at multiple temperatures simultaneously, with chains at high temperature able to more easily jump between minima and chains at low temperature to fully explore the minima. Like DREAM it has a differential evolution stepper, but this version uses the chain history as the population rather than maintaining a population at each temperature.

This is an experimental algorithm which does not yet perform well.

When to use

When complete, parallel tempering should be used for problems with widely spaced local minima which dream cannot fit.

Options

--steps=n is the number of iterations to include in the Markov chain. Each iteration updates the full population. The population size scales with the number of fitted parameters.

--burn=n is the number of iterations to required for the Markov chain to converge to the equilibrium distribution. If the fit ends early, the tail of the burn will be saved to the start of the steps.

--CR=v is the differential evolution crossover ratio to use when computing step size and direction. Use a small value to step through the dimensions one at a time, or a large value to step through all at once.

-nT=k, -Tmin=v and --Tmax=v specify a log-spaced initial distribution of temperatures. The default is 25 points between 0.1 and 10. DREAM runs at a fixed temperature of 1.0.

Use --fit=pt to select parallel tempering from the commandline.

References

3.7 Bumps Options

\textit{Bumps} has a number of options available to control the fits and the output. On the command line, each option is either --option if it is True/False or --option=value if the option takes a value. The fit control form is used by graphical users interfaces to set the optimizer and its controls and stopping conditions. The long form name of the the option will be used on the form. Not all controls will appear on the form, and will be set from the command line.

\textsuperscript{13} Swendsen, R. H.; Wang J. S. Replica Monte Carlo simulation of spin glasses \textit{Physical Review Letters} 1986, 57, 2607-2609
Need to describe the array of output files produced by optimizers, particularly dream. Some of them (convergence plot, model plot, par file, model file) are common to all. Others (mcmc points) are specific to one optimizer.

## 3.7.1 Bumps Command Line

Usage:

```
bumps [options] modelfile [modelargs]
```

The modelfile is a Python script (i.e., a series of Python commands) which sets up the data, the models, and the fittable parameters. The model arguments are available in the modelfile as `sys.argv[1:]`. Model arguments may not start with `'-``. The options all start with `'-`' and can appear in any order anywhere on the command line.

## 3.7.2 Problem Setup

### --pars

Set initial parameter values from a previous fit. The par file is a list of lines with parameter name followed by parameter value on each line. The parameters must appear with the same name and in the same order as the fitted parameters in the model. Additional parameters are ignored. Missing parameters are filled using LHS. `--preview` will show the model parameters.

### --shake

Set random initial values for the parameters in the model. Note that shake happens after `--simulate` so that you can simulate a random model, shake it, then try to recover its initial values.

### --simulate

Simulate a dataset using the initial problem parameters. This is useful when setting up a model before an experiment to see what data it might produce, and for seeing how well the fitting program might recover the parameters of interest.

### --simrandom

Simulate a dataset using random initial parameters. Because `--shake` is applied after `--simulate`, we need a separate way to shake the parameters before simulating the model.

### --noise

Set the noise percentage on the simulated data. The default is 5 for 5% normally distributed uncertainty in the measured values. Use `--noise=data` to use the uncertainty on a dataset in the simulation.

### --seed

Set a specific seed to the random number generator. This happens before shaking and simulating so that fitting tests, and particularly failures, can be reliably reproduced. The numpy random number generator is used for all values, so any consistency guarantees between versions of bumps over time and across platforms depends on the consistency of
the numpy generators. If no seed is specified then one will be generated and printed so that the fit can be rerun with the same random sequence.

### 3.7.3 Stopping Conditions

**--steps**

Steps is the number of iterations that the algorithm will perform. The meaning of iterations will differ from optimizer to optimizer. In the case of population based optimizers such as *Differential Evolution*, each step is an update to every member of the population. For local descent optimizers such as *Nelder-Mead Simplex* each step is an iteration of the algorithm. *DREAM* uses steps plus *–burn* for the total number of iterations.

**--samples**

Samples sets the number of function evaluations. This is an alternative for setting the number of iterations of the algorithm, used when *--steps* is zero. Population optimizers perform *–pop* times the number of parameters in the fit for each step of the operation, so given the desired number of samples, you can control the number of steps. The number of samples is particularly convenient for *DREAM* (the only optimizer for which it is implemented at the moment), where 100,000 samples are needed to estimate the 1-sigma interval to 2 digits of accuracy (assuming an approximately gaussian distribution), and 1,000,000 samples are needed for the 95% confidence interval. Like *--steps*, the total evaluations does not include any *–burn* iterations.

**--ftol**

$f(x)$ tolerance uses differences in the function value to decide when the fit is complete. The different fitters will interpret this in different ways. The Newton descent algorithms (*Quasi-Newton BFGS, Levenberg-Marquardt*) will use this as the minimum improvement of the function value with each step. The population-based algorithms (*Differential Evolution, Nelder-Mead Simplex*) will use the maximum difference between highest and lowest value in the population. *DREAM* does not use this stopping condition.

**--xtol**

$x$ tolerance uses differences in the parameter value to decide when the fit is complete. The different fitters will interpret this in different ways. The Newton descent algorithms (*Quasi-Newton BFGS, Levenberg-Marquardt*) will use this as the minimum change in the parameter values with each step. The population-based algorithmgs (*Differential Evolution, Nelder-Mead Simplex*) will use the maximum difference between highest and lowest parameter in the population. *DREAM* does not use this stopping condition.

**--time**

Max time is the maximum running time of the optimizer. This forces the optimizer to stop even if tolerance or steps conditions are not met. It is particularly useful for batch jobs run in an environment where the queuing system stops the job unceremoniously when the time allocation is complete. Time is checked between iterations, so be sure to set it well below the queue allocation so that it does not stop in the middle of an iteration, and so that it has time to save its state.
--alpha

Convergence is the test criterion to use when deciding if stopping conditions are met. This is for the variety of stopping tests built into the DREAM algorithm. Usual values are --alpha=0.01 or --alpha=0.05. Note that various stopping criteria depend on the number of samples and the chain length (where chain length x #pars x #pop = #samples), so there is no definitive value to use for alpha, but larger values will allow the fit to stop sooner.

3.7.4 Optimizer Controls

--fit

Fit Algorithm selects the optimizer. The available optimizers are:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>amoeba</td>
<td>Nelder-Mead Simplex</td>
</tr>
<tr>
<td>de</td>
<td>Differential Evolution</td>
</tr>
<tr>
<td>dream</td>
<td>DREAM</td>
</tr>
<tr>
<td>lm</td>
<td>Levenberg-Marquardt</td>
</tr>
<tr>
<td>newton</td>
<td>Quasi-Newton BFGS</td>
</tr>
<tr>
<td>pt</td>
<td>Parallel Tempering</td>
</tr>
<tr>
<td>ps</td>
<td>Particle Swarm</td>
</tr>
<tr>
<td>rl</td>
<td>Random Lines</td>
</tr>
</tbody>
</table>

The default fit method is --fit=amoeba.

--pop

Population determines the size of the population. For Differential Evolution and DREAM it is a scale factor, where the number of individuals, \( k \), is equal to the number of fitted parameters times pop. For Nelder-Mead Simplex the number of individuals is one plus the number of fitted parameters, as determined by the size of the simplex.

--init

Initializer is used by population-based algorithms (DREAM) to set the initial population. The options are as follows:

- *lhs* (latin hypersquare), which chops the bounds within each dimension in \( k \) equal sized chunks where \( k \) is the size of the population and makes sure that each parameter has at least one value within each chunk across the population.
- *eps* (epsilon ball), in which the entire initial population is chosen at random from within a tiny hypersphere centered about the initial point.
- *cov* (covariance matrix), in which the uncertainty is estimated using the covariance matrix at the initial point, and points are selected at random from the corresponding gaussian ellipsoid.
- *rand* (uniform random), in which the points are selected at random within the bounds of the parameters.

Nelder-Mead Simplex uses --radius to initialize its simplex. Differential Evolution uses a random number from the prior distribution for the parameter, if any.
**--burn**

*Burn-in Steps* is the number of iterations to required for the Markov chain to converge to the equilibrium distribution. If the fit ends early, the tail of the burn will be saved to the start of the steps. *DREAM* uses burn plus steps as the total number of iterations to run.

**--thin**

*Thinning* is used by the Markov chain analysis to give samples time to wander to different points in parameter space. In an ideal chain, there would be no correlation between points in the chain other than that which is dictated by the equilibrium distribution. However, if the space has complicated boundaries and taking a step can easily lead to a highly improbable point, then the chain may be stuck at the same value for long periods of time. If this is observed, then thinning can be used to only keep every \( n \)th step, giving the saved chain a better opportunity for good mixing.

**--CR**

*Crossover ratio* indicates the proportion of mixing which occurs with each iteration. This is a value in [0,1] giving the probability that each individual dimension will be selected for update in the next generation.

**--outliers**

*Outliers* is used to identify chains that are stuck in high local minima during dream burn-in. Options are:

- *iqr*: Use the interquartile range to determine the width of the distribution then exclude all chains whose log likelihood is more than two standard deviations below the first quartile.
- *grubbs*: Use a t-test to determine whether the samples in each chain are significantly different from the mean.
- *mahal*: Use the mahalanobis distance to determine whether the lowest probability chain is close to the remaining chain in parameter space. Only this chain will be marked as an outlier if the test fails.
- *none*: Don’t do any outlier trimming.

The default is **--outliers=none**. Outlier removal occurs every \( 2n \) steps where \( n \) is \#samples/(#pars #pop), or when the convergence test indicates the chains are stable.

Note that outliers are marked at the end of the fit using IQR and not included in the statistics, though they are saved in the MCMC files. This is independent of the **--outliers** setting.

**--F**

*Scale* is a factor applied to the difference vector before adding it to the parent in differential evolution.

**--radius**

*Simplex radius* is the radius of the initial simplex in *Nelder-Mead Simplex*.

**--nT**

*# Temperatures* is the number of temperature chains to run using parallel tempering. Default is 25.
--Tmin

*Min temperature* is the minimum temperature in the log-spaced series of temperatures to run using parallel tempering. Default is 0.1.

--Tmax

*Max temperature* is the maximum temperature in the log-spaced series of temperatures to run using parallel tempering. Default is 10.

--starts

*Starts* is the number of times to run the fit from random starting points.

--keep_best

If *Keep best* is set, then the each subsequent restart for the multi-start fitter keeps the best value from the previous fit(s).

### 3.7.5 Execution Controls

--store

Directory in which to store the results of the fit. Fits produce multiple files and plots. Rather than cluttering up the current directory, all the outputs are written to the store directory along with a copy of the model file.

--overwrite

If the store directory already exists then you need to include overwrite on the command line to reuse it. While inconvenient, this prevents accidental overwriting of fits that may have taken hours to generate.

--checkpoint

Save fit state every --checkpoint=n hours. [dream only]

--resume

Continue fit from a previous store directory. Use --resume or --resume= to reuse the existing store directory.

--parallel

Run fit using multiprocessing for parallelism. Use “--parallel=0” for all CPUs or “--parallel=n” for only “n” CPUs.

--mpi

Run fit using MPI for parallelism. Use command “mpirun -n cpus …” to run bumps for MPI. This will usually be the last line of a queue submission script. Be sure to include --time=... to limit the fit to run within the queue allocation time.
--batch

Run fit in batch mode. Progress updates are sent to STORE/MODEL.mon, and can be monitored using tail -f (unix, mac). When the fit is complete, the plot png files are created as usual, but the interactive plots are not shown. This allows you to set up a sequence of runs in a shell script where the first run completes before the next run starts. Batch is also useful for cluster computing where the cluster nodes do not have access to the outside network and can’t display an interactive window. Batch is automatic when running with --mpi.

--stepmon

Create a log file tracking each point examined during the fit. This does not provide any real utility except for generating plots of the population over time, which can be useful for understanding the different fitting methods.

3.7.6 Output Controls

`--err`

Show uncertainties at the end of the fit using the square root of the diagonals of the covariance matrix. See --cov.

--cov

Compute the covariance matrix for the model at the minimum. With gaussian uncertainties on the data, bumps is minimizing the sum of squares, so the Jacobian matrix is used for the covariance, formed from the numerical derivative of each residual with respect to each parameter. If the likelihood function is not a simple sum of squared residuals, then the Hessian matrix is used for the covariance, formed from the numerical derivative of the likelihood with respect to pairs of parameters.

--entropy

Calculate entropy is a flag which indicates whether entropy should be computed for the final fit. Entropy an estimate of the number of bits of information available from the fit. Use “--entropy=method” to specify the entropy calculation method. This can be one of:

- gmm: fit sample to a gaussian mixture model (GMM) with $5\sqrt{d}$ components where $d$ is the number fitted parameters and estimate entropy by sampling from the GMM.
- llf: estimates likelihood scale factor from ratio of density estimate to model likelihood, then computes Monte Carlo entropy from sample; this does not work for marginal likelihood estimates. DOI:10.1109/CCA.2010.5611198
- mvn: fit sample to a multi-variate Gaussian and return the entropy of the best fit gaussian; uses bootstrap to estimate uncertainty. This method is only valid if the sample distribution is approximately Gaussian.
- wnn: estimate entropy from weighted nearest-neighbor distances in sample. Note: use with caution. The results from this implementation are not consistent with other methods. DOI:10.1214/18-AOS1688

--plot

For problems that have different view options for plotting, select the default option to display. For example, when fitting a power law to a dataset, you may want to choose log or linear as the output plot type.
**--trim**

*Burn-in trim* finds the “burn point” after which the DREAM Markov chains appear to have converged and ignores all points before it when plotting or computing covariance and entropy. The trimmed points are still written to the MCMC output files so they will be available when the fit is resumed. Use `--trim=true` to set trimming.

**--noshow**

*No show* suppresses the plot window after the fit. This is done automatically when `--batch` is selected.

### 3.7.7 Bumps Controls

**--preview**

If the command contains `preview` then display model but do not perform a fitting operation. Use this to see the initial model before running a fit. It will also show the fit range.

**--chisq**

If the command contains `chisq` then show $\chi^2$ and exit. Use this to check that the model does not have any syntax errors.

**--resynth**

Run a resynth uncertainty analysis on the model. After finding a good minimum, you can rerun bumps with:

```
bumps --store=T1 --pars=T1/model.par --fit=amoeba --resynth=20 model.py
```

This will generate 20 data simulated datasets using the initial data values as the mean and the data uncertainty as the standard deviation. Each of these datasets will be fit with the specified optimizer, and the resulting parameters saved in `T1/model.rsy`. On completion, the parameter values can be loaded into python and averaged or histogrammed.

**--time_model**

Run the model `--steps` times and find the average run time per step. If `--parallel` is used, then the models will be run in parallel.

**--profile**

Run the model `--steps` times using the python profiler. This can be useful for identifying slow parts of your model definition, or alternatively, finding out that the model runtime is smaller than the Bumps overhead. Use a larger value of steps for better statistics.

### 3.7.8 Special Options

**--edit**

If the command contains `edit` then start the Bumps user interface so that you can interact with the model, adjusting fitted parameters with a slider and seeing how they impact the result.
--help, -h, -?

Use -?, -h or --help to show a brief description of each command line option.

-i, -m, -c, -p

The bumps program can be used as a python interpreter with numpy, scipy, matplotlib and bumps packages available. This is useful if you do not have python set up on your system, and you are using a bundled executable like Bumps or Refl1D on windows. Even if you have python, you may want to run the bumps post-analysis scripts through the bumps command which already has the appropriate path set up to bumps on your system.

The options are:

• -i: run an interactive interpreter.

• -m package.module: run a module as main. This is similar to python -m package.module with the python interpreter.

• -c expression: run a python command and quit.

• -p script.py: run a python script.

### 3.8 Calculating Entropy

Entropy is a measure of how much uncertainty is in the parameters. We can start with the simple case of a discrete parameter which can take on limited set of values. Using the formula for discrete entropy:

\[ H(x) = - \sum_x p(x) \log_2(x) \]

where \( x \) is the set of possible states of the parameter, we can examine a simple system with four states of equal probability:

Before the experiment, the entropy is \(-4(1/4) \log_2(1/4) = 2\) bits. After the experiment, which eliminates the states on the right, only two states are remaining with an entropy of 1 bit. The difference in entropy before and after the experiment is the information gain, which is 1 bit in this case.

Extending this concept to continuous parameters, we use:

\[ h(x) = - \int_{x \in X} p(x) \log_2(x) dx \]
For a parameter which is normally distributed, $x \sim N(\mu, \sigma)$, the entropy is:

$$h(x) = \frac{1}{2} \log_2(2\pi e \sigma^2)$$

Consider an experiment in which the parameter uncertainty $\sigma$ is reduced from $\sigma = 1$ before the experiment to $\sigma = \frac{1}{2}$ after the experiment:

![Entropy Comparison](image)

This experiment reduces the entropy from 2.05 bits to 1.05 bits, for an information gain of 1 bit.

For a multivariate normal $N(\bar{\mu}, \Sigma)$, the entropy is

$$h(N) = \frac{n}{2} \log_2(2\pi e) + \frac{1}{2} \log_2 |\Sigma|$$

where $n$ is the number of fitting parameters and $\Sigma$ is the covariance matrix relating the parameters. For an uncorrelated system, this is proportional to $\sum_{i=1}^{n} \log_2 \sigma_i$, with the individual parameter uncertainties $\sigma_i$. In effect, the entropy is a measure of overall uncertainty resulting after the fit.

Within bumps, most models start with a uniform prior distribution for the parameters set using the $x.range(low,high)$ or $x.pm(delta)$ for some parameter $x$. Some models set the prior probability to a normal distribution using $x.dev(sigma)$. Arbitrary prior probability distributions can be set using $x.bounds = Distribution(D)$ where $D$ is a distribution following the scipy.stats interface. The uncertainty on the data points does not directly enter into the entropy calculation. Instead, it has a direct influence on the calculation of the probability of seeing the data given the parameter, and so it influences the probability of the parameters after the fit. Increasing the error bars will increase the variance in the parameter estimation which will increase the entropy.

There are three ways that bumps can evaluate entropy. For the fitters which return a sample from the posterior distribution, such as DREAM, BUMPS can estimate the entropy directly from the sample. If the distribution is approximately normal, we can compute the covariance matrix from the sample and use the formula above for the multivariate normal. For the remaining fitters, we can use an estimate of the covariance matrix that results from the fit (Levenberg-Marquardt, BFGS), or we can compute the Hessian at the minimum (differential evolution, Nelder-Mead simplex). Again, this can be used in the formula above to give an estimate of the entropy.

We can use the difference in entropy between fits for experimental design. After setting up the model system, we can simulate a dataset using the expected statistics from the experiment, then fit the simulated data. This will give us the the expected uncertainty on our individual parameters, and the overall entropy. We can then play with different experimental parameters such as instrument configurations, sample variants and measurement time and select a combination which provides the most information about the parameters of interest. This can be done from the command line using $--simulate$, $--noise$ and $--entropy$.

The information gain from the fit is not quite meaningful. We can calculate the prior entropy by looking at the fitting range of the parameters, and the particular choice of fitting ranges can alter the output of the fit. So for example, if we set the fitting range to eliminate solutions, we will have reduced the prior entropy as well as the posterior entropy, and likely decreased the number of bits of information gain. Conversely, if the fit converges to the same distribution regardless of the parameter range, we can drive the information gain to infinity by setting an unbounded input range.
4.1 bounds - Parameter constraints

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pm</td>
<td>Return the tuple ((-v-dv,-v+dv)), where (-v) is a ‘nice’ number near to the value of expr.</td>
</tr>
<tr>
<td>pmp</td>
<td>Return the tuple ((-v-%v,-v+%v)), where (-v) is a ‘nice’ number near to the value of expr.</td>
</tr>
<tr>
<td>pm_raw</td>
<td>Return the tuple ([v-dv,v+dv]).</td>
</tr>
<tr>
<td>pmp_raw</td>
<td>Return the tuple ([v-%v,v+%v]).</td>
</tr>
<tr>
<td>nice_range</td>
<td>Given a range, return an enclosing range accurate to two digits.</td>
</tr>
<tr>
<td>init_bounds</td>
<td>Returns a bounds object of the appropriate type given the arguments.</td>
</tr>
</tbody>
</table>

Parameter bounds and prior probabilities.

Parameter bounds encompass several features of our optimizers.

First and most trivially they allow for bounded constraints on parameter values.

Secondly, for parameter values known to follow some distribution, the bounds encodes a penalty function as the value strays from its nominal value. Using a negative log likelihood cost function on the fit, then this value naturally contributes to the overall likelihood measure.
Predefined bounds are:

*Unbounded*
  - range (-inf, inf)
*BoundedBelow*
  - range (base, inf)
*BoundedAbove*
  - range (-inf, base)
*Bounded*
  - range (low, high)
*Normal*
  - range (-inf, inf) with gaussian probability
*SoftBounded*
  - range (low, high) with gaussian probability outside

New bounds can be defined following the abstract base class interface defined in `Bounds`, or using `Distribution(rv)` where `rv` is a scipy.stats continuous distribution.

For generating bounds given a value, we provide a few helper functions:

```

v +/- d:   pm(x,dx) or pm(x,-dm,+dp) or pm(x,+dp,-dm)
         return (x-dm,x+dm) limited to 2 significant digits
v +/- p%: pmp(x,p) or pmp(x,-pm,+pp) or pmp(x,+pp,-pm)
         return (x-pm*x/100, x+pp*x/100) limited to 2 sig. digits
pm_raw(x,dx) or raw_pm(x,-dm,+dp) or raw_pm(x,+dp,-dm)
         return (x-dm,x+dm)
pm_raw(x,p) or raw_pmp(x,-pm,+pp) or raw_pmp(x,+pp,-pm)
         return (x-pm*x/100, x+pp*x/100)
nice_range(lo,hi)
         return (lo,hi) limited to 2 significant digits
```

**bumps.bounds.pm** (v, plus=None, minus=None, limits=None)

Return the tuple (~v-dv,~v+dv), where ~expr is a ‘nice’ number near to the value of expr. For example:

```
>>> r = pm(0.78421, 0.0023145)
>>> print("%g - %g"%r)
0.7818 - 0.7866
```

If called as pm(value, +dp, -dm) or pm(value, -dm, +dp), return (~v-dm, ~v+dp).

**bumps.bounds.pmp** (v, plus=None, minus=None, limits=None)

Return the tuple (~v-%v,~v+%v), where ~expr is a ‘nice’ number near to the value of expr. For example:

```
>>> r = pmp(0.78421, 10)
>>> print("%g - %g"%r)
0.7 - 0.87
```

If called as pmp(value, +pp, -pm) or pmp(value, -pm, +pp), return (~v-pm%v, ~v+pp%v).

**bumps.bounds.pm_raw** (v, plus=None)

Return the tuple [v-dv,v+dv].

If called as pm_raw(value, +dp, -dm) or pm_raw(value, -dm, +dp), return (v-dm, v+dp).
bumps.bounds.pmp_raw(v, plus, minus=None)
Return the tuple [v-%v, v+%v]
If called as pmp_raw(value, +pp, -pm) or pmp_raw(value, -pm, +pp), return (v-pm%v, v+pp%v).

bumps.bounds.nice_range(bounds)
Given a range, return an enclosing range accurate to two digits.

bumps.bounds.init_bounds(v)
Returns a bounds object of the appropriate type given the arguments.
This is a helper factory to simplify the user interface to parameter objects.

class bumps.bounds.Bounds
Bases: object

Bounds abstract base class.

A range is used for several purposes. One is that it transforms parameters between unbounded and bounded forms depending on the needs of the optimizer.

Another is that it generates random values in the range for stochastic optimizers, and for initialization.

A third is that it returns the likelihood of seeing that particular value for optimizers which use soft constraints. Assuming the cost function that is being optimized is also a probability, then this is an easy way to incorporate information from other sorts of measurements into the model.

get01(x)
Convert value into [0,1] for optimizers which are bounds constrained.
This can also be used as a scale bar to show approximately how close to the end of the range the value is.

getfull(x)
Convert value into (-inf,inf) for optimizers which are unconstrained.

limits = (-inf, inf)
nllf(value)
Return the negative log likelihood of seeing this value, with likelihood scaled so that the maximum probability is one.
For uniform bounds, this either returns zero or inf. For bounds based on a probability distribution, this returns values between zero and inf. The scaling is necessary so that indefinite and semi-definite ranges return a sensible value. The scaling does not affect the likelihood maximization process, though the resulting likelihood is not easily interpreted.

put01(v)
Convert [0,1] into value for optimizers which are bounds constrained.

putfull(v)
Convert (-inf,inf) into value for optimizers which are unconstrained.

random(n=1, target=1.0)
Return a randomly generated valid value.

    target gives some scale independence to the random number generator, allowing the initial value of the parameter to influence the randomly generated value. Otherwise fits without bounds have too large a space to search through.

residual(value)
Return the parameter ‘residual’ in a way that is consistent with residuals in the normal distribution. The primary purpose is to graphically display exceptional values in a way that is familiar to the user. For fitting, the scaled likelihood should be used.
To do this, we will match the cumulative density function value with that for N(0,1) and find the corresponding percent point function from the N(0,1) distribution. In this way, for example, a value to the right of 2.275% of the distribution would correspond to a residual of -2, or 2 standard deviations below the mean.

For uniform distributions, with all values equally probable, we use a value of +/-4 for values outside the range, and 0 for values inside the range.

```python
start_value()
Return a default starting value if none given.

to_dict()
```

```python
class bumps.bounds.Unbounded
Bases: bumps.bounds.Bounds

Unbounded parameter.

The random initial condition is assumed to be between 0 and 1.

The probability is uniformly 1/inf everywhere, which means the negative log likelihood of P is inf everywhere. A value inf will interfere with optimization routines, and so we instead choose P == 1 everywhere.

```python
get01(x)
Convert value into [0,1] for optimizers which are bounds constrained.

This can also be used as a scale bar to show approximately how close to the end of the range the value is.

getfull(x)
Convert value into (-inf,inf) for optimizers which are unconstrained.

limits = (-inf, inf)
```

```python
nllf(value)
Return the negative log likelihood of seeing this value, with likelihood scaled so that the maximum probability is one.

For uniform bounds, this either returns zero or inf. For bounds based on a probability distribution, this returns values between zero and inf. The scaling is necessary so that indefinite and semi-definite ranges return a sensible value. The scaling does not affect the likelihood maximization process, though the resulting likelihood is not easily interpreted.

```
put01(v)
Convert [0,1] into value for optimizers which are bounds constrained.

putfull(v)
Convert (-inf,inf) into value for optimizers which are unconstrained.

```

random(n=1, target=1.0)
Return a randomly generated valid value.

target gives some scale independence to the random number generator, allowing the initial value of the parameter to influence the randomly generated value. Otherwise fits without bounds have too large a space to search through.

residual(value)
Return the parameter ‘residual’ in a way that is consistent with residuals in the normal distribution. The primary purpose is to graphically display exceptional values in a way that is familiar to the user. For fitting, the scaled likelihood should be used.

To do this, we will match the cumulative density function value with that for N(0,1) and find the corresponding percent point function from the N(0,1) distribution. In this way, for example, a value to the right
of 2.275% of the distribution would correspond to a residual of -2, or 2 standard deviations below the mean.

For uniform distributions, with all values equally probable, we use a value of +/-4 for values outside the range, and 0 for values inside the range.

**start_value()**
Return a default starting value if none given.

**to_dict()**

```python
class bumps.bounds.Bounded(lo, hi)
Bases: bumps.bounds.Bounds
```

Bounded range.

[lo,hi] <-> [0,1] scale is simple linear [lo,hi] <-> (-inf,inf) scale uses exponential expansion

While technically the probability of seeing any value within the range is 1/range, for consistency with the semi-infinite ranges and for a more natural mapping between nllf and chisq, we instead set the probability to 0. This choice will not affect the fits.

**get01(x)**
Convert value into [0,1] for optimizers which are bounds constrained.

This can also be used as a scale bar to show approximately how close to the end of the range the value is.

**getfull(x)**
Convert value into (-inf,inf) for optimizers which are unconstrained.

```python
limits = (-inf, inf)
```

**nllf(value)**
Return the negative log likelihood of seeing this value, with likelihood scaled so that the maximum probability is one.

For uniform bounds, this either returns zero or inf. For bounds based on a probability distribution, this returns values between zero and inf. The scaling is necessary so that indefinite and semi-definite ranges return a sensible value. The scaling does not affect the likelihood maximization process, though the resulting likelihood is not easily interpreted.

**put01(v)**
Convert [0,1] into value for optimizers which are bounds constrained.

**putfull(v)**
Convert (-inf,inf) into value for optimizers which are unconstrained.

**random(n=1, target=1.0)**
Return a randomly generated valid value.

```
target gives some scale independence to the random number generator, allowing the initial value of the parameter to influence the randomly generated value. Otherwise fits without bounds have too large a space to search through.
```

**residual(value)**
Return the parameter ‘residual’ in a way that is consistent with residuals in the normal distribution. The primary purpose is to graphically display exceptional values in a way that is familiar to the user. For fitting, the scaled likelihood should be used.

To do this, we will match the cumulative density function value with that for N(0,1) and find the corresponding percent point function from the N(0,1) distribution. In this way, for example, a value to the right of 2.275% of the distribution would correspond to a residual of -2, or 2 standard deviations below the mean.

---

**4.1. bounds - Parameter constraints**
For uniform distributions, with all values equally probable, we use a value of +/-4 for values outside the range, and 0 for values inside the range.

\textbf{start\_value()}

Return a default starting value if none given.

\textbf{to\_dict()}

class \texttt{bumps.bounds.BoundedAbove}(base)

\begin{itemize}
  \item \texttt{base} \texttt{Bounds}
\end{itemize}

Semidefinite range bounded above.

[-inf,base] <-> [0,1] uses logarithmic compression [-inf,base] <-> (-inf,inf) is direct below base-1, 1/(base-x)

Logarithmic compression works by converting sign*m*2^e+base to sign*(e+1023+m), yielding a value in

\texttt{[0,2048]} This can then be converted to a value in [0,1].

Note that the likelihood function is problematic: the true probability of seeing any particular value in the range
is infinitesimal, and that is indistinguishable from values outside the range. Instead we say that P = 1 in range,
and 0 outside.

\textbf{get01}(x)

Convert value into [0,1] for optimizers which are bounds constrained.

This can also be used as a scale bar to show approximately how close to the end of the range the value is.

\textbf{getfull}(x)

Convert value into (-inf,inf) for optimizers which are unconstrained.

\textbf{limits} = (-inf, inf)

\textbf{nllf(value)}

Return the negative log likelihood of seeing this value, with likelihood scaled so that the maximum probability
is one.

For uniform bounds, this either returns zero or inf. For bounds based on a probability distribution, this
returns values between zero and inf. The scaling is necessary so that indefinite and semi-definite ranges return
a sensible value. The scaling does not affect the likelihood maximization process, though the resulting likelihood
is not easily interpreted.

\textbf{put01}(v)

Convert [0,1] into value for optimizers which are bounds constrained.

\textbf{putfull}(v)

Convert (-inf,inf) into value for optimizers which are unconstrained.

\textbf{random}(n=1, target=1.0)

Return a randomly generated valid value.

\textit{target} gives some scale independence to the random number generator, allowing the initial value of the
parameter to influence the randomly generated value. Otherwise fits without bounds have too large a space
to search through.

\textbf{residual(value)}

Return the parameter ‘residual’ in a way that is consistent with residuals in the normal distribution. The primary purpose is to graphically display exceptional values in a way that is familiar to the user. For fitting, the scaled likelihood should be used.

To do this, we will match the cumulative density function value with that for N(0,1) and find the corresponding percent point function from the N(0,1) distribution. In this way, for example, a value to the right
of 2.275% of the distribution would correspond to a residual of -2, or 2 standard deviations below the mean.

For uniform distributions, with all values equally probable, we use a value of +/-4 for values outside the range, and 0 for values inside the range.

**start_value()**
Return a default starting value if none given.

**to_dict()**

```python
class bumps.bounds.BoundedBelow(base)
    Bases: bumps.bounds.Bounds

Semidefinite range bounded below.

The random initial condition is assumed to be within 1 of the maximum.

[base,inf] <-> (-inf,inf) is direct above base+1, -1/(x-base) below [base,inf] <-> [0,1] uses logarithmic compression.

Logarithmic compression works by converting sign*m*2^e+base to sign*(e+1023+m), yielding a value in [0,2048]. This can then be converted to a value in [0,1].

Note that the likelihood function is problematic: the true probability of seeing any particular value in the range is infinitesimal, and that is indistinguishable from values outside the range. Instead we say that P = 1 in range, and 0 outside.

**get01(x)**
Convert value into [0,1] for optimizers which are bounds constrained.

This can also be used as a scale bar to show approximately how close to the end of the range the value is.

**getfull(x)**
Convert value into (-inf,inf) for optimizers which are unconstrained.

**limits = (-inf, inf)**

**nllf(value)**
Return the negative log likelihood of seeing this value, with likelihood scaled so that the maximum probability is one.

For uniform bounds, this either returns zero or inf. For bounds based on a probability distribution, this returns values between zero and inf. The scaling is necessary so that indefinite and semi-definite ranges return a sensible value. The scaling does not affect the likelihood maximization process, though the resulting likelihood is not easily interpreted.

**put01(v)**
Convert [0,1] into value for optimizers which are bounds constrained.

**putfull(v)**
Convert (-inf,inf) into value for optimizers which are unconstrained.

**random(n=1, target=1.0)**
Return a randomly generated valid value.

*target* gives some scale independence to the random number generator, allowing the initial value of the parameter to influence the randomly generated value. Otherwise fits without bounds have too large a space to search through.

**residual(value)**
Return the parameter ‘residual’ in a way that is consistent with residuals in the normal distribution. The primary purpose is to graphically display exceptional values in a way that is familiar to the user. For fitting, the scaled likelihood should be used.
To do this, we will match the cumulative density function value with that for N(0,1) and find the corresponding percent point function from the N(0,1) distribution. In this way, for example, a value to the right of 2.275% of the distribution would correspond to a residual of -2, or 2 standard deviations below the mean.

For uniform distributions, with all values equally probable, we use a value of +/-4 for values outside the range, and 0 for values inside the range.

```python
start_value()
    Return a default starting value if none given.

to_dict()

class bumps.bounds.Distribution(dist)
    Bases: bumps.bounds.Bounds

Parameter is pulled from a distribution.

dist must implement the distribution interface from scipy.stats. In particular, it should define methods rvs, nllf, cdf and ppf and attributes args and dist.name.

get01(x)
    Convert value into [0,1] for optimizers which are bounds constrained.

    This can also be used as a scale bar to show approximately how close to the end of the range the value is.

getfull(x)
    Convert value into (-inf,inf) for optimizers which are unconstrained.

limits = (-inf, inf)

nllf(value)
    Return the negative log likelihood of seeing this value, with likelihood scaled so that the maximum probability is one.

    For uniform bounds, this either returns zero or inf. For bounds based on a probability distribution, this returns values between zero and inf. The scaling is necessary so that indefinite and semi-definite ranges return a sensible value. The scaling does not affect the likelihood maximization process, though the resulting likelihood is not easily interpreted.

put01(v)
    Convert [0,1] into value for optimizers which are bounds constrained.

putfull(v)
    Convert (-inf,inf) into value for optimizers which are unconstrained.

random(n=1, target=1.0)
    Return a randomly generated valid value.

    target gives some scale independence to the random number generator, allowing the initial value of the parameter to influence the randomly generated value. Otherwise fits without bounds have too large a space to search through.

residual(value)
    Return the parameter ‘residual’ in a way that is consistent with residuals in the normal distribution. The primary purpose is to graphically display exceptional values in a way that is familiar to the user. For fitting, the scaled likelihood should be used.

    To do this, we will match the cumulative density function value with that for N(0,1) and find the corresponding percent point function from the N(0,1) distribution. In this way, for example, a value to the right of 2.275% of the distribution would correspond to a residual of -2, or 2 standard deviations below the mean.
For uniform distributions, with all values equally probable, we use a value of +/-4 for values outside the range, and 0 for values inside the range.

