Bumps: Curve Fitting and Uncertainty Analysis

Release 1.0.5rc1.post9+g569fed8

Paul Kienzle

Jun 11, 2026

CONTENTS

CHAPTER

ONE

GETTING STARTED

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

Bumps 1.0.5rc1.post9+g569fed8 is provided in a self-contained Python environment:

- Windows installer: bumps-1.0.5rc1.post9+g569fed8-Windows-x86_64-installer.exe
- Apple installer: bumps-1.0.5rc1.post9+g569fed8-Darwin-arm64.dmg
- Apple (Intel) installer: bumps-1.0.5rc1.post9+g569fed8-Darwin-x86_64.dmg
- Linux self-contained package: bumps-1.0.5rc1.post9+g569fed8-Linux-x86_64.tar.gz

The Windows installer installs to the *AppData/Local* directory by default, and adds a shortcut to the Start menu, optionally adding a desktop shortcut. It also provides an uninstaller through the Control Panel (add/remove programs)

The Apple .dmg installer unpacks to the Applications directory. You can start the application by double-clicking on the *bumps_webview.app*. To uninstall just drag the entire app to the trash from your finder.

For Linux (HPC), you can download the self-contained package and unpack it to a directory of your choice. The package contains a python environment with all the required dependencies, including the webview interface. To run the application, change into the unpacked directory and run:

```
./bin/python -m bumps
```

For Debian/Ubuntu Linux, bumps is provided as a package [pre-1.0 as of this writing]:

```
sudo apt install python3-bumps
```

For other linux or for the latest version you will need to install bumps as a python package.

1.1.1 Python install

Bumps is available on PyPI so you can install directly into a python environment with pip. To avoid conflicts between python applications it is good practice to create a separate python environment for each one.

We recommend using miniforge. This installs a conda system with the "conda-forge" channel for packages. Versions are available for Windows, MacOS and Linux.

To create your environment and install bumps use:

```
conda create --name bumps python
conda activate bumps
pip install bumps
```

You could instead download Python directly from Python.org. Again, recommended practice is to use an isolated python environment:

```
python -m venv bumps
. bumps/bin/activate
pip install bumps
```

1.1.2 Running bumps

Fitting problems in bumps are defined in python files or jupyter notebooks. You can retrieve the example *curve.py* model here

To run the webview interface with your problem showing in a browser window use:

```
bumps curve.py
```

To run in batch mode with no interactive interface use:

```
bumps -b curve.py --session=fit.h5
```

This runs a complete fit, appending the results to the session file T1.hdf. To later view the fit results use:

```
bumps --session=fit.h5
```

There are many command line options for controlling the fit. For a complete list use:

```
bumps -h
```

1.1.3 Jupyter notebooks

The webview interface can be run inside a Jupyter notebook. This allows you to interact with the server from within the notebook, providing a more integrated experience for users who are already working in a Jupyter environment.

You will need to set up Jupyter. You can install jupyter into your bumps environment with pip and run the Jupyter server from there. If you are using jupyterhub, you can install and run ipykernel in your bumps environment to make it available:

```
pip install ipykernel
python -m ipykernel install --user --name bumps --display-name "bumps"
```

If running on colab or similar, then you can install bumps from within a notebook cell using pip:

```
%pip install bumps
```

To start webview, use the following code cell:

```
import asyncio
from bumps.webview.server import start_bumps_server, api

# Start the server
await start_bumps_server()
```

A link to the server will be printed in the notebook output. You can open this link in a browser to access the server.

In a different cell you can define a problem and load it into the server using the *api* module:

```
# Define a problem
from bumps.fitproblem import FitProblem

model = MyFitnessClass()
...

problem = FitProblem([model])
await api.set_problem(problem)
```

1.1.4 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.

The following command should build the fast stepper binary module:

```
python -m bumps.dream.build_compiled
```

If you have installed from source, you must first check out the random123 library:

```
git clone --branch v1.14.0 https://github.com/DEShawResearch/random123.git bumps/dream/
↪random123
python -m bumps.dream.build_compiled
```

If this fails you can try running the compiler directly. First find the path to the bumps directory:

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

Change into that directory and compile the module:

```
(cd path/to/bumps/dream && cc compiled.c -I ./random123/include/ -O2 -DMAX_THREADS=64 -
↪fopenmp -shared -lm -o _compiled.so -fPIC)
```

Note: clang doesn't support OpenMP, so on macOS use:

```
(cd path/to/bumps/dream && cc compiled.c -I ./random123/include/ -O2 -DMAX_THREADS=64 -
↪shared -lm -o _compiled.so -fPIC)
```

Make sure MAX_THREADS is at least the number of processors on your system otherwise you will need to set `OMP_NUM_THREADS=MAX_THREADS` in your environment before running bumps.

1.2 Contributing Changes

The bumps package is a community project, and we welcome contributions from anyone. The package is developed collaboratively on Github - if you don't have an account yet, you can sign up for free. For direct write access to the repository, it is required that your account have two-factor authentication enabled. You may also want to configure your account to use SSH keys for authentication.

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:

https://github.com/bumps/bumps

You will need the git source control software for your computer. This can be downloaded from the git page, or you can use an integrated development environment (IDE) such as PyCharm or VS Code, which may have git built in.

1.2.1 Getting the Code

To get the code, you will need to clone the repository. If you are planning on making only a few small changes, you can clone the repository directly, make the changes, document and test, then send a patch (see *Simple patches* below).

If you are planning on making larger changes, you should fork the repository on github, make the changes in your fork, then issue a pull request to the main repository (see *Larger changes* below).

Note

If you are working on a fork, the clone line is slightly different:

```
git clone https://github.com/YourGithubAccount/bumps
```

You will also need to keep your fork up to date with the main repository. You can do this by adding the main repository as a remote, fetching the changes, then merging them into your fork.

```
# Add the main repository as a remote
git remote add bumps

# Fetch the changes from the main repository
git fetch bumps

# Merge the changes into your fork
git merge bumps/master

# Push the changes to your fork
git push
```

1.2.2 Run from source

When working from the source tree it is helpful to install bumps in developer mode. This allows you to change the files in place and see the changes the next time you run the program. It also includes all the additional packages needed for building documentation and testing.

To set up and install the developer version use:

```
cd bumps
conda create --name bumps-dev python
conda activate bumps-dev
pip install -e .[dev]
```

This puts bumps and bumps-webview on your path while leaving the files in place. Any changes you do to the files will appear when you next run the program.

The webview client uses modern web technologies such as TypeScript, Vue.js, and Plotly. These are pre-compiled and included with the python wheel for pip installs and also in the binary installers, but when running from source you will need to build the client package before starting the server.

To install *nodejs* and build the client use:

```
conda install nodejs
python -m bumps.webview.build_client
```

This will download the necessary dependencies to build the client package and save it to the *bumps/webview/client/dist* directory. You need to run the *build_client* command whenever you change the javascript for the webview interface.

If you already have a python environment with the necessary dependencies and you don't want to install the package into your environment (for example, because you are testing out a fork in another source tree), then you can change to the bumps directory and run the package in place:

```
python -m bumps --batch ... # for the command line interface
python -m bumps ... # for the webview interface
```

1.2.3 Building Documentation

The HTML documentation requires the sphinx, docutils, and jinja2 packages in python.

The PDF documentation requires a working LaTex installation, including the following packages:

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

To build the documentation use:

```
(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.

1.2.4 Making Changes

Simple patches

If you want to make one or two tiny changes, it is easiest to clone the repository, make the changes, then send a patch. This is the simplest way to contribute to the project.

As you make changes to the package, you can see what you have done using git:

```
git status
git diff
```

Please update the documentation and add tests for your changes. We use doctests on all of our examples so that we know our documentation is correct. More thorough tests are found in test directory. You can run these tests via pytest, or via the convenience Makefile target:

```
pytest
# or
make test
```

When all the tests run, create a patch and send it to paul.kienzle@nist.gov:

```
git diff > patch
```

Pre-commit hooks

Bumps uses pre-commit to run automated checks and linting/formatting on the code before it is committed.

First, activate the Python environment in which you installed bumps. Then, install the pre-commit hooks by running:

```
pre-commit install
```

This will install the pre-commit hooks in your git repository. The pre-commit hooks will run every time you commit changes to the repository. If the checks fail, the commit will be aborted.

You can run the checks manually by running:

```
pre-commit run
```

To see what actions are being run, inspect the *.pre-commit-config.yaml* file in the root of the repository.

Larger changes

For a larger set of changes, you should fork bumps on github, and issue pull requests for each part.

After you have tested your changes, you will need to push them to your github fork:

```
git commit -a -m "short sentence describing what the change is for"
git push
```

Good commit messages are a bit of an art. Ideally you should be able to read through the commit messages and create a "what's new" summary without looking at the actual code.

Make sure your fork is up to date before issuing a pull request. You can track updates to the original bumps package using:

```
git remote add bumps https://github.com/bumps/bumps
git fetch bumps
git merge bumps/master
git push
```

When making changes, you need to take care that they work on different versions of python. Using conda makes it convenient to maintain multiple independent environments. You can create a new environment for testing with, for example:

```
conda create -n py312 python=3.12
conda activate py312
pip install -e .[dev]
pytest
```

When all the tests pass, issue a pull request from your github account.

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

1.2.5 Building an installer (all platforms)

To build a packed distribution for Windows, you will need to install conda-pack in your base conda environment. If you don't already have a base interpreter, install that as well (e.g. on Windows) from conda-forge:

```
conda install -c conda-forge conda-pack bash
```

Then you can build the packed distribution using:

```
bash extra/build_conda_packed.sh
```

This will create a packed distribution in the dist directory.

1.2.6 Creating a New Release

A developer with maintainer status can tag a new release and publish a package to the Python Package Index (PyPI). Bumps uses versioningit to generate the version number from the latest tag in the git repository.

1. Make sure all tests are passing on the master branch
2. On GitHub, go to Releases page and click on Draft a new release
3. Click on *Choose Tag* and type in a new version number
    - For a new alpha pre-release, choose an alpha tag like "1.2.3a1"
    - For a beta pre-release type a version number like "1.2.3b1"
    - For a full release use a tag like "1.2.3" (following semver guidelines)
4. Click on *Generate release notes* button
5. Edit the generated notes as desired
6. Check the *Set as pre-release* box below the release notes for alpha- or beta-releases
    - (leave *Set as the latest release* box checked for full releases)
7. Click *Publish Release*

At this point, some workflow jobs will publish the new version:

1. the job defined in .github/workflows/test-publish.yml will run the tests one more time, then publish the new version to PyPi
2. the job defined in .github/workflows/build-distributables.yml will create installers for *MacOS*, *Windows* and *Linux* and attach them to the release page.

1.3 License

Bumps is in the public domain.

Code in individual files has copyright and license set by individual authors.

1.3.1 bumps.gui, bumps.quasinewton

1.3.2 DREAM

1.3.3 Random123

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 DAMAGE.

1.3.4 MPFit

The original version of this software, called LMFIT, was written in FORTRAN as part of the MINPACK-1 package by XXX.

Craig Markwardt converted the FORTRAN code to IDL. The information for the IDL version is:

> Craig B. Markwardt, NASA/GSFC Code 662, Greenbelt, MD 20770 craigm@lheamail.gsfc.nasa.gov UPDATED VERSIONs can be found on my WEB PAGE:
>
> http://cow.physics.wisc.edu/~craigm/idl/idl.html

Mark Rivers created this Python version from Craig's IDL version.
Mark Rivers, University of Chicago Building 434A, Argonne National Laboratory 9700 South Cass Avenue, Argonne, IL 60439 rivers@cars.uchicago.edu Updated versions can be found at http://cars.uchicago.edu/software

1.3.5 bumps.simplex

```
# ******NOTICE***************
# From optimize.py module by Travis E. Oliphant
#
# You may copy and use this module as you see fit with no
# guarantee implied provided you keep this notice in all copies.
# *****END NOTICE************
```

1.3.6 bumps.cli.warn_with_traceback

From http://stackoverflow.com/questions/22373927/get-traceback-of-warnings answered by mgab (2014-03-13) edited by Gareth Rees (2015-11-28)

1.3.7 bumps.dream.entropy.Timer

Based on: Eli Bendersky https://stackoverflow.com/a/5849861 Extended with tic/toc by Paul Kienzle

1.3.8 bumps.dream.entropy.MultivariateT.rvs

From farhawa on stack overflow https://stackoverflow.com/questions/29798795/multivariate-student-t-distribution-with-python

1.3.9 bumps.lsqerror.comb

From dheerosaur https://stackoverflow.com/questions/4941753/is-there-a-math-ncr-function-in-python/4941932#4941932

1.4 Credits

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

It can by referenced by DOI: 10.5281/zenodo.15730428. (citation text available at that link)

Or cited as:

> Kienzle, P.A., Krycka, J., Patel, N., & Sahin, I. (2011). Bumps (Version 1.0.5rc1.post9+g569fed8) [Computer Software]. College Park, MD: University of Maryland. Retrieved Jun 11, 2026.

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.

CHAPTER

TWO

TUTORIAL

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, σ_y, where vector y is the value measured in conditions x, and σ_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:

```
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.

```
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.