**start_value()**
Return a default starting value if none given.

**to_dict()**

```python
class bumps.bounds.Normal(mean=0, std=1)

Bases: bumps.bounds.Distribution
```

Parameter is pulled from a normal distribution.

If you have measured a parameter value with some uncertainty (e.g., the film thickness is 35+/-5 according to TEM), then you can use this measurement to restrict the values given to the search, and to penalize choices of this fitting parameter which are different from this value.

*mean* is the expected value of the parameter and *std* is the 1-sigma standard deviation.

**get01(x)**
Convert value into [0,1] for optimizers which are bounds constrained.

This can also be used as a scale bar to show approximately how close to the end of the range the value is.

**getfull(x)**
Convert value into (-inf,inf) for optimizers which are unconstrained.

**limits = (-inf, inf)**

**nllf(value)**
Return the negative log likelihood of seeing this value, with likelihood scaled so that the maximum probability is one.

For uniform bounds, this either returns zero or inf. For bounds based on a probability distribution, this returns values between zero and inf. The scaling is necessary so that indefinite and semi-definite ranges return a sensible value. The scaling does not affect the likelihood maximization process, though the resulting likelihood is not easily interpreted.

**put01(v)**
Convert [0,1] into value for optimizers which are bounds constrained.

**putfull(v)**
Convert (-inf,inf) into value for optimizers which are unconstrained.

**random(n=1, target=1.0)**
Return a randomly generated valid value.

*target* gives some scale independence to the random number generator, allowing the initial value of the parameter to influence the randomly generated value. Otherwise fits without bounds have too large a space to search through.

**residual(value)**
Return the parameter ‘residual’ in a way that is consistent with residuals in the normal distribution. The primary purpose is to graphically display exceptional values in a way that is familiar to the user. For fitting, the scaled likelihood should be used.

To do this, we will match the cumulative density function value with that for N(0,1) and find the corresponding percent point function from the N(0,1) distribution. In this way, for example, a value to the right of 2.275% of the distribution would correspond to a residual of -2, or 2 standard deviations below the mean.

For uniform distributions, with all values equally probable, we use a value of +/-4 for values outside the range, and 0 for values inside the range.

---

4.1. **bounds - Parameter constraints**
**Bumps: Curve Fitting and Uncertainty Analysis, Release 0.9.0**

```python
start_value()
    Return a default starting value if none given.

to_dict()

class bumps.bounds.BoundedNormal (mean=0, std=1, limits=(-inf, inf))
    Bases: bumps.bounds.Bounds

    truncated normal bounds

g01(x)
    Convert value into [0,1] for optimizers which are bounds constrained.
    This can also be used as a scale bar to show approximately how close to the end of the range the value is.

gfull(x)
    Convert value into (-inf,inf) for optimizers which are unconstrained.

limits = (-inf, inf)
nllf(value)
    Return the negative log likelihood of seeing this value, with likelihood scaled so that the maximum probability is one.

put01(v)
    Convert [0,1] into value for optimizers which are bounds constrained.

putfull(v)
    Convert (-inf,inf) into value for optimizers which are unconstrained.

random(n=1, target=1.0)
    Return a randomly generated valid value, or an array of values

residual(value)
    Return the parameter 'residual' in a way that is consistent with residuals in the normal distribution. The primary purpose is to graphically display exceptional values in a way that is familiar to the user. For fitting, the scaled likelihood should be used.

    For the truncated normal distribution, we can just use the normal residuals.

start_value()
    Return a default starting value if none given.

to_dict()

class bumps.bounds.SoftBounded (lo, hi, std=None)
    Bases: bumps.bounds.Bounds

    Parameter is pulled from a stretched normal distribution.

    This is like a rectangular distribution, but with gaussian tails.

    The intent of this distribution is for soft constraints on the values. As such, the random generator will return values like the rectangular distribution, but the likelihood will return finite values based on the distance from the from the bounds rather than returning infinity.

    Note that for bounds constrained optimizers which force the value into the range [0,1] for each parameter we don’t need to use soft constraints, and this acts just like the rectangular distribution.

g01(x)
    Convert value into [0,1] for optimizers which are bounds constrained.

    This can also be used as a scale bar to show approximately how close to the end of the range the value is.
```
getfull(x)
   Convert value into (-inf,inf) for optimizers which are unconstrained.

limits = (-inf, inf)

nllf(value)
   Return the negative log likelihood of seeing this value, with likelihood scaled so that the maximum probability is one.

   For uniform bounds, this either returns zero or inf. For bounds based on a probability distribution, this returns values between zero and inf. The scaling is necessary so that indefinite and semi-definite ranges return a sensible value. The scaling does not affect the likelihood maximization process, though the resulting likelihood is not easily interpreted.

put01(v)
   Convert [0,1] into value for optimizers which are bounds constrained.

putfull(v)
   Convert (-inf,inf) into value for optimizers which are unconstrained.

random(n=1, target=1.0)
   Return a randomly generated valid value.

   target gives some scale independence to the random number generator, allowing the initial value of the parameter to influence the randomly generated value. Otherwise fits without bounds have too large a space to search through.

residual(value)
   Return the parameter ‘residual’ in a way that is consistent with residuals in the normal distribution. The primary purpose is to graphically display exceptional values in a way that is familiar to the user. For fitting, the scaled likelihood should be used.

   To do this, we will match the cumulative density function value with that for N(0,1) and find the corresponding percent point function from the N(0,1) distribution. In this way, for example, a value to the right of 2.275% of the distribution would correspond to a residual of -2, or 2 standard deviations below the mean.

   For uniform distributions, with all values equally probable, we use a value of +/-4 for values outside the range, and 0 for values inside the range.

start_value()
   Return a default starting value if none given.

to_dict()

4.2 bspline - B-Spline interpolation library

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bspline</td>
<td>Evaluate the B-spline with control points y at positions xt in [0,1].</td>
</tr>
<tr>
<td>pbs</td>
<td>Evaluate the parametric B-spline px(t),py(t).</td>
</tr>
</tbody>
</table>

BSpline calculator.

Given a set of knots, compute the cubic B-spline interpolation.

bumps.bspline.bspline(y, xt, clamp=True)

   Evaluate the B-spline with control points y at positions xt in [0,1].

   The spline goes through the control points at the ends. If clamp is True, the derivative of the spline at both ends
is zero. If clamp is False, the derivative at the ends is equal to the slope connecting the final pair of control points.

B-spline knots are chosen to be equally spaced within [0,1].

Bumps.bspline.pbs(x, y, t, clamp=True, parametric=True)
Evaluate the parametric B-spline px(t),py(t).

x and y are the control points, and t are the points in [0,1] at which they are evaluated. The x values are sorted so that the spline describes a function.

The spline goes through the control points at the ends. If clamp is True, the derivative of the spline at both ends is zero. If clamp is False, the derivative at the ends is equal to the slope connecting the final pair of control points.

If parametric is False, then parametric points t’ are chosen such that x(t’) = t.

The B-spline knots are chosen to be equally spaced within [0,1].

4.3 cheby - Freeform - Chebyshev

<table>
<thead>
<tr>
<th>profile</th>
<th>Evaluate the chebyshev approximation c at points x.</th>
</tr>
</thead>
<tbody>
<tr>
<td>cheby_approx</td>
<td>Return the coefficients for the order n chebyshev approximation to function f evaluated over the range [low,high].</td>
</tr>
<tr>
<td>cheby_val</td>
<td>Evaluate the chebyshev approximation c at points x.</td>
</tr>
<tr>
<td>cheby_points</td>
<td>Return the points in at which a function must be evaluated to generate the order n Chebyshev approximation function.</td>
</tr>
<tr>
<td>cheby_coeff</td>
<td>Compute chebyshev coefficients for a polynomial of order n given the function evaluated at the chebyshev points for order n.</td>
</tr>
</tbody>
</table>

Freeform modeling with Chebyshev polynomials.

Chebyshev polynomials \(T_k\) form a basis set for functions over \([-1, 1]\). The truncated interpolating polynomial \(P_n\) is a weighted sum of Chebyshev polynomials up to degree \(n\):

\[
f(x) \approx P_n(x) = \sum_{k=0}^{n} c_k T_k(x)
\]

The interpolating polynomial exactly matches \(f(x)\) at the chebyshev nodes \(z_k\) and is near the optimal polynomial approximation to \(f\) of degree \(n\) under the maximum norm. For well behaved functions, the coefficients \(c_k\) decrease rapidly, and furthermore are independent of the degree \(n\) of the polynomial.

The models can either be defined directly in terms of the Chebyshev coefficients \(c_k\) with method = ‘direct’, or in terms of control points \((z_k, f(z_k))\) at the Chebyshev nodes cheby_points() with method = ‘interp’. Bounds on the parameters are easier to control using ‘interp’, but the function may oscillate wildly outside the bounds. Bounds on the oscillation are easier to control using ‘direct’, but the shape of the profile is difficult to control.

Bumps.cheby.profile(c, t, method)
Evaluate the chebyshev approximation c at points x.

If method is ‘direct’ then \(c_i\) are the coefficients for the chebyshev polynomials \(T_i\) yielding \(P = \sum_i c_i T_i(x)\).

If method is ‘interp’ then \(c_i\) are the values of the interpolated function \(f\) evaluated at the chebyshev points returned by cheby_points().
bumps.cheby.\texttt{cheby\_approx}(n, f, range=(0, 1))

Return the coefficients for the order \( n \) Chebyshev approximation to function \( f \) evaluated over the range \([\text{low}, \text{high}]\).

bumps.cheby.\texttt{cheby\_val}(c, x)

Evaluate the Chebyshev approximation \( c \) at points \( x \).

The values \( c_i \) are the coefficients for the Chebyshev polynomials \( T_i \) yielding \( p(x) = \sum_i c_i T_i(x) \).

bumps.cheby.\texttt{cheby\_points}(n, range=(0, 1))

Return the points in which a function must be evaluated to generate the order \( n \) Chebyshev approximation function.

Over the range \([-1,1]\), the points are \( p_k = \cos(\pi(2k + 1)/(2n)) \). Adjusting the range to \([x_L, x_R]\), the points become \( x_k = \frac{1}{2}(p_k - x_L + 1)/(x_R - x_L) \).

bumps.cheby.\texttt{cheby\_coeff}(f)

Compute Chebyshev coefficients for a polynomial of order \( n \) given the function evaluated at the Chebyshev points for order \( n \).

This can be used as the basis of a direct interpolation method where the \( n \) control points are positioned at \( \text{cheby\_points}(n) \).

### 4.4 cli - Command line interface

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{main}</td>
<td>Run the bumps program with the command line interface.</td>
</tr>
<tr>
<td>\texttt{install_plugin}</td>
<td>Replace symbols in \texttt{bumps.plugin} with application</td>
</tr>
<tr>
<td></td>
<td>specific methods.</td>
</tr>
<tr>
<td>\texttt{set_mplconfig}</td>
<td>Point the matplotlib config dir to %LOCALAPPDATA%{appdatadir}mplconfig.</td>
</tr>
<tr>
<td>\texttt{config_matplotlib}</td>
<td>Setup matplotlib to use a particular backend.</td>
</tr>
<tr>
<td>\texttt{load_model}</td>
<td>Load a model file.</td>
</tr>
<tr>
<td>\texttt{preview}</td>
<td>Show the problem plots and parameters.</td>
</tr>
<tr>
<td>\texttt{load_best}</td>
<td>Reload individual parameter values from a saved .par file.</td>
</tr>
<tr>
<td>\texttt{save_best}</td>
<td>Save the fit data, including parameter values, uncertainties and plots.</td>
</tr>
<tr>
<td>\texttt{resynth}</td>
<td>Generate maximum likelihood fits to resynthesized data sets.</td>
</tr>
</tbody>
</table>

Bumps command line interface.

The functions in this module are used by the bumps command to implement the command line interface. Bumps plugin models can use them to create stand alone applications with a similar interface. For example, the Refl1D application uses the following:

```python
from . import fitplugin
import bumps.cli
bumps.cli.set_mplconfig(appdatadir='Refl1D')
bumps.cli.install_plugin(fitplugin)
bumps.cli.main()
```

After completing a set of fits on related systems, a post-analysis script can use \texttt{load\_model()} to load the problem definition and \texttt{load\_best()} to load the best value found in the fit. This can be used for example in experiment design, where you look at the expected parameter uncertainty when fitting simulated data from a range of experimental
Bumps: Curve Fitting and Uncertainty Analysis, Release 0.9.0

 systems.

bumps.cli.main()
    Run the bumps program with the command line interface.
    Input parameters are taken from sys.argv.

bumps.cli.install_plugin(p)
    Replace symbols in bumps.plugin with application specific methods.

bumps.cli.set_mplconfig(appdatadir)
    Point the matplotlib config dir to %LOCALAPPDATA%{appdatadir}mplconfig.

bumps.cli.config_matplotlib(backend=None)
    Setup matplotlib to use a particular backend.
    The backend should be ‘WXAgg’ for interactive use, or ‘Agg’ for batch. This distinction allows us to run in
    environments such as cluster computers which do not have wx installed on the compute nodes.
    This function must be called before any imports to pylab. To allow this, modules should not import pylab at the
    module level, but instead import it for each function/method that uses it. Exceptions can be made for modules
    which are completely dedicated to plotting, but these modules should never be imported at the module level.

bumps.cli.load_model(path, model_options=None)
    Load a model file.
    path contains the path to the model file.
    model_options are any additional arguments to the model. The sys.argv variable will be set such that sys.argv[1:]
    == model_options.

bumps.cli.preview(problem, view=None)
    Show the problem plots and parameters.

bumps.cli.load_best(problem, path)
    Reload individual parameter values from a saved .par file.
    If the label does not exist in the file, use the value from the model as the default value. Ignore labels that do not
    exist in the model. In that way we can load parameters from an old fit with minimal fuss, even as we add, delete
    and move parameters in the model. If any parameters are missing, set problem.undefined to the a boolean index
    of the undefined parameters.
    There is an interaction with –init=eps and the par file. If any parameters are missing from the par file they will
    be randomized across the entire parameter range using the equivalent of –init=lhs. That means you can drop a
    # at the beginning of the line in the .par file and that parameter will be shuffled on restart, with the remaining
    parameters starting near the initial value.

bumps.cli.save_best(fitdriver, problem, best, view=None)
    Save the fit data, including parameter values, uncertainties and plots.
    fitdriver is the fitter that was used to drive the fit.
    problem is a FitProblem instance.
    best is the parameter set to save.

bumps.cli.re synth(fitdriver, problem, mapper, opts)
    Generate maximum likelihood fits to resynthesized data sets.
    fitdriver is a bumps.fitters.FitDriver object with a fitter already chosen.
    problem is a bumps.fitproblem.FitProblem() object. It should be initialized with optimal values for
    the parameters.
    mapper is one of the available bumps.mapper classes.
opts is a `bumps.options.BumpsOpts` object representing the command line parameters.

## 4.5 curve - Model a fit function

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Curve</td>
<td>Model a measurement with a user defined function.</td>
</tr>
<tr>
<td>PoissonCurve</td>
<td>Model a measurement with Poisson uncertainty.</td>
</tr>
<tr>
<td>plot_err</td>
<td>DEPRECATED: subclass Curve and override the plot function.</td>
</tr>
</tbody>
</table>

Build a bumps model from a function and data.

### 4.5.1 Example

Given a function `sin_model` which computes a sine wave at times \( t \):

```python
from numpy import sin
def sin_model(t, freq, phase):
    return sin(2*pi*(freq*t + phase))
```

and given data \((y, dy)\) measured at times \( t \), we can define the fit problem as follows:

```python
from bumps.names import *
M = Curve(sin_model, t, y, dy, freq=20)
```

The `freq` and `phase` keywords are optional initial values for the model parameters which otherwise default to zero. The model parameters can be accessed as attributes on the model to set fit range:

```python
M.freq.range(2, 100)
M.phase.range(0, 1)
```

As usual, you can initialize or assign parameter expressions to the the parameters if you want to tie parameters together within or between models.

Note: there is sometimes difficulty getting bumps to recognize the function during fits, which can be addressed by putting the definition in a separate file on the python path. With the windows binary distribution of bumps, this can be done in the problem definition file with the following code:

```python
import os
from bumps.names import *
sys.path.insert(0, os.getcwd())
```

The model function can then be imported from the external module as usual:

```python
from sin_model import sin_model
```

```python
class bumps.curve.Curve(fn, x, y=None, name=",", labels=None, plot=None, plot_x=None, **kwargs)
    Bases: object
    
    Model a measurement with a user defined function.
    
    The function \( fn(x; p1, p2, \ldots) \) should return the expected value \( y \) for each point \( x \) given the parameters \( p1, p2, \) etc. \( dy \) is the uncertainty for each measured value \( y \). If not specified, it defaults to 1. Multi-valued functions, which return multiple \( y \) values for each \( x \) value, should have \( x \) as a vector of length \( n \) and \( y, dy \) as arrays of size \( [n, k] \).```
Initial values for the parameters can be set as $p=value$ arguments to Curve. If no value is set, then the initial value will be taken from the default value given in the definition of $fn$, or set to 0 if the parameter is not defined with an initial value. Arbitrary non-fittable data can be passed to the function as parameters, but only if the parameter is given a default value of None in the function definition, and has the initial value set as an argument to Curve. Defining $state=dict(key=value, ...)$ before Curve, and calling Curve as Curve(..., **state) works pretty well.

Curve takes the following special keyword arguments:

- **name** is added to each parameter name when the parameter is defined. The filename for the data is a good choice, since this allows you to keep the parameters straight when fitting multiple datasets simultaneously.

- **plot** is an alternative plotting function. The function should be defined as $plot(x,y,dy,fy,**kw)$. The keyword arguments will be filled with the values of the parameters used to compute $fy$. It will be easiest to list the parameters you need to make your plot as positional arguments after $x,y,dy,fy$ in the plot function declaration. For example, $plot(x,y,dy, fy, p^3, **kw)$ will make the value of parameter $p^3$ available as a variable in your function. The special keyword **view** will be a string containing linear, log, logx, or loglog. If only showing the residuals, the string will be residual.

- **plot_x** is an array giving the sample points to use when plotting the theory function, if different from the $x$ values at which the function is sampled. Use this to draw a smooth curve between the fitted points. This value is ignored if you provide your own plot function.

- **labels** are the axis labels for the plot. This should include units in parentheses. If the function is multi-valued then use ['x axis', 'y axis', 'line 1', 'line 2', ...].

The data uncertainty is assumed to follow a gaussian distribution. If measurements draw from some other uncertainty distribution, then subclass Curve and replace nllf with the correct probability given the residuals. See the implementation of PoissonCurve for an example.

```python
nllf()
numpoints()
parameters()
plot(view=None)
residuals()
save(basename)
simulate_data(noise=None)
theory(x=None)
update()
```

**class** bumps.curve.PoissonCurve($fn, x, y, name=", **fnkw)$

Bases: bumps.curve.Curve

Model a measurement with Poisson uncertainty.

The nllf is calculated using Poisson probabilities, but the curve itself is displayed using the approximation that $\sigma_y \approx \sqrt{(y)}$.

See Curve for details.

```python
nllf()
numpoints()
parameters()
plot(view=None)
```
```python
residuals()
save(basename)
simulate_data(noise=None)
theory(x=None)
update()
```
4.7 errplot - Plot sample profile uncertainty

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>reload_errors</code></td>
<td>Reload the MCMC state and compute the model confidence intervals.</td>
</tr>
<tr>
<td><code>calc_errors_from_state</code></td>
<td>Compute confidence regions for a problem from the Align the sample profiles and compute the residual difference from the measured data for a set of points returned from DREAM.</td>
</tr>
<tr>
<td><code>calc_errors</code></td>
<td>Align the sample profiles and compute the residual difference from the measured data for a set of points.</td>
</tr>
<tr>
<td><code>show_errors</code></td>
<td>Display the confidence regions returned by <code>calc_errors()</code>.</td>
</tr>
</tbody>
</table>

Estimate model uncertainty from random sample.

MCMC uncertainty analysis gives the uncertainty on the model parameters rather than the model itself. For example, when fitting a line to a set of data, the uncertainty on the slope and the intercept does not directly give you the uncertainty in the expected value of \( y \) for a given value of \( x \).

The routines in bumps.errplot allow you to generate confidence intervals on the model using a random sample of MCMC parameters. After calculating the model \( y \) values for each sample, one can generate 68% and 95% contours for a set of sampling points \( x \). This can apply even to models which are not directly measured. For example, in scattering inverse problems the scattered intensity is the value measured, but the fitting parameters describe the real space model that is being probed. It is the uncertainty in the real space model that is of primary interest.

Since bumps knows only the probability of seeing the measured value given the input parameters, it is up to the model itself to calculate and display the confidence intervals on the model and the expected values for the data points. This is done using the bumps.plugin architecture, so application writers can provide the appropriate functions for their data types. Eventually this capability will move to the model definition so that different types of models can be processed in the same fit.

For a completed MCMC run, four steps are required:

1. reload the fitting problem and the MCMC state
2. select a set of sample points
3. evaluate model confidence intervals from sample points
4. show model confidence intervals

`reload_errors()` performs steps 1, 2 and 3, returning `errs`. If the fitting problem and the MCMC state are already loaded, then use `calc_errors_from_state()` to perform steps 2 and 3, returning `errs`. If alternative sampling is desired, then use `calc_errors()` on a given set of points to perform step 3, returning `errs`. Once `errs` has been calculated and returned by one of these methods, call `show_errors()` to perform step 4.

`bumps.errplot.reload_errors(model, store, nshown=50, random=True)`

Reload the MCMC state and compute the model confidence intervals.

The loaded error data is a sample from the fit space according to the fit parameter uncertainty. This is a subset of the samples returned by the DREAM MCMC sampling process.

`model` is the name of the model python file

`store` is the name of the store directory containing the dream results

`nshown` and `random` are as for `calc_errors_from_state()`.

Returns `errs` for `show_errors()`. 

96 Chapter 4. Reference: bumps
bumps.errplot.calc_errors_from_state(problem, state, nshown=50, random=True, portion=1.0)

Compute confidence regions for a problem from the Align the sample profiles and compute the residual difference from the measured data for a set of points returned from DREAM.

*nshown* is the number of samples to include from the state.

*random* is True if the samples are randomly selected, or False if the most recent samples should be used. Use random if you have poor mixing (i.e., the parameters tend to stay fixed from generation to generation), but not random if your burn-in was too short, and you want to select from the end.

Returns *errs* for *show_errors()*.

bumps.errplot.calc_errors(problem, points)

Align the sample profiles and compute the residual difference from the measured data for a set of points.

The return value is arbitrary. It is passed to the *show_errors()* plugin for the application. Returns *errs* for *show_errors()*.

bumps.errplot.show_errors(errs)

Display the confidence regions returned by *calc_errors()*.

The content of *errs* depends on the active plugin.

### 4.8 fitproblem - Interface between models and fitters

**Fitness**

Manage parameters, data, and theory function evaluation.

**FitProblem**

Return a fit problem instance for the fitness function(s).

**load_problem**

Load a problem definition from a python script file.

**BaseFitProblem**

See *FitProblem()*.

**MultiFitProblem**

Weighted fits for multiple models.

Interface between the models and the fitters.

*Fitness* defines the interface that model evaluators can follow. These models can be bundled together into a *FitProblem()* and sent to *bumps.fitters.FitDriver* for optimization and uncertainty analysis.

Summary of problem attributes:

```python
# Used by fitters
nllf(p: Optional[Vector]) -> float  # main calculation
bounds() -> Tuple(Vector, Vector)  # or equivalent sequence
setp(p: Vector) -> None
getp() -> Vector
residuals() -> Vector  # for LM, MPFit
parameter_residuals() -> Vector  # for LM, MPFit
constraints_nllf() -> float  # for LM, MPFit; constraint cost is spread across the
  -> individual residuals
randomize() -> None  # for multistart
resynth_data() -> None  # for Monte Carlo resampling of maximum likelihood
restore_data() -> None  # for Monte Carlo resampling of maximum likelihood
name: str  # DREAM uses this
chisq() -> float
chisq_str() -> str
labels() -> List[str]
summarize() -> str
```

(continues on next page)
Bumps: Curve Fitting and Uncertainty Analysis, Release 0.9.0

show() -> None
load(input_path: str) -> None
save(output_path: str) -> None
plot(figfile: str, view: str) -> None

# Set/used by bumps.cli
model_reset() -> None # called by load_model
path: str # set by load_model
title: str = filename # set by load_model
name: str # set by load_model
options: List[str] # from sys.argv[1:]
undefined: List[int] # when loading a save .par file, these parameters weren’t defined
store: str # set by make_store
output_path: str # set by make_store
simulate_data(noise: float) -> None # for --simulate in opts
cov() -> Matrix # for --cov in opts

class bumps.fitproblem.Fitness

    Bases: object

    Manage parameters, data, and theory function evaluation.

    See Complex models for a detailed explanation.

    nllf()
    Return the negative log likelihood value of the current parameter set.

    numpoints()
    Return the number of data points.

    parameters()
    return the parameters in the model.
    model parameters are a hierarchical structure of lists and dictionaries.

    plot(view='linear')
    Plot the model to the current figure. You only get one figure, but you can make it as complex as you want.
    This will be saved as a png on the server, and composed onto a results web page.

    residuals()
    Return residuals for current theory minus data.
    Used for Levenburg-Marquardt, and for plotting.

    restore_data()
    Restore the original data in the model (after resynth).

    resynth_data()
    Generate fake data based on uncertainties in the real data. For Monte Carlo resynth-refit uncertainty
    analysis. Bootstrapping?

    save(basename)
    Save the model to a file based on basename+extension. This will point to a path to a directory on a remote
    machine; don’t make any assumptions about information stored on the server. Return the set of files saved
    so that the monitor software can make a pretty web page.

    to_dict()

    update()
    Called when parameters have been updated. Any cached values will need to be cleared and the model
    reevaluated.
bumps.fitproblem.FitProblem(*args, **kw)
    Return a fit problem instance for the fitness function(s).

For an individual model:

    fitness is a Fitness instance.

For a set of models:

    models is a sequence of Fitness instances.

    weights is an optional scale factor for each model. A weighted fit returns nllf \( L = \sum w_k^2 L_k \). If an individual nllf is the sum squared residuals then this is equivalent to scaling the measurement uncertainty by \( 1/w \). Unless the measurement uncertainty is unknown, weights should be in \([0, 1]\), representing an unknown systematic uncertainty spread across the individual measurements.

    freevars is parameter.FreeVariables instance defining the per-model parameter assignments. See Free Variables for details.

Additional parameters:

    name name of the problem

    constraints is a function which returns the negative log likelihood of seeing the parameters independent from the fitness function. Use this for example to check for feasible regions of the search space, or to add constraints that cannot be easily calculated per parameter. Ideally, the constraints nllf will increase as you go farther from the feasible region so that the fit will be directed toward feasible values.

    soft_limit is the constraints function cutoff, beyond which the penalty_nllf will be used and fitness nllf will not be calculated.

    penalty_nllf is the nllf to use for fitness when constraints is greater than soft_limit.

Total nllf is the sum of the parameter nllf, the constraints nllf and the depending on whether constraints is greater than soft_limit, either the fitness nllf or the penalty nllf.

New in 0.9.0: weights are now squared when computing the sum rather than linear.

bumps.fitproblem.load_problem(filename, options=None)
    Load a problem definition from a python script file.

    sys.argv is set to [file] + options within the context of the script.

    The user must define problem=FitProblem(...) within the script.

    Raises ValueError if the script does not define problem.

class bumps.fitproblem.BaseFitProblem(fitness, name=None, constraints=None, penalty_nllf=inf, soft_limit=inf, partial=False)

    Bases: object

    See FitProblem()

    bounds()
        Return the bounds fore each parameter a 2 x N array

    chisq()
        Return sum squared residuals normalized by the degrees of freedom.

        In the context of a composite fit, the reduced chisq on the individual models only considers the points and the fitted parameters within the individual model.

        Note that this does not include cost factors due to constraints on the parameters, such as sample_offset ~ N(0,0.01).
chisq_str()
Return a string representing the chisq equivalent of the nllf.

If the model has strictly gaussian independent uncertainties then the negative log likelihood function will return 0.5*sum(residuals**2), which is 1/2*chisq. Since we are printing normalized chisq, we multiply the model nllf by 2/DOF before displaying the value. This is different from the problem nllf function, which includes the cost of the prior parameters and the cost of the penalty constraints in the total nllf. The constraint value is displayed separately.

constraints_nllf()
Returns the cost of all constraints.

cov()
Return the covariance matrix as computed by numdifftools from the Hessian matrix for the problem at the current parameter values.

getp()
Returns the current value of the parameter vector.

has_residuals
True if the underlying fitness function defines residuals.

labels()
Return the list of labels, one per fitted parameter.

model_nllf()
Negative log likelihood of seeing data given model.

model_parameters()
Parameters associated with the model.

model_points()
Number of data points associated with the model.

model_reset()
Prepare for the fit.

This sets the parameters and the bounds properties that the solver is expecting from the fittable object. We also compute the degrees of freedom so that we can return a normalized fit likelihood.

If the set of fit parameters changes, then model_reset must be called.

model_update()
Update the model according to the changed parameters.

nllf(pvec=None)
compute the cost function for a new parameter set p.

this is not simply the sum-squared residuals, but instead is the negative log likelihood of seeing the data given the model parameters plus the negative log likelihood of seeing the model parameters. the value is used for a likelihood ratio test so normalization constants can be ignored. there is an additional penalty value provided by the model which can be used to implement inequality constraints. any penalty should be large enough that it is effectively excluded from the parameter space returned from uncertainty analysis.

the model is not actually calculated if the parameter nllf plus the constraint nllf are bigger than soft_limit, but instead it is assigned a value of penalty_nllf. this will prevent expensive models from spending time computing values in the unfeasible region.

parameter_nllf()
Returns negative log likelihood of seeing parameters p.

parameter_residuals()
Returns negative log likelihood of seeing parameters p.
**plot** \( (p=None, fignum=None, figfile=None, view=None) \)

Plot the problem state for the current parameter set.

The underlying Fitness object `plot` method is called with `view`. It should produce its plot on the current matplotlib figure. This method will add chisq to the plot and save it to a file.

**randomize** \( (n=None) \)
Generates a random model.

`randomize()` sets the model to a random value.

`randomize(n)` returns a population of \( n \) random models.

For indefinite bounds, the random population distribution is centered on initial value of the parameter, or 1. if the initial parameter is not finite.

**residuals**
Return the model residuals.

If the model is defined by \( y = f(x) + \epsilon \) for normally distributed error in the measurement \( y \) equal to \( \epsilon \sim N(0, \sigma^2) \), then residuals will be defined by \( R = (y - f(x))/\sigma \). If the measurement uncertainty is not normal, then the normal equivalent residuals should be defined so that the Levenberg-Marquardt fit behaves reasonably, and the plot of residuals gives an indication of which points are driving the fit.

**restore_data**
Restore original data after resynthesis.

**resynth_data**
Resynthesize data with noise from the uncertainty estimates.

**save** \( (basename) \)
Save the problem state for the current parameter set.

The underlying Fitness object `save` method is called, if it exists, so that theory values can be saved in a format suitable to the problem.

Uses `basename` as the base of any files that are created.

**setp** \( (pvec) \)
Set a new value for the parameters into the model. If the model is valid, calls model_update to signal that the model should be recalculated.

Returns True if the value is valid and the parameters were set, otherwise returns False.

**show** \( (_subs={}) \)
Print the available parameters to the console as a tree.

**simulate_data** \( (noise=None) \)
Simulate data with added noise

**stderr**
Return the 1-sigma uncertainty estimate for each parameter and the correlation matrix \( R \) as computed from the covariance returned by `cov`.

**summarize**
Return a table of current parameter values with range bars.

**to_dict**

**valid** \( (pvec) \)
Return true if the point is in the feasible region.
class bumps.fitproblem.MultiFitProblem(models, weights=None, name=None, constraints=None, soft_limit=inf, penalty_nllf=1000000.0, freevars=None)

Bases: bumps.fitproblem.BaseFitProblem

Weighted fits for multiple models. See FitProblem() for an explanation of weights.

bounds()
Return the bounds for each parameter a 2 x N array

chisq()
Return sum squared residuals normalized by the degrees of freedom.
In the context of a composite fit, the reduced chisq on the individual models only considers the points and the fitted parameters within the individual model.

Note that this does not include cost factors due to constraints on the parameters, such as sample_offset ~ N(0,0.01).

chisq_str()
Return a string representing the chisq equivalent of the nllf.
If the model has strictly gaussian independent uncertainties then the negative log likelihood function will return 0.5*sum(residuals**2), which is 1/2*chisq. Since we are printing normalized chisq, we multiply the model nllf by 2/DOF before displaying the value. This is different from the problem nllf function, which includes the cost of the prior parameters and the cost of the penalty constraints in the total nllf. The constraint value is displayed separately.

constraints_nllf()
Return the cost function for all constraints

cov()
Return the covariance matrix as computed by numdiff tools from the Hessian matrix for the problem at the current parameter values.

getp()
Returns the current value of the parameter vector.

has_residuals
True if all underlying fitness functions define residuals.

labels()
Return the list of labels, one per fitted parameter.

model_nllf()
Return cost function for all data sets

model_parameters()
Return parameters from all models

model_points()
Return number of points in all models

model_reset()
Prepare for the fit.
This sets the parameters and the bounds properties that the solver is expecting from the fittable object. We also compute the degrees of freedom so that we can return a normalized fit likelihood.
If the set of fit parameters changes, then model_reset must be called.

model_update()
Let all models know they need to be recalculated
models

Iterate over models, with free parameters set from model values

nllf (pvec=None)

compute the cost function for a new parameter set p.

this is not simply the sum-squared residuals, but instead is the negative log likelihood of seeing the data
given the model parameters plus the negative log likelihood of seeing the model parameters. the value is
used for a likelihood ratio test so normalization constants can be ignored. there is an additional penalty
value provided by the model which can be used to implement inequality constraints. any penalty should be
large enough that it is effectively excluded from the parameter space returned from uncertainty analysis.

the model is not actually calculated if the parameter nllf plus the constraint nllf are bigger than soft_limit,
but instead it is assigned a value of penalty_nllf. this will prevent expensive models from spending time
computing values in the unfeasible region.

parameter_nllf()

Returns negative log likelihood of seeing parameters p.

parameter_residuals()

Returns negative log likelihood of seeing parameters p.

plot (p=None, fignum=1, figfile=None, view=None)

Plot the problem state for the current parameter set.

The underlying Fitness object plot method is called with view. It should produce its plot on the current
matplotlib figure. This method will add chisq to the plot and save it to a file.

randomize (n=None)

Generates a random model.

randomize() sets the model to a random value.

randomize(n) returns a population of n random models.

For indefinite bounds, the random population distribution is centered on initial value of the parameter, or
1. if the initial parameter is not finite.

residuals()

Return the model residuals.

If the model is defined by $y = f(x) + \epsilon$ for normally distributed error in the measurement $y$ equal to
$\epsilon \sim N(0, \sigma^2)$, then residuals will be defined by $R = (y - f(x))/\sigma$. If the measurement uncertainty is
not normal, then the normal equivalent residuals should be defined so that the Levenberg-Marquardt fit
behaves reasonably, and the plot of residuals gives an indication of which points are driving the fit.

restore_data()

Restore original data after resynthesis.

resynth_data()

Resynthesize data with noise from the uncertainty estimates.

save (basename)

Save the problem state for the current parameter set.

The underlying Fitness object save method is called, if it exists, so that theory values can be saved in a
format suitable to the problem.

Uses basename as the base of any files that are created.

set_active_model (i)

Use free parameters from model $i$
setp\( (pvec) \)
Set a new value for the parameters into the model. If the model is valid, calls model_update to signal that the model should be recalculated.
Returns True if the value is valid and the parameters were set, otherwise returns False.

show()
Print the available parameters to the console as a tree.

simulate_data\( (noise=None) \)
Simulate data with added noise

stderr()
Return the 1-sigma uncertainty estimate for each parameter and the correlation matrix \( R \) as computed from the covariance returned by \( cov \).

summarize()
Return a table of current parameter values with range bars.

to_dict()

valid\( (pvec) \)
Return true if the point is in the feasible region

4.9 fitservice - Remote job plugin for fit jobs

ServiceMonitor
Display fit progress on the console

fitservice

Fit job definition for the distributed job queue.

class bumps.fitservice.ServiceMonitor\( (problem, path, progress=60, improvement=60) \)
Bases: bumps.monitor.TimedUpdate
Display fit progress on the console

config_history\( (history) \)
Indicate which fields are needed by the monitor and for what duration.

show_improvement\( (history) \)

show_progress\( (history) \)
bumps.fitservice.fitservice\( (request) \)

4.10 fitters - Wrappers for various optimization algorithms

<table>
<thead>
<tr>
<th>BFGSFit</th>
<th>BFGS quasi-newton optimizer.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CheckpointMonitor</td>
<td>Periodically save fit state so that it can be resumed later.</td>
</tr>
<tr>
<td>ConsoleMonitor</td>
<td>Display fit progress on the console</td>
</tr>
<tr>
<td>DEFit</td>
<td>Classic Storn and Price differential evolution optimizer.</td>
</tr>
<tr>
<td>DreamFit</td>
<td>DREAM wrapper for fit problems.</td>
</tr>
</tbody>
</table>
Table 10 – continued from previous page

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FitBase</strong></td>
<td>FitBase defines the interface from bumps models to the various fitting engines available within bumps.</td>
</tr>
<tr>
<td><strong>FitDriver</strong></td>
<td></td>
</tr>
<tr>
<td><strong>LevenbergMarquardtFit</strong></td>
<td>Levenberg-Marquardt optimizer.</td>
</tr>
<tr>
<td><strong>MPFit</strong></td>
<td>MPFit optimizer.</td>
</tr>
<tr>
<td><strong>MonitorRunner</strong></td>
<td>Adaptor which allows solvers to accept progress monitors.</td>
</tr>
<tr>
<td><strong>MultiStart</strong></td>
<td>Multi-start monte carlo fitter.</td>
</tr>
<tr>
<td><strong>PSFit</strong></td>
<td>Particle swarm optimizer.</td>
</tr>
<tr>
<td><strong>PTFit</strong></td>
<td>Parallel tempering optimizer.</td>
</tr>
<tr>
<td><strong>RLFit</strong></td>
<td>Random lines optimizer.</td>
</tr>
<tr>
<td><strong>Resampler</strong></td>
<td></td>
</tr>
<tr>
<td><strong>SimplexFit</strong></td>
<td>Nelder-Mead simplex optimizer.</td>
</tr>
<tr>
<td><strong>SnobFit</strong></td>
<td></td>
</tr>
<tr>
<td><strong>StepMonitor</strong></td>
<td>Collect information at every step of the fit and save it to a file.</td>
</tr>
<tr>
<td><strong>fit</strong></td>
<td>Simpified fit interface.</td>
</tr>
<tr>
<td><strong>load_history</strong></td>
<td>Load fitter details from a history file.</td>
</tr>
<tr>
<td><strong>parse_tolerance</strong></td>
<td>Register a new fitter with bumps, if it is not already there.</td>
</tr>
<tr>
<td><strong>register</strong></td>
<td>Save fitter details to a history file as JSON.</td>
</tr>
</tbody>
</table>

Interfaces to various optimizers.

```python
class bumps.fitters.BFGSFit(problem)
    Bases: bumps.fitters.FitBase
    BFGS quasi-newton optimizer.
    BFGS estimates Hessian and its cholesky decomposition, but initial tests give uncertainties quite different from the directly computed Jacobian in Levenburg-Marquardt or the Hessian estimated at the minimum by numdifftools.
    To use the internal 'H' and 'L' and save some computation time, then use:

    C = lsqerror.chol_cov(fit.result['L'])
    stderr = lsqerror.stderr(C)

    id = 'newton'
    name = 'Quasi-Newton BFGS'
    settings = [('steps', 3000), ('starts', 1), ('ftol', 1e-06), ('xtol', 1e-12)]
    solve(monitors=None, abort_test=None, mapper=None, **options)

class bumps.fitters.CheckpointMonitor(checkpoint, progress=1800)
    Bases: bumps.monitor.TimedUpdate
    Periodically save fit state so that it can be resumed later.

    checkpoint = None
    Function to call at each checkpoint.

    config_history(history)
    Indicate which fields are needed by the monitor and for what duration.

    show_improvement(history)
```

4.10. fitters - Wrappers for various optimization algorithms
show_progress(history)

class bumps.fitters.ConsoleMonitor(problem, progress=1, improvement=30)
    Bases: bumps.monitor.TimedUpdate
    Display fit progress on the console

    config_history(history)
        Indicate which fields are needed by the monitor and for what duration.

    show_improvement(history)
    show_progress(history)

class bumps.fitters.DEFit(problem)
    Bases: bumps.fitters.FitBase
    Classic Storn and Price differential evolution optimizer.

        id = 'de'
        load(input_path)
        name = 'Differential Evolution'
        save(output_path)

        settings = [(('steps', 1000), ('pop', 10), ('CR', 0.9), ('F', 2.0), ('ftol', 1e-08), ('xtol', 1e-06)]
        solve(monitors=None, abort_test=None, mapper=None, **options)

class bumps.fitters.DreamFit(problem)
    Bases: bumps.fitters.FitBase

        entropy(**kw)
        error_plot(figfile)
        id = 'dream'
        load(input_path)
        name = 'DREAM'
        plot(output_path)
        save(output_path)

        settings = [(('samples', 10000), ('burn', 100), ('pop', 10), ('init', 'eps'), ('thin', 1), ('alpha', 0.01), ('outliers', 'none'), ('trim', False), ('steps', 0)]
        show()
        solve(monitors=None, abort_test=None, mapper=None, **options)
        stderr()
            Approximate standard error as 1/2 the 68% interval for the sample, which is a more robust measure than
            the mean of the sample for non-normal distributions.

class bumps.fitters.DreamModel(problem=None, mapper=None)
    Bases: bumps.dream.model.MCMCModel
    DREAM wrapper for fit problems.

        bounds = None
        labels = None
        log_density(x)
map \((\text{pop})\)

\text{nllf} \((x)\)

Negative log likelihood of seeing models given \(x\)

\text{plot} \((x)\)

class bumps.fitters.FitBase \((\text{problem})\)

Bases: \text{object}

FitBase defines the interface from bumps models to the various fitting engines available within bumps.

Each engine is defined in its own class with a specific set of attributes and methods.

The \text{name} attribute is the name of the optimizer. This is just a simple string.

The \text{settings} attribute is a list of pairs (name, default), where the names are defined as fields in FitOptions. A best attempt should be made to map the fit options for the optimizer to the standard fit options, since each of these becomes a new command line option when running bumps. If that is not possible, then a new option should be added to FitOptions. A plugin architecture might be appropriate here, if there are reasons why specific problem domains might need custom fitters, but this is not yet supported.

Each engine takes a fit problem in its constructor.

The \text{solve()} method runs the fit. It accepts a monitor to track updates, a mapper to distribute work and key-value pairs defining the settings.

There are a number of optional methods for the fitting engines. Basically, all the methods in FitDriver first check if they are specialized in the fit engine before performing a default action.

The \text{load/}save methods load and save the fitter state in a given directory with a specific base file name. The fitter can choose a file extension to add to the base name. Some care is needed to be sure that the extension doesn’t collide with other extensions such as .mon for the fit monitor.

The \text{plot} method shows any plots to help understand the performance of the fitter, such as a convergence plot showing the the range of values in the population over time, as well as plots of the parameter uncertainty if available. The plot should work within is given a figure canvas to work with.

The \text{stderr/}cov methods should provide summary statistics for the parameter uncertainties. Some fitters, such as MCMC, will compute these directly from the population. Others, such as BFGS, will produce an estimate of the uncertainty as they go along. If the fitter does not provide these estimates, then they will be computed from numerical derivatives at the minimum in the FitDriver method.

solve \((\text{monitors=None, mapper=None, **options})\)

class bumps.fitters.FitDriver \((\text{fitclass=None, problem=None, monitors=None, abort_test=None, mapper=None, **options})\)

Bases: \text{object}

chisq()

clip()

Force parameters within bounds so constraints are finite.

The problem is updated with the new parameter values.

Returns a list of parameter names that were clipped.

cov()

Return an estimate of the covariance of the fit.

Depending on the fitter and the problem, this may be computed from existing evaluations within the fitter, or from numerical differentiation around the minimum.

If the problem uses \(\chi^2/2\) as its nllf, then the covariance is derived from the Jacobian:
x = fit.problem.getp()
J = bumps.lsqerror.jacobian(fit.problem, x)
cov = bumps.lsqerror.jacobian_cov(J)

Otherwise, the numerical differentiation will use the Hessian estimated from nllf:

x = fit.problem.getp()
H = bumps.lsqerror.hessian(fit.problem, x)
cov = bumps.lsqerror.hessian_cov(H)

entropy (method=None)
fit (resume=None)
load (input_path)
plot (output_path, view=None)
save (output_path)
show ()
show_cov ()
show_entropy (method=None)
show_err ()
Display the error approximation from the numerical derivative.
Warning: cost grows as the cube of the number of parameters.
stderr ()
Return an estimate of the standard error of the fit.
Depending on the fitter and the problem, this may be computed from existing evaluations within the fitter,
or from numerical differentiation around the minimum.
stderr_from_cov ()
Return an estimate of standard error of the fit from covariance matrix.
Unlike stderr, which uses the estimate from the underlying fitter (DREAM uses the MCMC sample for this),
stderr_from_cov estimates the error from the diagonal of the covariance matrix. Here, the covariance
matrix may have been estimated by the fitter instead of the Hessian.

class bumps.fitters.LevenbergMarquardtFit (problem)
Bases: bumps.fitters.FitBase
Levenberg-Marquardt optimizer.
cov ()
id = 'scipy.leastsq'
name = 'Levenberg-Marquardt (scipy.leastsq)'
settings = [("steps", 200), ('ftol', 1.5e-08), ('xtol', 1.5e-08)]
solve (monitors=None, abort_test=None, mapper=None, **options)

class bumps.fitters.MPFit (problem)
Bases: bumps.fitters.FitBase
MPFit optimizer.
id = 'lm'
name = 'Levenberg–Marquardt'

settings = [('steps', 200), ('ftol', 1e-10), ('xtol', 1e-10)]
solve(monitors=None, abort_test=None, mapper=None, **options)

class bumps.fitters.MonitorRunner(monitors, problem)
Bases: object
Adaptor which allows solvers to accept progress monitors.

class bumps.fitters.MultiStart(fitter)
Bases: bumps.fitters.FitBase
Multi-start monte carlo fitter.
This fitter wraps a local optimizer, restarting it a number of times to give it a chance to find a different local
minimum. If the keep_best option is True, then restart near the best fit, otherwise restart at random.

name = 'Multistart Monte Carlo'

settings = [('starts', 100)]
solve(monitors=None, mapper=None, **options)

class bumps.fitters.PSFit(problem)
Bases: bumps.fitters.FitBase
Particle swarm optimizer.

id = 'ps'

name = 'Particle Swarm'

settings = [('steps', 3000), ('pop', 1)]
solve(monitors=None, mapper=None, **options)

class bumps.fitters.PTFit(problem)
Bases: bumps.fitters.FitBase
Parallel tempering optimizer.

id = 'pt'

name = 'Parallel Tempering'

settings = [('steps', 400), ('nT', 24), ('CR', 0.9), ('burn', 100), ('Tmin', 0.1), ('Tmax', 10)]
solve(monitors=None, mapper=None, **options)

class bumps.fitters.RLFit(problem)
Bases: bumps.fitters.FitBase
Random lines optimizer.

id = 'rl'

name = 'Random Lines'

settings = [('steps', 3000), ('starts', 20), ('pop', 0.5), ('CR', 0.9)]
solve(monitors=None, abort_test=None, mapper=None, **options)

class bumps.fitters.Resampler(fitter)
Bases: bumps.fitters.FitBase
solve(**options)
import Bumps: Curve Fitting and Uncertainty Analysis, Release 0.9.0

```python
class bumps.fitters.SimplexFit(problem)
    Bases: bumps.fitters.FitBase
    Nelder-Mead simplex optimizer.
    id = 'amoeba'
    name = 'Nelder-Mead Simplex'
    settings = [('steps', 1000), ('starts', 1), ('radius', 0.15), ('xtol', 1e-06), ('ftol'

solve(monitors=None, abort_test=None, mapper=None, **options)
```

class bumps.fitters.SnobFit(problem)
    Bases: bumps.fitters.FitBase
    id = 'snobfit'
    name = 'SNOBFIT'
    settings = [('steps', 200)]

solve(monitors=None, mapper=None, **options)
```

class bumps.fitters.StepMonitor(problem, fid, fields=['step', 'time', 'value', 'point'])
    Bases: bumps.monitor.Monitor
    Collect information at every step of the fit and save it to a file.
    fid is the file to save the information to fields is the list of "step|time|value|point" fields to save
    The point field should be last in the list.
    FIELDS = ['step', 'time', 'value', 'point']

config_history(history)
    Indicate which fields are needed by the monitor and for what duration.

bumps.fitters.fit(problem, method='amoeba', verbose=False, **options)
Simplified fit interface.
Given a fit problem, the name of a fitter and the fitter options, it will run the fit and return the best value and standard error of the parameters. If verbose is true, then the console monitor will be enabled, showing progress through the fit and showing the parameter standard error at the end of the fit, otherwise it is completely silent.

Returns an OptimizeResult object containing “x” and “dx”. The dream fitter also includes the “state” object, allowing for more detailed uncertainty analysis. Optimizer information such as the stopping condition and the number of function evaluations are not yet included.

To run in parallel (with multiprocessing and dream):
```
from bumps.mapper import MPMapper
mapper = MPMapper.start_mapper(problem, None, cpu=0)  #cpu=0 for all CPUs
result = fit(problem, method="dream", mapper=mapper)
```

bumps.fitters.load_history(path)
Load fitter details from a history file.

bumps.fitters.parse_tolerance(options)

bumps.fitters.register(fitter, active=True)
Register a new fitter with bumps, if it is not already there.
active is False if you don’t want it showing up in the GUI selector.
bumps.fitters.save_history(path, state)
Save fitter details to a history file as JSON.
The content of the details are fitter specific.

4.11 formatnum - Format numbers and uncertainties

<table>
<thead>
<tr>
<th>format_value</th>
<th>Given value v and uncertainty dv, return a string v which is the value formatted with the appropriate number of digits.</th>
</tr>
</thead>
<tbody>
<tr>
<td>format_uncertainty</td>
<td>Value and uncertainty formatter.</td>
</tr>
<tr>
<td>format_uncertainty_compact</td>
<td>Given value v and uncertainty dv, return the compact representation v(##), where ## are the first two digits of the uncertainty.</td>
</tr>
<tr>
<td>format_uncertainty_pm</td>
<td>Given value v and uncertainty dv, return a string v +/- dv.</td>
</tr>
</tbody>
</table>

Format values and uncertainties nicely for printing.
The formatted value uses only the number of digits warranted by the uncertainty in the measurement.

format_value() shows the value without the uncertainty.
format_uncertainty_pm() shows the expanded format v +/- err.
format_uncertainty_compact() shows the compact format v(##), where the number in parenthesis is the uncertainty in the last two digits of v.
format_uncertainty() uses the compact format by default, but this can be changed to use the expanded +/- format by setting format_uncertainty.compact to False. This is a global setting which should be considered a user preference. Any library code that depends on a specific format style should use the corresponding formatting function.

If the uncertainty is 0 or not otherwise provided, the simple %g floating point format option is used.

Infinite and indefinite numbers are represented as inf and NaN.

Example:

```python
goldenibilidad = 757.2356, 0.01032
>>> print(format_uncertainty_pm(goldenibilidad, 0.01032))
757.236 +/- 0.010

>>> print(format_uncertainty_compact(goldenibilidad, 0.01032))
757.236(10)

>>> print(format_uncertainty(goldenibilidad, 0.01032))
757.236(10)

>>> format_uncertainty.compact = False

>>> print(format_uncertainty(goldenibilidad, 0.01032))
757.236 +/- 0.010

>>> format_uncertainty.compact = True  # restore default
```

bumps.formatnum.format_value(value, uncertainty)
Given value v and uncertainty dv, return a string v which is the value formatted with the appropriate number of digits.

bumps.formatnum.format_uncertainty(value, uncertainty)
Value and uncertainty formatter.
Either the expanded v +/- dv form or the compact v(##) form will be used depending on whether \texttt{format\_uncertainty\_compact} is True or False. The default is True.

\begin{Verbatim}
\texttt{bumps.formatnum.format\_uncertainty\_compact(value, uncertainty)}
\end{Verbatim}

Given \texttt{value} v and \texttt{uncertainty} dv, return the compact representation v(##), where ## are the first two digits of the uncertainty.

\begin{Verbatim}
\texttt{bumps.formatnum.format\_uncertainty\_pm(value, uncertainty)}
\end{Verbatim}

Given \texttt{value} v and \texttt{uncertainty} dv, return a string v +/- dv.

## 4.12 history - Optimizer evaluation trace

### History

<table>
<thead>
<tr>
<th>History</th>
<th>Collection of traces.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace</td>
<td>Value trace.</td>
</tr>
</tbody>
</table>

Log of progress through a computation.

Each cycle through a computation, a process can update its history, adding information about the number of function evaluations, the total time taken, the set of points evaluated and their values, the current best value and so on. The process can use this history when computing the next set of points to evaluate and when checking if the termination conditions are met. Any values that may be useful outside the computation, e.g., for logging or for updating the user, should be recorded. In the ideal case, the history is all that is needed to restart the process in case of a system crash.

History consists of a set of traces. The content of the traces themselves is provided by the computation, but various stake holders can use them. For example, the user may wish to log the set of points that have been evaluated and their values using the system logger and an optimizer may require a certain amount of history to calculate the next set of values.

New traces are provided using \texttt{History.provides()}. For example, the following adds traces for ‘value’ and ‘point’ to the history, and requires the best value on the two previous cycles in order to do its work:

\begin{Verbatim}
>>> from bumps.history import History
>>> h = History()
>>> h.provides(value=2, point=0)
\end{Verbatim}

Initially the history is empty:

\begin{Verbatim}
>>> print(len(h.value))
0
\end{Verbatim}

After three updates we see that only two values are kept:

\begin{Verbatim}
>>> h.update(value=2, point=[1,1,1])
>>> h.update(value=1, point=[1,0.5,1])
>>> h.update(value=0.5, point=[1,0.5,0.9])
>>> print(h.value)
Trace value: 0.5, 1
>>> print(len(h.value))
2
\end{Verbatim}

Note that point is not monitored since it is not required:

\begin{Verbatim}
>>> print(h.point[0])
Traceback (most recent call last):
  ...
IndexError: point has not accumulated enough history
\end{Verbatim}
Traces may be used as accumulators. In that case, the next value is added to the tail value before appending to the trace. For example:

```python
>>> h = History()
>>> h.provides(step=1)
>>> h.accumulate(step=1)
>>> h.accumulate(step=1)
>>> print(h.step[0])
2
```

```python
class bumps.history.History(**kw)
    Bases: object
    Collection of traces.
    Provided traces can be specified as key word arguments, name=length.

    accumulate(**kw)
        Extend the given traces with the provided values. The traced value will be the old value plus the new value.

    clear()
        Clear history, removing all traces

    provides(**kw)
        Specify additional provided fields.
        Raises AttributeError if trace is already provided or if the trace name matches the name of one of the history methods.

    requires(**kw)
        Specify required fields, and their history length.

    restore(state)
        Restore history to the state returned by a call to snapshot

    snapshot()
        Return a dictionary of traces { ‘name’: [v[n], v[n-1], ..., v[0]] }

    update(**kw)
        Extend the given traces with the provided values. The traced values are independent. Use accumulate if you want to add the new value to the previous value in the trace.
```

```python
class bumps.history.Trace(keep=1, name='trace')
    Bases: object
    Value trace.
    This is a stack-like object with items inserted at the beginning, and removed from the end once the maximum length keep is reached.

    len(trace) returns the number of items in the trace trace[i] returns the ith previous element in the history

    trace.requires(n) says how much history to keep

    trace.put(value) stores value

    trace.accumulate(value) adds value to the previous value before storing

    state = trace.snapshot() returns the values as a stack, most recent last

    trace.restore(state) restores a snapshot

    Note that snapshot/restore uses lists to represent numpy arrays, which may cause problems if the trace is capturing lists.

    accumulate(value)
        Add an item to the trace, shifting off from the beginning when the trace is full.
```
4.13 initpop - Population initialization strategies

**generate**  
Population initializer.

**cov_init**  
Initialize $n$ sets of random variables from a gaussian model.

**eps_init**  
Generate a random population using an epsilon ball around the current value.

**lhs_init**  
Latin hypercube sampling.

**random_init**  
Generate a random population from the problem parameters.

Population initialization strategies.

To start the analysis an initial population is required. This will be an array of size $M \times N$, where $M$ is the number of dimensions in the fitting problem and $N$ is the number of individuals in the population.

Normally the initialization will use a call to `generate()` with key-value pairs from the command line options. This will include the ‘init’ option, with the name of the strategy used to initialize the population.

Additional strategies like uniform box in $[0,1]$ or standard norm (rand(m,n) and randn(m,n) respectively), may also be useful.

bumps.initpop.generate(\textit{problem}, init='eps', pop=10, use_point=True, **options)  
Population initializer.

\textit{problem} is a fit problem with getp and bounds methods.

\textit{init} is ‘eps’, ‘cov’, ‘lhs’ or ‘random’, indicating which initializer should be used.

\textit{pop} is the population scale factor, generating \textit{pop} individuals for each parameter in the fit. If \textit{pop} < 0, generate a total of -\textit{pop} individuals regardless of the number of parameters.

\textit{use_point} is True if the initial value should be a member of the population.

Additional options are ignored so that generate can be called using all command line options.

bumps.initpop.cov_init(\textit{n}, initial, bounds, use_point=False, cov=None, dx=None)  
Initialize $n$ sets of random variables from a gaussian model.

The center is at $x$ with an uncertainty ellipse specified by the 1-sigma independent uncertainty values $dx$ or the full covariance matrix uncertainty $cov$.

For example, create an initial population for 20 sequences for a model with local minimum $x$ with covariance matrix $C$: 
pop = cov_init(cov=C, pars=p, n=20)

If `use_point` is True, then the current value of the parameters is returned as the first point in the population.

```python
bumps.initpop.eps_init(n, initial, bounds, use_point=False, eps=1e-06)
```
Generate a random population using an epsilon ball around the current value.

Since the initial population is contained in a small volume, this method is useful for exploring a local minimum around a point. Over time the ball will expand to fill the minimum, and perhaps tunnel through barriers to nearby minima given enough burn-in time.

eps is in proportion to the bounds on the parameter, or the current value of the parameter if the parameter is unbounded. This gives the initialization a bit of scale independence.

If `use_point` is True, then the current value of the parameters is returned as the first point in the population.

```python
bumps.initpop.lhs_init(n, initial, bounds, use_point=False)
```
Latin hypercube sampling.

Returns an array whose columns and rows each have \( n \) samples from equally spaced bins between `bounds=(xmin, xmax)` for the column. Unlike random, this method guarantees a certain amount of coverage of the parameter space. Consider, though that the diagonal matrix satisfies the LHS condition, and you can see that the guarantees are not very strong. A better methods, similar to sudoku puzzles, would guarantee coverage in each block of the matrix, but this is not yet implemented.

If `use_point` is True, then the current value of the parameters is returned as the first point in the population, preserving the the LHS property.

```python
bumps.initpop.random_init(n, initial, bounds, use_point=False, problem=None)
```
Generate a random population from the problem parameters.

Values are selected at random from the bounds of the problem according to the underlying probability density of each parameter. Uniform semi-definite and indefinite bounds use the standard normal distribution for the underlying probability, with a scale factor determined by the initial value of the parameter.

If `use_point` is True, then the current value of the parameters is returned as the first point in the population.

### 4.14 Isqerror - Least squares error analysis

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>chol_cov</code></td>
<td>Given the cholesky decomposition of the Hessian matrix ( H ), compute the covariance matrix ( C = H^{-1} ).</td>
</tr>
<tr>
<td><code>chol_stderr</code></td>
<td>Return parameter uncertainty from the Cholesky decomposition of the Hessian matrix, as returned, e.g., from the quasi-Newton optimizer BFGS or as calculated from <code>perturbed_hessian()</code> on the output of <code>hessian()</code> applied to the cost function problem.nllf.</td>
</tr>
<tr>
<td><code>comb</code></td>
<td>n choose r combination function.</td>
</tr>
<tr>
<td><code>corr</code></td>
<td>Convert covariance matrix ( C ) to correlation matrix ( R^2 ).</td>
</tr>
<tr>
<td><code>demo_hessian</code></td>
<td></td>
</tr>
<tr>
<td><code>demo_jacobian</code></td>
<td></td>
</tr>
<tr>
<td><code>demo_stderr_hilbert</code></td>
<td></td>
</tr>
<tr>
<td><code>demo_stderr_perturbed</code></td>
<td></td>
</tr>
<tr>
<td><code>gradient</code></td>
<td></td>
</tr>
<tr>
<td><code>hessian</code></td>
<td>Returns the derivative wrt to the fit parameters at point ( p ).</td>
</tr>
</tbody>
</table>

Continued on next page
Table 14 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hessian_cov</td>
<td>Given Hessian H, return the covariance matrix ( \text{inv}(H) ).</td>
</tr>
<tr>
<td>hilbert</td>
<td>Generate ill-conditioned Hilbert matrix of size ( n \times n ).</td>
</tr>
<tr>
<td>hilbertinv</td>
<td>Analytical inverse for ill-conditioned Hilbert matrix of size ( n \times n ).</td>
</tr>
<tr>
<td>jacobian</td>
<td>Returns the derivative wrt the fit parameters at point ( p ).</td>
</tr>
<tr>
<td>jacobian_cov</td>
<td>Given Jacobian J, return the covariance matrix ( \text{inv}(J^T J) ).</td>
</tr>
<tr>
<td>max_correlation</td>
<td>Return the maximum correlation coefficient for any pair of variables in correlation matrix Rsq.</td>
</tr>
<tr>
<td>perturbed_hessian</td>
<td>DEPRECATED Numerical testing has shown that the perturbed Hessian is too aggressive with its perturbation, and it is distorting the error too much, so use hessian_cov(H) instead.</td>
</tr>
<tr>
<td>stderr</td>
<td>Return parameter uncertainty from the covariance matrix ( C ).</td>
</tr>
</tbody>
</table>

Least squares error analysis.

Given a data set with gaussian uncertainty on the points, and a model which is differentiable at the minimum, the parameter uncertainty can be estimated from the covariance matrix at the minimum. The model and data are wrapped in a problem object, which must define the following methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getp()</td>
<td>get the current value of the model</td>
</tr>
<tr>
<td>setp(p)</td>
<td>set a new value in the model</td>
</tr>
<tr>
<td>nllf(p)</td>
<td>negative log likelihood function</td>
</tr>
<tr>
<td>residuals(p)</td>
<td>residuals around its current value</td>
</tr>
<tr>
<td>bounds()</td>
<td>get the bounds on the parameter ( p ) [optional]</td>
</tr>
</tbody>
</table>

\( \text{jacobian()} \) computes the Jacobian matrix \( J \) using numerical differentiation on residuals. Derivatives are computed using the center point formula, with two evaluations per dimension. If the problem has analytic derivatives with respect to the fitting parameters available, then these should be used to compute the Jacobian instead.

\( \text{hessian()} \) computes the Hessian matrix \( H \) using numerical differentiation on nllf.

\( \text{jacobian_cov()} \) takes the Jacobian and computes the covariance matrix \( C \). \( \text{hessian_cov()} \) takes the Hessian and computes the covariance matrix \( C \).

\( \text{corr()} \) uses the off-diagonal elements of \( C \) to compute correlation coefficients \( R_{ij}^2 \) between the parameters.

\( \text{stderr()} \) computes the uncertain \( \sigma_i \) from covariance matrix \( C \), assuming that the \( C_{\text{diag}} \) contains \( \sigma_i^2 \), which should be the case for functions which are approximately linear near the minimum.

\( \text{max_correlation()} \) takes \( R^2 \) and returns the maximum correlation.

The user should be shown the uncertainty \( \sigma_i \) for each parameter, and if there are strong parameter correlations (e.g., \( R_{\text{max}}^2 > 0.2 \)), the correlation matrix as well.

The bounds method for the problem is optional, and is used only to determine the step size needed for the numerical derivative. If bounds are not present and finite, the current value for the parameter is used as a basis to estimate step size.

\[ \text{bumps.lsqerror.chol_cov}(L) \]

Given the cholesky decomposition of the Hessian matrix \( H \), compute the covariance matrix \( C = H^{-1} \)

**Warning:** assumes \( H = L@L^T \) (numpy default) not \( H = U^T@U \) (scipy default).

\[ \text{bumps.lsqerror.chol_stderr}(L) \]

Return parameter uncertainty from the Cholesky decomposition of the Hessian matrix, as returned, e.g., from the
quasi-Newton optimizer BFGS or as calculated from `perturbed_hessian()` on the output of `hessian()` applied to the cost function problem.nllf.

Note that this calls chol_cov to compute the inverse from the Cholesky decomposition, so use stderr(C) if you are already computing C = chol_cov().

**Warning:** assumes $H = L@L.T$ (numpy default) not $H = U.T@U$ (scipy default).

**bumps.lsqerror.comb** $(n, r)$

$n$ choose $r$ combination function

**bumps.lsqerror.corr** $(C)$

Convert covariance matrix $C$ to correlation matrix $R^2$.

Uses $R = D^{-1}CD^{-1}$ where $D$ is the square root of the diagonal of the covariance matrix, or the standard error of each variable.

**bumps.lsqerror.demo_hessian**

**bumps.lsqerror.demo_jacobian**

**bumps.lsqerror.demo_stderr_hilbert** $(n=5)$

**bumps.lsqerror.demo_stderr_perturbed**

**bumps.lsqerror.gradient** *(problem, p=None, step=None)*

Returns the derivative wrt to the fit parameters at point $p$.

The current point is preserved.

**bumps.lsqerror.hessian_cov** $(H, tol=1e-15)$

Given Hessian $H$, return the covariance matrix inv($H$).

We provide some protection against singular matrices by setting singular values smaller than tolerance $tol$ (relative to the largest singular value) to zero (see np.linalg.pinv for details).

**bumps.lsqerror.hilbert** $(n)$

Generate ill-conditioned Hilbert matrix of size $n \times n$

**bumps.lsqerror.hilbertinv** $(n)$

Analytical inverse for ill-conditioned Hilbert matrix of size $n \times n$

**bumps.lsqerror.jacobian** *(problem, p=None, step=None)*

Returns the derivative wrt the fit parameters at point $p$.

Numeric derivatives are calculated based on step, where step is the portion of the total range for parameter $j$, or the portion of point value $p_j$ if the range on parameter $j$ is infinite.

The current point is preserved.

**bumps.lsqerror.jacobian_cov** $(J, tol=1e-08)$

Given Jacobian $J$, return the covariance matrix inv($J'J$).

We provide some protection against singular matrices by setting singular values smaller than tolerance $tol$ to the tolerance value.

**bumps.lsqerror.max_correlation** $(Rsq)$

Return the maximum correlation coefficient for any pair of variables in correlation matrix Rsq.

**bumps.lsqerror.perturbed_hessian** $(H, scale=None)$

**DEPRECATED** Numerical testing has shown that the perturbed Hessian is too aggressive with its perturbation, and it is distorting the error too much, so use hessian_cov($H$) instead.

Adjust Hessian matrix to be positive definite.
Returns the adjusted Hessian and its Cholesky decomposition.

```python
bumps.lsqerror.stderr(C)
```

Return parameter uncertainty from the covariance matrix C.

This is just the square root of the diagonal, without any correction for covariance.

If measurement uncertainty is unknown, scale the returned uncertainties by \( \sqrt{\chi^2_N} \), where \( \chi^2_N \) is the sum squared residuals divided by the degrees of freedom. This will match the uncertainty on the parameters to the observed scatter assuming the model is correct and the fit is optimal. This will also be appropriate for weighted fits when the true measurement uncertainty \( dy_i \) is known up to a scaling constant for all \( y_i \).

Standard error on `scipy.optimize.curve_fit` always includes the chisq correction, whereas `scipy.optimize.leastsq` never does.

### 4.15 mapper - Parallel processing implementations

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>AMQPMapper</code></td>
<td>Returns True if <code>problem</code> can be pickled.</td>
</tr>
<tr>
<td><code>MPIMapper</code></td>
<td></td>
</tr>
<tr>
<td><code>MPMapper</code></td>
<td></td>
</tr>
<tr>
<td><code>SerialMapper</code></td>
<td></td>
</tr>
<tr>
<td><code>can_pickle</code></td>
<td></td>
</tr>
<tr>
<td><code>nice</code></td>
<td></td>
</tr>
<tr>
<td><code>setpriority</code></td>
<td></td>
</tr>
</tbody>
</table>

Parallel and serial mapper implementations.

The API is a bit crufty since interprocess communication has evolved from the original implementation. And the names are misleading.

Usage:

```python
mapper = Mapper.start_mapper(problem, None, cpus)
result = mapper(points)
...  
mapper = Mapper.start_mapper(problem, None, cpus)
result = mapper(points)
Mapper.stop_mapper(mapper)
```

```python
class bumps.mapper.AMQPMapper
    Bases: object
    static start_mapper (problem, modelargs, cpus=0)
    static start_worker (problem)
    static stop_mapper (mapper)

class bumps.mapper.MPIMapper
    Bases: object
    static start_mapper (problem, modelargs, cpus=0)
    static start_worker (problem)
```

Start the worker process.

For the main process this does nothing and returns immediately. The worker processes never return.
Each worker sits in a loop waiting for the next batch of points for the problem, or for the next problem. Set `problem` is set to None, then exit the process and never

```python
class bumps.mapper.MPMapper
    manager = None
    namespace = None
    pool = None
    problem_id = 0
    static start_mapper (problem, modelargs, cpus=0)
    static start_worker (problem)
    static stop_mapper (mapper)
```

```python
class bumps.mapper.SerialMapper
    static start_mapper (problem, modelargs, cpus=0)
    static start_worker (problem)
    static stop_mapper (mapper)
```

```python
bumps.mapper.can_pickle (problem, check=False)
    Returns True if `problem` can be pickled.
    If this method returns False then MPMapper cannot be used and SerialMapper should be used instead.
    If `check` is True then call `nllf()` on the duplicated object as a “smoke test” to verify that the function will run after copying. This is not foolproof. For example, access to a database may work in the duplicated object because the connection is open and available in the current process, but it will fail when trying to run on a remote machine.
```

```python
bumps.mapper.setpriority (pid=None, priority=1)
    Set The Priority of a Windows Process. Priority is a value between 0-5 where 2 is normal priority and 5 is maximum. Default sets the priority of the current python process but can take any valid process ID.
```

### 4.16 monitor - Monitor fit progress

<table>
<thead>
<tr>
<th>Monitor</th>
<th>Base class for monitors.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logger</td>
<td>Keeps a record of all values for the desired fields.</td>
</tr>
<tr>
<td>TimedUpdate</td>
<td>Indicate progress every n seconds.</td>
</tr>
</tbody>
</table>

Progress monitors.