```
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-σ 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.

```
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.

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

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

```
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 --session=fit.h5
```

The `--fit=newton` option says to use the quasi-newton optimizer for not more than 100 steps. The `--session=fit.h5` option says to store the initial model, the fit results and any monitoring information in the HDF5 file `fit.h5`.

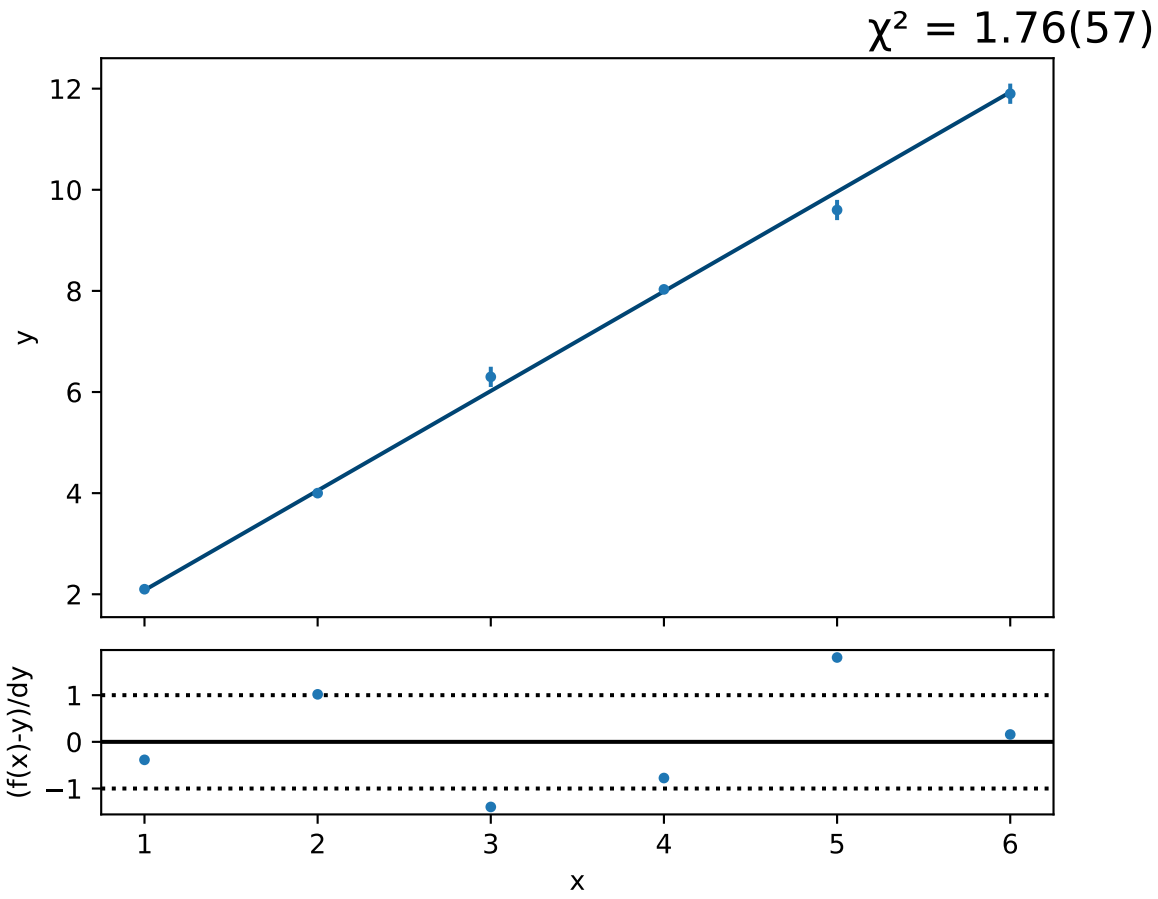
As the fit progresses, we are shown an iteration number and a cost value. The cost value is approximately the normalized χ^2_N. The value in parentheses is like the uncertainty in χ^2_N, in that a 1-σ change in parameter values should increase χ^2_N by that amount.

Here is the resulting fit:

All is well: Normalized χ^2_N is close to 1 and the line goes nicely through the data.

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.


χ² = 1.76(57)
12
10
8
6
4
2
y
1
0
−1
(f(x)-y)/dy
1
2
3
4
5
6
x

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, ...)* to propose rates for the various *x* values given the parameters, yielding parameter values *p1, p2, ...* 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 ± 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 λ 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:

$$E[\lambda] = \int_{0}^{\infty} \lambda \frac{\lambda^k e^{-\lambda}}{k!} d\lambda$$

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 λ 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 ± 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 λ and see which version best recovers the true parameters which generated the peak.

```
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.

```
x = np.linspace(5, 20, 345)
# y = np.random.poisson(peak(x, 1000, 12, 1.0, 1))
# y = np.random.poisson(peak(x, 300, 12, 1.5, 1))
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.

```
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[y == 0] = 0.5
    dy[y == 0] = 0.5
elif cond == "expected_zero":  # option 5: L = y +/- sqrt(y);  L[0] = 0 +/- 1
    dy = np.sqrt(y)
    dy[y == 0] = 1.0
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.

```
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.

```
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 ± 1 for $k = 1$ gives the best gaussian approximation.

Table 1: Fit results

#	method	background
0	poisson	1.0
1	expected	1.55
2	pearson	0.16
3	expected_mle	0.55
4	pearson_zero	0.34
5	expected_zero	0.75

2.1.3 Poisson simulation

For the poisson background estimation problem, poisson.py, we explore different options for estimating the rate parameter λ 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 underly rate λ. We do this by running a Poisson generator to draw thousands of samples of k from each of a range of values λ. By counting the number of times k occurs in each λ 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 λ 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:

$$d\hat{\lambda}^2 = \int_0^\infty (\lambda - \hat{\lambda})^2 P(\lambda|k) d\lambda$$

```
import numpy as np
from pylab import *
```

Generate a bunch of samples from different underlying rate parameters L in the range 0 to 20

```
P = np.random.poisson
L = linspace(0, 20, 1000)
X = P(L, size=(10000, len(L)))
```

Generate the distributions

```
P = dict((k, sum(X == k, axis=0) / sum(X == k)) for k 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 λ 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 :math:`\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
```

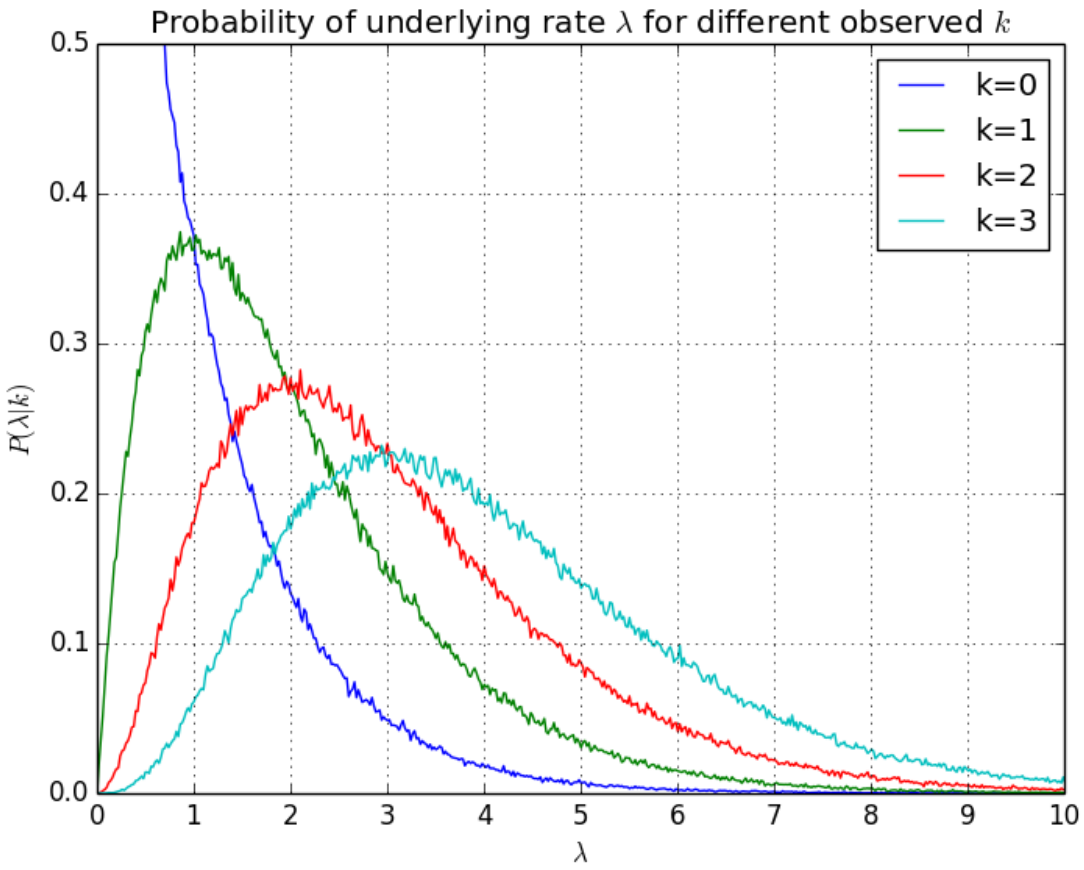


Fig. 1: The figure clearly shows that the maximum likelihood value for λ is equal to the observed counts k. Because the histogram is skew right, the expected value is a little larger, with an estimated value of $k + 1$, as seen from the output.

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

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

Define the ODE

```
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.

```
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


t, y, dy = simulate()
```

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.

```
M = Curve(g, t, y, dy, x0=1.0, a=1.0, b=1.0, plot_x=np.linspace(t[0], t[-1], 1000))
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:

https://scipy-cookbook.readthedocs.io/items/CoupledSpringMassSystem.html

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

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

```
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

```
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.

```
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]
```

(continues on next page)

(continued from previous page)

```
    w0 = [x1, y1, x2, y2]

    # 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(1):  # Make sure that the simulated data is the same each run
        data = ft + noise * np.random.randn(*ft.shape)
    return t, data, noise
```

(continues on next page)

(continued from previous page)

```
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.

```
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.

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)

```
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.

This uses a library of peak functions to model the peaks. In order for this to work peaks.py must be on your python path. For example, on linux or mac:

PYTHONPATH=. bumps -b –fit=dream model.py –session=peaks.h5

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*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 a 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.

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

Anticorrelated function

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

Fake data

```
sigma = 1
x = np.linspace(-1.0, 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

```
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.

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

```
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.

```
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.