Process monitors accept a `bumps.history.History` object each cycle and perform some sort of work.

```python
class bumps.monitor.Monitor
    bases: object
    Base class for monitors.
    config_history (history)
```
Indicate which fields are needed by the monitor and for what duration.

```python
class bumps.monitor.Logger(fields=(), table=None)
    Bases: bumps.monitor.Monitor
    Keeps a record of all values for the desired fields.
    fields is a list of history fields to store.
    table is an object with a store(field=value,...) method, which gets the current value of each field every time the
    history is updated.
    Call config_history() with the bumps.history.History object before starting so that the correct
    fields are stored.

class bumps.monitor.TimedUpdate(progress=60, improvement=5)
    Bases: bumps.monitor.Monitor
    Indicate progress every n seconds.
    The process should provide time, value, point, and step to the history update. Call config_history() with
    the bumps.history.History object before starting so that these fields are stored.
    progress is the number of seconds to go before showing progress, such as time or step number.
    improvement is the number of seconds to go before showing improvements to value.
    By default, the updater only prints step number and improved value. Subclass TimedUpdate with replaced
    show_progress() and show_improvement() to trigger GUI updates or show parameter values.

class bumps.monitor.TimedUpdate(progress=60, improvement=5)
    Bases: bumps.monitor.Monitor
    Indicate progress every n seconds.
    The process should provide time, value, point, and step to the history update. Call config_history() with
    the bumps.history.History object before starting so that these fields are stored.
    progress is the number of seconds to go before showing progress, such as time or step number.
    improvement is the number of seconds to go before showing improvements to value.
    By default, the updater only prints step number and improved value. Subclass TimedUpdate with replaced
    show_progress() and show_improvement() to trigger GUI updates or show parameter values.

class bumps.monitor.TimedUpdate(progress=60, improvement=5)
    Bases: bumps.monitor.Monitor
    Indicate progress every n seconds.
    The process should provide time, value, point, and step to the history update. Call config_history() with
    the bumps.history.History object before starting so that these fields are stored.
    progress is the number of seconds to go before showing progress, such as time or step number.
    improvement is the number of seconds to go before showing improvements to value.
    By default, the updater only prints step number and improved value. Subclass TimedUpdate with replaced
    show_progress() and show_improvement() to trigger GUI updates or show parameter values.
```

## 4.17 mono - Freeform - Monotonic Spline

<table>
<thead>
<tr>
<th>monospline</th>
<th>Monotonic cubic hermite interpolation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>hermite</td>
<td>Computes the cubic hermite polynomial $p(x_i)$.</td>
</tr>
<tr>
<td>count_inflections</td>
<td>Count the number of inflection points in a curve.</td>
</tr>
<tr>
<td>plot_inflections</td>
<td>Plot inflection points in a curve.</td>
</tr>
</tbody>
</table>

Monotonic spline modeling.

```python
bumps.mono.monospline(x, y, xt)
    Monotonic cubic hermite interpolation.
    Returns $p(x_i)$ where $p(x_i) = y_i$ and $p(x_i) \leq p(x_i)$ if $y_i \leq y_{i+1}$ for all $y_i$. Also works for decreasing values $y$, resulting in decreasing $p(x)$. If $y$ is not monotonic, then $p(x)$ may peak higher than any $y$, so this function is not suitable for a strict constraint on the interpolated function when $y$ values are unconstrained.
    http://en.wikipedia.org/wiki/Monotone_cubic_interpolation
```

```python
bumps.mono.hermite(x, y, m, xt)
    Computes the cubic hermite polynomial $p(x_i)$.
```
The polynomial goes through all points \((x_i, y_i)\) with slope \(m_i\) at the point.

```python
bumps.mono.count_inflections(x, y)
```
Count the number of inflection points in a curve.

```python
bumps.mono.plot_inflections(x, y)
```
Plot inflection points in a curve.

## 4.18 names - External interface

Exported names.

In model definition scripts, rather than importing symbols one by one, you can simply perform:

```python
from bumps.names import *
```

This is bad style for library and applications but convenient for model scripts.

The following symbols are defined:

- \(np\) for the \texttt{numpy} array package
- \texttt{sys} for the \texttt{python sys} module
- \texttt{inf} for infinity
- \texttt{pmath} for parameter expressions like \(2*pmath.sin(M.theta)\)
- \texttt{Parameter} for defining parameters
- \texttt{FreeVariables} for defining shared parameters
- \texttt{Distribution} for indicating prior  probability for a model parameter
- \texttt{Curve} for defining models from functions
- \texttt{PoissonCurve} for modelling data with Poisson uncertainty
- \texttt{PDF} for fitting a probability distribution directly
- \texttt{FitProblem} for defining the fit (see \texttt{BaseFitProblem} or \texttt{MultiFitProblem} for details, depending on whether you are fitting a single model or multiple models simultaneously).

## 4.19 options - Command line options processor

<table>
<thead>
<tr>
<th>BumpsOpts</th>
<th>Option parser for \textit{bumps}.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChoiceList</td>
<td></td>
</tr>
<tr>
<td>FIT_CONFIG</td>
<td>FitConfig singleton for the common case in which only one config is needed.</td>
</tr>
<tr>
<td>FitConfig</td>
<td>Fit settings configuration object.</td>
</tr>
<tr>
<td>ParseOpts</td>
<td>Options parser.</td>
</tr>
<tr>
<td>getopt</td>
<td>Process command line options.</td>
</tr>
<tr>
<td>parse_int</td>
<td></td>
</tr>
<tr>
<td>yesno</td>
<td></td>
</tr>
</tbody>
</table>
Option parser for bumps command line

class bumps.options.BumpsOpts(args)
    Bases: bumps.options.ParseOpts

Option parser for bumps.

FLAGS = {'batch', 'chisq', 'cov', 'edit', 'err', 'i', 'keep_best', 'mpi', 'multiprocessing-fork', 
          'noshow', 'overwrite', 'preview', 'profile', 'remote', 'shake', 'simulate', 'staj', 'stepmon', 'time_model', 'worker'}

IMPLICIT_VALUES = {'entropy': 'llf', 'parallel': '0', 'resume': '-'}

MINARGS = 1

PLOTTERS = ('linear', 'log', 'residuals')

TRANSPORTS = ('amqp', 'mp', 'mpi', 'celery')

USAGE = 'Usage: bumps [options] modelfile [modelargs]

The modelfile is a Python script (i.e., a series of Python ...
-p: command is the name of a python script
 -i
 start the interactive interpreter
 -?/-h/--help
 display this help
'

VALUES = {'CR', 'F', 'Tmax', 'Tmin', 'alpha', 'burn', 'c', 'checkpoint', 'entropy', 'fit', 
          'ftol', 'init', 'm', 'nT', 'p', 'p0', 'plot', 'ploth', 'plotn', 'prio', 'prob', 'p: 
          'q: ', 'q:0', 'queue', 'res', 'resample', 'residuals', 'resynth', 'samples', 'seed', 
          'starts', 'store', 'thin', 'time', 'transport', 'trim', 'view', 'xtol'}

alpha = 0.0

checkpoint = '0'

entropy = None

fit

fit_config = <bumps.options.FitConfig object>

meshsteps = 40

noise = '5'

notify = '

parallel = '

pars = None

plot

queue = None

resume = None

resynth = '0'

seed = '

starts = '1'

store = None

time = 'inf'

transport

trim = 'true'

view = None

class bumps.options.ChoiceList(*choices)
    Bases: object

bumps.options.FIT_CONFIG = <bumps.options.FitConfig object>

FitConfig singleton for the common case in which only one config is needed. There may be other use cases, such as saving the fit config along with the rest of the state so that on resume the fit options are restored, but in that case the application will not be using the singleton.
class bumps.options.FitConfig (default='amoeba', active=['amoeba', 'de', 'dream', 'newton', 'scipy.leastsq', 'lm'])

    Bases: object

    Fit settings configuration object.

    The command line parser will define a FitConfig object which contains the fitter that was given on the command line and all its options. For embedded bumps, which does not use the bumps command line parser, a new FitConfig object can be created with its own selected options.

    Attributes

    ids = [id, id, . . . ] is a list available fitters in “preferred” order. Depending on usage, you may want to sort them, or alternatively, sort by long name with [id for _,id in sorted((v,k) for k,v in self.names]

    fitters = {id: fitclass} maps ids to fitters.

    names = {id: name}* maps ids to long names

    settings = {id: [{option, default}, . . . ]} maps ids to default settings. The order of the settings is the preferred order to present the settings to the user in a GUI dialog for example.

    values = {id: {option: value, . . . }} maps ids to the settings for each fitter. Note that in the GUI, different fitters may have there settings recorded and preserved even when not selected.

    active_ids = [id, id, . . . ] is the list of fitters to show the user in a GUI dialog for example. The other fitters should still be available from the command line.

    default_id = id is the fitter to use by default.

    selected_id = id is the fitter that was selected, either by command line or by GUI.

    selected_values = {option: value} returns the settings for the current fitter.

    selected_name = name returns the name of the selected fitter.

    selected_fitter = FitClass returns the class of the selected fitter.

    selected_fitter

    selected_name

    selected_values

    set_from_cli (opts)

        Use the BumpsOpts command line parser values to set the selected fitter and its configuration options.

class bumps.options.ParseOpts (args)

    Bases: object

    Options parser.

    Subclass should define MINARGS, FLAGS, VALUES and USAGE.

    MINARGS is the minimum number of positional arguments.

    FLAGS is a set of arguments that may be present or absent.

    VALUES is a set of arguments that take values. Value checking can be done in the setter for each argument in the set. Default values should be set in the corresponding object attribute.

    USAGE is the help string to display for option “help”.

    The constructor will invoke the command line parser, leaving the values set by the command line as attribute values. Flag options will be True or False.

    FLAGS = { }

4.19. options - Command line options processor
**IMPLICIT_VALUES** = {}

Value to use if a value flag is is present without '='. This is different from the default value if the flag is not present, which is the default value set in the calling class.

**MINARGS** = 0

**USAGE** = ''

**VALUES** = {}

bumps.options.getopts()

Process command line options.

Option values will be stored as attributes in the returned object.

bumps.options.parse_int(value)

bumps.options.yesno(value)

### 4.20 parameter - Optimization parameter definition

<table>
<thead>
<tr>
<th>Alias</th>
<th>Parameter alias.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BaseParameter</strong></td>
<td>Root of the parameter class, defining arithmetic on parameters</td>
</tr>
<tr>
<td><strong>Constant</strong></td>
<td>An unmodifiable value.</td>
</tr>
<tr>
<td><strong>Constraint</strong></td>
<td></td>
</tr>
<tr>
<td><strong>FreeVariables</strong></td>
<td>A collection of parameter sets for a group of models.</td>
</tr>
<tr>
<td><strong>Function</strong></td>
<td>Delayed function evaluator.</td>
</tr>
<tr>
<td><strong>IntegerParameter</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Operator</strong></td>
<td>Parameter operator</td>
</tr>
<tr>
<td><strong>Parameter</strong></td>
<td>A parameter is a symbolic value.</td>
</tr>
<tr>
<td><strong>ParameterSet</strong></td>
<td>A parameter that depends on the model.</td>
</tr>
<tr>
<td><strong>Reference</strong></td>
<td>Create an adaptor so that a model attribute can be treated as if it were a parameter.</td>
</tr>
<tr>
<td><strong>acosd</strong></td>
<td>Return the arc cosine (measured in in degrees) of x.</td>
</tr>
<tr>
<td><strong>arccosd</strong></td>
<td>Return the arc cosine (measured in in degrees) of x.</td>
</tr>
<tr>
<td><strong>arcsind</strong></td>
<td>Return the arc sine (measured in in degrees) of x.</td>
</tr>
<tr>
<td><strong>arctan2d</strong></td>
<td>Return the arc tangent (measured in in degrees) of y/x.</td>
</tr>
<tr>
<td><strong>arctand</strong></td>
<td>Return the arc tangent (measured in in degrees) of x.</td>
</tr>
<tr>
<td><strong>asin</strong></td>
<td>Return the arc sine (measured in in degrees) of x.</td>
</tr>
<tr>
<td><strong>atan2d</strong></td>
<td>Return the arc tangent (measured in in degrees) of y/x.</td>
</tr>
<tr>
<td><strong>atand</strong></td>
<td>Return the arc tangent (measured in in degrees) of x.</td>
</tr>
<tr>
<td><strong>boxed_function</strong></td>
<td></td>
</tr>
<tr>
<td><strong>cosd</strong></td>
<td>Return the cosine of x (measured in in degrees).</td>
</tr>
<tr>
<td><strong>current</strong></td>
<td>Return the list of fittable parameters in no paraticular order.</td>
</tr>
<tr>
<td><strong>fittable</strong></td>
<td></td>
</tr>
<tr>
<td><strong>flatten</strong></td>
<td></td>
</tr>
<tr>
<td><strong>format</strong></td>
<td>Format parameter set for printing.</td>
</tr>
<tr>
<td><strong>function</strong></td>
<td>Convert a function into a delayed evaluator.</td>
</tr>
<tr>
<td><strong>randomize</strong></td>
<td>Set random values to the parameters in the parameter set, with values chosen according to the bounds.</td>
</tr>
<tr>
<td><strong>sind</strong></td>
<td>Return the sine of x (measured in in degrees).</td>
</tr>
</tbody>
</table>

Continued on next page
Fitting parameter objects.

Parameters are a big part of the interface between the model and the fitting engine. By saving and retrieving values and ranges from the parameter, the fitting engine does not need to be aware of the structure of the model.

Users can also perform calculations with parameters, tying together different parts of the model, or different models.

```python
class bumps.parameter.Alias(obj, attr, p=None, name=None)
    Bases: object

    Parameter alias.

    Rather than modifying a model to contain a parameter slot, allow the parameter to exist outside the model. The resulting parameter will have the full parameter semantics, including the ability to replace a fixed value with a parameter expression.

    Deprecated Reference does this better.

    parameters ()
    to_dict ()
    update ()
```

```python
class bumps.parameter.BaseParameter
    Bases: object

    Root of the parameter class, defining arithmetic on parameters

        arccos (**kw)
        Return the arc cosine (measured in radians) of x.

        arccosh (**kw)
        Return the inverse hyperbolic cosine of x.

        arcsin (**kw)
        Return the arc sine (measured in radians) of x.

        arcsinh (**kw)
        Return the inverse hyperbolic sine of x.

        arctan (**kw)
        Return the arc tangent (measured in radians) of x.

        arctanh (**kw)
        Return the inverse hyperbolic tangent of x.

        bounds
        Fit bounds

        ceil (**kw)
        Return the ceiling of x as an Integral.
```

4.20. parameter - Optimization parameter definition
This is the smallest integer >= x.

\textbf{cos (**kw)}

Return the cosine of x (measured in radians).

\textbf{cosh (**kw)}

Return the hyperbolic cosine of x.

\textbf{degrees (**kw)}

Convert angle x from radians to degrees.

\textbf{dev (std, mean=None, limits=None, sigma=None, mu=None)}

Allow the parameter to vary according to a normal distribution, with deviations from the mean added to the overall cost function for the model.

If \textit{mean} is None, then it defaults to the current parameter value.

If \textit{limits} are provide, then use a truncated normal distribution.

Note: \textit{sigma} and \textit{mu} have been replaced by \textit{std} and \textit{mean}, but are left in for backward compatibility.

\textbf{discrete = False}

\textbf{exp (**kw)}

Return e raised to the power of x.

\textbf{expm1 (**kw)}

Return \text{exp}(x)-1.

This function avoids the loss of precision involved in the direct evaluation of \text{exp}(x)-1 for small x.

\textbf{fittable = False}

\textbf{fixed = True}

\textbf{floor (**kw)}

Return the floor of x as an Integral.

This is the largest integer <= x.

\textbf{format ()}

Format the parameter, value and range as a string.

\textbf{log (x[, base=\text{math.e}])}

Return the logarithm of x to the given base.

If the base not specified, returns the natural logarithm (base e) of x.

\textbf{log10 (**kw)}

Return the base 10 logarithm of x.

\textbf{log1p (**kw)}

Return the natural logarithm of 1+x (base e).

The result is computed in a way which is accurate for x near zero.

\textbf{name = None}

\textbf{nllf ()}

Return -log(P) for the current parameter value.

\textbf{parameters ()}

\textbf{pdf (dist)}

Allow the parameter to vary according to any continuous scipy.stats distribution.
pm (plus, minus=None, limits=None)
Allow the parameter to vary as value +/- delta.

pm(delta) -> [value-delta, value+delta]

pm(plus, minus) -> [value+minus, value+plus]

In the plus/minus form, one of the numbers should be plus and the other minus, but it doesn’t matter which.

If limits are provided, bound the end points of the range to lie within the limits.

The resulting range is converted to “nice” numbers.

pmp (plus, minus=None, limits=None)
Allow the parameter to vary as value +/- percent.

pmp(percent) -> [value*(1-percent/100), value*(1+percent/100)]

pmp(plus, minus) -> [value*(1+minus/100), value*(1+plus/100)]

In the plus/minus form, one of the numbers should be plus and the other minus, but it doesn’t matter which.

If limits are provided, bound the end points of the range to lie within the limits.

The resulting range is converted to “nice” numbers.

radians (**kw)
Convert angle x from degrees to radians.

range (low, high)
Allow the parameter to vary within the given range.

residual ()
Return the z score equivalent for the current parameter value.

That is, the given the value of the parameter in the underlying distribution, find the equivalent value in the standard normal. For a gaussian, this is the z score, in which you subtract the mean and divide by the standard deviation to get the number of sigmas away from the mean. For other distributions, you need to compute the cdf of value in the parameter distribution and invert it using the ppf from the standard normal distribution.

sin (**kw)
Return the sine of x (measured in radians).

sinh (**kw)
Return the hyperbolic sine of x.

soft_range (low, high, std)
Allow the parameter to vary within the given range, or with Gaussian probability, stray from the range.

sqrt (**kw)
Return the square root of x.

tan (**kw)
Return the tangent of x (measured in radians).

tanh (**kw)
Return the hyperbolic tangent of x.

to_dict ()
Return a dict representation of the object.

trunc (**kw)
Truncates the Real x to the nearest Integral toward 0.

Uses the __trunc__ magic method.
valid()
    Return true if the parameter is within the valid range.

value = None
class bumps.parameter.Constant(value, name=None)
    Bases: bumps.parameter.BaseParameter
    An unmodifiable value.
arccos(**kw)
    Return the arc cosine (measured in radians) of x.
arccosh(**kw)
    Return the inverse hyperbolic cosine of x.
arcsin(**kw)
    Return the arc sine (measured in radians) of x.
arcsinh(**kw)
    Return the inverse hyperbolic sine of x.
arctan(**kw)
    Return the arc tangent (measured in radians) of x.
arctanh(**kw)
    Return the inverse hyperbolic tangent of x.
bounds
    Fit bounds
ceil(**kw)
    Return the ceiling of x as an Integral.
    This is the smallest integer >= x.
cos(**kw)
    Return the cosine of x (measured in radians).
cosh(**kw)
    Return the hyperbolic cosine of x.
degrees(**kw)
    Convert angle x from radians to degrees.
dev(std, mean=None, limits=None, sigma=None, mu=None)
    Allow the parameter to vary according to a normal distribution, with deviations from the mean added to
    the overall cost function for the model.
    If mean is None, then it defaults to the current parameter value.
    If limits are provide, then use a truncated normal distribution.
    Note: sigma and mu have been replaced by std and mean, but are left in for backward compatibility.
discrete = False
exp(**kw)
    Return e raised to the power of x.
expm1(**kw)
    Return exp(x)-1.
    This function avoids the loss of precision involved in the direct evaluation of exp(x)-1 for small x.
fittable = False
fixed = True

floor(**kw)
    Return the floor of x as an Integral.
    This is the largest integer <= x.

format()
    Format the parameter, value and range as a string.

log(x[, base=math.e])
    Return the logarithm of x to the given base.
    If the base not specified, returns the natural logarithm (base e) of x.

log10(**kw)
    Return the base 10 logarithm of x.

log1p(**kw)
    Return the natural logarithm of 1+x (base e).
    The result is computed in a way which is accurate for x near zero.

name = None

nllf()
    Return -log(P) for the current parameter value.

parameters()

pdf(dist)
    Allow the parameter to vary according to any continuous scipy.stats distribution.

pm(plus, minus=None, limits=None)
    Allow the parameter to vary as value +/- delta.
    pm(delta) -> [value-delta, value+delta]
    pm(plus, minus) -> [value+minus, value+plus]
    In the plus/minus form, one of the numbers should be plus and the other minus, but it doesn’t matter which.
    If limits are provided, bound the end points of the range to lie within the limits.
    The resulting range is converted to “nice” numbers.

pmp(plus, minus=None, limits=None)
    Allow the parameter to vary as value +/- percent.
    pmp(percent) -> [value*(1-percent/100), value*(1+percent/100)]
    pmp(plus, minus) -> [value*(1+minus/100), value*(1+plus/100)]
    In the plus/minus form, one of the numbers should be plus and the other minus, but it doesn’t matter which.
    If limits are provided, bound the end points of the range to lie within the limits.
    The resulting range is converted to “nice” numbers.

radians(**kw)
    Convert angle x from degrees to radians.

range(low, high)
    Allow the parameter to vary within the given range.
residual()
    Return the z score equivalent for the current parameter value.

    That is, the given the value of the parameter in the underlying distribution, find the equivalent value in
    the standard normal. For a gaussian, this is the z score, in which you subtract the mean and divide by the
    standard deviation to get the number of sigmas away from the mean. For other distributions, you need to
    compute the cdf of value in the parameter distribution and invert it using the ppf from the standard normal
    distribution.

sin(**kw)
    Return the sine of x (measured in radians).

sinh(**kw)
    Return the hyperbolic sine of x.

soft_range(low, high, std)
    Allow the parameter to vary within the given range, or with Gaussian probability, stray from the range.

sqrt(**kw)
    Return the square root of x.

tan(**kw)
    Return the tangent of x (measured in radians).

tanh(**kw)
    Return the hyperbolic tangent of x.
to_dict()
    Return a dict representation of the object.

trunc(**kw)
    Truncates the Real x to the nearest Integral toward 0.

    Uses the __trunc__ magic method.

valid()
    Return true if the parameter is within the valid range.

value

class bumps.parameter.Constraint (a, b, op_name, op_str=")
    Bases: object

class bumpsparameter.FreeVariables (names= None, **kw)
    Bases: object

    A collection of parameter sets for a group of models.

    names is the set of model names.

    The parameters themselves are specified as key=value pairs, with key being the attribute name which is used to
    retrieve the parameter set and value being a Parameter containing the parameter that is shared between the
    models.

    In order to evaluate the log likelihood of all models simultaneously, the fitting program will need to call
    set_model with the model index for each model in turn in order to substitute the values from the free vari-
    ables into the model. This allows us to share a common sample across multiple data sets, with each dataset
    having its own values for some of the sample parameters. The alternative is to copy the entire sample structure,
    sharing references to common parameters and creating new parameters for each model for the free parameters. Setting up these copies was inconvenient.

get_model (i)
    Get the parameters for model i as {reference: substitution}
parameters()
Return the set of free variables for all the models.

set_model(i)
Set the reference parameters for model i.

to_dict()

class bumps.parameter.Function(op, *args, **kw)
Bases: bumps.parameter.BaseParameter
Delayed function evaluator.

f.value evaluates the function with the values of the parameter arguments at the time f.value is referenced rather than when the function was invoked.

arccos(**kw)
Return the arc cosine (measured in radians) of x.

arccosh(**kw)
Return the inverse hyperbolic cosine of x.

arcsin(**kw)
Return the arc sine (measured in radians) of x.

arcsinh(**kw)
Return the inverse hyperbolic sine of x.

arctan(**kw)
Return the arc tangent (measured in radians) of x.

arctanh(**kw)
Return the inverse hyperbolic tangent of x.

args

bounds
Fit bounds

ceil(**kw)
Return the ceiling of x as an Integral.
This is the smallest integer >= x.

cos(**kw)
Return the cosine of x (measured in radians).

cosh(**kw)
Return the hyperbolic cosine of x.

degrees(**kw)
Convert angle x from radians to degrees.

dev(std, mean=None, limits=None, sigma=None, mu=None)
Allow the parameter to vary according to a normal distribution, with deviations from the mean added to the overall cost function for the model.

If mean is None, then it defaults to the current parameter value.

If limits are provide, then use a truncated normal distribution.

Note: sigma and mu have been replaced by std and mean, but are left in for backward compatibility.

 discrete = False
\textbf{exp(**kw)}

Return \(e\) raised to the power of \(x\).

\textbf{expm1(**kw)}

Return \(\exp(x)-1\).

This function avoids the loss of precision involved in the direct evaluation of \(\exp(x)-1\) for small \(x\).

\textbf{fittable = False}

\textbf{fixed = True}

\textbf{floor(**kw)}

Return the floor of \(x\) as an Integral.

This is the largest integer \(<= x\).

\textbf{format()}

Format the parameter, value and range as a string.

\textbf{kw}

\textbf{log(x[, base=math.e])}

Return the logarithm of \(x\) to the given base.

If the base not specified, returns the natural logarithm (base \(e\)) of \(x\).

\textbf{log10(**kw)}

Return the base 10 logarithm of \(x\).

\textbf{log1p(**kw)}

Return the natural logarithm of \(1+x\) (base \(e\)).

The result is computed in a way which is accurate for \(x\) near zero.

\textbf{name = None}

\textbf{nllf()}

Return \(-\log(P)\) for the current parameter value.

\textbf{op}

\textbf{parameters()}

\textbf{pdf(dist)}

Allow the parameter to vary according to any continuous scipy.stats distribution.

\textbf{pm(plus, minus=None, limits=None)}

Allow the parameter to vary as value +/- delta.

\(\text{pm(delta)} \rightarrow [\text{value-delta, value+delta}]\)

\(\text{pm(plus, minus)} \rightarrow [\text{value+minus, value+plus}]\)

In the \text{plus/minus} form, one of the numbers should be plus and the other minus, but it doesn’t matter which.

If \textit{limits} are provided, bound the end points of the range to lie within the limits.

The resulting range is converted to “nice” numbers.

\textbf{pmp(plus, minus=None, limits=None)}

Allow the parameter to vary as value +/- percent.

\(\text{pmp(percent)} \rightarrow [\text{value*(1-percent/100), value*(1+percent/100)}]\)

\(\text{pmp(plus, minus)} \rightarrow [\text{value*(1+minus/100), value*(1+plus/100)}]\)

In the \text{plus/minus} form, one of the numbers should be plus and the other minus, but it doesn’t matter which.
If limits are provided, bound the end points of the range to lie within the limits.

The resulting range is converted to “nice” numbers.

**radians (**kw\)**

Convert angle x from degrees to radians.

**range (low, high)**

Allow the parameter to vary within the given range.

**residual ()**

Return the z score equivalent for the current parameter value.

That is, the given the value of the parameter in the underlying distribution, find the equivalent value in
the standard normal. For a gaussian, this is the z score, in which you subtract the mean and divide by the
standard deviation to get the number of sigmas away from the mean. For other distributions, you need to
compute the cdf of value in the parameter distribution and invert it using the ppf from the standard normal
distribution.

**sin (**kw\)**

Return the sine of x (measured in radians).

**sinh (**kw\)**

Return the hyperbolic sine of x.

**soft_range (low, high, std)**

Allow the parameter to vary within the given range, or with Gaussian probability, stray from the range.

**sqrt (**kw\)**

Return the square root of x.

**tan (**kw\)**

Return the tangent of x (measured in radians).

**tanh (**kw\)**

Return the hyperbolic tangent of x.

**to_dict ()**

Return a dict representation of the object.

**trunc (**kw\)**

Truncates the Real x to the nearest Integral toward 0.

Uses the __trunc__ magic method.

**valid ()**

Return true if the parameter is within the valid range.

**value**

```python
class bumps.parameter.IntegerParameter (value=None, bounds=None, fixed=None, name=None, **kw)
```

Bases: *bumps.parameter.Parameter*

**arccos (**kw\)**

Return the arc cosine (measured in radians) of x.

**arccosh (**kw\)**

Return the inverse hyperbolic cosine of x.

**arcsin (**kw\)**

Return the arc sine (measured in radians) of x.

**arcsinh (**kw\)**

Return the inverse hyperbolic sine of x.
arctan (**kw)
   Return the arc tangent (measured in radians) of x.

arctanh (**kw)
   Return the inverse hyperbolic tangent of x.

bounds
   Fit bounds

ceil (**kw)
   Return the ceiling of x as an Integral.
   This is the smallest integer >= x.

clip_set (value)
   Set a new value for the parameter, clipping it to the bounds.

cos (**kw)
   Return the cosine of x (measured in radians).

cosh (**kw)
   Return the hyperbolic cosine of x.

classmethod default (value, **kw)
   Create a new parameter with the value and kw attributes, or return the existing parameter if value is already a parameter.

   The attributes are the same as those for Parameter, or whatever subclass cls of Parameter is being created.

degrees (**kw)
   Convert angle x from radians to degrees.

dev (std, mean=None, limits=None, sigma=None, mu=None)
   Allow the parameter to vary according to a normal distribution, with deviations from the mean added to the overall cost function for the model.

   If mean is None, then it defaults to the current parameter value.

   If limits are provide, then use a truncated normal distribution.

   Note: sigma and mu have been replaced by std and mean, but are left in for backward compatibility.

discrete = True

exp (**kw)
   Return e raised to the power of x.

expm1 (**kw)
   Return exp(x)-1.

   This function avoids the loss of precision involved in the direct evaluation of exp(x)-1 for small x.

feasible ()
   Value is within the limits defined by the model

fittable = True

fixed = True

floor (**kw)
   Return the floor of x as an Integral.

   This is the largest integer <= x.

format ()
   Format the parameter, value and range as a string.
\texttt{log(x[, \text{base}=\text{math.e}])}

Return the logarithm of \(x\) to the given base.

If the base not specified, returns the natural logarithm (base e) of \(x\).

\texttt{log10(**kw)}

Return the base 10 logarithm of \(x\).

\texttt{log1p(**kw)}

Return the natural logarithm of \(1+x\) (base e).

The result is computed in a way which is accurate for \(x\) near zero.

\texttt{name = None}

\texttt{nllf()}  
Return \(-\log(P)\) for the current parameter value.

\texttt{parameters()}  

\texttt{pdf(dist)}  
Allow the parameter to vary according to any continuous scipy.stats distribution.

\texttt{pm(plus, minus=None, limits=None)}  
Allow the parameter to vary as value +/- delta.

\begin{itemize}
  \item \texttt{pm(delta) -> [value-delta, value+delta]}
  \item \texttt{pm(plus, minus) -> [value+minus, value+plus]}
\end{itemize}

In the \texttt{plus/minus} form, one of the numbers should be plus and the other minus, but it doesn’t matter which.

If \texttt{limits} are provided, bound the end points of the range to lie within the limits.

The resulting range is converted to “nice” numbers.

\texttt{pmp(plus, minus=None, limits=None)}  
Allow the parameter to vary as value +/- percent.

\begin{itemize}
  \item \texttt{pmp(percent) -> [value*(1-percent/100), value*(1+percent/100)]}
  \item \texttt{pmp(plus, minus) -> [value*(1+minus/100), value*(1+plus/100)]}
\end{itemize}

In the \texttt{plus/minus} form, one of the numbers should be plus and the other minus, but it doesn’t matter which.

If \texttt{limits} are provided, bound the end points of the range to lie within the limits.

The resulting range is converted to “nice” numbers.

\texttt{radians(**kw)}  
Convert angle \(x\) from degrees to radians.

\texttt{randomize(rng=None)}  
Set a random value for the parameter.

\texttt{range(low, high)}  
Allow the parameter to vary within the given range.

\texttt{residual()}  
Return the z score equivalent for the current parameter value.

That is, the given the value of the parameter in the underlying distribution, find the equivalent value in the standard normal. For a gaussian, this is the z score, in which you subtract the mean and divide by the standard deviation to get the number of sigmas away from the mean. For other distributions, you need to compute the cdf of value in the parameter distribution and invert it using the ppf from the standard normal distribution.
**set**(*value*)
Set a new value for the parameter, ignoring the bounds.

**sin**(**kw**)
Return the sine of x (measured in radians).

**sinh**(**kw**)
Return the hyperbolic sine of x.

**soft_range**(*low, high, std*)
Allow the parameter to vary within the given range, or with Gaussian probability, stray from the range.

**sqrt**(**kw**)
Return the square root of x.

**tan**(**kw**)
Return the tangent of x (measured in radians).

**tanh**(**kw**)
Return the hyperbolic tangent of x.

**to_dict**()
Return a dict representation of the object.

**trunc**(**kw**)
Truncates the Real x to the nearest Integral toward 0.
Uses the __trunc__ magic method.

**valid**()
Return true if the parameter is within the valid range.

**class** bumps.parameter.Operator(*a, b, op_name, op_str*)
Parameter operator

**arccos**(**kw**)
Return the arc cosine (measured in radians) of x.

**arccosh**(**kw**)
Return the inverse hyperbolic cosine of x.

**arcsin**(**kw**)
Return the arc sine (measured in radians) of x.

**arcsinh**(**kw**)
Return the inverse hyperbolic sine of x.

**arctan**(**kw**)
Return the arc tangent (measured in radians) of x.

**arctanh**(**kw**)
Return the inverse hyperbolic tangent of x.

**bounds**
Fit bounds

**ceil**(**kw**)
Return the ceiling of x as an Integral.
This is the smallest integer >= x.
cos (**kw)**

Return the cosine of x (measured in radians).

**cosh (**kw)**

Return the hyperbolic cosine of x.

**degrees (**kw)**

Convert angle x from radians to degrees.

**dev (std, mean=None, limits=None, sigma=None, mu=None)**

Allow the parameter to vary according to a normal distribution, with deviations from the mean added to the overall cost function for the model.

If mean is None, then it defaults to the current parameter value.

If limits are provide, then use a truncated normal distribution.

Note: sigma and mu have been replaced by std and mean, but are left in for backward compatibility.

**discrete = False**

**dvalue**

**exp (**kw)**

Return e raised to the power of x.

**expm1 (**kw)**

Return exp(x)-1.

This function avoids the loss of precision involved in the direct evaluation of exp(x)-1 for small x.

**fittable = False**

**fixed = True**

**floor (**kw)**

Return the floor of x as an Integral.

This is the largest integer <= x.

**format ()**

Format the parameter, value and range as a string.

**log (x[, base=math.e])**

Return the logarithm of x to the given base.

If the base not specified, returns the natural logarithm (base e) of x.

**log10 (**kw)**

Return the base 10 logarithm of x.

**log1p (**kw)**

Return the natural logarithm of 1+x (base e).

The result is computed in a way which is accurate for x near zero.

**name = None**

**nllf ()**

Return -log(P) for the current parameter value.

**parameters ()**

**pdf (dist)**

Allow the parameter to vary according to any continuous scipy.stats distribution.
pm (plus, minus=None, limits=None)
   Allow the parameter to vary as value +/- delta.
   pm(delta) -> [value-delta, value+delta]
   pm(plus, minus) -> [value+minus, value+plus]
   In the plus/minus form, one of the numbers should be plus and the other minus, but it doesn’t matter which.
   If limits are provided, bound the end points of the range to lie within the limits.
   The resulting range is converted to “nice” numbers.

pmp (plus, minus=None, limits=None)
   Allow the parameter to vary as value +/- percent.
   pmp(percent) -> [value*(1-percent/100), value*(1+percent/100)]
   pmp(plus, minus) -> [value*(1+minus/100), value*(1+plus/100)]
   In the plus/minus form, one of the numbers should be plus and the other minus, but it doesn’t matter which.
   If limits are provided, bound the end points of the range to lie within the limits.
   The resulting range is converted to “nice” numbers.

radians (**kw)
   Convert angle x from degrees to radians.

range (low, high)
   Allow the parameter to vary within the given range.

residual ()
   Return the z score equivalent for the current parameter value.
   That is, the given the value of the parameter in the underlying distribution, find the equivalent value in the standard normal. For a gaussian, this is the z score, in which you subtract the mean and divide by the standard deviation to get the number of sigmas away from the mean. For other distributions, you need to compute the cdf of value in the parameter distribution and invert it using the ppf from the standard normal distribution.

sin (**kw)
   Return the sine of x (measured in radians).

sinh (**kw)
   Return the hyperbolic sine of x.

soft_range (low, high, std)
   Allow the parameter to vary within the given range, or with Gaussian probability, stray from the range.

sqrt (**kw)
   Return the square root of x.

tan (**kw)
   Return the tangent of x (measured in radians).

tanh (**kw)
   Return the hyperbolic tangent of x.

to_dict ()
   Return a dict representation of the object.

trunc (**kw)
   Truncates the Real x to the nearest Integral toward 0.
   Uses the __trunc__ magic method.
valid()
Return true if the parameter is within the valid range.

class bumps.parameter.Parameter (value=None, bounds=None, fixed=None, name=None, **kw)
Bases: bumps.parameter.BaseParameter
A parameter is a symbolic value.
It can be fixed or it can vary within bounds.

p = Parameter(3).pmp(10) # 3 +/- 10% p = Parameter(3).pmp(-5,10) # 3 in [2.85,3.3] rounded to 2 digits p = Parameter(3).pm(2) # 3 +/- 2 p = Parameter(3).pm(-1,2) # 3 in [2,5] p = Parameter(3).range(0,5) # 3 in [0,5]

It has hard limits on the possible values, and a range that should live within those hard limits. The value should lie within the range for it to be valid. Some algorithms may drive the value outside the range in order to satisfy soft It has a value which should lie within the range.

Other properties can decorate the parameter, such as tip for tool tip and units for units.

arccos (**kw)
Return the arc cosine (measured in radians) of x.
arccosh (**kw)
Return the inverse hyperbolic cosine of x.
arcsin (**kw)
Return the arc sine (measured in radians) of x.
arcsinh (**kw)
Return the inverse hyperbolic sine of x.
arctan (**kw)
Return the arc tangent (measured in radians) of x.
arctanh (**kw)
Return the inverse hyperbolic tangent of x.
bounds
Fit bounds
ceil (**kw)
Return the ceiling of x as an Integral.
This is the smallest integer >= x.
clip_set (value)
Set a new value for the parameter, clipping it to the bounds.
cos (**kw)
Return the cosine of x (measured in radians).
cosh (**kw)
Return the hyperbolic cosine of x.
classmethod default (value, **kw)
Create a new parameter with the value and kw attributes, or return the existing parameter if value is already a parameter.
The attributes are the same as those for Parameter, or whatever subclass cls of Parameter is being created.
degrees (**kw)
Convert angle x from radians to degrees.
dev \( (std, mean=None, limits=None, sigma=None, mu=None) \)
Allow the parameter to vary according to a normal distribution, with deviations from the mean added to the overall cost function for the model.

If \( mean \) is None, then it defaults to the current parameter value.

If \( limits \) are provide, then use a truncated normal distribution.

Note: \( sigma \) and \( mu \) have been replaced by \( std \) and \( mean \), but are left in for backward compatibility.

discrete = False

exp (**kw)
Return \( e \) raised to the power of \( x \).

expm1 (**kw)
Return \( \exp(x)-1 \).

This function avoids the loss of precision involved in the direct evaluation of \( \exp(x)-1 \) for small \( x \).

feasible()
Value is within the limits defined by the model

fittable = True

fixed = True

floor (**kw)
Return the floor of \( x \) as an Integral.

This is the largest integer \( \leq x \).

format()
Format the parameter, value and range as a string.

log \([x, base=math.e]\)  
Return the logarithm of \( x \) to the given base.

If the base not specified, returns the natural logarithm (base \( e \)) of \( x \).

log10 (**kw)
Return the base 10 logarithm of \( x \).

log1p (**kw)
Return the natural logarithm of \( 1+x \) (base \( e \)).

The result is computed in a way which is accurate for \( x \) near zero.

name = None

nllf()
Return \(-\log(P)\) for the current parameter value.

parameters()

pdf (dist)
Allow the parameter to vary according to any continuous scipy.stats distribution.

pm (plus, minus=None, limits=None)
Allow the parameter to vary as value +/- delta.

\[ \text{pm}(\text{delta}) \rightarrow [\text{value}-\text{delta}, \text{value}+\text{delta}] \]

\[ \text{pm}(\text{plus}, \text{minus}) \rightarrow [\text{value}+\text{minus}, \text{value}+\text{plus}] \]

In the plus/minus form, one of the numbers should be plus and the other minus, but it doesn’t matter which.
If \textit{limits} are provided, bound the end points of the range to lie within the limits.

The resulting range is converted to “nice” numbers.

\textbf{pmp (\texttt{plus}, \texttt{minus=None}, \texttt{limits=None})}

Allow the parameter to vary as value +/- percent.

\texttt{pmp(\texttt{percent})} \rightarrow [\texttt{value*}(1-\texttt{percent}/100), \texttt{value*}(1+\texttt{percent}/100)]

\texttt{pmp(\texttt{plus, minus})} \rightarrow [\texttt{value*}(1+\texttt{minus}/100), \texttt{value*}(1+\texttt{plus}/100)]

In the \textit{plus/minus} form, one of the numbers should be plus and the other minus, but it doesn’t matter which.

If \textit{limits} are provided, bound the end points of the range to lie within the limits.

The resulting range is converted to “nice” numbers.

\textbf{radians (**kw)}

Convert angle \texttt{x} from degrees to radians.

\textbf{randomize (\texttt{rng=None})}

Set a random value for the parameter.

\textbf{range (\texttt{low}, \texttt{high})}

Allow the parameter to vary within the given range.

\textbf{residual ()}

Return the \textit{z} score equivalent for the current parameter value.

That is, the given the value of the parameter in the underlying distribution, find the equivalent value in the standard normal. For a gaussian, this is the \textit{z} score, in which you subtract the mean and divide by the standard deviation to get the number of sigmas away from the mean. For other distributions, you need to compute the cdf of value in the parameter distribution and invert it using the ppf from the standard normal distribution.

\textbf{set (\texttt{value})}

Set a new value for the parameter, ignoring the bounds.

\textbf{sin (**kw)}

Return the sine of \texttt{x} (measured in radians).

\textbf{sinh (**kw)}

Return the hyperbolic sine of \texttt{x}.

\textbf{soft_range (\texttt{low, high, std})}

Allow the parameter to vary within the given range, or with Gaussian probability, stray from the range.

\textbf{sqrt (**kw)}

Return the square root of \texttt{x}.

\textbf{tan (**kw)}

Return the tangent of \texttt{x} (measured in radians).

\textbf{tanh (**kw)}

Return the hyperbolic tangent of \texttt{x}.

\textbf{to_dict ()}

Return a dict representation of the object.

\textbf{trunc (**kw)}

Truncates the Real \texttt{x} to the nearest Integral toward 0.

Uses the \texttt{__trunc__} magic method.
valid()
    Return true if the parameter is within the valid range.

t = None

class bumps.parameter.ParameterSet(object, names=None)

A parameter that depends on the model.

get_model(index)
    Get the reference and underlying model parameter for the nth model.

pm(*args, **kw)
    Like Parameter.pm(), but applied to all models.

pmp(*args, **kw)
    Like Parameter.pmp(), but applied to all models.

range(*args, **kw)
    Like Parameter.range(), but applied to all models.

set_model(index)
    Set the underlying model parameter to the value of the nth model.

to_dict()

values

class bumps.parameter.Reference(obj, attr, **kw)

Bases: bumps.parameter.Parameter

Create an adaptor so that a model attribute can be treated as if it were a parameter. This allows only direct
access, wherein the storage for the parameter value is provided by the underlying model.

Indirect access, wherein the storage is provided by the parameter, cannot be supported since the parameter has
no way to detect that the model is asking for the value of the attribute. This means that model attributes cannot
be assigned to parameter expressions without some trigger to update the values of the attributes in the model.

arccos(**kw)
    Return the arc cosine (measured in radians) of x.

arccosh(**kw)
    Return the inverse hyperbolic cosine of x.

arcsin(**kw)
    Return the arc sine (measured in radians) of x.

arcsinh(**kw)
    Return the inverse hyperbolic sine of x.

arctan(**kw)
    Return the arc tangent (measured in radians) of x.

arctanh(**kw)
    Return the inverse hyperbolic tangent of x.

bounds
    Fit bounds

to_dict()
    Return the ceiling of x as an Integral.
    This is the smallest integer >= x.
clip_set (value)
Set a new value for the parameter, clipping it to the bounds.

cos (**kw)
Return the cosine of x (measured in radians).

cosh (**kw)
Return the hyperbolic cosine of x.

classmethod default (value, **kw)
Create a new parameter with the value and kw attributes, or return the existing parameter if value is already a parameter.

The attributes are the same as those for Parameter, or whatever subclass cls of Parameter is being created.

degrees (**kw)
Convert angle x from radians to degrees.

dev (std, mean=None, limits=None, sigma=None, mu=None)
Allow the parameter to vary according to a normal distribution, with deviations from the mean added to the overall cost function for the model.

If mean is None, then it defaults to the current parameter value.

If limits are provide, then use a truncated normal distribution.

Note: sigma and mu have been replaced by std and mean, but are left in for backward compatibility.

discrete = False

exp (**kw)
Return e raised to the power of x.

expm1 (**kw)
Return exp(x)-1.

This function avoids the loss of precision involved in the direct evaluation of exp(x)-1 for small x.

feasible ()
Value is within the limits defined by the model

fittable = True

fixed = True

floor (**kw)
Return the floor of x as an Integral.

This is the largest integer <= x.

format ()
Format the parameter, value and range as a string.

log (x[, base=math.e])
Return the logarithm of x to the given base.

If the base not specified, returns the natural logarithm (base e) of x.

log10 (**kw)
Return the base 10 logarithm of x.

log1p (**kw)
Return the natural logarithm of 1+x (base e).

The result is computed in a way which is accurate for x near zero.
**name** = None

**nllf**()
Return -log(P) for the current parameter value.

**parameters**()

**pdf**(*dist*)
Allow the parameter to vary according to any continuous scipy.stats distribution.

**pm**(*plus*, *minus*=None, *limits*=None)
Allow the parameter to vary as value +/- delta.

- **pm(delta)** -> [value-delta, value+delta]
- **pm(plus, minus)** -> [value+minus, value+plus]

In the **plus/minus** form, one of the numbers should be plus and the other minus, but it doesn’t matter which.
If **limits** are provided, bound the end points of the range to lie within the limits.

The resulting range is converted to “nice” numbers.

**pmp**(*plus*, *minus*=None, *limits*=None)
Allow the parameter to vary as value +/- percent.

- **pmp(percent)** -> [value*(1-percent/100), value*(1+percent/100)]
- **pmp(plus, minus)** -> [value*(1+minus/100), value*(1+plus/100)]

In the **plus/minus** form, one of the numbers should be plus and the other minus, but it doesn’t matter which.
If **limits** are provided, bound the end points of the range to lie within the limits.

The resulting range is converted to “nice” numbers.

**radians**(**kw**)  
Convert angle x from degrees to radians.

**randomize**(*rng*=None)
Set a random value for the parameter.

**range**(*low*, *high*)
Allow the parameter to vary within the given range.

**residual**()
Return the z score equivalent for the current parameter value.

That is, the given the value of the parameter in the underlying distribution, find the equivalent value in the standard normal. For a gaussian, this is the z score, in which you subtract the mean and divide by the standard deviation to get the number of sigmas away from the mean. For other distributions, you need to compute the cdf of value in the parameter distribution and invert it using the ppf from the standard normal distribution.

**set**(*value*)
Set a new value for the parameter, ignoring the bounds.

**sin**(**kw**)  
Return the sine of x (measured in radians).

**sinh**(**kw**)  
Return the hyperbolic sine of x.

**soft_range**(*low*, *high*, *std*)  
Allow the parameter to vary within the given range, or with Gaussian probability, stray from the range.
**sqrt(**kw**)
Return the square root of x.

**tan(**kw**)
Return the tangent of x (measured in radians).

**tanh(**kw**)
Return the hyperbolic tangent of x.

**to_dict()
Return a dict representation of the object.

**trunc(**kw**)
Truncates the Real x to the nearest Integral toward 0.
Uses the __trunc__ magic method.

**valid()
Return true if the parameter is within the valid range.

**value
bumps.parameter.acosd(v)
Return the arc cosine (measured in in degrees) of x.

bumps.parameter.arccosd(v)
Return the arc cosine (measured in in degrees) of x.

bumps.parameter.arcsind(v)
Return the arc sine (measured in in degrees) of x.

bumps.parameter.arctan2d(dy, dx)
Return the arc tangent (measured in in degrees) of y/x. Unlike atan(y/x), the signs of both x and y are considered.

bumps.parameter.arctand(v)
Return the arc tangent (measured in in degrees) of x.

bumps.parameter.asind(v)
Return the arc sine (measured in in degrees) of x.

bumps.parameter.atan2d(dy, dx)
Return the arc tangent (measured in in degrees) of y/x. Unlike atan(y/x), the signs of both x and y are considered.

bumps.parameter.atand(v)
Return the arc tangent (measured in in degrees) of x.

bumps.parameter.boxed_function(f)

bumps.parameter.cosd(v)
Return the cosine of x (measured in in degrees).

bumps.parameter.current(s)

bumps.parameter.fittable(s)
Return the list of fittable parameters in no paraticular order.
Note that some fittable parameters may be fixed during the fit.

bumps.parameter.flatten(s)

bumps.parameter.format(p, indent=0, freevars={}, field=None)
Format parameter set for printing.
Note that this only says how the parameters are arranged, not how they relate to each other.

4.20. parameter - Optimization parameter definition
bumps.parameter.function(op)
   Convert a function into a delayed evaluator.
   The value of the function is computed from the values of the parameters at the time that the function value is requested rather than when the function is created.

bumps.parameter.randomize(s)
   Set random values to the parameters in the parameter set, with values chosen according to the bounds.

bumps.parameter.sind(v)
   Return the sine of x (measured in in degrees).

bumps.parameter.substitute(a)
   Return structure a with values substituted for all parameters.
   The function traverses lists, tuples and dicts recursively. Things which are not parameters are returned directly.

bumps.parameter.summarize(pars, sorted=False)
   Return a stylized list of parameter names and values with range bars suitable for printing.
   If sorted, then print the parameters sorted alphabetically by name.

bumps.parameter.tand(v)
   Return the tangent of x (measured in in degrees).

bumps.parameter.test_operator()

bumps.parameter.to_dict(p)

bumps.parameter.unique(s)
   Return the unique set of parameters
   The ordering is stable. The same parameters/dependencies will always return the same ordering, with the first occurrence first.

bumps.parameter.varying(s)
   Return the list of fitted parameters in the model.
   This is the set of parameters that will vary during the fit.

### 4.21 partemp - Parallel tempering optimizer

<table>
<thead>
<tr>
<th>parallel_tempering</th>
<th>Perform a MCMC walk using multiple temperatures in parallel.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Parallel tempering for continuous function optimization and uncertainty analysis.</td>
</tr>
<tr>
<td></td>
<td>The program performs Markov chain Monte Carlo exploration of a probability density function using a combination of random and differential evolution updates.</td>
</tr>
<tr>
<td></td>
<td>bumps.partemp.parallel_tempering(nllf, p, bounds, T=None, steps=1000, CR=0.9, burn=1000, monitor=&lt;function every_ten&gt;, logfile=None)</td>
</tr>
<tr>
<td></td>
<td>Perform a MCMC walk using multiple temperatures in parallel.</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td></td>
</tr>
<tr>
<td>nllf</td>
<td>[function(vector) -&gt; float] Negative log likelihood function to be minimized. $\chi^2/2$ is a good choice for curve fitting with no prior restraints on the possible input parameters.</td>
</tr>
<tr>
<td>p</td>
<td>[vector] Initial value</td>
</tr>
</tbody>
</table>
bounds  [vector, vector] Box constraints on the parameter values. No support for indefinite or semi-definite programming at present

T  [vector | 0 < T[0] < T[1] < ...] Temperature vector. Something like logspace(-1,1,10) will give you 10 logarithmically spaced temperatures between 0.1 and 10. The maximum temperature T[-1] determines the size of the barriers that can be easily jumped. Note that the number of temperature values limits the amount of parallelism available in the algorithm, so it may gather statistics more quickly, though it will not necessarily converge any faster.

steps = 1000  [int] Length of the accumulation vector. The returned history will store this many values for each temperature. These values can be used in a weighted histogram to determine parameter uncertainty.

burn = 1000  [int | [0,inf)] Number of iterations to perform in addition to steps. Only the last steps points will be preserved for each temperature. Since the value should be in the same order as steps to be sure that the full history is acquired.

CR = 0.9  [float | [0,1]] Cross-over ratio. This is the differential evolution crossover ratio to use when computing step size and direction. Use a small value to step through the dimensions one at a time, or a large value to step through all at once.

monitor = every_ten  [function(int step, vector x, float fx) -> None] Function to called at every iteration with the step number the best point and the best value.

logfile = None  [string] Name of the file which will log the history of every accepted step. Note that this includes all of the burn steps, so it can get very large.

Returns

history  [History] Structure containing best, best_point and buffer. best is the best nllf value seen and best_point is the parameter vector which yielded best. The list buffer contains lists of tuples (step, temperature, nllf, x) for each temperature.

4.22 pdfwrapper - Model a probability density function

| DirectProblem | Build model from negative log likelihood function f(p). |
| PDF | Build a model from a function. |
| VectorPDF | Build a model from a function. |

Build a bumps model from a function.

The PDF class uses introspection to convert a negative log likelihood function nllf(m1,m2,...) into a bumps.fitproblem.Fitness class that has fittable parameters m1, m2, ....

There is no attempt to manage data or uncertainties, except that an additional plot function can be provided to display the current value of the function in whatever way is meaningful.

The note regarding user defined functions in bumps.curve apply here as well.

class bumps.pdfwrapper.DirectProblem (f, p0, bounds=None, dof=1, labels=None, plot=None)
   Bases: object
      Build model from negative log likelihood function f(p).
      Vector p of length n defines the initial value.
      bounds defines limiting values for p as [(p1_low, p1_high), (p2_low, p2_high), ...]. If all parameters are have the same bounds, use bounds=np.tile([low,high],[n,1]).
Unlike PDF, no parameter objects are defined for the elements of \( p \), so all are fitting parameters.

bounds()
chisq()
chisq_str()
getp()

has_residuals = False

labels()
model_parameters()
model_reset()
model_update()
nllf (pvec=None)

plot (p=None, fignum=None, figfile=None, view=None)

Plot the model to the current figure. You only get one figure, but you can make it as complex as you want. This will be saved as a png on the server, and composed onto a results web page.

randomize (n=None)

setp (p)

show()

summarize()

class bumps.pdfwrapper.PDF (fn, name='', plot=None, dof=1, **kw)

Build a model from a function.

This model can be fitted with any of the bumps optimizers.

\( fn \) is a function that returns the negative log likelihood of seeing its input parameters.

The fittable parameters are derived from the parameter names in the function definition, with \( name \) prepended to each parameter.

The optional \( plot \) function takes the same arguments as \( fn \), with an additional \( view \) argument which may be set from the bumps command line. If provide, it should provide a visual indication of the function value and uncertainty on the current matplotlib.pyplot figure.

Additional keyword arguments are treated as the initial values for the parameters, or initial ranges if par=(min,max). Otherwise, the default is taken from the function definition (if the function uses par=value to define the parameter) or is set to zero if no default is given in the function.

chisq()

chisq_str()

has_residuals = False

nllf()

Call self as a function.

numpoints()

Return the number of data points.
parameters()
return the parameters in the model.
model parameters are a hierarchical structure of lists and dictionaries.

plot (view=None)
Plot the model to the current figure. You only get one figure, but you can make it as complex as you want. This will be saved as a png on the server, and composed onto a results web page.

class bumps.pdfwrapper.VectorPDF (fn, p, name="", plot=None, dof=1, labels=None, **kw)
Bases: object
Build a model from a function.
This model can be fitted with any of the bumps optimizers.

fn is a function that returns the negative log likelihood of seeing its input parameters.
Vector p of length n defines the initial value. Unlike PDF, VectorPDF operates on a parameter vector p rather than individual parameters p1, p2, etc. Default parameter values p must be provided in order to determine the number of parameters.

labels are the names of the individual parameters. If not present, the name for parameter k defaults to pk. Each label is prefixed by name.
The optional plot function takes the same arguments as fn, with an additional view argument which may be set from the bumps command line. If provide, it should provide a visual indication of the function value and uncertainty on the current matplotlib.pyplot figure.

Additional keyword arguments are treated as the initial values for the parameters, or initial ranges if par=(min,max). Otherwise, the default is taken from the function definition (if the function uses par=value to define the parameter) or is set to zero if no default is given in the function.

chisq()
chisq_str()

has_residuals = False
nllf()
Call self as a function.

numpoints()
Return the number of data points.

parameters()
return the parameters in the model.
model parameters are a hierarchical structure of lists and dictionaries.

plot (view=None)
Plot the model to the current figure. You only get one figure, but you can make it as complex as you want. This will be saved as a png on the server, and composed onto a results web page.

residuals()
Return residuals for current theory minus data.
Used for Levenburg-Marquardt, and for plotting.

4.23 plotutil - Plotting utilities
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>auto_shift</code></td>
<td>Return a y-offset coordinate transform for the current axes.</td>
</tr>
<tr>
<td><code>coordinated_colors</code></td>
<td>Return a set of coordinated colors as c['base</td>
</tr>
<tr>
<td><code>dhsv</code></td>
<td>Modify color on hsv scale.</td>
</tr>
<tr>
<td><code>next_color</code></td>
<td>Return the next color in the plot color cycle.</td>
</tr>
<tr>
<td><code>plot_quantiles</code></td>
<td>Return quantiles and values for a list of confidence intervals.</td>
</tr>
<tr>
<td><code>form_quantiles</code></td>
<td>Plot quantile curves for a set of lines.</td>
</tr>
</tbody>
</table>

Pylab plotting utilities.

**bumps.plotutil.auto_shift(offset)**
- Return a y-offset coordinate transform for the current axes.

Each call to `auto_shift` increases the y-offset for the next line by the given number of points (with 72 points per inch).

Example:

```python
from matplotlib import pyplot as plt
from bumps.plotutil import auto_shift
trans = auto_shift(plt.gca())
plot(x, y, trans=trans)
```

**bumps.plotutil.coordinated_colors(base=None)**
- Return a set of coordinated colors as c['base|light|dark'].

If `base` is not provided, use the next color in the color cycle as the base. Light is bright and pale, dark is dull and saturated.

**bumps.plotutil.dhsv(color, dh=0.0, ds=0.0, dv=0.0, da=0.0)**
- Modify color on hsv scale.

`dv` change intensity, e.g., +0.1 to brighten, -0.1 to darken. `dh` change hue `ds` change saturation `da` change transparency

Color can be any valid matplotlib color. The hsv scale is [0,1] in each dimension. Saturation, value and alpha scales are clamped to [0,1] after changing. The hue scale wraps between red to violet.

Example

Make sea green 10% darker:

```python
>>> from bumps.plotutil import dhsv
>>> darker = dhsv('seagreen', dv=-0.1)
>>> print([int(v*255) for v in darker])
[37, 113, 71, 255]
```

**bumps.plotutil.next_color()**
- Return the next color in the plot color cycle.

Example:

```python
from matplotlib import pyplot as plt
from bumps.plotutil import next_color, dhsv
color = next_color()
plt.errorbar(x, y, yerr=dy, fmt='.', color=color)
```
# Draw the theory line with the same color as the data, but darker
plt.plot(x, y, '-', color=dhsv(color, dv=-0.2))

bumps.plotutil.plot_quantiles(x, y, contours, color, alpha=None)
Plot quantile curves for a set of lines.

- x is the x coordinates for all lines.
- y is the y coordinates, one row for each line.
- contours is a list of confidence intervals expressed as percents.
- color is the color to use for the quantiles. Quantiles are draw as a filled region with alpha transparency. Higher probability regions will be covered with multiple contours, which will make them lighter and more saturated.
- alpha is the transparency level to use for all fill regions. The default value, alpha=2./(#contours+1), works pretty well.

bumps.plotutil.form_quantiles(y, contours)
Return quantiles and values for a list of confidence intervals.
- contours is a list of confidence interfaces [a, b,...] expressed as percents.

Returns:
- quantiles is a list of intervals [[a_low, a_high], [b_low, b_high], ...] in [0,1].
- values is a list of intervals [[A_low, A_high], ...] with one entry in A for each row in y.

## 4.24 plugin - Domain branding

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>new_model</td>
<td>Return a new empty model or None.</td>
</tr>
<tr>
<td>load_model</td>
<td>Return a model stored within a file.</td>
</tr>
<tr>
<td>calc_errors</td>
<td>Gather data needed to display uncertainty in the model and the data.</td>
</tr>
<tr>
<td>show_errors</td>
<td>Display the model with uncertainty on the current figure.</td>
</tr>
<tr>
<td>data_view</td>
<td>Panel factory for the data tab in the GUI.</td>
</tr>
<tr>
<td>model_view</td>
<td>Panel factory for the model tab in the GUI.</td>
</tr>
</tbody>
</table>

Bumps plugin architecture.

With sophisticated models, developers need to be able to provide tools such as model builders and data viewers. Some of these will be tools for the GUI, such as views. Others will be tools to display results.

This file defines the interface that can be defined by your own application so that it interacts with models of your type. Define your own model package with a module plugin.py.

Create a main program which looks like:

```python
if __name__ == '__main__':
    import multiprocessing
    multiprocessing.freeze_support()

    import bumps.cli
    import mypackage.plugin
```

(continues on next page)
You should be able to use this as a driver program for your application.

Note: the plugin architecture is likely to change radically as more models are added to the system, particularly so that we can accommodate simultaneous fitting of data taken using different experimental techniques. For now, only one plugin at a time is supported.

```python
bumps.cli.install_plugin(mypackage.plugin)
bumps.cli.main()
```

You should be able to use this as a driver program for your application.

Note: the plugin architecture is likely to change radically as more models are added to the system, particularly so that we can accommodate simultaneous fitting of data taken using different experimental techniques. For now, only one plugin at a time is supported.

```python
bumps.plugin.new_model()
    Return a new empty model or None.
    Called in response to >File >New from the GUI. Creates a new empty model. Also triggered if GUI is started without a model.

bumps.plugin.load_model(filename)
    Return a model stored within a file.
    This routine is for specialized model descriptions not defined by script.
    If the filename does not contain a model of the appropriate type (e.g., because the extension is incorrect), then return None.
    No need to load pickles or script models. These will be attempted if load_model returns None.

bumps.plugin.calc_errors(problem, sample)
    Gather data needed to display uncertainty in the model and the data.
    Returns an object to be passed later to show_errors().

bumps.plugin.show_errors(errs)
    Display the model with uncertainty on the current figure.
    errs is the data returned from calc_errs.

bumps.plugin.data_view()
    Panel factory for the data tab in the GUI.
    If your model has an adequate show() function this should not be necessary.

bumps.plugin.model_view()
    Panel factory for the model tab in the GUI.
    Return None if not present.
```

### 4.25 pmath - Parametric versions of standard functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>exp</code></td>
<td>Return e raised to the power of x.</td>
</tr>
<tr>
<td><code>log</code></td>
<td>Return the logarithm of x to the given base.</td>
</tr>
<tr>
<td><code>log10</code></td>
<td>Return the base 10 logarithm of x.</td>
</tr>
<tr>
<td><code>sqrt</code></td>
<td>Return the square root of x.</td>
</tr>
<tr>
<td><code>sin</code></td>
<td>Return the sine of x (measured in radians).</td>
</tr>
<tr>
<td><code>cos</code></td>
<td>Return the cosine of x (measured in radians).</td>
</tr>
<tr>
<td><code>tan</code></td>
<td>Return the tangent of x (measured in radians).</td>
</tr>
<tr>
<td><code>asin</code></td>
<td>Return the arc sine (measured in radians) of x.</td>
</tr>
<tr>
<td><code>acos</code></td>
<td>Return the arc cosine (measured in radians) of x.</td>
</tr>
<tr>
<td><code>atan</code></td>
<td>Return the arc tangent (measured in radians) of x.</td>
</tr>
</tbody>
</table>
### Table 25 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>atan2</td>
<td>Return the arc tangent (measured in radians) of y/x.</td>
</tr>
<tr>
<td>sind</td>
<td>Return the sine of x (measured in degrees).</td>
</tr>
<tr>
<td>cosd</td>
<td>Return the cosine of x (measured in degrees).</td>
</tr>
<tr>
<td>tand</td>
<td>Return the tangent of x (measured in degrees).</td>
</tr>
<tr>
<td>asind</td>
<td>Return the arc sine (measured in degrees) of x.</td>
</tr>
<tr>
<td>acosd</td>
<td>Return the arc cosine (measured in degrees) of x.</td>
</tr>
<tr>
<td>atand</td>
<td>Return the arc tangent (measured in degrees) of x.</td>
</tr>
<tr>
<td>atan2d</td>
<td>Return the arc tangent (measured in degrees) of y/x.</td>
</tr>
<tr>
<td>sinh</td>
<td>Return the hyperbolic sine of x.</td>
</tr>
<tr>
<td>cosh</td>
<td>Return the hyperbolic cosine of x.</td>
</tr>
<tr>
<td>tanh</td>
<td>Return the hyperbolic tangent of x.</td>
</tr>
<tr>
<td>asinh</td>
<td>Return the inverse hyperbolic sine of x.</td>
</tr>
<tr>
<td>acosh</td>
<td>Return the inverse hyperbolic cosine of x.</td>
</tr>
<tr>
<td>atanh</td>
<td>Return the inverse hyperbolic tangent of x.</td>
</tr>
<tr>
<td>degrees</td>
<td>Convert angle x from radians to degrees.</td>
</tr>
<tr>
<td>radians</td>
<td>Convert angle x from degrees to radians.</td>
</tr>
<tr>
<td>sum</td>
<td>Return the sum of a ‘start’ value (default: 0) plus an iterable of numbers</td>
</tr>
<tr>
<td>prod</td>
<td>Return the product of a sequence of numbers.</td>
</tr>
</tbody>
</table>

Standard math functions for parameter expressions.

```python
bumps.pmath.exp(*args, **kw)
    Return e raised to the power of x.

bumps.pmath.log(x[, base=math.e])
    Return the logarithm of x to the given base.
    If the base not specified, returns the natural logarithm (base e) of x.

bumps.pmath.log10(*args, **kw)
    Return the base 10 logarithm of x.

bumps.pmath.sqrt(*args, **kw)
    Return the square root of x.

bumps.pmath.sin(*args, **kw)
    Return the sine of x (measured in radians).

bumps.pmath.cos(*args, **kw)
    Return the cosine of x (measured in radians).

bumps.pmath.tan(*args, **kw)
    Return the tangent of x (measured in radians).

bumps.pmath.asin(*args, **kw)
    Return the arc sine (measured in radians) of x.

bumps.pmath.acos(*args, **kw)
    Return the arc cosine (measured in radians) of x.

bumps.pmath.atan(*args, **kw)
    Return the arc tangent (measured in radians) of x.

bumps.pmath.atan2(*args, **kw)
    Return the arc tangent (measured in radians) of y/x.
    Unlike atan(y/x), the signs of both x and y are considered.
```

4.25. pmath - Parametric versions of standard functions
bumps.pmath.sind(*args, **kw)
    Return the sine of x (measured in in degrees).

bumps.pmath.cosd(*args, **kw)
    Return the cosine of x (measured in in degrees).

bumps.pmath.tand(*args, **kw)
    Return the tangent of x (measured in in degrees).

bumps.pmath.asind(*args, **kw)
    Return the arc sine (measured in in degrees) of x.

bumps.pmath.acosd(*args, **kw)
    Return the arc cosine (measured in in degrees) of x.

bumps.pmath.atan(*args, **kw)
    Return the arctangent (measured in in degrees) of x.

bumps.pmath.atan2d(*args, **kw)
    Return the arctangent (measured in in degrees) of y/x. Unlike atan(y/x), the signs of both x and y are considered.

bumps.pmath.sinh(*args, **kw)
    Return the hyperbolic sine of x.

bumps.pmath.cosh(*args, **kw)
    Return the hyperbolic cosine of x.

bumps.pmath.tanh(*args, **kw)
    Return the hyperbolic tangent of x.

bumps.pmath.asinh(*args, **kw)
    Return the inverse hyperbolic sine of x.

bumps.pmath.acosh(*args, **kw)
    Return the inverse hyperbolic cosine of x.

bumps.pmath.atanh(*args, **kw)
    Return the inverse hyperbolic tangent of x.

bumps.pmath.degrees(*args, **kw)
    Convert angle x from radians to degrees.

bumps.pmath.radians(*args, **kw)
    Convert angle x from degrees to radians.

bumps.pmath.sum(*args, **kw)
    Return the sum of a ‘start’ value (default: 0) plus an iterable of numbers
    When the iterable is empty, return the start value. This function is intended specifically for use with numeric values and may reject non-numeric types.

bumps.pmath.prod(*args, **kw)
    Return the product of a sequence of numbers.