```
domain = 1
```

Uniform cost function.

```
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


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():
    """
```

(continues on next page)

(continued from previous page)

```
    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
    to modify the parameter view to show 100% of the range rather than
    the 95% CI cutoff in current plots (code in bumps.dream.varplot.plot_var).
    """
    a, b = M.a.value, M.b.value
    return 0 if all(0.0 < xk / domain < 1.5 for xk in (a, b)) else 1e6 + sum((xk /␣
↪domain) ** 2 for xk in (a, b))
```

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
```

(continues on next page)

(continued from previous page)

```
M.a.range(-2 * domain, 2 * domain)
M.b.range(-2 * domain, 2 * domain)

# 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 paramters, 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.

```
import sys
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

```
USAGE = """
Usage: bumps check_entropy.py dist p1 p2 ...

where dist is one of the distributions in scipy.stats.distributions and
p1, p2, ... are the arguments for the distribution in the order that they
appear. For example, for the normal distribution, x ~ N(3, 0.8), use:

    bumps --fit=dream --entropy --session=fit.h5 check_entropy.py --args norm 3 0.2
"""


def _mu_sigma(mu, sigma):
    sigma = np.asarray(sigma)
    if len(sigma.shape) == 1:
        sigma = np.diag(sigma**2)
    if mu is None:
        mu = np.zeros(sigma.shape[0])
    return mu, sigma


def mvn(sigma, mu=None):
    mu, sigma = _mu_sigma(mu, sigma)
    return multivariate_normal(mean=mu, cov=sigma)


def mvskewn(alpha, sigma, mu=None):
    sigma = np.asarray(sigma)
    assert len(sigma.shape) == 1
    if mu is None:
        mu = np.zeros(sigma.shape[0])
    Dk = [distributions.skewnorm(alpha, m, s) for m, s in zip(mu, sigma)]
    return Joint(Dk)


def mvt(df, sigma, mu=None):
    mu, sigma = _mu_sigma(mu, sigma)
```

(continues on next page)

(continued from previous page)

```
    return MultivariateT(mu=mu, sigma=sigma, df=df)


def mvcauchy(sigma, mu=None):
    mu, sigma = _mu_sigma(mu, sigma)
    return MultivariateT(mu=mu, sigma=sigma, df=1)


DISTS = {
    "mvn": mvn,
    "mvt": mvt,
    "mvskewn": mvskewn,
    "mvcauchy": mvcauchy,
    "mvu": Box,
}
if len(sys.argv) > 1:
    dist_name = sys.argv[1]
    D_class = DISTS.get(dist_name, None)
    if D_class is None:
        D_class = getattr(distributions, dist_name, None)
    if D_class is None:
        print("unknown distribution " + dist_name)
        sys.exit()
    args = [
        [[float(vjk) for vjk in vj.split(",")] for vj in v.split(";")]
        if ";" in v
        else [float(vj) for vj in v.split(",")]
        if "," in v
        else float(v)
        for v in sys.argv[2:]
    ]
    D = D_class(*args)
else:
    print(USAGE)
    sys.exit()
```

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.

```
def D_nllf(x):
    return -D.logpdf(x)


dim = getattr(D, "dim", 1)
if dim == 1:
    M = PDF(D_nllf, x=0.9)
    M.x.range(-inf, inf)
else:
    M = VectorPDF(D_nllf, np.ones(dim))
    for k in range(dim):
```

(continues on next page)

(continued from previous page)

```
        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.

```
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:

```
t 84            # close to normal
t 4             # high kurtosis
uniform -5 100  # high entropy
cauchy 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 standard normal at (1,2,3)
mvt 4 1,1,1,1,1 # 5-D multivariate t-distribution with df=4 at origin
mvu 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:

```
bumps peak.py N --entropy --session=fit.h5 --fit=dream
```

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

```
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
```

(continues on next page)

(continued from previous page)

```
# 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 give 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:

```
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.util 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))
```

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 contraints of the form $a < b$ for parameters a and paramter 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 $a <= b$ 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 neglible, and will not skew the posterior distribution statistics.

Define the model as usual

```
from bumps.names import *


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


# Simulated data for f(x)=mx+b with m=1.972(20) and b=0.11(5)
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)
```

Attach the constraints to the problem. In this case we will force m < b using a list of inequality constraints. Note that we are starting deep in the infeasible region, so use the default penalty_nllf = 1e6

```
problem = FitProblem(M, constraints=[M.m < M.b])
```

CHAPTER

THREE

USER'S GUIDE

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.

> # 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:

```
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:

```
problem = FitProblem(models)
```

In fact, this is the only requirement of the fit file. The Bumps engine loads the fit file, retieves 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 *FitProblem* 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:

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

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

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 χ^2 optimization problem and the maximum likelihood problem. Given Gaussian uncertainty for data measurements, we find that data y_i measured with uncertainty σ_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

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

Note that this is the unnormalized χ^2, whose expected value is the number of degrees of freedom in the model, not the reduced χ^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 χ^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 overestimate 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° and

an uncertainty of $0.2°$, we can set

$$-\log P(\text{model}) = -\frac{1}{2}\frac{(\theta - 3)^2}{0.2^2}$$

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 ± 3.5 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 has arguments defining the parameters. For example, to fit a line you would need:

```
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:

```
# 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:

```
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*:

```
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:

```
M.m.range(-inf,inf)
M.b.range(-inf,inf)
problem = FitProblem(M)
```

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[i]*.

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. χ^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 *PyMCProblem* and interested parties are directed therein for a working example.

3.3.6 External constraints

3.4 Parameters

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:

```
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:

```
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:

```
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.

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:

```
free = FreeVariables(names=["model1", "model2", ...],
                     p1=model.p1, p2=model.p2, ...)
...
problem = FitProblem(experiments, freevars=free)
```

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:

```
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:

```
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.,

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

3.5 Fitting

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. In batch mode the command line will be something like:

```
bumps --fit=amoeba --steps=1000 --starts=10 --parallel model.py --session=fit.h5 --
↪export=T1
```

Here, the results will be stored in the HDF5 file `fit.h5`. The `T1` directory will contain the current model in *model.py*, the best fit result in *model.par* and 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. The results will be appended to the session file. Note that the model.py file isn't required on reload:

```
bumps --fit=amoeba --steps=1000 --starts=20 --parallel --session=fit.h5
```

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 --session=fit.h5
```

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 --session=fit.h5
```

Some fitters save the complete state of the fitter on termination so that the fit can be resumed. Use `--resume` to resume from DREAM or DE.

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 χ^2 above the minimum corresponds to 1-$sigma$ confidence interval. By following this contour you can find the set of all points ξ 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 *--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 *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 `--init=random` distributes the initial points using a uniform distribution across the space of the parameters. Latin hypersquares `--init=lhs` improves on random by making sure that there is on value for each subrange of every variable. The covariance population `--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 `--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.

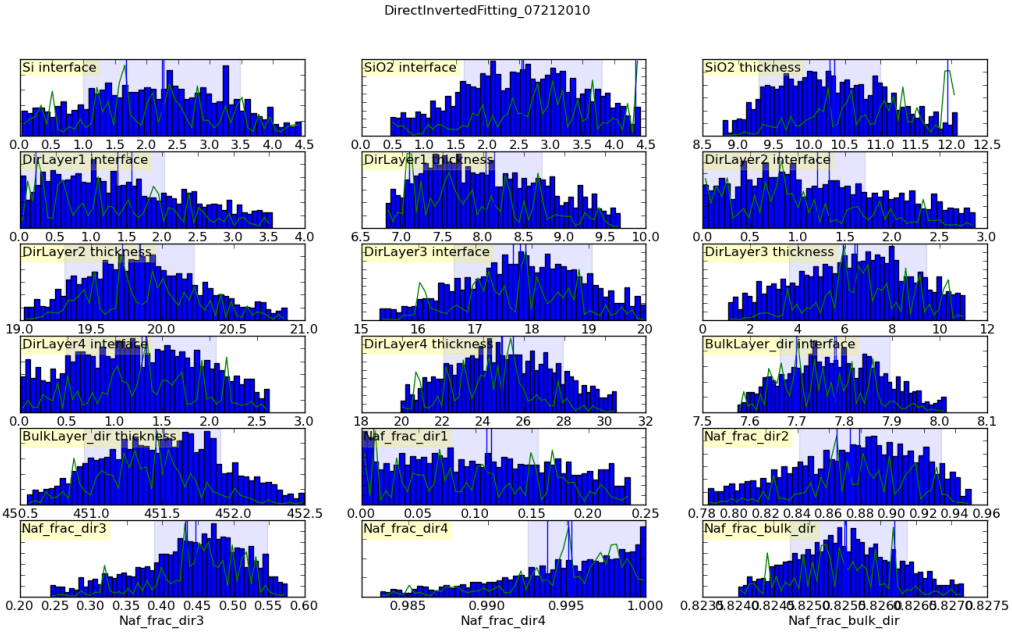
The Markov chain will take time to converge on a stable population. This burn in time needs to be specified at the start of the analysis. After burn, DREAM will collect all points visited for N iterations of the algorithm. If the burn time was long enough, the resulting points can be used to estimate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

A common command line for running DREAM is:

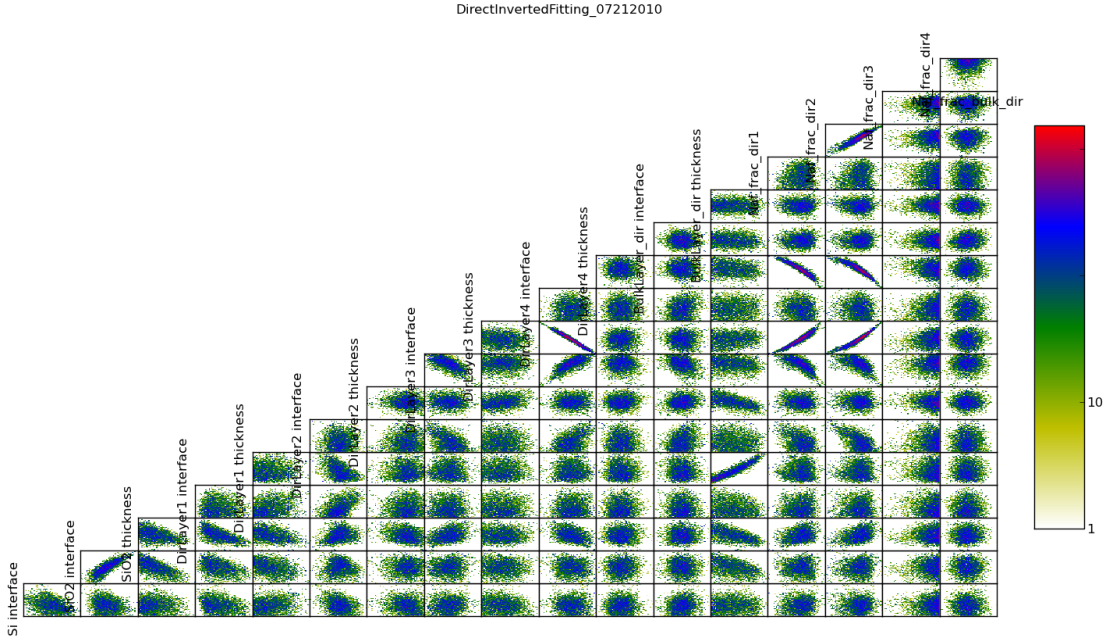
```
bumps --fit=dream --burn=1000 --samples=1e5 --init=cov --parallel model.py --session=fit.
↪h5
```