4.26 pymcfit - Wrapper for pyMC models

PyMCProblem

Bumps wrapper for PyMC models.
4.27 quasinewton - BFGS quasi-newton optimizer

```
4.27. quasinewton - BFGS quasi-newton optimizer

quasinewton Run a quasinewton optimization on the problem.

BFGS quasi-newton optimizer.

All modules in this file are implemented from the book “Numerical Methods for Unconstrained Optimization and Nonlinear Equations” by J.E. Dennis and Robert B. Schnabel (Only a few minor modifications are done).

The interface is through the quasinewton() function. Here is an example call:

```python
n = 2
x0 = [-0.9 0.9]'
fn = lambda p: (1-p[0])**2 + 100*(p[1]-p[0]**2)**2
grad = lambda p: array([-2*(1-p[0]) - 400*(p[1]-p[0]**2)*p[0], 200*p[1]])
Sx = ones(n,1)
typf = 1
machedps = eps
teta = eps
maxstep = 100
gradtola = 1e-6
steptol = 1e-12
# do not let steptol larger than 1e-9
itnlimit = 1000
result = quasinewton(fn, x0, grad, Sx, typf,
                      macheps, eta, maxstep, gradtola, steptol, itnlimit)
print("status code %d"%result['status'])
print("x_min=%s, f(x_min)=%g%g"%(str(result['x']),result['fx']))
print("iterations, function calls, linesearch function calls",
      result['iterations'],result['evals'],result['linesearch_evals'])
```
bumps.quasinewton.quasinewton (fn, x0=None, grad=None, Sx=None, typf=1, macheps=None, eta=None, maxstep=100, gradtol=1e-06, steptol=1e-12, itnlimit=2000, abort_test=None, monitor=<function <lambda>>>)

Run a quasinewton optimization on the problem.

fn(x) is the cost function, which takes a point x and returns a scalar fx.

x0 is the initial point

grad is the analytic gradient (if available)

Sx is a scale vector indicating the typical values for parameters in the fitted result. This is used for a variety of things such as setting the step size in the finite difference approximation to the gradient, and controlling numerical accuracy in calculating the Hessian matrix. If for example some of your model parameters are in the order of 1e-6, then Sx for those parameters should be set to 1e-6. Default: [1, ...,]

typf is the typical value for f(x) near the minimum. This is used along with gradtol to check the gradient stopping condition. Default: 1

macheps is the minimum value that can be added to 1 to produce a number not equal to 1. Default: numpy.finfo(float).eps

eta adapts the numerical gradient calculations to machine precision. Default: macheps

maxstep is the maximum step size in any gradient step, after normalizing by Sx. Default: 100

gradopt is a stopping condition for the fit based on the amount of improvement expected at the next step. Default: 1e-6

steptol is a stopping condition for the fit based on the size of the step. Default: 1e-12

itnlimit is the maximum number of steps to take before stopping. Default: 2000

abort_test is a function which tests whether the user has requested abort. Default: None.

monitor(x,fx,step) is called every iteration so that a user interface function can monitor the progress of the fit. Default: lambda **kw: True

Returns the fit result as a dictionary:

status is a status code indicating why the fit terminated. Turn the status code into a string with STATUS[result.status]. Status values vary from 1 to 9, with 1 and 2 indicating convergence and the remaining codes indicating some form of premature termination.

x is the minimum point

fx is the value fn(x) at the minimum

H is the approximate Hessian matrix, which is the inverse of the covariance matrix

L is the cholesky decomposition of H+D, where D is a small correction to force H+D to be positive definite. To compute parameter uncertainty

iterations is the number of iterations

evals is the number of function evaluations

linesearch_evals is the number of function evaluations for line search

4.28 random_lines - Random lines and particle swarm optimizers
**random_lines**

Random lines is a population based optimizer which using quadratic fits along randomly oriented directions.

**particle_swarm**

Particle swarm is a population based optimizer which uses force and momentum to select candidate points.

Random Lines Algorithm finds the optimal minimum of a function.


bumps.random_lines.random_lines(cfo, NP, CR=0.9, epsilon=1e-10, abort_test=None, maxiter=1000)

Random lines is a population based optimizer which using quadratic fits along randomly oriented directions.

cfo is the cost function object. This is a dictionary which contains the following keys:

- **cost** is the function to be optimized. If parallel_cost exists, it should accept a list of points, not just a single point on each evaluation.
- **n** is the problem dimension
- **x0** is the initial point
- **x1** and **x2** are lower and upper bounds for each parameter

**monitor** is a callable which is called each iteration using callback(step, x, fx, k), where **step** is the iteration number, **x** is the population, **fx** is value of the cost function for each member of the population and **k** is the index of the best point in the population.

**f_opt** is the target value of the optimization

NP is the number of fit parameters

**CR** is the cross-over ratio, which is the proportion of dimensions that participate in any random orientation vector.

**epsilon** is the convergence criterion.

**abort_test** is a callable which indicates whether an external processes requests the fit to stop.

**maxiter** is the maximum number of generations

Returns success, num_evals, f(x_best), x_best.

bumps.random_lines.particle_swarm(cfo, NP, epsilon=1e-10, maxiter=1000)

Particle swarm is a population based optimizer which uses force and momentum to select candidate points.

cfo is the cost function object. This is a dictionary which contains the following keys:

- **cost** is the function to be optimized. If parallel_cost exists, it should accept a list of points, not just a single point on each evaluation.
- **n** is the problem dimension
- **x0** is the initial point
- **x1** and **x2** are lower and upper bounds for each parameter

**monitor** is a callable which is called each iteration using callback(step, x, fx, k), where **step** is the iteration number, **x** is the population, **fx** is value of the cost function for each member of the population and **k** is the index of the best point in the population.

**f_opt** is the target value of the optimization
NP is the number of fit parameters

$\epsilon$ is the convergence criterion.

abort_test is a callable which indicates whether an external processes requests the fit to stop.

maxiter is the maximum number of generations

Returns success, num_evals, f(x_best), x_best.

4.29 simplex - Nelder-Mead simplex optimizer (amoeba)

Downhill simplex optimizer.

*bumps.simplex.simplex* (f, x0=None, bounds=None, radius=0.05, xtol=0.0001, ftol=0.0001, maxiter=None, update_handler=None, abort_test=<function dont_abort>)

Minimize a function using Nelder-Mead downhill simplex algorithm.

This optimizer is also known as Amoeba (from Numerical Recipes) and the Neuler-Mead simplex algorithm. This is not the simplex algorithm for solving constrained linear systems.

Downhill simplex is a robust derivative free algorithm for finding minima. It proceeds by choosing a set of points (the simplex) forming an n-dimensional triangle, and transforming that triangle so that the worst vertex is improved, either by stretching, shrinking or reflecting it about the center of the triangle. This algorithm is not known for its speed, but for its simplicity and robustness, and is a good algorithm to start your problem with.

Parameters:

- **f** [callable f(x,*args)] The objective function to be minimized.
- **x0** [ndarray] Initial guess.
- **bounds** [(ndarray,ndarray) or None] Bounds on the parameter values for the function.
- **radius** [float] Size of the initial simplex. For bounded parameters (those which have finite lower and upper bounds), radius is clipped to a value in (0,0.5] representing the portion of the range to use as the size of the initial simplex.

Returns: Result (*park.simplex.Result*)

- **x** [ndarray] Parameter that minimizes function.
- **fx** [float] Value of function at minimum: $f_{opt} = func(x_{opt})$.
- **iters** [int] Number of iterations performed.
- **calls** [int] Number of function calls made.
- **success** [boolean] True if fit completed successfully.

Other Parameters:

- **xtol** [float] Relative error in xopt acceptable for convergence.
- **ftol** [number] Relative error in func(xopt) acceptable for convergence.
- **maxiter** [int=200*N] Maximum number of iterations to perform. Defaults
**update_handler** [callable] Called after each iteration, as callback(k,n,xk,fxk), where k is the current iteration, n is the maximum iteration, xk is the simplex and fxk is the value of the simplex vertices. xk[0],fxk[0] is the current best.

**abort_test** [callable] Called after each iteration, as callback(), to see if an external process has requested stop.

*Notes*

Uses a Nelder-Mead simplex algorithm to find the minimum of function of one or more variables.

### 4.30 util - Miscellaneous functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>kbhit</code></td>
<td>Check whether a key has been pressed on the console.</td>
</tr>
<tr>
<td><code>profile</code></td>
<td>Profile a function called with the given arguments.</td>
</tr>
<tr>
<td><code>pushdir</code></td>
<td>Change directories for the duration of a with statement.</td>
</tr>
<tr>
<td><code>push_seed</code></td>
<td>Set the seed value for the random number generator.</td>
</tr>
<tr>
<td><code>redirect_console</code></td>
<td>Console output redirection context</td>
</tr>
</tbody>
</table>

Miscellaneous utility functions.

bumps.util.`kbhit`()

- Check whether a key has been pressed on the console.

bumps.util.`profile`(fn, *args, **kw)

- Profile a function called with the given arguments.

class bumps.util.`pushdir`(path)

- Change directories for the duration of a with statement.

**Example**

Show that the original directory is restored:

```python
>>> import sys, os
>>> original_wd = os.getcwd()
>>> with pushdir(sys.path[0]):
...    pushed_wd = os.getcwd()
...    first_site = os.path.abspath(sys.path[0])
...    assert pushed_wd == first_site
>>> restored_wd = os.getcwd()
>>> assert original_wd == restored_wd
```

class bumps.util.`push_seed`(seed=None)

- Set the seed value for the random number generator.

When used in a with statement, the random number generator state is restored after the with statement is complete.

**Parameters**

- `seed` [int or array_like, optional] Seed for RandomState

**Example**
Seed can be used directly to set the seed:

```python
>>> from numpy.random import randint
>>> push_seed(24)
<...push_seed object at...>
>>> print(randint(0,1000000,3))
[242082  899  211136]
```

Seed can also be used in a with statement, which sets the random number generator state for the enclosed computations and restores it to the previous state on completion:

```python
>>> with push_seed(24):
...     print(randint(0,1000000,3))
[242082  899  211136]
```

Using nested contexts, we can demonstrate that state is indeed restored after the block completes:

```python
>>> with push_seed(24):
...     print(randint(0,1000000))
...     with push_seed(24):
...         print(randint(0,1000000,3))
...     print(randint(0,1000000))
242082
242082  899  211136
899
```

The restore step is protected against exceptions in the block:

```python
>>> with push_seed(24):
...     print(randint(0,1000000))
...     try:
...         with push_seed(24):
...             print(randint(0,1000000,3))
...             raise Exception()
...     except Exception:
...         print("Exception raised")
...     print(randint(0,1000000))
242082
242082  899  211136
Exception raised
899
```

class bumps.util.redirect_console (stdout=None, stderr=None)

Bases: object

Console output redirection context

The output can be redirected to a string, to an already opened file (anything with a write attribute), or to a filename which will be opened for the duration of the with context. Unless stderr is specified, then both standard output and standard error are redirected to the same file. The open file handle is returned on enter, and (if it was not an already opened file) it is closed on exit.

If no file is specified, then output is redirected to a StringIO object, which has a getvalue() method which can retrieve the string. The StringIO object is deleted when the context ends, so be sure to retrieve its value within the redirect_console context.

**Example**

Show that output is captured in a file:
Output can also be captured to a string:

```python
>>> with redirect_console() as fid:
...   print("captured to string")
...   captured_string = fid.getvalue()
>>> print(captured_string.strip())
captured to string
```

### 4.31 wsolve - Weighted linear and polynomial solver with uncertainty

**wsolve**

Given a linear system \( y = Ax + \delta y \), estimates \( x \) and \( \delta x \).

**wpolyfit**

Return the polynomial of degree \( n \) that minimizes \( \sum (p(x_i) - y_i)^2 / \sigma_i^2 \).

**LinearModel**

Model evaluator for linear solution to \( Ax = y \).

**PolynomialModel**

Model evaluator for best fit polynomial \( p(x) = y + / - \delta y \).

Weighted linear and polynomial solver with uncertainty.

Given \( A \bar{x} = \bar{y} \pm \delta \bar{y} \), solve using \( s = wsolve(A,y,dy) \)

wsolve uses the singular value decomposition for increased accuracy.

The uncertainty in the solution is estimated from the scatter in the data. Estimates the uncertainty for the solution from the scatter in the data.

The returned model object \( s \) provides:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.x</td>
<td>solution</td>
</tr>
<tr>
<td>s.std</td>
<td>uncertainty estimate assuming no correlation</td>
</tr>
<tr>
<td>s.nnorm</td>
<td>residual norm</td>
</tr>
<tr>
<td>s.DoF</td>
<td>degrees of freedom</td>
</tr>
<tr>
<td>s.cov</td>
<td>covariance matrix</td>
</tr>
<tr>
<td>s.ci(p)</td>
<td>confidence intervals at point ( p )</td>
</tr>
<tr>
<td>s.pi(p)</td>
<td>prediction intervals at point ( p )</td>
</tr>
<tr>
<td>s(p)</td>
<td>predicted value at point ( p )</td>
</tr>
</tbody>
</table>

### 4.31.1 Example

Weighted system:
```python
>>> import numpy as np
>>> from bumps import wsolve

>>> A = np.array([[1, 2, 3], [2, 1, 3], [1, 1, 1]], dtype='d')

>>> dy = [0.2, 0.01, 0.1]

>>> y = [14.16, 13.01, 6.15]

>>> s = wsolve.wsolve(A, y, dy)

>>> print(', '.join("\%0.2f +/- \%0.2f"\%(a, b) for a, b in zip(s.x, s.std)))
1.05 +/- 0.17, 2.20 +/- 0.12, 2.91 +/- 0.12
```

Note there is a counter-intuitive result that scaling the estimated uncertainty in the data does not affect the computed uncertainty in the fit. This is the correct result — if the data were indeed selected from a process with ten times the uncertainty, you would expect the scatter in the data to increase by a factor of ten as well. When this new data set is fitted, it will show a computed uncertainty increased by the same factor. Monte Carlo simulations bear this out. The conclusion is that the dataset carries its own information about the variance in the data, and the weight vector serves only to provide relative weighting between the points.

```
>>> bumps.wsolve.wsolve(A, y, dy=1, rcond=1e-12)

Given a linear system \(y = Ax + \delta y\), estimates \(x\) and \(\delta x\).

\(A\) is an \(n \times m\) array of measurement points.

\(y\) is an \(n \times k\) array or vector of length \(n\) of measured values at \(A\).

\(dy\) is a scalar or an \(n \times 1\) array of uncertainties in the values at \(A\).

```

Returns `LinearModel`.

```
>>> bumps.wsolve.wpolyfit(x, y, dy=1, degree=None, origin=False)

Return the polynomial of degree \(n\) that minimizes \(\sum (p(x_i) - y_i)^2 / \sigma_i^2\).

if origin is True, the fit should go through the origin.

```

Returns `PolynomialModel`.

```
class bumps.wsolve.LinearModel(x=None, DoF=None, SVinv=None, rnorm=None)

Bases: object

Model evaluator for linear solution to \(Ax = y\).

Use \(s(A)\) to compute the predicted value of the linear model \(s\) at points given on the rows of \(A\).

Computes a confidence interval (range of likely values for the mean at \(x\)) or a prediction interval (range of likely values seen when measuring at \(x\)). The prediction interval gives the width of the distribution at \(x\). This should be the same regardless of the number of measurements you have for the value at \(x\). The confidence interval gives the uncertainty in the mean at \(x\). It should get smaller as you increase the number of measurements. Error bars in the physical sciences usually show a \(1 - \alpha\) confidence value of \(\text{erfc}(1/\sqrt{2})\), representing a \(1 - \sigma\) standard deviation of uncertainty in the mean.

Confidence intervals for the expected value of the linear system evaluated at a new point \(w\) are given by the \(t\) distribution for the selected interval \(1 - \alpha\), the solution \(x\), and the number of degrees of freedom \(n - p\): \n\[
\begin{align*}
    w^T x & \pm \frac{\alpha/2}{n-p} \sqrt{\text{var}(w)}
\end{align*}
\]
where the variance \(\text{var}(w)\) is given by:

\[
\text{var}(w) = \sigma^2 (w^T (A^T A)^{-1} w)
\]

Prediction intervals are similar, except the variance term increases to include both the uncertainty in the predicted value and the variance in the data:

\[
\text{var}(w) = \sigma^2 (1 + w^T (A^T A)^{-1} w)
\]

```

162 Chapter 4. Reference: bumps
DoF = None
    number of degrees of freedom in the solution

ci (A, sigma=1)
    Compute the calculated values and the confidence intervals for the linear model evaluated at A.
    
    sigma=1 corresponds to a 1 − σ confidence interval
    Confidence intervals are sometimes expressed as 1 − α values, where α = \text{erfc}(σ/\sqrt{2})

cov
    covariance matrix [\text{inv}(A'A); O(n^3)]

P
    p-value probability of rejection

pi (A, p=0.05)
    Compute the calculated values and the prediction intervals for the linear model evaluated at A.
    
    p=0.05 corresponds to the 95% prediction interval.

rnorm = None
    2-norm of the residuals ||y − Ax||_2

std
    solution standard deviation [sqrt(var); O(n^2)]

var
    solution variance [\text{diag}(cov); O(n^2)]

x = None
    solution to the equation Ax = y

class bumps.wsolve.PolynomialModel (x, y, dy, s, origin=False)
    Bases: object
    
    Model evaluator for best fit polynomial \( p(x) = y + / − \delta y \).
    Use \( p(x) \) for PolynomialModel \( p \) to evaluate the polynomial at all points in the vector \( x \).

DoF = None
    number of degrees of freedom in the solution

ci (x, sigma=1)
    Evaluate the polynomial and the confidence intervals at x.
    
    sigma=1 corresponds to a 1-sigma confidence interval

coeff = None
    polynomial coefficients

cov
    covariance matrix
    Note that the ones column will be absent if origin is True.

degree = None
    polynomial degree

der (x)
    Evaluate the polynomial derivative at x.

origin = None
    True if polynomial goes through the origin
p

- p-value probability of rejection

\( \pi(x, p=0.05) \)

- Evaluate the polynomial and the prediction intervals at \( x \).
  - \( p = 1 - \alpha = 0.05 \) corresponds to 95% prediction interval

plot (ci=1, pi=0)

- \( \text{plot} \) evaluates the polynomial and the prediction intervals at \( x \).
  - \( \text{p} = \text{1-alpha} = 0.05 \) corresponds to 95% prediction interval.

\( rnorm = \text{None} \)

- 2-norm of the residuals \( \| y - Ax \|_2 \)

std

- solution standard deviation

var

- solution variance

---

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bounds</td>
<td>Parameter bounds and prior probabilities.</td>
</tr>
<tr>
<td>bspline</td>
<td>BSpline calculator.</td>
</tr>
<tr>
<td>cheby</td>
<td>Freeform modeling with Chebyshev polynomials.</td>
</tr>
<tr>
<td>cli</td>
<td>Bumps command line interface.</td>
</tr>
<tr>
<td>curve</td>
<td>Build a bumps model from a function and data.</td>
</tr>
<tr>
<td>data</td>
<td>Data handling utilities.</td>
</tr>
<tr>
<td>errplot</td>
<td>Estimate model uncertainty from random sample.</td>
</tr>
<tr>
<td>fitproblem</td>
<td>Interface between the models and the fitters.</td>
</tr>
<tr>
<td>fitservice</td>
<td>Fit job definition for the distributed job queue.</td>
</tr>
<tr>
<td>fitters</td>
<td>Interfaces to various optimizers.</td>
</tr>
<tr>
<td>formatnum</td>
<td>Format values and uncertainties nicely for printing.</td>
</tr>
<tr>
<td>history</td>
<td>Log of progress through a computation.</td>
</tr>
<tr>
<td>initpop</td>
<td>Population initialization strategies.</td>
</tr>
<tr>
<td>lsqerror</td>
<td>Least squares error analysis.</td>
</tr>
<tr>
<td>mapper</td>
<td>Parallel and serial mapper implementations.</td>
</tr>
<tr>
<td>monitor</td>
<td>Progress monitors.</td>
</tr>
<tr>
<td>mono</td>
<td>Monotonic spline modeling.</td>
</tr>
<tr>
<td>names</td>
<td>Exported names.</td>
</tr>
<tr>
<td>options</td>
<td>Option parser for bumps command line</td>
</tr>
<tr>
<td>parameter</td>
<td>Fitting parameter objects.</td>
</tr>
<tr>
<td>partemp</td>
<td>Parallel tempering for continuous function optimization and uncertainty analysis.</td>
</tr>
<tr>
<td>pdfwrapper</td>
<td>Build a bumps model from a function.</td>
</tr>
<tr>
<td>plotutil</td>
<td>Pylab plotting utilities.</td>
</tr>
<tr>
<td>plugin</td>
<td>Bumps plugin architecture.</td>
</tr>
<tr>
<td>pmath</td>
<td>Standard math functions for parameter expressions.</td>
</tr>
<tr>
<td>pymcfit</td>
<td>Bumps wrapper for PyMC models.</td>
</tr>
<tr>
<td>quasinewton</td>
<td>BFGS quasi-newton optimizer.</td>
</tr>
<tr>
<td>random_lines</td>
<td>Random Lines Algorithm finds the optimal minimum of a function.</td>
</tr>
<tr>
<td>simplex</td>
<td>Downhill simplex optimizer.</td>
</tr>
<tr>
<td>util</td>
<td>Miscellaneous utility functions.</td>
</tr>
<tr>
<td>wsolve</td>
<td>Weighted linear and polynomial solver with uncertainty.</td>
</tr>
</tbody>
</table>
5.1 acr - A C Rencher normal outlier test

ACR upper percentiles critical value for test of single multivariate normal outlier.

From the method given by Wilks (1963) and approaching to a F distribution function by the Yang and Lee (1987) formulation, we compute the critical value of the maximum squared Mahalanobis distance to detect outliers from a normal multivariate sample.

We can generate all the critical values of the maximum squared Mahalanobis distance presented on the Table XXXII of by Barnett and Lewis (1978) and Table A.6 of Rencher (2002). Also with any given significance level (alpha).

Example:

```python
>>> print("%.4f"%ACR(3, 25, 0.01))
13.1753
```

Created by:

A. Trujillo-Ortiz, R. Hernandez-Walls, A. Castro-Perez and K. Barba-Rojo
Facultad de Ciencias Marinas
Universidad Autonoma de Baja California
Apdo. Postal 453
Ensenada, Baja California
Mexico.
strujo@uabc.mx

Copyright. August 20, 2006.

To cite this file, this would be an appropriate format:
The function’s name is given in honour of Dr. Alvin C. Rencher for his invaluable contribution to multivariate statistics with his text ‘Methods of Multivariate Analysis’.

References:

bumps.dream.acr.ACR \( (p, n, \alpha=0.05) \)
Return critical value for test of single multivariate normal outlier using the Mahalanobis distance metric.

\( p \) is the number of independent variables, \( n \) is the number of samples, and \( \alpha \) is the significance level cutoff (default=0.05).

### 5.2 bounds - Bounds handling

<table>
<thead>
<tr>
<th>make_bounds_handler</th>
<th>Return a bounds object which can update the bounds.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bounds</td>
<td>Base class for all times of bounds objects.</td>
</tr>
<tr>
<td>ReflectBounds</td>
<td>Reflect parameter values into bounded region</td>
</tr>
<tr>
<td>ClipBounds</td>
<td>Clip values to bounded region</td>
</tr>
<tr>
<td>FoldBounds</td>
<td>Wrap values into the bounded region</td>
</tr>
<tr>
<td>RandomBounds</td>
<td>Randomize values into the bounded region</td>
</tr>
<tr>
<td>IgnoreBounds</td>
<td>Leave values outside the bounded region</td>
</tr>
</tbody>
</table>

Bounds handling.

Use \( \text{bounds}(\text{low}, \text{high}, \text{style}) \) to create a bounds handling object. This function operates on a point \( x \), transforming it so that all dimensions are within the bounds. Options are available, including reflecting, wrapping, clipping or randomizing the point, or ignoring the bounds.

The returned bounds object should have an \( \text{apply}(x) \) method which transforms the point \( x \).

bumps.dream.bounds.make_bounds_handler \( (\text{bounds}, \text{style}='\text{reflect}') \)
Return a bounds object which can update the bounds.

Bounds handling \textit{style} name is one of:

- \textit{reflect}: reflect off the boundary
- \textit{clip}: stop at the boundary
- \textit{fold}: wrap values to the other side of the boundary
- \textit{randomize}: move to a random point in the bounds
- \textit{none}: ignore the bounds

With semi-infinite intervals folding and randomizing aren’t well defined, and reflection is used instead.
With finite intervals the reflected or folded point may still be outside the bounds (which can happen if the step size is too large), and a random uniform value is used instead.

```python
class bumps.dream.bounds.Bounds
    Bases: object
    Base class for all times of bounds objects.

    apply(x)
        Force x values within bounds

    c_interface = None
    high = None
    low = None

class bumps.dream.bounds.ReflectBounds(low, high)
    Bases: bumps.dream.bounds.Bounds
    Reflect parameter values into bounded region

    apply(y)
        Update x so all values lie within bounds
        Returns x for convenience. E.g., y = bounds.apply(x+0)

    c_interface = None
    high = None
    low = None

class bumps.dream.bounds.ClipBounds(low, high)
    Bases: bumps.dream.bounds.Bounds
    Clip values to bounded region

    apply(y)
        Force x values within bounds

    c_interface = None
    high = None
    low = None

class bumps.dream.bounds.FoldBounds(low, high)
    Bases: bumps.dream.bounds.Bounds
    Wrap values into the bounded region

    apply(y)
        Force x values within bounds

    c_interface = None
    high = None
    low = None

class bumps.dream.bounds.RandomBounds(low, high)
    Bases: bumps.dream.bounds.Bounds
    Randomize values into the bounded region

    apply(y)
        Force x values within bounds
```

5.2. bounds - Bounds handling
c_interface = None
high = None
low = None

class bumps.dream.bounds.IgnoreBounds(low=None, high=None)
    Bases: bumps.dream.bounds.Bounds

    Leave values outside the bounded region

    apply(y)
    Force x values within bounds

c_interface = None
high = None
low = None

5.3 core - DREAM core

<table>
<thead>
<tr>
<th>Dream</th>
<th>Data structure containing the details of the running DREAM analysis code.</th>
</tr>
</thead>
</table>

DiffeRential Evolution Adaptive Metropolis algorithm

DREAM runs multiple different chains simultaneously for global exploration, and automatically tunes the scale and orientation of the proposal distribution using differential evolution. The algorithm maintains detailed balance and ergodicity and works well and efficient for a large range of problems, especially in the presence of high-dimensionality and multimodality.

DREAM developed by Jasper A. Vrugt and Cajo ter Braak

This algorithm has been described in:


For more information please read:


Copyright (c) 2008, Los Alamos National Security, LLC All rights reserved.

Copyright 2008. Los Alamos National Security, LLC. This software was produced under U.S. Government contract DE-AC52-06NA25396 for Los Alamos National Laboratory (LANL), which is operated by Los Alamos National Security, LLC for the U.S. Department of Energy. The U.S. Government has rights to use, reproduce, and distribute this software.

NEITHER THE GOVERNMENT NOR LOS ALAMOS NATIONAL SECURITY, LLC MAKES ANY WARRANTY, EXPRESS OR IMPLIED, OR ASSUMES ANY LIABILITY FOR THE USE OF THIS SOFTWARE. If software is modified to produce derivative works, such modified software should be clearly marked, so as not to confuse it with the version available from LANL.

Additionally, redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of Los Alamos National Security, LLC, Los Alamos National Laboratory, LANL the U.S. Government, nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY LOS ALAMOS NATIONAL SECURITY, LLC AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL LOS ALAMOS NATIONAL SECURITY, LLC OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

MATLAB code written by Jasper A. Vrugt, Center for NonLinear Studies (CNLS)

Written by Jasper A. Vrugt: vrugt@lanl.gov

Version 0.5: June 2008 Version 1.0: October 2008 Adaption updated and generalized CR implementation

2010-04-20 Paul Kienzle * Convert to python
class bumps.dream.core.Dream(**kw)
    Bases: object
    Data structure containing the details of the running DREAM analysis code.
    
    CR = None
    CR_spacing = 'linear'
    DE_eps = 0.05
    DE_noise = 1e-06
    DE_pairs = 3
    DE_snooker_rate = 0.1
    DE_steps = 10
    DR_scale = 1
    alpha = 0.01
        convergence criteria
    bounds_style = 'reflect'
    burn = 0
    draws = 100000
    goalseek_interval = 1e+100
    goalseek_minburn = 1000
    goalseek_optimizer = None
    model = None
    outlier_test = 'none'
    population = None
    sample(state=None, abort_test=<function Dream.<lambda>>)
        Pull the requisite number of samples from the distribution
    state = None
    thinning = 1
    use_delayed_rejection = False

5.4 corrplot - Correlation plots

Corr2d Generate and manage 2D correlation histograms.

2-D correlation histograms
Generate 2-D correlation histograms and display them in a figure.
Uses false color plots of density.

class bumps.dream.corrplot.Corr2d(data, labels=None, **kw)
    Bases: object
    Generate and manage 2D correlation histograms.
5.5 crossover - Adaptive crossover support

<table>
<thead>
<tr>
<th>Crossover</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crossover</td>
<td>Fixed weight crossover ratios.</td>
</tr>
<tr>
<td>BaseAdaptiveCrossover</td>
<td>Adapted weight crossover ratios.</td>
</tr>
<tr>
<td>AdaptiveCrossover</td>
<td>Adapted weight crossover ratios.</td>
</tr>
<tr>
<td>LogAdaptiveCrossover</td>
<td>Adapted weight crossover ratios, log-spaced.</td>
</tr>
</tbody>
</table>

Crossover ratios

The crossover ratio (CR) determines what percentage of parameters in the target vector are updated with difference vector selected from the population. In traditional differential evolution a CR value is chosen somewhere in \([0, 1]\) at the start of the search and stays constant throughout. DREAM extends this by allowing multiple CRs at the same time with different probabilities. Adaptive crossover adjusts the relative weights of the CRs based on the average distance of the steps taken when that CR was used. This distance will be zero for unsuccessful metropolis steps, and so the relative weights on those CRs which generate many unsuccessful steps will be reduced.

5.5.1 Usage

1. Traditional differential evolution:

   ```python
crossover = Crossover(CR=CR)
   ```

2. Weighted crossover ratios:

   ```python
crossover = Crossover(CR=[CR1, CR2, ...], weight=[weight1, weight2, ...])
   ```

The weights are normalized to one, and default to equally weighted CRs.

3. Adaptive weighted crossover ratios:

   ```python
crossover = AdaptiveCrossover(N)
   ```

The CRs are set to \([1/N, 2/N, \ldots 1]\), and start out equally weighted. The weights are adapted during burn-in (10% of the runs) and fixed for the remainder of the analysis.

5.5.2 Compatibility Notes

For `Extra.pCR == 'Update'` in the matlab interface use:

```python
CR = AdaptiveCrossover(Ncr=MCMCPar.nCR)
```

For `Extra.pCR != 'Update'` in the matlab interface use:

```python
CR = Crossover(CR=[1./Ncr], pCR=[1])
```

```
class bumps.dream.crossover.Crossover (CR, weight=None)
   Bases: object
```

5.5. crossover - Adaptive crossover support
Fixed weight crossover ratios.

CR is a scalar if there is a single crossover ratio, or a vector of numbers in (0, 1].

weight is the relative weighting of each CR, or None for equal weights.

adapt()  
Update CR weights based on the available adaptation data.

reset()  
update(xold, xnew, used)  
Gather adaptation data on xold, xnew for each CR that was used in step N.

class bumps.dream.crossover.BaseAdaptiveCrossover
Bases: object
Adapted weight crossover ratios.

adapt()  
Update CR weights based on the available adaptation data.

reset()  
update(xold, xnew, used)  
Gather adaptation data on xold, xnew for each CR that was used in step N.

weight = None

class bumps.dream.crossover.AdaptiveCrossover(N)
Bases: bumps.dream.crossover.BaseAdaptiveCrossover
Adapted weight crossover ratios.

N is the number of CRs to use. CR is set to [1/N, 2/N, . . . , 1], with initial weights [1/N, 1/N, . . . , 1/N].

adapt()  
Update CR weights based on the available adaptation data.

reset()  
update(xold, xnew, used)  
Gather adaptation data on xold, xnew for each CR that was used in step N.

weight = None

class bumps.dream.crossover.LogAdaptiveCrossover(dim, N=4.5)
Bases: bumps.dream.crossover.BaseAdaptiveCrossover
Adapted weight crossover ratios, log-spaced.

dim is the number of dimensions in the problem. N is the number of CRs to use per decade.

CR is set to [k(dim)] where k is log-spaced from 1 to dim. The CRs start equally weighted as [1, . . . , 1]/len(CR).

N should be around 4.5. This gives good low end density, with 1, 2, 3, and 5 parameters changed at a time, and proceeds up to 60% and 100% of parameters each time. Lower values of N give too few high density CRs, and higher values give too many low density CRs.

adapt()  
Update CR weights based on the available adaptation data.

reset()  
update(xold, xnew, used)  
Gather adaptation data on xold, xnew for each CR that was used in step N.
weight = None

5.6 diffev - Differential evolution MCMC stepper

de_step

Generates offspring using METROPOLIS HASTINGS monte-carlo markov chain

Differential evolution MCMC stepper.

bumps.dream.diffev.de_step(Nchain, pop, CR, max_pairs=2, eps=0.05, snooker_rate=0.1, noise=1e-06, scale=1.0)
Generates offspring using METROPOLIS HASTINGS monte-carlo markov chain
The number of chains may be smaller than the population size if the population is selected from both the current generation and the ancestors.

5.7 entropy - Entropy calculation

entropy

Return entropy estimate and uncertainty from a random sample.

gmm_entropy
Use sklearn.mixture.BayesianGaussianMixture to estimate entropy.

cov_entropy
Entropy estimate from covariance matrix C

wnn_entropy
Weighted Kozachenko-Leonenko nearest-neighbour entropy calculation.

MVNEntropy
Multivariate normal entropy approximation.

Estimate entropy after a fit.
The gmm_entropy() function computes the entropy from a Gaussian mixture model. This provides a reasonable estimate even for non-Gaussian distributions. This is the recommended method for estimating the entropy of a sample.
The cov_entropy() method computes the entropy associated with the covariance matrix. This covariance matrix can be estimated during the fitting procedure (BFGS updates an estimate of the Hessian matrix for example), or computed by estimating derivatives when the fit is complete.
The MVNEntropy class estimates the covariance from an MCMC sample and uses this covariance to estimate the entropy. This gives a better estimate of the entropy than the equivalent direct calculation, which requires many more samples for a good kernel density estimate. The reject_normal attribute is True if the MCMC sample is significantly different from normal. Unfortunately, this almost always the case for any reasonable sample size that isn’t strictly gaussian.
The entropy() function computes the entropy directly from a set of MCMC samples, normalized by a scale factor computed from the kernel density estimate at a subset of the points.

There are many other entropy calculations implemented within this file, as well as a number of sampling distributions for which the true entropy is known. Furthermore, entropy was computed against dream output and checked for consistency. None of the methods is truly excellent in terms of minimum sample size, maximum dimensions and speed, but many of them are pretty good.


5.7. entropy - Entropy calculation
The following is an informal summary of the results from different algorithms applied to DREAM output:

```python
from bumps.dream.entropy import Timer as T

# Try MVN ... only good for normal distributions, but very fast
with T(): M = entropy.MVNEntropy(drawn.points)
print("Entropy from MVN: \$s\$
str(M))

# Try wnn ... no good.
with T(): S_wnn, Serr_wnn = entropy.ww_entropy(drawn.points, n_est=20000)
print("Entropy from wnn: \$s\$
str(S_wnn))

# Try wnn with bootstrap ... still no good.
with T(): S_wnn, Serr_wnn = entropy.ww_bootstrap(drawn.points)
print("Entropy from wnn bootstrap: \$s\$
str(S_wnn))

# Try wnn entropy with thinning ... still no good.
#drawn = self.draw(portion=portion, vars=vars,
#selection=selection, thin=10)
with T(): S_wnn, Serr_wnn = entropy.ww_entropy(drawn.points)
print("Entropy from wnn with thinning: \$s\$
str(S_wnn))

# Try wnn with gmm ... still no good
with T(): S_wnn, Serr_wnn = entropy.ww_entropy(drawn.points, n_est=20000, gmm=20)
print("Entropy from wnn with gmm: \$s\$
str(S_wnn))

# Try pure gmm ... pretty good
with T(): S_gmm, Serr_gmm = entropy.gmm_entropy(drawn.points, n_est=10000)
print("Entropy from pure gmm: \$s\$
str(S_gmm))

# Try kde from statsmodels ... pretty good
with T(): S_kde_stats = entropy.kde_entropy_statsmodels(drawn.points, n_est=10000)
print("Entropy from kde statsmodels: \$s\$
str(S_kde_stats))

# Try kde from sklearn ... pretty good
with T(): S_kde = entropy.kde_entropy_sklearn(drawn.points, n_est=10000)
print("Entropy from kde sklearn: \$s\$
str(S_kde))

# Try kde from sklearn at points from gmm ... pretty good
with T(): S_kde_gmm = entropy.kde_entropy_sklearn_gmm(drawn.points, n_est=10000)
print("Entropy from kde+gmm: \$s\$
str(S_kde_gmm))

# Try Kramer ... pretty good, but doesn’t support marginal entropy
with T(): S, Serr = entropy.entropy(drawn.points, drawn.logp, N_entropy=n_est)
print("Entropy from Kramer: \$s\$
str(S))
```

`bumps.dream.entropy.entropy(points, logp, N_entropy=10000, N_norm=2500)`

Return entropy estimate and uncertainty from a random sample.

- `points` is a set of draws from an underlying distribution, as returned by a Markov chain Monte Carlo process for example.
- `logp` is the log-likelihood for each draw.
- `N_norm` is the number of points $k$ to use to estimate the posterior density normalization factor $P(D) = \hat{N}$, converting from $\log(P(D|M)P(M))$ to $\log(P(D|M)P(M)/P(D))$. The relative uncertainty $\Delta\hat{S}/\hat{S}$ scales with $\sqrt{k}$, with the default $N_{\text{norm}}=2500$ corresponding to 2% relative uncertainty. Computation cost is $O(nk)$ where $n$ is number of points in the draw.
- `N_entropy` is the number of points used to estimate the entropy $\hat{S} = -\int P(M|D)\log P(M|D)$ from the
normalized log likelihood values.

```
bumps.dream.entropy.gmm_entropy(points, n_est=None, n_components=None)
```

Use sklearn.mixture.BayesianGaussianMixture to estimate entropy.

- `points` are the data points in the sample.
- `n_est` are the number of points to use in the estimation; default is 10,000 points, or 0 for all the points.
- `n_components` are the number of Gaussians in the mixture. Default is $5\sqrt{d}$ where $d$ is the number of dimensions.

Returns estimated entropy and uncertainty in the estimate.

This method uses BayesianGaussianMixture from scikit-learn to build a model of the point distribution, then uses Monte Carlo sampling to determine the entropy of that distribution. The entropy uncertainty is computed from the variance in the MC sample scaled by the number of samples. This does not incorporate any uncertainty in the sampling that generated the point distribution or the uncertainty in the GMM used to model that distribution.

```
bumps.dream.entropy.cov_entropy(C)
```

Entropy estimate from covariance matrix C

```
bumps.dream.entropy.wnn_entropy(points, k=None, weights=True, n_est=None, gmm=None)
```

Weighted Kozachenko-Leonenko nearest-neighbour entropy calculation.

- `k` is the number of neighbours to consider, with default $k = n^{1/3}$
- `n_est` is the number of points to use for estimating the entropy, with default $n_{est} = n$
- `weights` is True for default weights, False for unweighted (using the distance to the kth neighbour only), or a vector of weights of length $k$.
- `gmm` is the number of gaussians to use to model the distribution using a gaussian mixture model. Default is 0, and the points represent an empirical distribution.

Returns entropy $H$ in bits and its uncertainty.