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

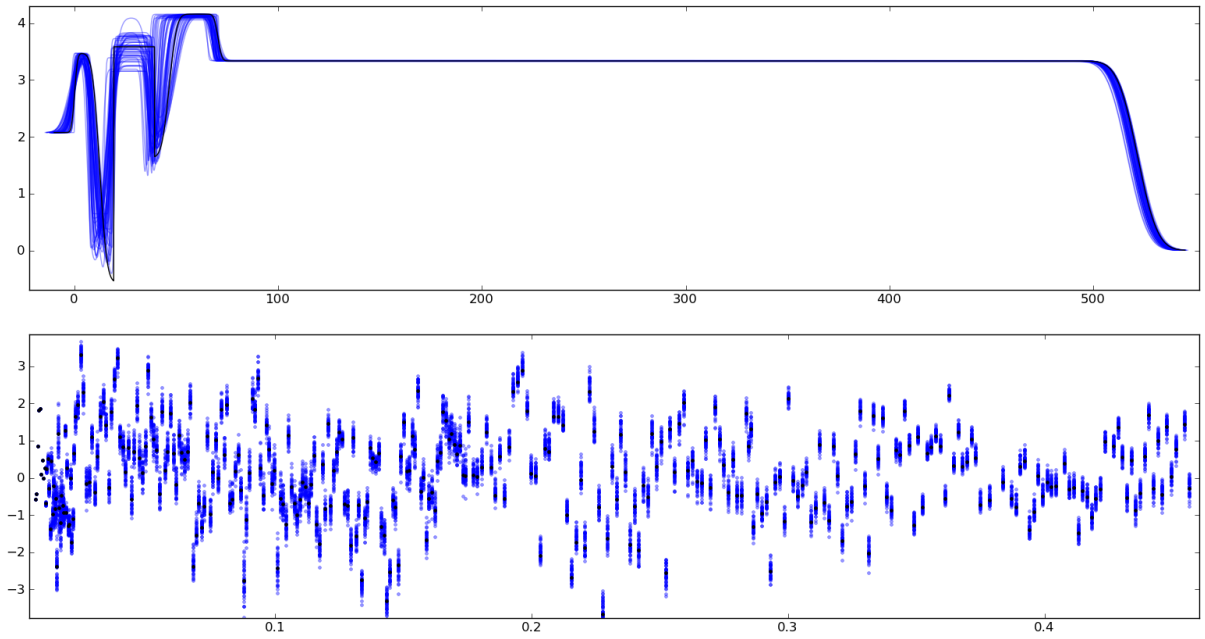
digits of precision on the 95% interval, approximately 1000000 samples drawn from the distribution are required, or steps = 1000000/(#parameters #pop). The 68% interval will require fewer draws, though how many has not yet been determined.



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 an perform uncertainty analysis operations after the fact.

First you need to import some functions:

```
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:

```
store = "/tmp/t1"   # path to the --export=/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:
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_corrmatrix(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 ρ and thickness t of the constituent layers using $\Gamma = 0.06955(\rho_1 t_1 + \rho_2 t_2)$. Using intermediate values for $\rho_1 t_1$ and $\rho_2 t_2$ 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_Ph4.9 NRW_0M_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)  24.892  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:

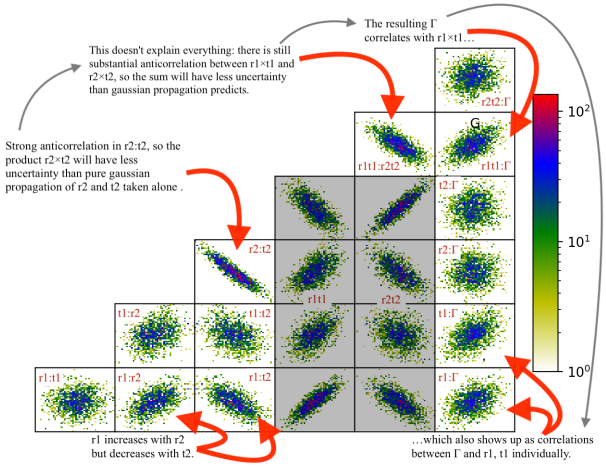
```
from uncertainties import ufloat as U
C = 0.06955
r1t1 = U(0.3321, 0.0098) * U(50.37, 0.89)
r2t2 = U(1.199, 0.022) * U(24.90, 0.80)
G = C*(r1t1 + r2t2)
print("r1*t1 =", r1t1)
print("r2*t2 =", r2t2)
print("G =", C*(r1t1 + r2t2))
```

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 ρ_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 r2:t2 having a very narrow diagonal (hence strong correlation) and r1t1:r2×t2 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)
reload(dream.corrplot)
state.show()
```

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:

```
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"))
```

(continues on next page)

(continued from previous page)

```
chisq = problem.chisq

print("chisq", chisq)
```

Assuming your model script is in model.py and you have run a fit with `--export=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:

```
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):
    Q,dQ,R,dR,theory = p.Q, p.dQ, p.R, p.dR, th[1]
    # ... insert reflectivity plotting code here ...
```

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

```
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:

```
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
```

(continues on next page)

(continued from previous page)

```
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):
    Q,dQ,R,dR,theory = p.Q, p.dQ, p.R, p.dR, th[1]
    # ... insert reflectivity plotting code here ...

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 agressively 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 `--Tmin=0.1` through `--Tmax=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 --session=fit.h5
```

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 --session=fit.h5
```

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

```
bumps fit.h5
```

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 (*Levenberg-Marquardt*, *Quasi-Newton BFGS*) tend to be fast but they will find local minima only, while the population algorithms (*DREAM*, *Differential Evolution*) are more robust and likely slower. *Nelder-Mead Simplex* is somewhere between, with a small population keeping the search local but more robust than the gradient descent algorithms.

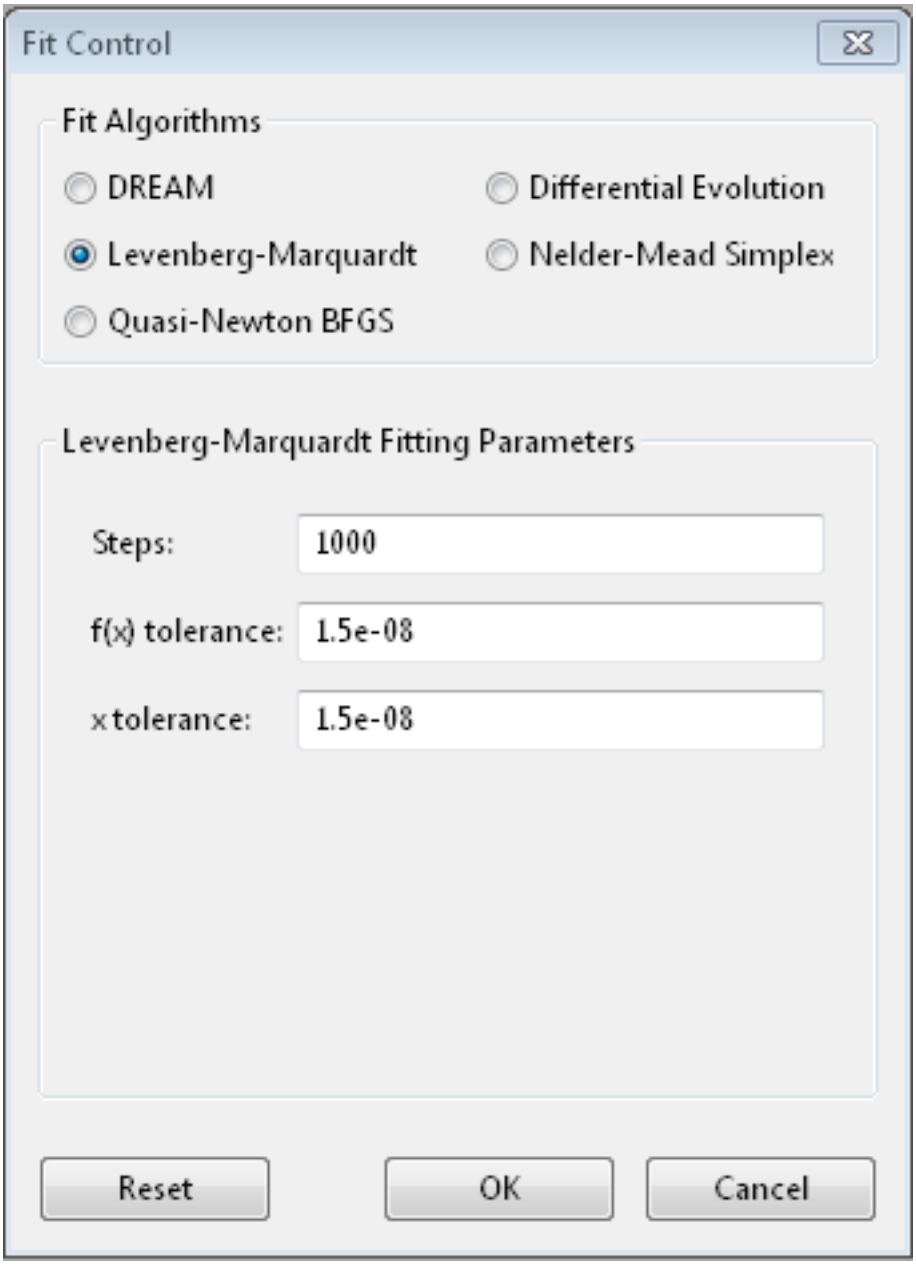
Each algorithm has its own set of control parameters for adjusting the search process and the stopping conditions. The same option may mean slightly different things to different optimizers. The Bumps package provides a dialog box for selecting the optimizer and its options when running the fit wx application. 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, *DREAM* uses a random walk to explore the parameter space near the minimum, showing pair-wise correlations amongst the parameter values. In order for *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 (*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[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.

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-σ) or 95% (2-σ) 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.

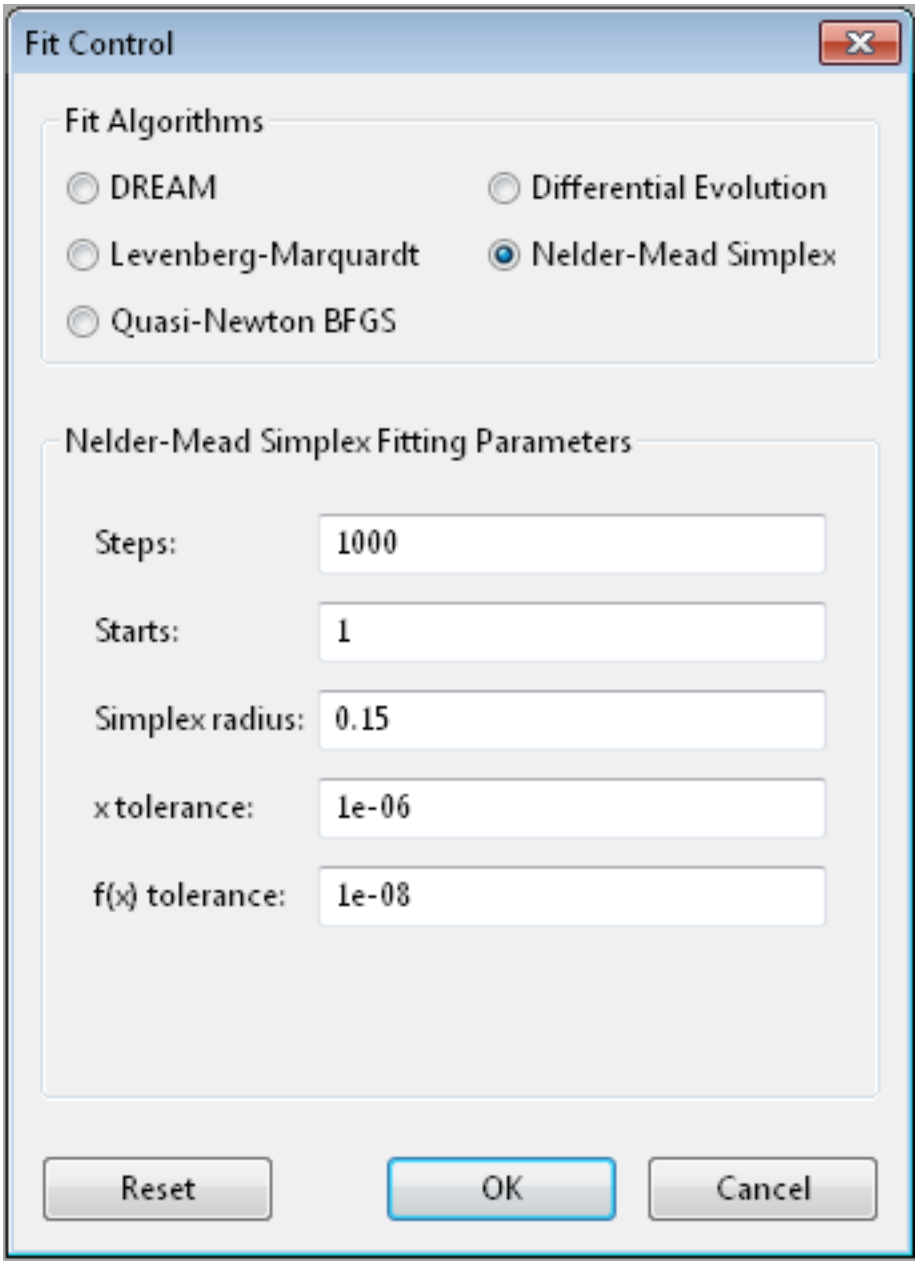
Notes

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

[1] Levenberg, K. *Quarterly Journal of Applied Mathematics* 1944, II (2), 164–168.
[2] Marquardt, D. W. *Journal of the Society for Industrial and Applied Mathematics* 1963, 11 (2), 431–441. DOI: 10.1137/0111030

References

3.6.2 Nelder-Mead Simplex



The Nelder-Mead[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 *Numerical Recipes*[4] wherein they describe the search as acting like an amoeba, squeezing through narrow valleys as it makes its way down to the minimum.

[3] Nelder, J. A.; Mead, R. *The Computer Journal* 1965, 7 (4), 308–313. DOI: 10.1093/comjnl/7.4.308

[4] Press, W. H.; Flannery, B. P.; Teukolsky, S. A.; Vetterling, W. T. In *Numerical Recipes in C: The Art of Scientific Computing, Second Edition*; Cambridge University Press: Cambridge; New York, 1992; pp 408–412.

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.

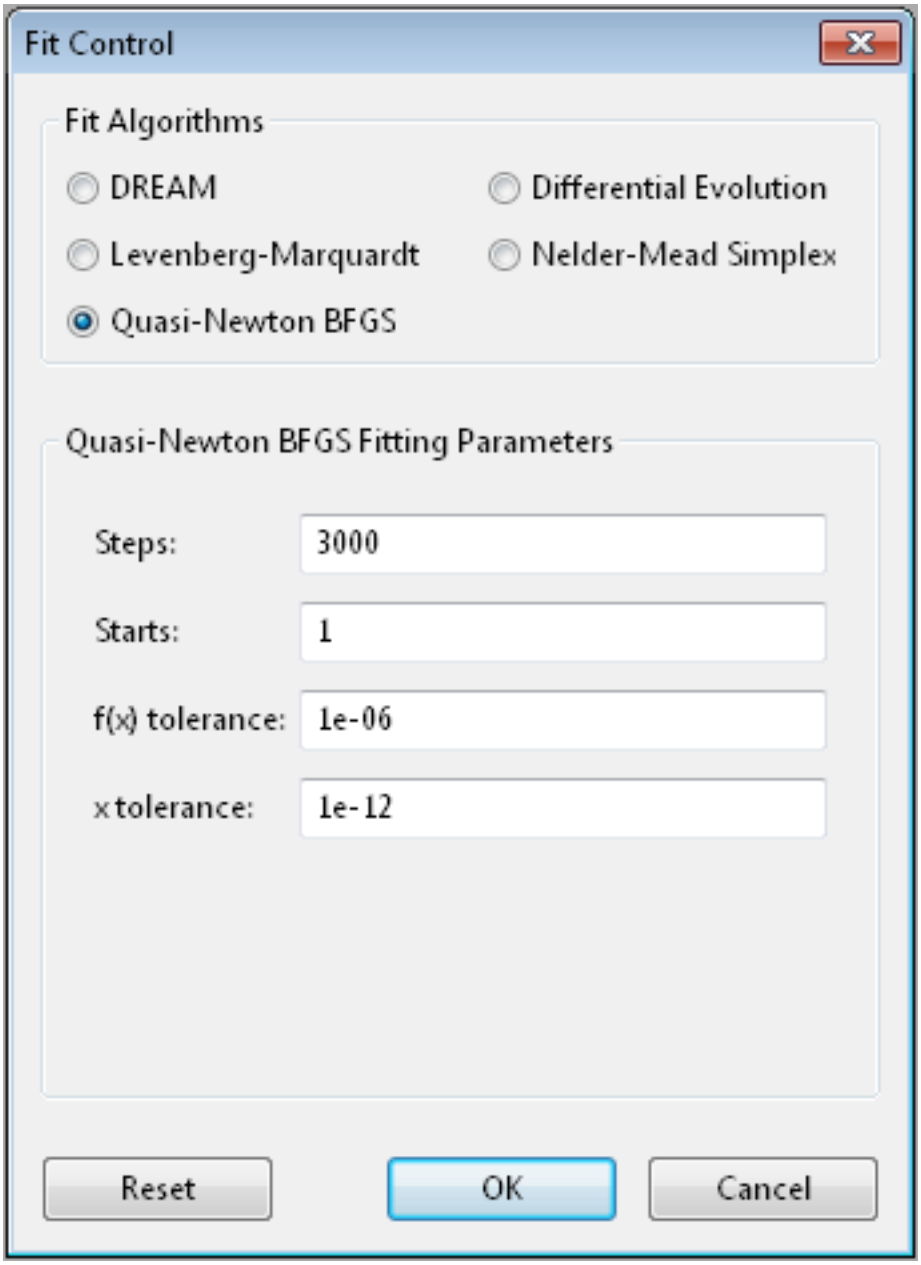
x tolerance and *f(x) tolerance* are used to determine when the fit has reached the point where no significant improvement is expected. If the simplex is tiny (that is, the corners are close to each other) and flat (that is, the values at the corners are close to each other), then the fit will terminate. 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.

[5] Dennis, J. E.; Schnabel, R. B. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*; Society for Industrial and Applied Mathematics: Philadelphia, 1987.

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.

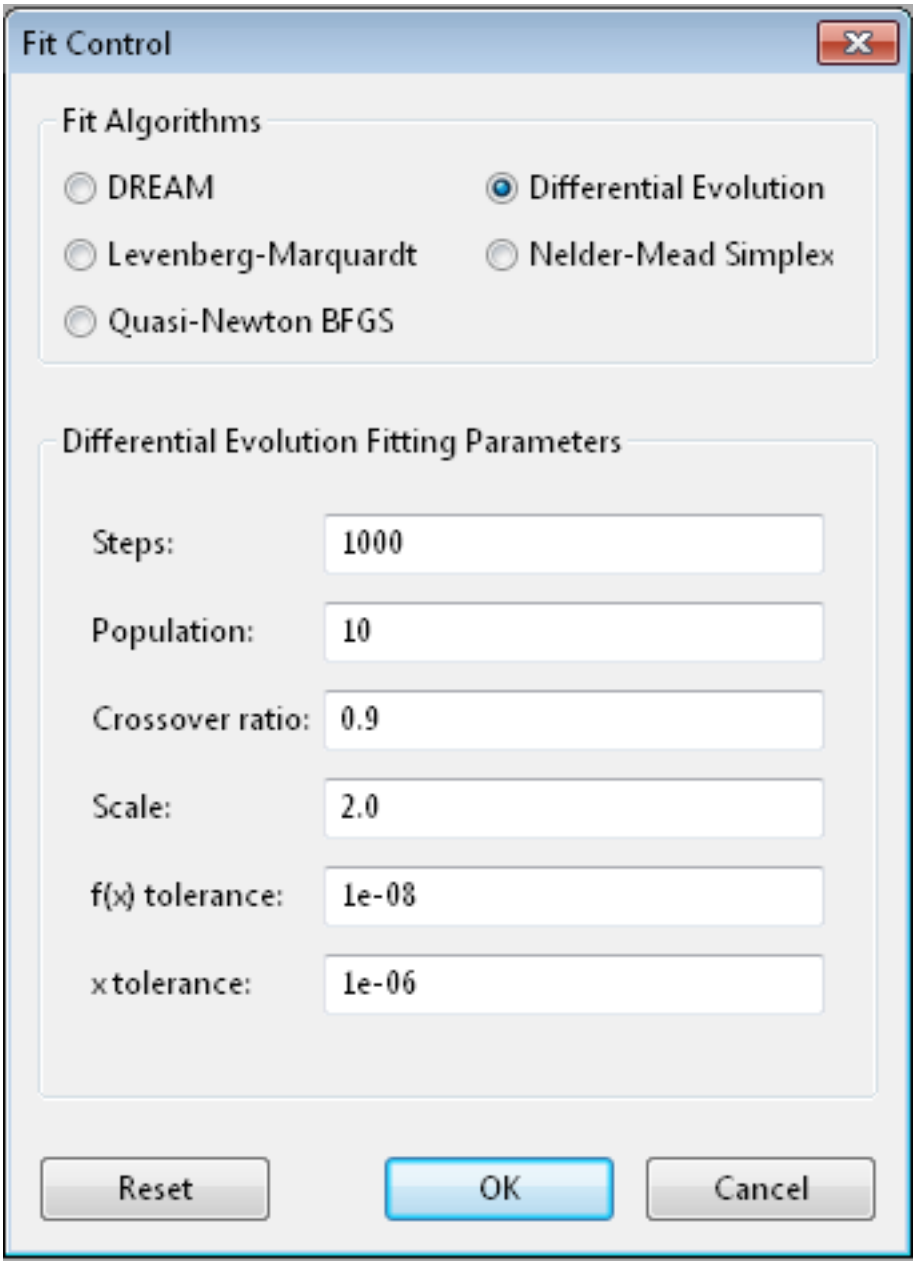
f(x) tolerance and *x tolerance* are used to determine when the fit has reached the point where no significant improvement is expected. If the function is small or the step is too short then the fit will terminate. Use `--ftol=v` and `--xtol=v` from the command line.

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 command line.

References

3.6.4 Differential Evolution



Differential evolution[6] 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

[6] Storn, R.; Price, K. *Journal of Global Optimization* 1997, 11 (4), 341–359. DOI: 10.1023/A:1008202821328

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 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 μ and standard deviation σ, 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.

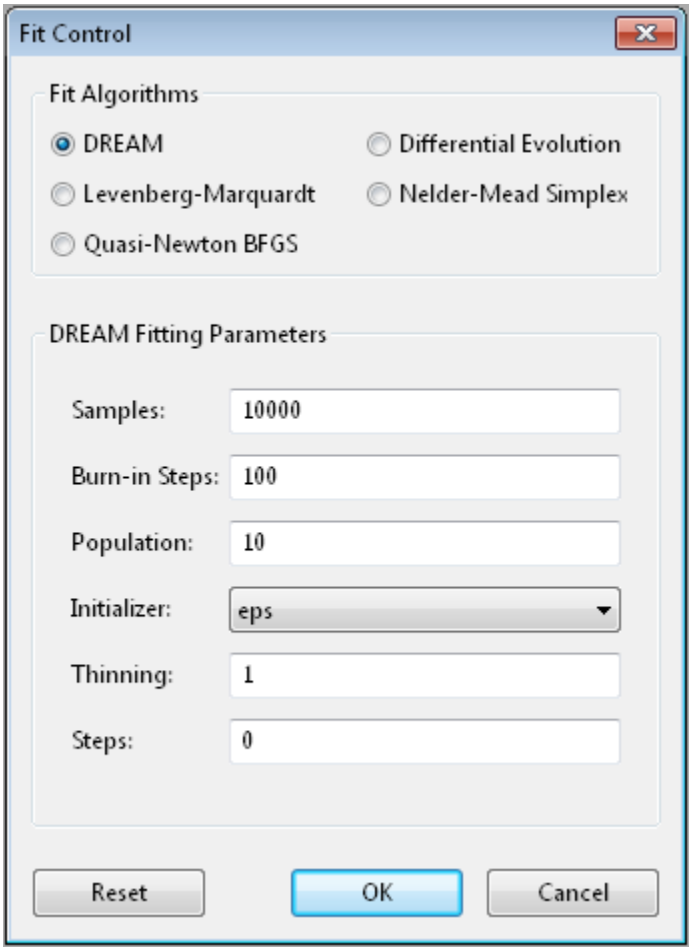
Scale determines how much to scale each difference vector before adding it to the candidate point. The selected mutation algorithm chooses a scale factor uniformly in $[0, F]$. Use `--F=v` from the command line.

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 command line.

References

3.6.5 DREAM



DREAM[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 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.

[7] Vrugt, J. A.; Ter Braak, C. J. F.; Diks, C. G. H.; Robinson, B. A.; Hyman, J. M.; Higdon, D. International Journal of Nonlinear Sciences and Numerical Simulation, 2009, 10 (3), 273–290. DOI: 10.1515/IJNSNS.2009.10.3.273

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 α 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 α 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 #samples/(#pars #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 (population x 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 command line. 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.

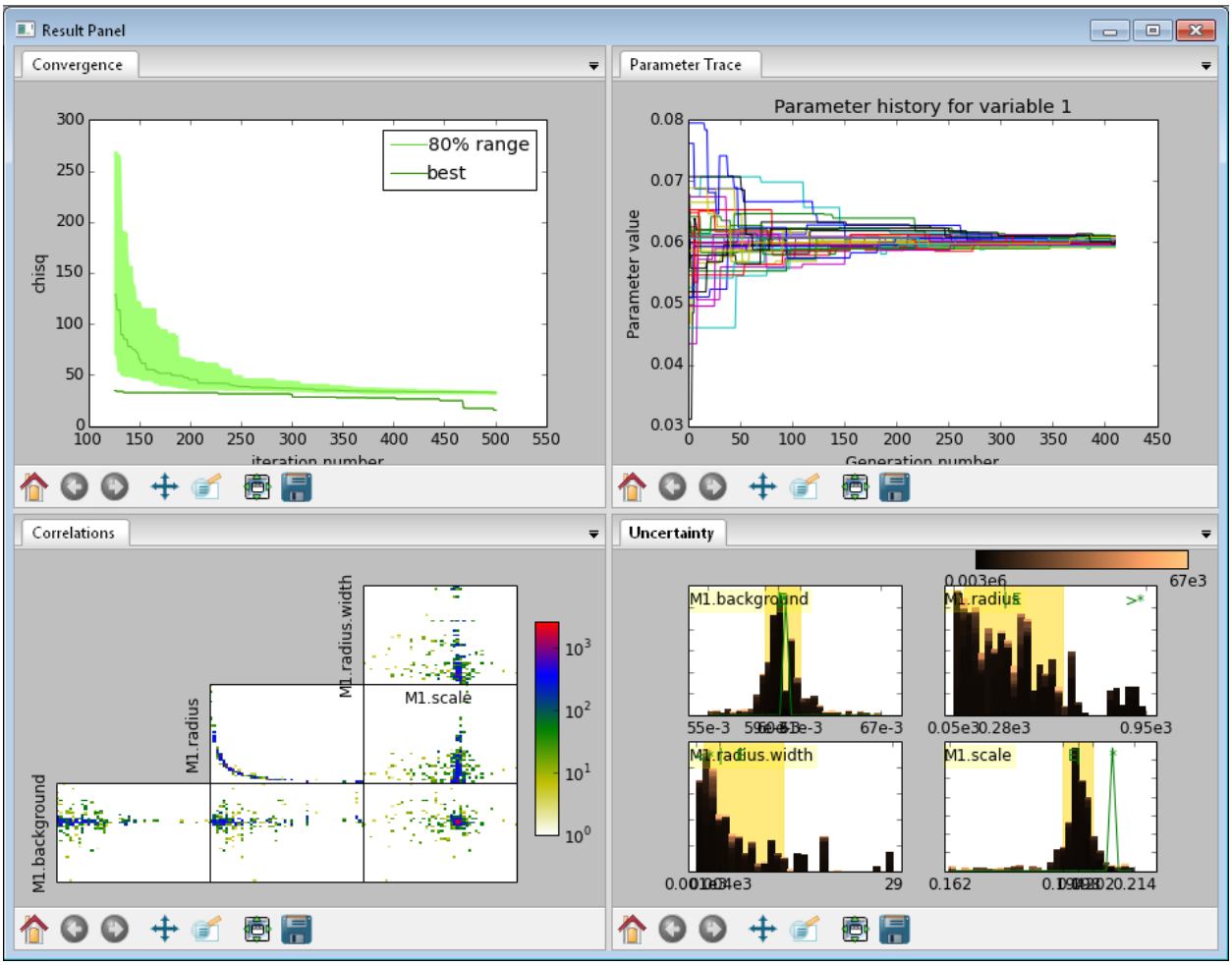


Fig. 1: This DREAM fit is incomplete, as can be seen on all four plots. The *Convergence* plot is still decreasing, the *Parameter Trace* plot shows a reduction in the mixing of Markov chain values, the *Correlation* plots are fuzzy and mostly empty, or show obvious correlations, and the *Uncertainty* plot shows black histograms (indicating that there are a few stray values far away from the best) whilst the green maximum likelihood spikes do not match the histogram (indicating that the region around the best value has not been adequately explored).

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 $\int xP(x)dx$, which is just the expected value of the probability distribution for

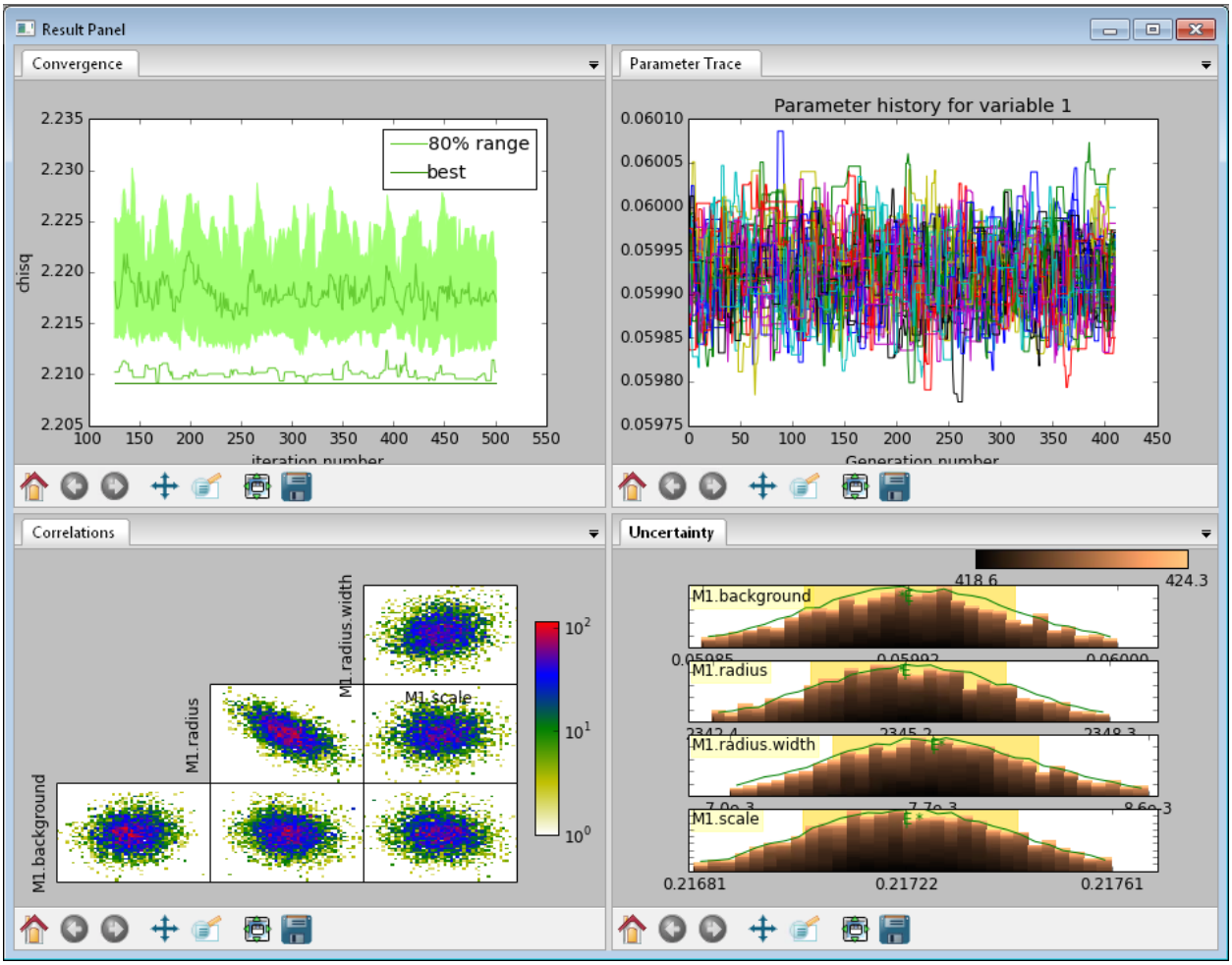


Fig. 2: This DREAM fit completed successfully. The *Convergence* plot is flat, the *Parameter Trace* plot is flat and messy indicating good mixing of the Markov chain values, the *Correlation* plots show nice defined blobs (with a bit of correlation between the *M1.radius* parameter and the *M1.radius. width* parameter), and the uncertainty plots show a narrow range of -log(P) values in the mostly brown histograms well-matched to the green constrained maximum likelihood line.