```
class bumps.dream.entropy.MVNEntropy(x, alpha=0.05, max_points=1000)
```

Bases: object

Multivariate normal entropy approximation.

Uses Mardia’s multivariate skewness and kurtosis test to estimate normality.

- `x` is a set of points
- `alpha` is the cutoff for the normality test.
- `max_points` is the maximum number of points to use when checking normality. Since the normality test is $O(n^2)$ in memory and time, where $n$ is the number of points, `max_points` defaults to 1000. The entropy is computed from the full dataset.

The returned object has the following attributes:

- `p_kurtosis` is the p-value for the kurtosis normality test
- `p_skewness` is the p-value for the skewness normality test
- `reject_normal` is True if either the the kurtosis or the skew test fails
- `entropy` is the estimated entropy of the best normal approximation to the distribution
5.8 **exppow - Exponential power density parameter calculator**

**exppow Pars**

Return \( w(B) \) and \( c(B) \) for the exponential power density:

\[
p(v|S, B) = \frac{w(B)}{S} \exp \left( -c(B) |v/S|^{2/(1+B)} \right)
\]

\( B \) in (-1,1] is a measure of kurtosis:

- \( B = 1 \): double exponential
- \( B = 0 \): normal
- \( B \to -1 \): uniform


5.9 **formatnum - Format values and uncertainties nicely for printing**

**format_value**

Given value \( v \) and uncertainty \( dv \), return a string \( v \) which is the value formatted with the appropriate number of digits.

**format_uncertainty**

Value and uncertainty formatter.

**format_uncertainty_compact**

Given value \( v \) and uncertainty \( dv \), return the compact representation \( v(##) \), where ## are the first two digits of the uncertainty.

**format_uncertainty_pm**

Given value \( v \) and uncertainty \( dv \), return a string \( v +/\pm dv \).

Format values and uncertainties nicely for printing.

The formatted value uses only the number of digits warranted by the uncertainty in the measurement.

**format_value()** shows the value without the uncertainty.

**format_uncertainty_pm()** shows the expanded format \( v +/\pm err \).

**format_uncertainty_compact()** shows the compact format \( v(##) \), where the number in parenthesis is the uncertainty in the last two digits of \( v \).

**format_uncertainty()** uses the compact format by default, but this can be changed to use the expanded +/- format by setting format_uncertainty.compact to False. This is a global setting which should be considered a user preference. Any library code that depends on a specific format style should use the corresponding formatting function.

If the uncertainty is 0 or not otherwise provided, the simple %g floating point format option is used.

Infinite and indefinite numbers are represented as inf and NaN.

Example:
bumps.dream.formatnum.format_value(value, uncertainty)
Given value v and uncertainty dv, return a string v which is the value formatted with the appropriate number of digits.

bumps.dream.formatnum.format_uncertainty(value, uncertainty)
Value and uncertainty formatter.
Either the expanded v +/- dv form or the compact v(##) form will be used depending on whether format_uncertainty.compact is True or False. The default is True.

bumps.dream.formatnum.format_uncertainty_compact(value, uncertainty)
Given value v and uncertainty dv, return the compact representation v(##), where ## are the first two digits of the uncertainty.

bumps.dream.formatnum.format_uncertainty_pm(value, uncertainty)
Given value v and uncertainty dv, return a string v +/- dv.

## 5.10 gelman - R-statistic convergence test

gelman
Calculates the R-statistic convergence diagnostic

Convergence test statistic from Gelman and Rubin, 1992.[1]


bumps.dream.gelman.gelman(sequences, portion=0.5)
Calculates the R-statistic convergence diagnostic


## 5.11 geweke - Geweke convergence test

geweke
Calculates the Geweke convergence diagnostic


bumps.dream.geweke.geweke(sequences, portion=0.25)
Calculates the Geweke convergence diagnostic

Refer to:
5.12 initpop - Population initialization routines

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lhs_init</td>
<td>Latin Hypercube Sampling</td>
</tr>
<tr>
<td>cov_init</td>
<td>Initialize N sets of random variables from a gaussian model.</td>
</tr>
</tbody>
</table>

Population initialization routines.

To start the analysis an initial population is required. This will be an array of size \( M \times N \), where \( M \) is the number of dimensions in the fitting problem and \( N \) is the number of Markov chains.

Two functions are provided:

1. \( \text{lhs} \_ \text{init}(N, \text{bounds}) \) returns a latin hypercube sampling, which tests every parameter at each of \( N \) levels.
2. \( \text{cov}_\text{init}(N, x, \text{cov}) \) returns a Gaussian sample along the ellipse defined by the covariance matrix, \( \text{cov} \). Covariance defaults to \( \text{diag}(dx) \) if \( dx \) is provided as a parameter, or to \( I \) if it is not.

Additional options are random box: \( \text{rand}(M, N) \) or random scatter: \( \text{randn}(M, N) \).

bumps.dream.initpop.lhs_init(\( N, \text{bounds} \))  
Latin Hypercube Sampling

Returns an array whose columns each have \( N \) samples from equally spaced bins between \( \text{bounds}=(\text{xmin}, \text{xmax}) \) for the column. DREAM bounds objects, with \( \text{bounds} \).low and \( \text{bounds} \).high can be used as well.

Note: Indefinite ranges are not supported.

bumps.dream.initpop.cov_init(\( N, x, \text{cov}=\text{None}, dx=\text{None} \))  
Initialize \( N \) sets of random variables from a gaussian model.

The center is at \( x \) with an uncertainty ellipse specified by the 1-sigma independent uncertainty values \( dx \) or the full covariance matrix uncertainty \( \text{cov} \).

For example, create an initial population for 20 sequences for a model with local minimum \( x \) with covariance matrix \( C \):

```
pop = cov_init(cov=C, x=x, N=20)
```

5.13 ksmirnov - Kolmogorov-Smirnov test for MCMC convergence

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ksmirnov</td>
<td>Kolmogorov-Smirnov test of similarity between the empirical distribution at the start and at the end of the chain.</td>
</tr>
</tbody>
</table>

Kolmogorov-Smirnov test for MCMC convergence.

Use the K-S tests to compare the distribution of values at the front of the chain to that at the end of the chain. If the distributions are significantly different, then the MCMC chain has not converged.

bumps.dream.ksmirnov.ksmirnov(seq, portion=0.25, filter_order=15)  
Kolmogorov-Smirnov test of similarity between the empirical distribution at the start and at the end of the chain.

Apply a median filter (filter=15) on neighbouring K-S values to reduce variation in the test statistic value.
5.14 mahal - Mahalanobis distance calculator

`mahalanobis`  
Returns the distances of the observations from a reference set.

Mahalanobis distance calculator

Compute the Mahalanobis distance between observations and a reference set. The principle components of the reference set define the basis of the space for the observations. The simple Euclidean distance is used within this space.

`bumps.dream.mahal.mahalanobis(Y, X)`  
Returns the distances of the observations from a reference set.
Observations are stored in rows $Y$ and the reference set in $X$.

5.15 metropolis - MCMC step acceptance test

`metropolis`  
Metropolis rule for acceptance or rejection

`metropolis_dr`  
Delayed rejection metropolis

MCMC step acceptance test.

`bumps.dream.metropolis.metropolis(xtry, logp_try, xold, logp_old, step_alpha)`  
Metropolis rule for acceptance or rejection
Generates the next generation, $newgen$ from:

\[
\begin{align*}
x_{\text{new}}[k] &= x[k] & \text{if } U > \alpha \\
&= x_{\text{old}}[k] & \text{if } U \leq \alpha
\end{align*}
\]

where $\alpha$ is $p/p_{\text{old}}$ and accept is $U > \alpha$.

Returns $x_{\text{new}}$, $\log p_{\text{new}}$, $\alpha$, accept

`bumps.dream.metropolis.metropolis_dr(xtry, logp_try, x, logp, xold, logp_old, alpha12, R)`  
Delayed rejection metropolis

5.16 model - MCMC model types

<table>
<thead>
<tr>
<th>Model</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MCMCModel</code></td>
<td>MCMC model abstract base class.</td>
</tr>
<tr>
<td><code>Density</code></td>
<td>Construct an MCMC model from a probability density function.</td>
</tr>
<tr>
<td><code>LogDensity</code></td>
<td>Construct an MCMC model from a log probability density function.</td>
</tr>
<tr>
<td><code>Simulation</code></td>
<td>Construct an MCMC model from a simulation function.</td>
</tr>
<tr>
<td><code>MVNormal</code></td>
<td>Multivariate normal negative log likelihood function</td>
</tr>
<tr>
<td><code>Mixture</code></td>
<td>Create a mixture model from a list of weighted density models.</td>
</tr>
</tbody>
</table>

MCMC model types
5.16.1 Usage

First create a `bumps.dream.bounds.Bounds` object. This stores the ranges available on the parameters, and controls how values outside the range are handled:

```python
M_bounds = bounds(minx, maxx, style='reflect|clip|fold|randomize|none')
```

For simple functions you can use one of the existing models.

If your model $f$ computes the probability density, use `Density`:

```python
M = Density(f, bounds=M_bounds)
```

If your model $f$ computes the log probability density, use `LogDensity`:

```python
M = LogDensity(f, bounds=M_bounds)
```

If your model $f$ computes a simulation which returns a vector, and you have data associated with the simulation, use `Simulation`:

```python
M = Simulation(f, data=data, bounds=M_bounds)
```

The measurement data can have a 1-sigma uncertainty associated with it, as well as a gamma factor if the uncertainty distribution has non-Gaussian kurtosis associated with it.

Multivariate normal distribution:

```python
M = MVNormal(mu, sigma)
```

Mixture models:

```python
M = Mixture(M1, w1, M2, w2, ...)
```

For more complex functions, you can subclass `MCMCModel`:

```python
class Model(MCMCModel):
    def __init__(self, ..., bounds=None, ...):
        ...
        self.bounds = bounds
        ...
    def nlnf(self, x):
        "Return the negative log likelihood of seeing x"
        p = probability of seeing x
        return -log(p)
M = Model(..., bounds=M_bounds, ...)
```

The MCMC program uses only two methods from the model:

```python
apply_bounds(pop)
log_density(pop)
```

If your model provides these methods, you will not need to subclass `MCMCModel` in order to interact with DREAM.

5.16.2 Compatibility with matlab DREAM

First generate a bounds handling function:
M_bounds = bounds(ParRange.minn, ParRange.maxn)

Then generate a model, depending on what kind of function you have.

Option 1. Model directly computes posterior density:

```python
model = Density(f, bounds=M_bounds)
```

Option 2. Model computes simulation, data has known 1-sigma uncertainty:

```python
model = Simulation(f, data=Measurement.MeasData, bounds=M_bounds, sigma=Measurement.Sigma, gamma = MCMCPar.Gamma)
```

Option 3. Model computes simulation, data has unknown 1-sigma uncertainty:

```python
model = Simulation(f, data=Measurement.MeasData, bounds=M_bounds, gamma = MCMCPar.Gamma)
```

Option 4. Model directly computes log posterior density:

```python
model = LogDensity(f, bounds=M_bounds)
```

Option 5 is like option 2 but the reported likelihoods do not take the 1-sigma uncertainty into account. The metropolis steps are still based on the 1-sigma uncertainty, so use the style given in option 2 for this case.

```python
class bumps.dream.model.MCMCModel
    Bases: object
    MCMCM model abstract base class.
    Each model must have a negative log likelihood function which operates on a point x, returning the negative log likelihood, or inf if the point is outside the domain.

    bounds = None
    labels = None
    log_density(x)
    map(pop)
    nllf(x)
    plot(x)

class bumps.dream.model.Density(f, bounds=None, labels=None)
    Bases: bumps.dream.model.MCMCModel
    Construct an MCMC model from a probability density function.
    f is the density function

    bounds = None
    labels = None
    log_density(x)
    map(pop)
    nllf(x)
    plot(x)
```
class bumps.dream.model.LogDensity(f, bounds=None, labels=None)
    Bases: bumps.dream.model.MCMCModel
    Construct an MCMC model from a log probability density function.
    
f is the log density function
    
    bounds = None
    labels = None
    log_density(x)
    map(pop)
    nllf(x)
    plot(x)

class bumps.dream.model.Simulation(f=None, bounds=None, data=None, sigma=1, gamma=0, labels=None)
    Bases: bumps.dream.model.MCMCModel
    Construct an MCMC model from a simulation function.
    
f is the function which simulates the data data is the measurement(s) to compare it to sigma is the 1-sigma uncertainty of the measurement(s). gamma in (-1, 1] represents kurtosis on the data measurement uncertainty.
    
    Data is assumed to come from an exponential power density:

\[
p(v|S, G) = \frac{w(G)}{S} \exp\left(-c(G) \frac{|v|}{S}^{(2/(1+G))}\right)
\]

where S is sigma and G is gamma.

The values of sigma and gamma can be uniform or can vary with the individual measurement points.

Certain values of gamma select particular distributions:

- G = 0: normal
- G = 1: double exponential
- G -> -1: uniform

    bounds = None
    labels = None
    log_density(x)
    map(pop)
    nllf(x)
    plot(x)

class bumps.dream.model.MVNormal(mu, sigma)
    Bases: bumps.dream.model.MCMCModel
    multivariate normal negative log likelihood function

    bounds = None
    labels = None
    log_density(x)
    map(pop)
    nllf(x)
    plot(x)
class bumps.dream.model.Mixture(*args)

Create a mixture model from a list of weighted density models.

MixtureModel( M1, w1, M2, w2, ... )

Models M1, M2, ... are MCMC models with M.nllf(x) returning the negative log likelihood of x. Weights w1, w2, ... are arbitrary scalars.

bounds = None
labels = None
log_density(x)
map(pop)
nllf(x)
plot(x)

5.17 outliers - Chain outlier tests

identify_outliers

Determine which chains have converged on a local maximum much lower than the maximum likelihood.

Chain outlier tests.

bumps.dream.outliers.identify_outliers(test, llf, x=None)

Determine which chains have converged on a local maximum much lower than the maximum likelihood.

test is the name of the test to use (one of IQR, Grubbs, Mahal or none). IQR rejects any chains with mean log likelihood more than twice the inter-quartile range below the value of the 25% quartile. The Grubbs method uses a t-test to determine which chains have a mean log likelihood extremely far below the mean across all the chains. The Mahal test looks at the head of the chain with the worst mean log likelihood and marks it as an outlier if it is far from the centroid of the population. This assumes that the posterior is approximately gaussian, which is not true in general.

llf is a set of log likelihood values for all chains, which is an array of shape (chain len, num chains)

x is the current population with one point for each each, which is an array of shape (num chains, num vars). This is only used for the Mahal test.

Returns an integer array of outlier indices.

5.18 state - Sampling history for MCMC

MCMCDraw
load_state
save_state

Sampling history for MCMC.

MCMC keeps track of a number of things during sampling.
The results may be queried as follows:

<table>
<thead>
<tr>
<th>Function</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>draws, generation, thinning</td>
<td>draws, generation, thinning</td>
</tr>
<tr>
<td>sample(condition)</td>
<td>draws, points, logp</td>
</tr>
<tr>
<td>logp()</td>
<td>draws, logp</td>
</tr>
<tr>
<td>acceptance_rate()</td>
<td>draws, AR</td>
</tr>
<tr>
<td>chains()</td>
<td>draws, chains, logp</td>
</tr>
<tr>
<td>CR_weight()</td>
<td>draws, CR_weight</td>
</tr>
<tr>
<td>best()</td>
<td>best_x, best_logp</td>
</tr>
<tr>
<td>outliers()</td>
<td>outliers</td>
</tr>
<tr>
<td>show()</td>
<td>save(file)/load(file)</td>
</tr>
</tbody>
</table>

Data is stored in circular arrays, which keeps the last N generations and throws the rest away.

draws is the total number of draws from the sampler.

generation is the total number of generations.

thinning is the number of generations per stored sample.

draws[i] is the number of draws including those required to produce the information in the corresponding return vector. Note that draw numbers need not be linearly spaced, since techniques like delayed rejection will result in a varying number of samples per generation.

logp[i] is the set of log likelihoods, one for each member of the population. The logp() method returns the complete set, and the sample() method returns a thinned set, with on element of logp[i] for each vector point[i, :].

AR[i] is the acceptance rate at generation i, showing the proportion of proposed points which are accepted into the population.

chains[i, :, :] is the set of points in the differential evolution population at thinned generation i. Ideally, the thinning rate of the MCMC process is chosen so that thinned generations i and i+1 are independent samples from the posterior distribution, though there is a chance that this may not be the case, and indeed, some points in generation i+1 may be identical to those in generation i. Actual generation number is i*thinning.

points[i, :] is the ith point in a returned sample. The i is just a place holder; there is no inherent ordering to the sample once they have been extracted from the chains. Note that the sample may be from a marginal distribution.

R[i] is the Gelman R statistic measuring convergence of the Markov chain.

CR_weight[i] is the set of weights used for selecting between the crossover ratios available to the candidate generation process of differential evolution. These will be fixed early in the sampling, even when adaptive differential evolution is selected.

outliers[i] is a vector containing the thinned generation number at which an outlier chain was removed, the id of the chain that was removed and the id of the chain that replaced it. We leave it to the reader to decide if the cloned samples, point[:generation, :, removed_id], should be included in further analysis.

best_logp is the highest log likelihood observed during the analysis and best_x is the corresponding point at which it was observed.

generation is the last generation number

class bumps.dream.state.MCMCDraw (Ngen, Nthin, Nupdate, Nvar, Npop, Ncr, thinning)

    Bases: object

    CR_weight()
        Return the crossover ratio weights to be used in the next generation.

        For example, to see if the adaptive CR is stable use:

        ```r
        draw, weight = state.CR_weight()
        plot(draw, weight)
        ```
See `crossover` for details.

**Ncr**
Number of parameters in the fit

**Ngen**

**Npop**

**Nsamples**

**Nthin**

**Nupdate**

**Nvar**

**acceptance_rate()**
Return the iteration number and the acceptance rate for that iteration.

For example, to plot the acceptance rate over time:

```python
draw, AR = state.acceptance_rate()
plot(draw, AR)
```

**best()**
Return the best point seen and its log likelihood.

**chains()**
Returns the observed Markov chains and the corresponding likelihoods.

The return value is a tuple `(draws, chains, logp)`.

- `draws` is the number of samples taken up to and including the samples for the current generation.
- `chains` is a three dimensional array of generations X chains X vars giving the set of points observed for each chain in every generation. Only the thinned samples are returned.
- `logp` is a two dimensional array of generation X population giving the log likelihood of observing the set of variable values given in chains.

**derive_vars(fn, labels=None)**
Generate derived variables from the current sample, adding columns for the derived variables to each sample of every chain.

The new columns are treated as part of the sample.

- `fn` is a function taking points `p[:, k]` for `k` in 0 . . . samples and returning a set of derived variables `pj[k]` for each sample `k`. The variables can be returned as any kind of sequence including an array or a tuple with one entry per variable. The caller uses `asarray` to convert the returned variables into a `vars X samples` array. For convenience, a single variable can be returned by itself.
- `labels` are the labels to use for the derived variables.

The following example adds the new variable `x+y = P[0] + P[1]`:

```python
state.derive_vars(lambda p: p[0]+p[1], labels="x+y")
```

**draw(portion=1.0, vars=None, selection=None, thin=1)**
Return a sample from the posterior distribution.

- `portion` is the portion of each chain to use
- `vars` is a list of variables to return for each point
selection sets the range each parameter in the returned distribution, using \{variable: (low, high)\}. Missing variables use the full range.

thin takes every nth item.

To plot the distribution for parameter p1:

```python
draw = state.draw()
hist(draw.points[:, 0])
```

To plot the interdependence of p1 and p2:

```python
draw = state.sample()
plot(draw.points[:, 0], draw.points[:, 1], '.')
```

entropy (vars=None, portion=1.0, selection=None, n_est=10000, thin=None, method=None)

Return entropy estimate and uncertainty from an MCMC draw.

portion is the portion of each chain to use

vars is the set of variables to marginalize over. It is None for the visible variables, or a list of variables.

vars is the list of variables to use for marginalization.

selection sets the range each parameter in the returned distribution, using \{variable: (low, high)\}. Missing variables use the full range.

n_est is the number of points to use from the draw when estimating the entropy (default=10000).

thin is the amount of thinning to use when selecting points from the draw.

method determines which entropy calculation to use:

- gmm: fit sample to a gaussian mixture model (GMM) with \(5\sqrt{d}\) components where \(d\) is the number fitted parameters and estimate entropy by sampling from the GMM.
- llf: estimates likelihood scale factor from ratio of density estimate to model likelihood, then computes Monte Carlo entropy from sample; this does not work for marginal likelihood estimates. DOI:10.1109/CCA.2010.5611198
- mvn: fit sample to a multi-variate Gaussian and return the entropy of the best fit gaussian; uses bootstrap to estimate uncertainty.
- wnn: estimate entropy from nearest-neighbor distances in sample. DOI:10.1214/18-AOS1688

gelman ()

Compute the R-statistic for the current frame

keep_best ()

Place the best point at the end of the last good chain.

Good chains are defined by mark_outliers.

Because the Markov chain is designed to wander the parameter space, the best individual seen during the random walk may have been observed during the burn-in period, and may no longer be present in the chain. If this is the case, replace the final point with the best, otherwise swap the positions of the final and the best.

labels

logp (full=False)

Return the iteration number and the log likelihood for each point in the individual sequences in that iteration.

For example, to plot the convergence of each sequence:
draw, logp = state.logp()
plot(draw, logp)

Note that draw[i] represents the total number of samples taken, including those for the samples in logp[i]. If full is True, then return all chains, not just good chains.

**logp_slice** *(n)*
Return a slice of the logp chains, either the first \( n \) if \( n > 0 \) or the last \( n \) if \( n < 0 \). Avoids unrolling the circular buffer if possible.

**mark_outliers** *(test= ‘IQR’, portion=1.0)*
Mark some chains as outliers but don’t remove them. This can happen after drawing is complete, so that chains that did not converge are not included in the statistics.

*test* is ‘IQR’, ‘Mahol’ or ‘none’.
*portion* indicates what portion of the samples should be included in the outlier test. The default is to include all of them.

**min_slice** *(n)*
Return the minimum logp for \( n \) slices, from the head if positive or the tail if negative.

This is a specialized function so it can be fast. Convergence can be quickly rejected if the min in a short head is smaller than the min in a long tail. Unfortunately, if the data is wrapped, then the max function will cost extra.

**outliers** *
Return a list of outlier removal operations.

Each outlier operation is a tuple giving the thinned generation in which it occurred, the old chain id and the new chain id.

The chains themselves have already been updated to reflect the removal.

Curiously, it is possible for the maximum likelihood seen so far to be removed by this operation.

**remove_outliers** *(x, logp, test= ‘IQR’)*
Replace outlier chains with clones of good ones. This should happen early in the sampling processes so the clones have an opportunity to evolve their own identity. Only the head of the chain is modified.

*state* contains the chains, with log likelihood for each point.

*x, logp* are the current population and the corresponding log likelihoods; these are updated with cloned chain values.

test is the name of the test to use (one of IQR, Grubbs, Mahal or none). See **outliers**. identify_outliers() for details.

Updates *state, x* and *logp* to reflect the changes.

Returns a list of the outliers that were removed.

**resize** *(Ngen, Nthin, Nupdate, Nvar, Npop, Ncr, thinning)*

**sample** (**kw**)
Return a sample from the posterior distribution.

Deprecated use **draw()** instead.

**save** *(filename)*

**set_integer_vars** *(labels)*
Indicate that variables should be considered integer variables when computing statistics.
```
set_visible_vars(labels)
show(portion=1.0, figfile=None)
stable_best()
    Return the best point seen and its log likelihood.
title = None
trim_portion()
bumps.dream.state.load_state(filename, skip=0, report=0, derived_vars=0)
bumps.dream.state.save_state(state, filename)
```

### 5.19 stats - Statistics helper functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VarStats</td>
<td>Statistics helper functions.</td>
</tr>
<tr>
<td>var_stats</td>
<td></td>
</tr>
<tr>
<td>format_vars</td>
<td></td>
</tr>
<tr>
<td>parse_var</td>
<td>Parse a line returned by format_vars back into the statistics for the variable on that line.</td>
</tr>
<tr>
<td>stats</td>
<td>Find mean and standard deviation of a set of weighted samples.</td>
</tr>
<tr>
<td>credible_interval</td>
<td>Find the credible interval covering the portion $ci$ of the data.</td>
</tr>
<tr>
<td>shortest_credible_interval</td>
<td>Find the credible interval covering the portion $ci$ of the data.</td>
</tr>
</tbody>
</table>

Statistics helper functions.