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.[9]

Table 1: Example fit output

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

The *Convergence* plot shows the range of χ^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 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 χ^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 $p_k == x$). 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 χ^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:

[9] JCGM. *Evaluation of measurement data — Supplement 1 to the "Guide to the expression of uncertainty in measurement" — Propagation of distributions using a Monte Carlo method*; Joint Committee for Guides in Metrology, JCGM 101:2008; Geneva, Switzerland, 2008; p 90. http://www.bipm.org/utils/common/documents/jcgm/JCGM_101_2008_E.pdf

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.

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

`--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.

`--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 `--fit=ps` to select particle swarm from the command line.

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

[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

[10] Berrett, T. B.; Samworth, R.J.; Yuan, M.; *Efficient multivariate entropy estimation via k-nearest neighbour distances.* Annals of Statistics 2019, 47 (1), 288-318. DOI: 10.1214/18-AOS1688

[11] Kennedy, J.; Eberhart, R. Particle Swarm Optimization *Proceedings of IEEE International Conference on Neural Networks. IV.* 1995; pp 1942–1948. DOI: 10.1109/ICNN.1995.48896

[12] Sahin, I. *An International Journal of Optimization and Control: Theories & Applications (IJOCTA)* 2013, 3 (2), 111–119.

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

`--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.

`--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 command line.

References

3.6.8 Parallel Tempering

Parallel tempering[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 command line.

[13] Swendsen, R. H.; Wang J. S. Replica Monte Carlo simulation of spin glasses *Physical Review Letters* 1986, 57, 2607-2609

References

3.7 Bumps Options

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.

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.

`--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 algorithgms (*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 the number 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:

amoeba	*Nelder-Mead Simplex*
de	*Differential Evolution*
dream	*DREAM*
lm	*Levenberg-Marquardt*
newton	*Quasi-Newton BFGS*
pt	*Parallel Tempering*
ps	*Particle Swarm*
rl	*Random Lines*

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 that 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

--export

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.

`--session`

Path to the HDF5 session file used to store the problem and the fit results. Run bumps with that session file to view the output and the plots.

`--resume`

Continue the most recent fit in the current session file.

`--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*.

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*.

Note: not currently available.

`--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.

Note: not currently available.

`--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 calcualation 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

Note: not currently available.

`--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.

3.7.7 Bumps Controls

`--chisq`

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

3.7.8 Special Options

`--webview`

If the command contains *webview* 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.

`--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.

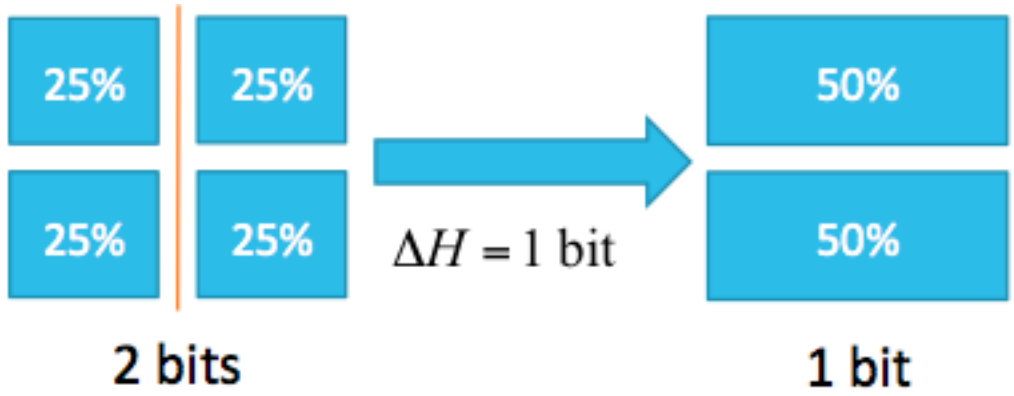
Note: not currently available.

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.

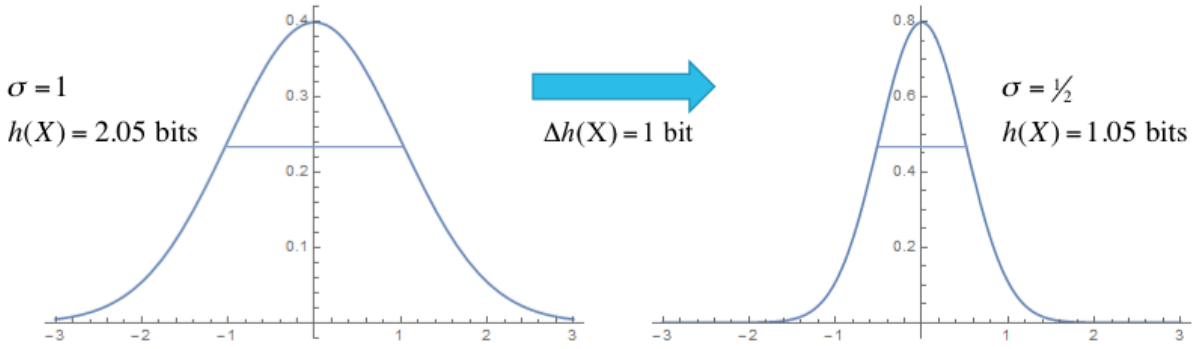
Extending this concept to continuous parameters, we use:

$$h(x) = -\int_{x \in X} p(x) \log_2(x) dx$$

For a parameter which is normally distributed, $x \sim N(\mu, \sigma)$, the entropy is:

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

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



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

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

where n is the number of fitting parameters and Σ 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 σ_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.

CHAPTER

FOUR

REFERENCE: BUMPS

4.1 bounds - Parameter constraints

pm	Return the tuple (~v-dv,~v+dv), where ~expr is a 'nice' number near to to the value of expr. For example::.
pmp	Return the tuple (~v-%v,~v+%v), where ~expr is a 'nice' number near to the value of expr. For example::.
pm_raw	Return the tuple [v-dv,v+dv].
pmp_raw	Return the tuple [v-%v,v+%v]
nice_range	Given a range, return an enclosing range accurate to two digits.
init_bounds	Returns a bounds object of the appropriate type given the arguments.
DistProtocol	Protocol for a distribution object, implementing the scipy.stats interface.
Bounds	Bounds abstract base class.
Unbounded	Unbounded parameter.
Bounded	Bounded range.
BoundedAbove	Semidefinite range bounded above.
BoundedBelow	Semidefinite range bounded below.
Distribution	Parameter is pulled from a distribution.
Normal	Parameter is pulled from a normal distribution.
BoundedNormal	truncated normal bounds
SoftBounded	Parameter is pulled from a stretched normal distribution.

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
```