```python
class bumps.dream.stats.VarStats(**kw):
    Bases: object
bumps.dream.stats.var_stats(draw, vars=None)
bumps.dream.stats.format_vars(all_vstats)
bumps.dream.stats.parse_var(line)
    Parse a line returned by format_vars back into the statistics for the variable on that line.
bumps.dream.stats.stats(x, weights=None)
    Find mean and standard deviation of a set of weighted samples.
    Note that the median is not strictly correct (we choose an endpoint of the sample for the case where the median falls between two values in the sample), but this is good enough when the sample size is large.
bumps.dream.stats.credible_interval(x, ci, weights=None)
    Find the credible interval covering the portion $ci$ of the data.
    $x$ are samples from the posterior distribution.
    $ci$ is a set of intervals in $[0,1]$. For a $1-\sigma$ interval use $ci=erf(1/sqrt(2))$, or 0.68. About 1e5 samples are needed for 2 digits of precision on a $1-\sigma$ credible interval. For a 95% interval, about 1e6 samples are needed for 2 digits of precision. At least 1000 points are needed for an unbiased result, otherwise the resulting interval will be shorter than expected (tested on a variety of distributions including exponential, cauchy, gaussian, beta and gamma).
```
weights is a vector of weights for each x, or None for unweighted. One could weight points according to temperature in a parallel tempering dataset.

Returns an array \([x_1\_low, x_1\_high], [l_2\_low, x_2\_high], \ldots\) where \([x_i\_low, x_i\_high]\) are the starting and ending values for credible interval \(i\).

This function is faster if the inputs are already sorted.

**bumps.dream.stats.shortest_credible_interval**(x, ci=0.95, weights=None)

Find the credible interval covering the portion \(ci\) of the data.

\(x\) are samples from the posterior distribution. \(ci\) is the interval size in (0,1], and defaults to 0.95. For a 1-sigma interval use \(ci=\text{erf}(1/\sqrt{2})\). weights is a vector of weights for each x, or None for unweighted.

Returns the minimum and maximum values of the interval. If \(ci\) is a vector, return a vector of intervals.

This function is faster if the inputs are already sorted.

About 1e6 samples are needed for 2 digits of precision on a 95% credible interval, or 1e5 for 2 digits on a 1-sigma credible interval.

To remove bias towards toward smaller intervals, the midpoints between the surrounding intervals are used as the end points.

### 5.20 tile - Split a rectangle into n panes

**max_tile_size**

Determine the maximum sized tile possible.

Split a rectangle into n panes.

**bumps.dream.tile.max_tile_size**(tile_count, rect_size)

Determine the maximum sized tile possible.

Keyword arguments: tile_count – Number of tiles to fit rect_size – 2-tuple of rectangle size as (width, height)

### 5.21 util - Miscellaneous utilities

**draw**

Select k things from a pool of n without replacement.

**console**

Start the python console with the local variables available.

Miscellaneous utilities.

**bumps.dream.util.draw**(k, n)

Select k things from a pool of n without replacement.

**bumps.dream.util.console**( )

Start the python console with the local variables available.

console() should be the last thing in the file, after sampling and showing the default plots.

### 5.22 varplot - Plot histograms for individual parameters

**5.22. varplot - Plot histograms for individual parameters** 189
Build layout for histogram plots
bumps.dream.varplot.var_plot_size(n)
bumps.dream.varplot.plot_vars(draw, all_vstats, **kw)
bumps.dream.varplot.plot_var(draw, vstats, var, cbar, nbins=30)

5.23 views - MCMC plotting methods

plot_all
plot_corr
plot_corrmatrix
plot_trace
plot_logp
format_vars

MCMC plotting methods.
bumps.dream.views.plot_all(state, portion=1.0, figfile=None)
bumps.dream.views.plot_corr(draw, vars=(0, 1))
bumps.dream.views.plot_corrmatrix(draw, nbins=50)
bumps.dream.views.plot_trace(state, var=0, portion=None)
bumps.dream.views.plot_logp(state, portion=None)
bumps.dream.views.format_vars(all_vstats)

acr ACR upper percentiles critical value for test of single multivariate normal outlier.
bounds Bounds handling.
core DiffeRential Evolution Adaptive Metropolis algorithm
corrplot 2-D correlation histograms
crossover Crossover ratios
diffev Differential evolution MCMC stepper.
entropy Estimate entropy after a fit.
exppow Exponential power density parameter calculator.
formatnum Format values and uncertainties nicely for printing.
gelman Convergence test statistic from Gelman and Rubin, 1992.[1]
initpop Population initialization routines.
ksmirnov Kolmogorov-Smirnov test for MCMC convergence.
mahal Mahalanobis distance calculator
metropolis MCMC step acceptance test.
<table>
<thead>
<tr>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>model</td>
<td>MCMC model types</td>
</tr>
<tr>
<td>outliers</td>
<td>Chain outlier tests.</td>
</tr>
<tr>
<td>state</td>
<td>Sampling history for MCMC.</td>
</tr>
<tr>
<td>stats</td>
<td>Statistics helper functions.</td>
</tr>
<tr>
<td>tile</td>
<td>Split a rectangle into n panes.</td>
</tr>
<tr>
<td>util</td>
<td>Miscellaneous utilities.</td>
</tr>
<tr>
<td>varplot</td>
<td>Build layout for histogram plots</td>
</tr>
<tr>
<td>views</td>
<td>MCMC plotting methods.</td>
</tr>
</tbody>
</table>
bumps.bounds, 79
bumps.bspline, 89
bumps.cheby, 90
bumps.cli, 91
bumps.curve, 93
bumps.data, 95
bumps.dream.acr, 165
bumps.dream.bounds, 166
bumps.dream.core, 168
bumps.dream.corrplot, 170
bumps.dream.crossover, 171
bumps.dream.diffev, 173
bumps.dream.entropy, 173
bumps.dream.exppow, 176
bumps.dream.formatnum, 176
bumps.dream.gelman, 177
bumps.dream.geweke, 177
bumps.dream.initpop, 178
bumps.dream.ksmirnov, 178
bumps.dream.mahal, 179
bumps.dream.metropolis, 179
bumps.dream.model, 179
bumps.dream.outliers, 183
bumps.dream.state, 183
bumps.dream.stats, 188
bumps.dream.tile, 189
bumps.dream.util, 189
bumps.dream.varplot, 190
bumps.dream.views, 190
bumps.errplot, 96
bumps.fitproblem, 97
bumps.fitservice, 104
bumps.fitters, 105
bumps.formatnum, 111
bumps.history, 112
bumps.initpop, 114
bumps.lsqerror, 116
bumps.mapper, 118
bumps.monitor, 119
bumps.mono, 120
bumps.names, 121
bumps.options, 122
bumps.parameter, 125
bumps.partemp, 146
bumps.pdfwrapper, 147
bumps.plotutil, 150
bumps.plugin, 151
bumps.pmath, 153
bumps.pymcfit, 154
bumps.quasineutron, 155
bumps.random_lines, 157
bumps.simplex, 158
bumps.util, 159
bumps.wsolve, 161
A

acceptance_rate() (bumps.dream.state.MCMCDraw method), 185
accumulate() (bumps.history.History method), 113
accumulate() (bumps.history.Trace method), 113
acos() (in module bumps.pmath), 153
acosd() (in module bumps.parameter), 145
acosd() (in module bumps.pmath), 154
acosh() (in module bumps.pmath), 154
ACR() (in module bumps.dream.acr), 166
adapt() (bumps.dream.crossover.AdaptiveCrossover method), 172
adapt() (bumps.dream.crossover.BaseAdaptiveCrossover method), 172
adapt() (bumps.dream.crossover.Crossover method), 172
adapt() (bumps.dream.crossover.LogAdaptiveCrossover method), 172
AdaptiveCrossover (class in bumps.dream.crossover), 172
Alias (class in bumps.parameter), 125
alpha (bumps.dream.core.Dream attribute), 120
alpha (bumps.options.BumpsOpts attribute), 122
AMQPMapper (class in bumps.mapper), 118
apply() (bumps.dream.bounds.Bounds method), 167
apply() (bumps.dream.bounds.ClipBounds method), 167
apply() (bumps.dream.bounds.FoldBounds method), 167
apply() (bumps.dream.bounds.IgnoreBounds method), 168
apply() (bumps.dream.bounds.RandomBounds method), 167
apply() (bumps.dream.bounds.ReflectionBounds method), 167
arccos() (bumps.parameter.BaseParameter method), 125
arccos() (bumps.parameter.Function method), 131
arccos() (bumps.parameter.IntegerParameter method), 133
arccos() (bumps.parameter.Operator method), 136
arccos() (bumps.parameter.Parameter method), 139
arccos() (bumps.parameter.Reference method), 142
arccosh() (bumps.parameter.BaseParameter method), 125
arccosh() (bumps.parameter.Constant method), 128
arccosh() (bumps.parameter.Function method), 131
arccosh() (bumps.parameter.IntegerParameter method), 133
arccosh() (bumps.parameter.Operator method), 136
arccosh() (bumps.parameter.Parameter method), 139
arccosh() (bumps.parameter.Reference method), 142
arcsin() (bumps.parameter.BaseParameter method), 125
arcsin() (bumps.parameter.Constant method), 128
arcsin() (bumps.parameter.Function method), 131
arcsin() (bumps.parameter.IntegerParameter method), 133
arcsin() (bumps.parameter.Operator method), 136
arcsin() (bumps.parameter.Parameter method), 139
arcsin() (bumps.parameter.Reference method), 142
arcsinh() (bumps.parameter.BaseParameter method), 125
arcsinh() (bumps.parameter.Constant method), 128
arcsinh() (bumps.parameter.Function method), 131
arcsinh() (bumps.parameter.IntegerParameter method), 133
arcsinh() (bumps.parameter.Operator method), 136
arcsinh() (bumps.parameter.Parameter method), 139
arcsinh() (bumps.parameter.Reference method), 142
arctan() (bumps.parameter.BaseParameter method), 125
arctan() (bumps.parameter.Constant method), 128
arctan() (bumps.parameter.Function method), 131
arctan()  (bumps.parameter.IntegerParameter method), 134
arctan()  (bumps.parameter.Operator method), 136
arctan()  (bumps.parameter.Parameter method), 139
arctan2d()  (bumps.parameter.Reference method), 142
arctan2d()  (in module bumps.parameter), 145
arctand()  (in module bumps.parameter), 145
arctanh()  (bumps.parameter.BaseParameter method), 125
arctanh()  (bumps.parameter.Constant method), 128
arctanh()  (bumps.parameter.Function method), 131
arctanh()  (bumps.parameter.IntegerParameter method), 134
arctanh()  (bumps.parameter.Operator method), 136
arctanh()  (bumps.parameter.Parameter method), 139
arctanh()  (bumps.parameter.Reference method), 142
args()  (bumps.parameter.Function attribute), 131
asin()  (in module bumps.pmath), 153
asind()  (in module bumps.parameter), 145
asind()  (in module bumps.pmath), 154
asinh()  (in module bumps.pmath), 154
atan()  (in module bumps.pmath), 153
atan2()  (in module bumps.pmath), 153
atan2d()  (in module bumps.parameter), 145
atan2d()  (in module bumps.pmath), 154
atand()  (in module bumps.parameter), 145
atand()  (in module bumps.pmath), 154
atanh()  (in module bumps.pmath), 154
auto_shift()  (in module bumps.plotutil), 150

B
BaseAdaptiveCrossover  (class in bumps.dream.crossover), 172
BaseFitProblem  (class in bumps.fitproblem), 99
BaseParameter  (class in bumps.parameter), 125
best()  (bumps.dream.state.MCMCDraw method), 185
BFGSFit  (class in bumps.fitters), 105
Bounded  (class in bumps.bounds), 83
BoundedAbove  (class in bumps.bounds), 84
BoundedBelow  (class in bumps.bounds), 85
BoundedNormal  (class in bumps.bounds), 88
bounds()  (bumps.dream.model.Density attribute), 181
bounds()  (bumps.dream.model.LogDensity attribute), 182
bounds()  (bumps.dream.model.MCMCModel attribute), 181
bounds()  (bumps.dream.model.Mixture attribute), 183
bounds()  (bumps.dream.model.MVNormal attribute), 182
bounds()  (bumps.dream.model.Simulation attribute), 182
bounds()  (bumps.fitters.DreamModel attribute), 106
bounds()  (bumps.parameter.BaseParameter attribute), 125
bounds()  (bumps.parameter.Constant attribute), 128
bounds()  (bumps.parameter.Function attribute), 131
bounds()  (bumps.parameter.IntegerParameter attribute), 134
bounds()  (bumps.parameter.Operator attribute), 136
bounds()  (bumps.parameter.Parameter attribute), 139
bounds()  (bumps.parameter.Reference attribute), 142
Bounds  (class in bumps.bounds), 81
Bounds  (class in bumps.dream.bounds), 167
bounds()  (bumps.fitproblem.BaseFitProblem method), 99
bounds()  (bumps.fitproblem.MultiFitProblem method), 102
bounds()  (bumps.pdfwrapper.DirectProblem method), 148
bounds()  (bumps.pymcfit.PyMCProblem method), 155
bounds_style()  (bumps.dream.core.Dream attribute), 170
boxed_function()  (in module bumps.parameter), 145
bspline()  (in module bumps.bspline), 89
bumps.bounds()  (module), 79
bumps.bspline()  (module), 89
bumps.cheby()  (module), 90
bumps.cli()  (module), 91
bumps.curve()  (module), 93
bumps.data()  (module), 95
bumps.dream.acr()  (module), 165
bumps.dream.bounds()  (module), 166
bumps.dream.core()  (module), 168
bumps.dream.corrplot()  (module), 170
bumps.dream.crossover()  (module), 171
bumps.dream.diffev()  (module), 173
bumps.dream.entropy()  (module), 173
bumps.dream.exppow()  (module), 176
bumps.dream.formatnum()  (module), 176
bumps.dream.gelman()  (module), 177
bumps.dream.geweke()  (module), 177
bumps.dream.initpop()  (module), 178
bumps.dream.ksmirnov()  (module), 178
bumps.dream.mahal()  (module), 179
bumps.dream.metropolis()  (module), 179
bumps.dream.model()  (module), 179
bumps.dream.outliers()  (module), 183
bumps.dream.state()  (module), 183
bumps.dream.stats()  (module), 188
bumps.dream.tile()  (module), 189
bumps.dream.util()  (module), 189
bumps.dream.varplot()  (module), 190
bumps.dream.views()  (module), 190
bumps.errplot()  (module), 96
bumps.fitproblem()  (module), 97
bumps.fitservice()  (module), 104
bumps.fitters()  (module), 105
bumps.formatnum()  (module), 111
bumps.history()  (module), 112
config_history() (bumps.monitor.TimedUpdate method), 120
config_matplotlib() (in module bumps.cli), 92
console() (in module bumps.dream.util), 189
ConsoleMonitor (class in bumps.fitters), 106
Constant (class in bumps.parameter), 128
Constraint (class in bumps.parameter), 130
constraints_nllf() (bumps.fitproblem.BaseFitProblem method), 100
constraints_nllf() (bumps.fitproblem.MultiFitProblem method), 102
correlated_colors() (in module bumps.plotutil), 150
corr() (in module bumps.lsqerror), 117
Corr2d (class in bumps.dream.corrplot), 170
cos() (bumps.parameter.BaseParameter method), 126
cos() (bumps.parameter.Parameter method), 129
cos() (bumps.parameter.Parameter method), 131
cos() (bumps.parameter.IntegerParameter method), 134
cos() (bumps.parameter.Operator method), 136
cos() (bumps.parameter.Parameter method), 139
cos() (bumps.parameter.Parameter method), 141
cos() (bumps.parameter.Parameter method), 143
cosh() (bumps.parameter.BaseParameter method), 126
cosh() (bumps.parameter.Constant method), 128
cosh() (bumps.parameter.Function method), 131
cosh() (bumps.parameter.IntegerParameter method), 134
cosh() (bumps.parameter.Operator method), 137
cosh() (bumps.parameter.Parameter method), 139
cosh() (bumps.parameter.Parameter method), 141
cosh() (bumps.parameter.Parameter method), 143
cosh() (in module bumps.pmath), 154
count_inflections() (in module bumps.mono), 121
cov (bumps.wsolve.LinearModel attribute), 163
cov (bumps.wsolve.PolynomialModel attribute), 163
cov() (bumps.fitproblem.BaseFitProblem method), 100
cov() (bumps.fitproblem.BaseFitProblem method), 100
cov() (bumps.fitproblem.MultiFitProblem method), 102
cov() (bumps.fitters.FitDriver method), 107
cov() (bumps.fitters.LevenbergMarquardtFit method), 108
cov_entropy() (in module bumps.dream.entropy), 175
cov_init() (in module bumps.dream.initpop), 178
cov_init() (in module bumps.initpop), 178
CR (bumps.dream.core.Dream attribute), 170
CR_spacing (bumps.dream.core.Dream attribute), 170
CR_weight() (bumps.dream.state.MCMCDraw method), 184
credible_interval() (in module bumps.dream.stats), 188
Crossover (class in bumps.dream.crossover), 171
current() (in module bumps.parameter), 145
curve() (in module bumps.curve), 93

data_view() (in module bumps.plugin), 152
der() (bumps.wsolve.PolynomialModel method), 163
degree() (bumps.wsolve.PolynomialModel attribute), 163
degrees() (bumps.parameter.BaseParameter method), 126
degrees() (bumps.parameter.Constant method), 128
degrees() (bumps.parameter.Function method), 131
degrees() (bumps.parameter.IntegerParameter method), 134
degrees() (bumps.parameter.Operator method), 137
degrees() (bumps.parameter.Parameter method), 139
degrees() (bumps.parameter.Parameter method), 141
degrees() (bumps.parameter.Parameter method), 143
degrees() (in module bumps.pmath), 154
demo_hessian() (in module bumps.lsqerror), 117
demo_jacobian() (in module bumps.lsqerror), 117
demo_stderr_hilbert() (in module bumps.lsqerror), 117
demo_stderr_perturbed() (in module bumps.lsqerror), 117
density() (in module bumps.dream.model), 181
derive_vars() (bumps.dream.state.MCMCDraw method), 185
dev() (bumps.parameter.BaseParameter method), 126
dev() (bumps.parameter.Constant method), 128
dev() (bumps.parameter.Function method), 131
dev() (bumps.parameter.IntegeParameter method), 134
dev() (bumps.parameter.Operator method), 137
dev() (bumps.parameter.Parameter method), 139
dev() (bumps.parameter.Reference method), 143
dhs() (in module bumps.plotutil), 150
DirectProblem (class in bumps.pdfwrapper), 147

198

Index
Bumps: Curve Fitting and Uncertainty Analysis, Release 0.9.0

discrete (bumps.parameter.BaseParameter attribute), 126
discrete (bumps.parameter.Constant attribute), 128
discrete (bumps.parameter.Function attribute), 131
discrete (bumps.parameter.IntegerParameter attribute), 134
discrete (bumps.parameter.Operator attribute), 137
discrete (bumps.parameter.Parameter attribute), 140
discrete (bumps.parameter.Reference attribute), 143
Distribution (class in bumps.bounds), 86
DoF (bumps.wsolve.LinearModel attribute), 162
DoF (bumps.wsolve.PolynomialModel attribute), 163
DR_scale (bumps.dream.core.Dream attribute), 170
draw() (bumps.dream.core.Dream method), 170
draw() (in module bumps.state.MCMCDraw method), 185
draw() (in module bumps.state.MCMCDraw method), 185
draws (bumps.dream.core.Dream attribute), 170
Dream (class in bumps.dream.core), 169
DreamFit (class in bumps.fitters), 106
DreamModel (class in bumps.fitters), 106
dvalue (bumps.parameter.Operator attribute), 137

E
entropy (bumps.options.BumpsOpts attribute), 122
entropy() (bumps.dream.core.Dream method), 186
entropy() (bumps.fitters.DreamFit method), 106
entropy() (bumps.fitters.FitDriver method), 108
entropy() (in module bumps.dream.entropy), 174
eps_init() (in module bumps.initentro), 115
error_plot() (bumps.fitters.DreamFit method), 106
exp() (bumps.parameter.BaseParameter method), 126
exp() (bumps.parameter.Constant method), 128
exp() (bumps.parameter.Function method), 131
exp() (bumps.parameter.IntegerParameter method), 134
exp() (bumps.parameter.Operator method), 137
exp() (bumps.parameter.Parameter method), 140
exp() (bumps.parameter.Reference method), 143
exp() (in module bumps.pmath), 153
expm1() (bumps.parameter.BaseParameter method), 126
expm1() (bumps.parameter.Constant method), 128
expm1() (bumps.parameter.Function method), 132
expm1() (bumps.parameter.IntegerParameter method), 134
expm1() (bumps.parameter.Operator method), 137
expm1() (bumps.parameter.Parameter method), 140
expm1() (bumps.parameter.Reference method), 143
exppow_pars() (in module bumps.dream.exppow), 176
feasible() (bumps.parameter.IntegerParameter method), 134
feasible() (bumps.parameter.Parameter method), 140
feasible() (bumps.parameter.Reference method), 143
FIELDS (bumps.fitters.StepMonitor attribute), 110
fit (bumps.options.BumpsOpts attribute), 122
fit() (bumps.fitters.FitDriver method), 108
fit() (in module bumps.fitters), 110
fit_config (bumps.options.BumpsOpts attribute), 122
FIT_CONFIG (in module bumps.options), 122
FitBase (class in bumps.fitters), 107
FitConfig (class in bumps.options), 122
FitDriver (class in bumps.fitters), 107
Fitness (class in bumps.fitproblem), 98
fitservice() (in module bumps.fitservice), 104
fittable (bumps.parameter.BaseParameter attribute), 126
fittable (bumps.parameter.Constant attribute), 128
fittable (bumps.parameter.Function attribute), 132
fittable (bumps.parameter.IntegerParameter attribute), 134
fittable (bumps.parameter.Operator attribute), 137
fittable (bumps.parameter.Parameter attribute), 140
fittable (bumps.parameter.Reference attribute), 143
fittable() (in module bumps.parameter), 145
fixed (bumps.parameter.BaseParameter attribute), 126
fixed (bumps.parameter.Constant attribute), 128
fixed (bumps.parameter.Function attribute), 132
fixed (bumps.parameter.IntegerParameter attribute), 134
fixed (bumps.parameter.Operator attribute), 137
fixed (bumps.parameter.Parameter attribute), 140
fixed (bumps.parameter.Reference attribute), 143
FLAGS (bumps.options.BumpsOpts attribute), 122
FLAGS (bumps.options.ParseOpts attribute), 123
flatten() (in module bumps.pmath), 145
floor() (bumps.parameter.BaseParameter method), 126
floor() (bumps.parameter.Constant method), 129
floor() (bumps.parameter.Function method), 132
floor() (bumps.parameter.IntegerParameter method), 134
floor() (bumps.parameter.Operator method), 137
floor() (bumps.parameter.Parameter method), 140
floor() (bumps.parameter.Reference method), 143
FoldBounds (class in bumps.dream.bounds), 167
form_quantiles() (in module bumps.plotutil), 151
format() (bumps.parameter.BaseParameter method), 126
format() (bumps.parameter.Constant method), 129
format() (bumps.getParameter.Function method), 132

Index
id (bumps.fitters.PTFit attribute), 109
id (bumps.fitters.RLFit attribute), 109
id (bumps.fitters.SimplexFit attribute), 110
id (bumps.fitters.SnobFit attribute), 110
identify_outliers() (in module bumps.dream.outliers), 183
IgnoreBounds (class in bumps.dream.bounds), 168
IMPLICIT_VALUES (bumps.options.BumpsOpts attribute), 122
IMPLICIT_VALUES (bumps.options.ParseOpts attribute), 123
indfloat () (in module bumps.data), 95
init_bounds () (in module bumps.bounds), 81
install_plugin () (in module bumps.cli), 92
IntegerParameter (class in bumps.parameter), 133
J
jacobian() (in module bumps.lsqerror), 117
jacobian_cov() (in module bumps.lsqerror), 117
K
kbhit () (in module bumps.util), 159
keep_best () (bumps.dream.state.MCMCDraw method), 186
ksmirnov () (in module bumps.dream.ksmirnov), 178
kw (bumps.parameter.Function attribute), 132
L
labels (bumps.dream.model.Density attribute), 181
labels (bumps.dream.model.LogDensity attribute), 182
labels (bumps.dream.model.MCMCModel attribute), 181
labels (bumps.dream.model.Mixture attribute), 183
labels (bumps.dream.model.MVNormal attribute), 182
labels (bumps.dream.model.Simulation attribute), 182
labels (bumps.dream.state.MCMCDraw attribute), 186
labels (bumps.fitters.DreamModel attribute), 106
labels () (bumps.fitproblem.BaseFitProblem method), 100
labels () (bumps.fitproblem.MultiFitProblem method), 102
labels () (bumps.pdfwrapper.DirectProblem method), 148
labels () (bumps.pymcfit.PyMCMCPMethod method), 155
LevenbergMarquardtFit (class in bumps.fitters), 108
lhs_init () (in module bumps.dream.initpop), 178
lhs_init () (in module bumps.initpop), 115
limits (bumps.bounds.Bounded attribute), 83
limits (bumps.bounds.BoundedAbove attribute), 84
limits (bumps.bounds.BoundedBelow attribute), 85
limits (bumps.bounds.BoundedNormal attribute), 88
limits (bumps.bounds.Bounded attribute), 81
limits (bumps.bounds.Distribution attribute), 86
limits (bumps.bounds.Normal attribute), 87
limits (bumps.bounds.SoftBounded attribute), 89
limits (bumps.bounds.Unbounded attribute), 82
LinearModel (class in bumps.wsolve), 162
load () (bumps.fitters.DEFit method), 106
load () (bumps.fitters.DreamFit method), 106
load () (bumps.fitters.FitDriver method), 108
load_best () (in module bumps.cli), 92
load_history () (in module bumps.fitters), 110
load_model () (in module bumps.cli), 92
load_model () (in module bumps.fitters), 152
load_problem () (in module bumps.fitproblem), 99
load_state () (in module bumps.dream.state), 188
log () (bumps.parameter.BaseParameter method), 126
log () (bumps.parameter.Constant method), 129
log () (bumps.parameter.Function method), 132
log () (bumps.parameter.IntegerParameter method), 134
log () (bumps.parameter.Operator method), 137
log () (bumps.parameter.Parameter method), 140
log () (bumps.parameter.Reference method), 143
log () (in module bumps.pmath), 153
log10() (bumps.parameter.BaseParameter method), 126
log10() (bumps.parameter.Constant method), 129
log10() (bumps.parameter.Function method), 132
log10() (bumps.parameter.IntegerParameter method), 135
log10() (bumps.parameter.Operator method), 137
log10() (bumps.parameter.Parameter method), 140
log10() (bumps.parameter.Reference method), 143
log() (in module bumps.pmath), 153
loglp() (bumps.parameter.BaseParameter method), 126
loglp() (bumps.parameter.Constant method), 129
loglp() (bumps.parameter.Function method), 132
loglp() (bumps.parameter.IntegerParameter method), 135
loglp() (bumps.parameter.Operator method), 137
loglp() (bumps.parameter.Parameter method), 140
loglp() (bumps.parameter.Reference method), 143
log_density() (bumps.dream.model.Density method), 181
log_density() (bumps.dream.model.LogDensity method), 181
log_density() (bumps.dream.model.MCMCModel method), 181
log_density() (bumps.dream.model.Mixture method), 183
log_density() (bumps.dream.model.MVNormal method), 182
log_density() (bumps.dream.model.Simulation method), 182

Index 201
log_density () (bumps.fitters.DreamModel method), 106
LogAdaptiveCrossover (class in bumps.dream.core), 172
LogDensity (class in bumps.dream.core), 181
Logger (class in bumps.monitor), 120
logp () (bumps.dream.core.MCMCDraw method), 186
logp_slice () (bumps.dream.core.MCMCDraw method), 187
LogNormal () (bumps.dream.core.Dream method), 106
LogNormal (class in bumps.dream.core), 106
logp() (bumps.dream.core.MCMCDraw method), 186
logp_slice() (bumps.dream.core.MCMCDraw method), 187
logp_slice (bumps.dream.core.ClipBounds attribute), 167
logp_slice (bumps.dream.core.IgnoreBounds attribute), 167
logp_slice (bumps.dream.core.RandomBounds attribute), 168
logp_slice (bumps.dream.core.ReflectBounds attribute), 167

M
mahalanobis () (in module bumps.dream.mahal), 179
main () (in module bumps.cli), 92
make_bounds_handler () (in module bumps.dream.bounds), 166
manager (bumps.mapper.MPMapper attribute), 119
map () (bumps.dream.core.Density method), 181
map () (bumps.dream.core.LogDensity method), 182
map () (bumps.dream.core.MCMCDraw method), 181
map () (bumps.dream.core.Mixture method), 183
map () (bumps.dream.core.MVNormal method), 182
map () (bumps.dream.core.MVNormal method), 182
map () (bumps.fitters.DreamModel method), 106
mark_outliers () (bumps.dream.core.MCMCDraw method), 187
max_correlation () (in module bumps.lsqerror), 117
max_tile_size () (in module bumps.dream.tile), 189
MCMCDraw (class in bumps.dream.core), 184
MCMCDraw (class in bumps.dream.core), 184
meshsteps (bumps.options.BumpsOpts attribute), 122
metropolis () (in module bumps.dream.metropolis), 179
metropolis_dr () (in module bumps.dream.metropolis), 179
min_slice () (bumps.dream.core.MCMCDraw method), 187
MINARGS (bumps.options.BumpsOpts attribute), 122
MINARGS (bumps.options.ParseException attribute), 124
Mixture (class in bumps.dream.core), 182
model () (bumps.dream.core.Dream attribute), 170
model_nllf() (bumps.fitproblem.BaseFitProblem method), 100
model_nllf() (bumps.fitproblem.MultiFitProblem method), 102
model_parameters() (bumps.fitproblem.BaseFitProblem method), 102
model_parameters() (bumps.fitproblem.MultiFitProblem method), 102
model_points() (bumps.fitproblem.MultiFitProblem method), 100
model_points() (bumps.fitproblem.BaseFitProblem method), 100
model_reset() (bumps.fitproblem.MCMCDraw method), 102
model_reset() (bumps.fitproblem.MultiFitProblem method), 102
model_reset() (bumps.pdfwrapper.DirectProblem method), 148
model_reset() (bumps.pdfwrapper.DirectProblem method), 148
model_reset() (bumps.pdfwrapper.DirectProblem method), 148
model_reset() (bumps.pdfwrapper.DirectProblem method), 148
model_update() (bumps.fitproblem.MultiFitProblem method), 100
model_update() (bumps.fitproblem.BaseFitProblem method), 102
model_update() (bumps.fitproblem.MultiFitProblem method), 102
model_update() (bumps.pdfwrapper.DirectProblem method), 148
model_update() (bumps.pdfwrapper.DirectProblem method), 148
model_view () (in module bumps.plugin), 152
models (bumps.fitproblem.MultiFitProblem attribute), 102
Monitor (class in bumps.monitor), 119
MonitorRunner (class in bumps.fitters), 109
monospline () (in module bumps.mono), 120
MPFit (class in bumps.fitters), 108
MPIMapper (class in bumps.mapper), 118
MPIMapper (class in bumps.mapper), 118
MultiFitProblem (class in bumps.fitproblem), 101
MultiStart (class in bumps.fitters), 109
MVNEntropy (class in bumps.dream.entropy), 175
MVNormal (class in bumps.dream.core), 182

N
name (bumps.fitters.BFGSFit attribute), 105
name (bumps.fitters.DEFit attribute), 106
name (bumps.fitters.DreamFit attribute), 106
name (bumps.fitters.LevenbergMarquardtFit attribute), 108
name (bumps.fitters.MPFit attribute), 108
name (bumps.fitters.MultiFit attribute), 108
name (bumps.fitters.MultiStart attribute), 109
name (bumps.fitters.PSFit attribute), 109
name (bumps.fitters.RLFit attribute), 109
name (bumps.fitters.SimplexFit attribute), 110
name (bumps.fitters.SnoBfit attribute), 110
name (bumps.parameter.BaseParameter attribute), 126
name (bumps.parameter.Constant attribute), 129

202 Index
name (bumps.parameter.Function attribute), 132
name (bumps.parameter.IntegerParameter attribute), 135
name (bumps.parameter.Operator attribute), 137
name (bumps.parameter.Parameter attribute), 140
name (bumps.parameter.Reference attribute), 143
namespace (bumps.mapper.MPMapper attribute), 119
Ncr (bumps.dream.state.MCMCDraw attribute), 185
new_model() (in module bumps.plugin), 152
next_color() (in module bumps.plotutil), 150
Ngen (bumps.dream.state.MCMCDraw attribute), 185
nice() (in module bumps.mapper), 119
nice_range() (in module bumps.bounds), 81
nllf() (bumps.bounds.Bounded method), 83
nllf() (bumps.bounds.BoundedAbove method), 84
nllf() (bumps.bounds.BoundedBelow method), 85
nllf() (bumps.bounds.BoundedNormal method), 88
nllf() (bumps.bounds.Bounded method), 81
nllf() (bumps.bounds.Distribution method), 86
nllf() (bumps.bounds.Normal method), 87
nllf() (bumps.bounds.SoftBounded method), 89
nllf() (bumps.bounds.Unbounded method), 82
nllf() (bumps.curve.Curve method), 94
nllf() (bumps.curve.PoissonCurve method), 94
nllf() (bumps.dream.model.Density method), 181
nllf() (bumps.dream.model.LogDensity method), 182
nllf() (bumps.dream.model.MCMCMCModel method), 181
nllf() (bumps.dream.model.Mixture method), 183
nllf() (bumps.dream.model.MVNormal method), 182
nllf() (bumps.dream.model.Simulation method), 182
nllf() (bumps.fitproblem.BaseFitProblem method), 100
nllf() (bumps.fitproblem.Fitness method), 98
nllf() (bumps.fitproblem.MultiFitProblem method), 103
nllf() (bumps.fitters.DreamModel method), 107
nllf() (bumps.parameter.BaseParameter method), 126
nllf() (bumps.parameter.Constant method), 129
nllf() (bumps.parameter.Function method), 132
nllf() (bumps.parameter.IntegerParameter method), 135
nllf() (bumps.parameter.Operator method), 137
nllf() (bumps.parameter.Parameter method), 140
nllf() (bumps.parameter.Reference method), 144
nllf() (bumps.pdfwrapper.DirectProblem method), 148
nllf() (bumps.pdfwrapper.PDF method), 148
nllf() (bumps.pdfwrapper.VectorPDF method), 149
nllf() (bumps.pyqmcfit.PyMCPProblem method), 155
noise (bumps.options.BumpsOpts attribute), 122
Normal (class in bumps.bounds), 87
notify (bumps.options.BumpsOpts attribute), 122
Npop (bumps.dream.state.MCMCDraw attribute), 185
Nsamples (bumps.dream.state.MCMCDraw attribute), 185
Nthin (bumps.dream.state.MCMCDraw attribute), 185
numpoints() (bumps.curve.Curve method), 94
numpoints() (bumps.curve.PoissonCurve method), 94
numpoints() (bumps.fitproblem.Fitness method), 98
numpoints() (bumps.pdfwrapper.PDF method), 148
numpoints() (bumps.pdfwrapper.VectorPDF method), 149
Nupdate (bumps.dream.state.MCMCDraw attribute), 185
Nvar (bumps.dream.state.MCMCDraw attribute), 185

O

op (bumps.parameter.Function attribute), 132
Operator (class in bumps.parameter), 136
origin (bumps.wsolve.PolynomialModel attribute), 163
outlier_test (bumps.dream.core.Dream attribute), 170
outliers() (bumps.dream.state.MCMCDraw method), 187

P

p (bumps.wsolve.LinearModel attribute), 163
p (bumps.wsolve.PolynomialModel attribute), 163
parallel (bumps.options.BumpsOpts attribute), 122
parallel_tempering() (in module bumps.partemp), 146
Parameter (class in bumps.parameter), 139
parameter_nllf() (bumps.fitproblem.BaseFitProblem method), 100
parameter_nllf() (bumps.fitproblem.MultiFitProblem method), 103
parameter_residuals() (bumps.fitproblem.BaseFitProblem method), 100
parameter_residuals() (bumps.fitproblem.MultiFitProblem method), 103
parameters() (bumps.curve.Curve method), 94
parameters() (bumps.curve.PoissonCurve method), 94
parameters() (bumps.fitproblem.Fitness method), 98
parameters() (bumps.parameter.Alias method), 125
parameters() (bumps.parameter.BaseParameter method), 126
parameters() (bumps.parameter.Constant method), 129
parameters() (bumps.parameter.FreeVariables method), 130
parameters() (bumps.parameter.Function method), 132
parameters() (bumps.parameter.IntegerParameter method), 135
randomize() (bumps.bounds.SoftBounded method), 89
randomize() (bumps.bounds.BoundedBelow method), 85
randomize() (bumps.bounds.BoundedNormal method), 87
putfull() (bumps.bounds.Distribution method), 86
putfull() (bumps.bounds.Bounded method), 83
putfull() (bumps.bounds.BoundedAbove method), 84
putfull() (bumps.bounds.BoundedBelow method), 85
putfull() (bumps.bounds.BoundedNormal method), 88
putfull() (bumps.bounds.Distribution method), 86
putfull() (bumps.bounds.Normal method), 87
putfull() (bumps.bounds.SoftBounded method), 89
putfull() (bumps.bounds.Unbounded method), 82
PyMCProblem (class in bumps.pymcfit), 154
Q
quasinewton() (in module bumps.quasinewton), 155
queue (bumps.options.BumpsOpts attribute), 122
R
R() (bumps.dream.corrplot.Corr2d method), 170
radians() (bumps.parameter.BaseParameter method), 127
radians() (bumps.parameter.Constant method), 129
radians() (bumps.parameter.Function method), 133
radians() (bumps.parameter.IntegerParameter method), 135
radians() (bumps.parameter.Operator method), 138
radians() (bumps.parameter.Parameter method), 141
radians() (bumps.parameter.Reference method), 144
radians() (in module bumps.pmath), 154
random() (bumps.bounds.Bounded method), 83
random() (bumps.bounds.BoundedAbove method), 84
random() (bumps.bounds.BoundedBelow method), 85
random() (bumps.bounds.BoundedNormal method), 88
random() (bumps.bounds.Bounded method), 81
random() (bumps.bounds.Distribution method), 86
random() (bumps.bounds.Normal method), 87
random() (bumps.bounds.SoftBounded method), 89
random() (bumps.bounds.Unbounded method), 82
random_init() (in module bumps.initpop), 115
random_lines() (in module bumps.random_lines), 157
RandomBounds (class in bumps.dream.bounds), 167
randomize() (bumps.fitproblem.BaseFitProblem method), 101
randomize() (bumps.fitproblem.MultiFitProblem method), 103
randomize() (bumps.parameter.IntegerParameter method), 135
randomize() (bumps.parameter.Parameter method), 141
randomize() (bumps.parameter.Reference method), 144
randomize() (bumps.pdfwrapper.DirectProblem method), 148
randomize() (bumps.pymcfit.PyMCProblem method), 155
randomize() (in module bumps.parameter), 146
range() (bumps.parameter.BaseParameter method), 127
range() (bumps.parameter.Constant method), 129
range() (bumps.parameter.Function method), 133
range() (bumps.parameter.IntegerParameter method), 135
range() (bumps.parameter.Operator method), 138
range() (bumps.parameter.Parameter method), 141
range() (bumps.parameter.ParameterSet method), 142
range() (bumps.parameter.Reference method), 144
direct_face (class in bumps.util), 160
Reference (class in bumps.parameter), 142
ReflectBounds (class in bumps.dream.bounds), 167
register() (in module bumps.fitters), 110
reload_errors() (in module bumps.errplot), 96
remove_outliers()
  (bumps.dream.state.MCMCDraw method), 187
requires() (bumps.history.History method), 113
requires() (bumps.history.Trace method), 113
Resampler (class in bumps.fitters), 109
reset() (bumps.dream.crossover.AdaptiveCrossover method), 172
reset() (bumps.dream.crossover.BaseAdaptiveCrossover method), 172
reset() (bumps.dream.crossover.Crossover method), 172
reset() (bumps.dream.crossover.LogAdaptiveCrossover method), 172
residual() (bumps.bounds.Bounded method), 83
residual() (bumps.bounds.BoundedAbove method), 84
residual() (bumps.bounds.BoundedBelow method), 85
residual() (bumps.bounds.Bounded method), 81
residual() (bumps.bounds.Distribution method), 86
residual() (bumps.bounds.Normal method), 87
residual() (bumps.bounds.SoftBounded method), 89
residual() (bumps.bounds.Unbounded method), 82
residual() (bumps.bounds.BoundedNormal method), 88
residual() (bumps.bounds.Bounded method), 81
residual() (bumps.bounds.Distribution method), 86
residual() (bumps.bounds.Normal method), 87
residual() (bumps.bounds.SoftBounded method), 89
residual() (bumps.bounds.Unbounded method), 82
residual() (bumps.parameter.BaseParameter method), 127
residual() (bumps.parameter.Constant method), 129
residual() (bumps.parameter.Function method), 133
residual() (bumps.parameter.IntegerParameter method), 135
residual() (bumps.parameter.Operator method), 138
residual() (bumps.parameter.Parameter method), 141
residual() (bumps.parameter.Reference method), 144
residuals() (bumps.curve.Curve method), 94
residuals() (bumps.curve.PoissonCurve method), 94
residuals() (bumps.fitproblem.BaseFitProblem method), 101
residuals() (bumps.fitproblem.Fitness method), 98
residuals() (bumps.fitproblem.MultiFitProblem method), 103
residuals() (bumps.pdfwrapper.VectorPDF method), 149
resize() (bumps.dream.state.MCMCDraw method), 187
restore() (bumps.history.History method), 113
restore() (bumps.history.Trace method), 114
restore_data() (bumps.fitproblem.BaseFitProblem method), 101
restore_data() (bumps.fitproblem.Fitness method), 98
restore_data() (bumps.fitproblem.MultiFitProblem method), 103
resume (bumps.options.BumpsOpts attribute), 122
resynth (bumps.options.BumpsOpts attribute), 122
resynth() (in module bumps.cli), 92
resynth_data() (bumps.fitproblem.BaseFitProblem method), 101
resynth_data() (bumps.fitproblem.Fitness method), 98
resynth_data() (bumps.fitproblem.MultiFitProblem method), 103
RLFit (class in bumps.fitters), 109
rnorm (bumps.wsolve.LinearModel attribute), 163
rnorm (bumps.wsolve.PolynomialModel attribute), 164
save_best() (in module bumps.cli), 92
save_history() (in module bumps.fitters), 110
save_state() (in module bumps.dream.state), 188
seed (bumps.options.BumpsOpts attribute), 122
selected_fitter (bumps.options.FitConfig attribute), 123
selected_name (bumps.options.FitConfig attribute), 123
selected_values (bumps.options.FitConfig attribute), 123
SerialMapper (class in bumps.mapper), 119
ServiceMonitor (class in bumps.fitservice), 104
set() (bumps.parameter.IntegerParameter method), 135
set() (bumps.parameter.Parameter method), 141
set_active_model() (bumps.fitproblem.MultiFitProblem method), 103
set_from_cli() (bumps.options.FitConfig method), 123
set_integer_vars() (bumps.dream.state.MCMCDraw method), 187
set_model() (bumps.parameter.FreeVariables method), 131
set_model() (bumps.parameter.ParameterSet method), 142
set_mplconfig() (in module bumps.cli), 92
set_visible_vars() (bumps.dream.state.MCMCDraw method), 187
setp() (bumps.fitproblem.BaseFitProblem method), 101
setp() (bumps.fitproblem.MultiFitProblem method), 103
setp() (bumps.pdfwrapper.DirectProblem method), 148
setp() (bumps.pymcfit.PyMCProblem method), 155
setpriority() (in module bumps.mapper), 119
settings (bumps.fitters.BFGSFit attribute), 105
settings (bumps.fitters.DEFit attribute), 106
settings (bumps.fitters.DreamFit attribute), 106
settings (bumps.fitters.LevenbergMarquardtFit attribute), 108
settings (bumps.fitters.MPFFit attribute), 109
settings (bumps.fitters.MultFit attribute), 109
settings (bumps.fitters.PSFit attribute), 109
settings (bumps.fitters.PTFit attribute), 109
settings (bumps.fitters.SimplexFit attribute), 110
settings (bumps.fitters.SnObFit attribute), 110
shortest_credible_interval() (in module bumps.dream.stats), 189

S
sample() (bumps.dream.core.Dream method), 170
sample() (bumps.dream.state.MCMCDraw method), 187
save() (bumps.curve.Curve method), 94
save() (bumps.curve.PoissonCurve method), 95
save() (bumps.dream.state.MCMCDraw method), 187
save() (bumps.fitproblem.BaseFitProblem method), 101
save() (bumps.fitproblem.Fitness method), 98
save() (bumps.fitproblem.MultiFitProblem method), 103
save() (bumps.fitters.DEFit method), 106
save() (bumps.fitters.DreamFit method), 106
save() (bumps.fitters.FitDriver method), 108
to_dict() (bumps.fitproblem.MultiFitProblem method), 104

valid() (bumps.fitproblem.BaseFitProblem method), 101
valid() (bumps.fitproblem.MultiFitProblem method), 104
valid() (bumps.parameter.BaseParameter method), 127
valid() (bumps.parameter.Constant method), 130
valid() (bumps.parameter.Function method), 133
valid() (bumps.parameter.IntegerParameter method), 136
valid() (bumps.parameter.Operator method), 138
valid() (bumps.parameter.Parameter method), 141
valid() (bumps.parameter.ParameterSet method), 142
valid() (bumps.parameter.Reference method), 145
to_dict() (bumps.parameter.Alias method), 125
to_dict() (bumps.parameter.BaseParameter method), 127
to_dict() (bumps.parameter.Constant method), 130
to_dict() (bumps.parameter.FreeVariables method), 131
to_dict() (bumps.parameter.Function method), 133
to_dict() (bumps.parameter.IntegerParameter method), 136
to_dict() (bumps.parameter.Operator method), 138
to_dict() (bumps.parameter.Parameter method), 141
to_dict() (bumps.parameter.ParameterSet method), 142
to_dict() (bumps.parameter.Reference method), 145
transport (bumps.options.BumpsOpts attribute), 122

U
Unbounded (class in bumps.bounds), 82

unique() (in module bumps.parameter), 146

V
valid() (bumps.fitproblem.BaseFitProblem method), 101
valid() (bumps.fitproblem.MultiFitProblem method), 104
valid() (bumps.parameter.BaseParameter method), 127
valid() (bumps.parameter.Constant method), 130
valid() (bumps.parameter.Function method), 133
valid() (bumps.parameter.IntegerParameter method), 136
valid() (bumps.parameter.Operator method), 138
valid() (bumps.parameter.Parameter method), 141
valid() (bumps.parameter.Reference method), 145

wsolve() (bumps.wsolve.LinearModel attribute), 163

W
weight (bumps.dream.crossover.AdaptiveCrossover attribute), 172
weight (bumps.dream.crossover.BaseAdaptiveCrossover attribute), 172
weight (bumps.dream.crossover.LogAdaptiveCrossover attribute), 172

X
x (bumps.wsolve.LinearModel attribute), 163

Y
yesno() (in module bumps.options), 124