(continues on next page)

(continued from previous page)

```
    range (-inf, base)
Bounded
    range (low, high)
Normal
    range (-inf, inf) with gaussian probability
BoundedNormal
    range (low, high) with gaussian probability within
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)
pmp_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
```

class bumps.bounds.**Bounded**(*lo: float*, *hi: float*)

Bases: *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.

property dof

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.

hi: float

property limits

lo: float

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.

penalty(*v*) → float

return a (differentiable) nonzero value when outside the bounds

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.

satisfied(*v*) → bool

start_value()

Return a default starting value if none given.

to_dict()

class bumps.bounds.**BoundedAbove**(*base: float*)

Bases: *Bounds*

Semidefinite range bounded above.

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

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.

base: float

property dof

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.

property limits

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.

penalty(*v*) → float

return a (differentiable) nonzero value when outside the bounds

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: float = 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.

satisfied(*v*) → bool

start_value()

Return a default starting value if none given.

to_dict()

class bumps.bounds.**BoundedBelow**(*base: float*)

Bases: *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.

base: float

property dof

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.

property limits

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.

penalty(*v*) → float

return a (differentiable) nonzero value when outside the bounds

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: float = 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.

satisfied(*v*) → bool

start_value()

Return a default starting value if none given.

to_dict()

type = 'BoundedBelow'

class bumps.bounds.**BoundedNormal**(*mean: float = 0*, *std: float = 1*, *limits=(-inf, inf)*, *hi='inf'*, *lo='-inf'*)

Bases: *Bounds*

truncated normal bounds

property dof

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.

hi: float | Literal['inf']

property limits

lo: float | Literal['-inf']

mean: float = 0.0

nllf(*value*)

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

penalty(*v*) → float

return a (differentiable) nonzero value when outside the bounds

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.

satisfied(*v*) → bool

start_value()

Return a default starting value if none given.

std: float = 1.0

to_dict()

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.

property dof

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.

property limits

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.

penalty(*v*) → float

return a (differentiable) nonzero value when outside the bounds

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.

satisfied(*v*) → bool

start_value()

Return a default starting value if none given.

to_dict()

class bumps.bounds.**DistProtocol**(**args*, ***kwargs*)

Bases: Protocol

Protocol for a distribution object, implementing the scipy.stats interface. (also including args, kwds and name)

args: Tuple[float, ...]

cdf(*value: float*) → float

kwds: Dict[str, Any]

name: str

nnlf(*value: float*) → float

pdf(*value: float*) → float

ppf(*value: float*) → float

rvs(*n: int*) → float

class bumps.bounds.**Distribution**(*dist*)

Bases: *Bounds*

Parameter is pulled from a distribution.

dist must implement the distribution interface from scipy.stats, described in the DistProtocol class.

dist: *DistProtocol* = None

property dof

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.

property limits

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.

penalty(*v*) → float

return a (differentiable) nonzero value when outside the bounds

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.

satisfied(*v*) → bool

start_value()

Return a default starting value if none given.

to_dict()

class bumps.bounds.**Normal**(*mean: float = 0.0*, *std: float = 1.0*)

Bases: *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.

class is 'frozen' because a new object should be created if *mean* or *std* are changed.

dist: *DistProtocol* = **None**

property dof

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.

property limits

mean: float = 0.0

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.

penalty(*v*) → float

return a (differentiable) nonzero value when outside the bounds

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.

satisfied(*v*) → bool

start_value()

Return a default starting value if none given.

std: float = 1.0

to_dict()

class bumps.bounds.**SoftBounded**(*lo: float = 0.0*, *hi: float = 1.0*, *std: float = 1.0*)

Bases: *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.

property dof

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.

hi: float = 1.0

property limits

lo: float = 0.0

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.

penalty(*v*) → float

return a (differentiable) nonzero value when outside the bounds

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.

satisfied(*v*) → bool

start_value()

Return a default starting value if none given.

std: float = 1.0

to_dict()

class bumps.bounds.**Unbounded**(**args*, ***kw*)

Bases: *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.

property dof

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.

property limits

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.

penalty(*v*) → float

return a (differentiable) nonzero value when outside the bounds

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.

satisfied(*v*) → bool

start_value()

Return a default starting value if none given.

to_dict()

type = 'Unbounded'

bumps.bounds.**init_bounds**(*v*) → *Bounds*

Returns a bounds object of the appropriate type given the arguments.

This is a helper factory to simplify the user interface to parameter objects.

bumps.bounds.**nice_range**(*bounds*)

Given a range, return an enclosing range accurate to two digits.

bumps.bounds.**pm**(*v*, *plus*, *minus=None*, *limits: Tuple[float | Literal['-inf'], float | Literal['inf']] | None = None*)

Return the tuple (~v-dv,~v+dv), where ~expr is a 'nice' number near to 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.**pm_raw**(*v*, *plus*, *minus=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**(*v*, *plus*, *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
>>> r = pmp(0.78421, 0.1)
>>> print("%g - %g"%r)
0.7834 - 0.785
```

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

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).

4.2 bspline - B-Spline interpolation library

bspline	Evaluate the B-spline with control points *y* at positions *xt* in [0,1].
pbs	Evaluate the parametric B-spline px(t),py(t).

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

profile	Evaluate the chebyshev approximation c at points x.
cheby_approx	Return the coefficients for the order n chebyshev approximation to function f evaluated over the range [low,high].
cheby_val	Evaluate the chebyshev approximation c at points x.
cheby_points	Return the points in at which a function must be evaluated to generate the order n Chebyshev approximation function.
cheby_coeff	Compute chebyshev coefficients for a polynomial of order n given the function evaluated at the chebyshev points for order n.

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:

$$f(x) \approx P_n(x) = \sum_{k=0}^{n} c_i T_k(x)$$

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.**cheby_approx**(*n*, *f*, *range=(0, 1)*)

Return the coefficients for the order n chebyshev approximation to function f evaluated over the range [low,high].

bumps.cheby.**cheby_coeff**(*fx*)

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 cheby_points(n).

bumps.cheby.**cheby_points**(*n*, *range=(0, 1)*)

Return the points in at 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.**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.**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()*.

4.4 cli - Command line interface

main	Run the bumps program with the command line interface.
install_plugin	Replace symbols in *bumps.plugin* with application specific methods.
set_mplconfig	Point the matplotlib config dir to %LOCALAPPDATA%{appdatadir}mplconfig.
config_matplotlib	Setup matplotlib to use a particular backend.
load_model	*** DEPRECATED***.
preview	Show the problem plots and parameters.
load_pars	Reload individual parameter values from a saved .par file.
save_best	Save the fit data, including parameter values, uncertainties and plots.
resynth	Generate maximum likelihood fits to resynthesized data sets.

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:

```
from . import fitplugin
from bumps.plugin import install_plugin
from bumps.plotutil import set_mplconfig
from bumps.cli import main as bumps_main
set_mplconfig(appdatadir='Refl1D')
install_plugin(fitplugin)
bumps_main()
```

After completing a set of fits on related systems, a post-analysis script can use *load_model()* to load the problem definition and *load_pars()* 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 systems.

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 matplotlib. To allow this, modules should not import matplotlib 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.**install_plugin**(*p*)

Replace symbols in *bumps.plugin* with application specific methods.

bumps.cli.**load_model**(*path: Path | str*, *model_options: list[str] | None = None*)

*** DEPRECATED***. Use fitproblem.load_problem(path, [args=...]) instead.

bumps.cli.**load_pars**(*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.**main**()

Run the bumps program with the command line interface.

Input parameters are taken from sys.argv.

bumps.cli.**preview**(*problem*, *view=None*)

Show the problem plots and parameters.

bumps.cli.**resynth**(*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.

bumps.cli.**save_best**(*fitdriver*, *problem=None*, *best=None*, *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.**set_mplconfig**(*appdatadir*)

Point the matplotlib config dir to %LOCALAPPDATA%{appdatadir}mplconfig.

4.5 curve - Model a fit function

Curve	Model a measurement with a user defined function.
PoissonCurve	Model a measurement with Poisson uncertainty.
plot_err	**DEPRECATED**: subclass Curve and override the plot function.

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*:

```
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:

```
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:

```
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:

```
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:

```
from sin_model import sin_model
```

class bumps.curve.**Curve**(*fn: Callable*, *x: _Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*, *y: _Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*, *dy: _Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str] | None = None*, *name: str | None = ''*, *labels: List[str] | None = None*, *plotter: Callable[[...], None] | None = None*, *plot_x: ndarray[tuple[Any, ...], dtype[_ScalarT]] | None = None*, *plot: Callable | None = None*, *pars: Dict[str,* Parameter*] | None = None*, *state: Dict[str, Any] | None = None*, ***kwargs*)

Bases: `object`

Model a measurement with a user defined function.

The function *fn(x,p1,p2,...)* 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 arguments after *x,y,dy,fy* in the plot function declaration. For example, *plot(x,y,dy,fy,p3,**kw)* will make the value of parameter *p3* 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.

dy: ndarray[tuple[Any, ...], dtype[_ScalarT]]

Data uncertainty

fn: Callable

labels: List[str]

name: str

Name of the model

nllf()

numpoints()

parameters()

pars: Dict[str, *Parameter*] = None

Fittable parameters to the model

plot(*view=None*)

plot_x: ndarray[tuple[Any, ...], dtype[_ScalarT]] | None

plotter: Callable | None

register_webview_plot(*plot_title: str*, *plot_function: Callable*, *change_with: Literal['parameter', 'uncertainty']*)

residuals()

save(*basename*)

simulate_data(*noise=None*)

state: Dict[str, Any]

Nonfittable parameters set during initialization. Values should be serializable.

theory(*x=None*)

update()

property webview_plots

x: ndarray[tuple[Any, ...], dtype[_ScalarT]]

y: ndarray[tuple[Any, ...], dtype[_ScalarT]]

class bumps.curve.**PoissonCurve**(*fn*, *x*, *y*, *dy=None*, *name=''*, ***fnkw*)

Bases: *Curve*

Model a measurement with Poisson uncertainty.

The nllf is calculated using Poisson probabilities, but the curve itself is displayed using the approximation that $\sigma_y \approx \sqrt{(y)}$.

Note that the *dy* argument is ignored. It is only present for deserialization of the Curve base class.

See *Curve* for details.

dy: ndarray[tuple[Any, ...], dtype[_ScalarT]]

Data uncertainty

fn: Callable

labels: `List[str]`

name: `str`

Name of the model

nllf()

numpoints()

parameters()

pars: `Dict[str, Parameter] = None`

Fittable parameters to the model

plot(*view=None*)

plot_x: `ndarray[tuple[Any, ...], dtype[_ScalarT]] | None`

plotter: `Callable | None`

register_webview_plot(*plot_title: str*, *plot_function: Callable*, *change_with: Literal['parameter', 'uncertainty']*)

residuals()

save(*basename*)

simulate_data(*noise=None*)

state: `Dict[str, Any]`

Nonfittable parameters set during initialization. Values should be serializable.

theory(*x=None*)

update()

property webview_plots

x: `ndarray[tuple[Any, ...], dtype[_ScalarT]]`

y: `ndarray[tuple[Any, ...], dtype[_ScalarT]]`

bumps.curve.**plot_err**(*x*, *y*, *dy*, *fy*, *view=None*, ***kw*)

DEPRECATED: subclass Curve and override the plot function.

Plot data *y* and error *dy* against *x*.

view is one of linear, log, logx or loglog.

4.6 data - Data handling utilities

indfloat	Convert string to float, with support for inf and nan.
parse_file	Parse a file into a header and data.

Data handling utilities.

bumps.data.**indfloat**(*s*)

Convert string to float, with support for inf and nan.

Example:

```
>>> from numpy import isinf, isnan
>>> print(isinf(indfloat('inf')))
True
>>> print(isinf(indfloat('-inf')))
True
>>> print(isnan(indfloat('nan')))
True
```

bumps.data.**parse_file**(*file*, *keysep=None*, *sep=None*, *comment='#'*)

Parse a file into a header and data.

Return a (header, data) pair, where header is a key: value dictionary and data is a numpy array.

The header section is list of key value pairs, with the *comment* character at the start of each line. Key and value will be separated by *keysep*, or by spaces if *keysep* = *None*. The data section is a sequence of floating point numbers separated by *sep*, or by spaces if *sep* is None. inf and nan are parsed as inf and nan. Comments at the end of the data line will be ignored. Data points can be commented out by including a comment character at the start of the data line, assuming the next character is a digit, plus, or decimal separator.

Quotes around keys are removed. For compatibility with the old interface, quotes around values are removed as well.

Special hack for binned data: if the first column contains bin edges, then the last row will only have the bin edge. To make the array square, we replace the bin edges with bin centers. The original bins can be found in the header using the 'bins' key (unless that key already exists in the header, in which case the key will be ignored).

4.7 errplot - Plot sample profile uncertainty

reload_errors	Reload the MCMC state and compute the model confidence intervals.
calc_errors_from_state	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.
calc_errors	Align the sample profiles and compute the residual difference from the measured data for a set of points.
show_errors	Display the confidence regions returned by *calc_errors()*.

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.**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.**calc_errors_from_state**(*problem*, *state*, *nshown=50*, *random=True*, *portion: float | None = None*)

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.

Returns *errs* for *show_errors()*.

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()*.

bumps.errplot.**show_errors**(*errs*, *fig=None*, *save=None*)

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	*models* is a sequence of *Fitness* instances. Note that they

continues on next page

Table 8 – continued from previous page

CovarianceMixin	Add methods for *cov*, *show_cov* and *show_err* to a bumps problem definition.
load_problem	Load a model file.

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:

```
# 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
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_problem
path: str  # set by load_problem
name: str # set by load_problem
title: str = filename # set by load_problem
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.**CovarianceMixin**

Bases: `object`

Add methods for *cov*, *show_cov* and *show_err* to a bumps problem definition.

This is done as a mixin because not all problems are FitProblem. See for example *bumps.pdfwrapper.PDF*.

cov(*x*)

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 has residuals available, 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)
```

show_cov(*x*, *cov*)

show_err(*x*, *dx*)

Display the error approximation from the covariance matrix.

err is the standard deviation computed from the covariance matrix. It is available as *result.dx* from the simple fitter, or using:

```
from bumps import lsqerror

dx = lsqerror.stderr(problem.cov(x))
```

Warning: cost to compute cov grows as the cube of the number of parameters.

class bumps.fitproblem.**FitProblem**(*models: FitnessType | List[FitnessType]*, *weights=None*, *name=None*, *constraints=None*, *penalty_nllf=None*, *freevars=None*, *auto_tag=False*)

Bases: Generic[FitnessType], *CovarianceMixin*

models is a sequence of *Fitness* instances. Note that they do not need to all be of the same class.

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 list of Constraint objects, which have a method to calculate the nllf for that constraint. Also supports an alternate form which cannot be serialized: 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.

penalty_nllf is the nllf to use for *fitness* when *constraints* or model parameter bounds are not satisfied. The total nllf is the squared distance from the boundary plus the penalty so that the derivative points the search back to the feasible region. The penalty should be larger than any nllf you might see near the boundary so that the fit doesn't get stuck outside, but small enough that penalty plus distance is different from penalty. The default is 1e12.

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.

bounds()

Return the bounds for each parameter as a 2 x N array

chisq(*nllf: float | ndarray[tuple[Any, ...], dtype[_ScalarT]] | None = None*, *norm: bool = True*, *compact: bool = True*)

Returns chisq as a floating point value.

See documentation for *chisq_str()*.

chisq_str(*nllf: float | None = None*, *norm: bool = True*, *compact: bool = True*)

Return a string representing the chisq equivalent of the nllf.

If *nllf* is provided then use that instead of calling the model evaluator. Fail if *compact* is False.

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 cost of the penalty constraints in the total nllf.

Parameter priors, if any, are treated as independent models in the total nllf. The constraint value is displayed separately.

Deprecated: *norm:bool* and *compact:bool* are ignored.

constraints: Sequence[*Constraint*] | None

constraints_nllf() → Tuple[float, List[str]]

Return the cost function for all constraints

cov(*x*)

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 has residuals available, 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)
```

property dof

property fitness

freevars: *FreeVariables* | None

getp()

Returns the current value of the parameter vector.

property has_residuals

True if all underlying fitness functions define residuals.

labels() → List[str]

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

property models

Iterate over models, with free parameters set from model values

name: str | None

nllf(*pvec=None*) → float

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 any of the parameters are out of bounds, or any of the constraints are not satisfied, but instead are assigned a value of *penalty_nllf*. This will prevent expensive models from spending time computing values in the unfeasible region.

property num_models

parameter_nllf() → Tuple[float, List[str]]

Returns negative log likelihood of seeing parameters p.

parameter_residuals()

Returns negative log likelihood of seeing parameters p.

property parameters

Return the list of fitted parameters.

penalty_nllf: float | Literal['inf']

plot(*p=None*, *fignum=1*, *figfile=None*, *view=None*, *model_indices=None*)

push_model(*index*)

Fetch model *index* with the appropriate free variables substituted.

On completion of the context, restore the parameters for the active model.

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()

restore_data()

Restore original data after resynthesis.

resynth_data()

Resynthesize data with noise from the uncertainty estimates.

save(*basename*)

set_active_model(*index*)

Fetch model *index* with the appropriate free variables substituted.

This will remain the active model until a new active model is selected.

Operations like chisq_str() or plot() which cycle through the models will restore the parameters upon completion.

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()

show_cov(*x*, *cov*)

show_err(*x*, *dx*)

Display the error approximation from the covariance matrix.

err is the standard deviation computed from the covariance matrix. It is available as *result.dx* from the simple fitter, or using:

```
from bumps import lsqerror

dx = lsqerror.stderr(problem.cov(x))
```

Warning: cost to compute cov grows as the cube of the number of parameters.

simulate_data(*noise=None*)

Simulate data with added noise

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

weights: List[float] | Literal[None]

class bumps.fitproblem.**Fitness**(**args*, ***kwargs*)

Bases: Protocol

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() → List[*Parameter*]

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.

update()

Called when parameters have been updated. Any cached values will need to be cleared and the model reevaluated.

bumps.fitproblem.**load_problem**(*path: Path | str*, *args: list[str] | None = None*)

Load a model file.

path contains the path to the model file. This could be a python script or a previously saved problem, serialized as .json, .cloudpickle, .pickle or .dill

args are any additional arguments to the model. The sys.argv variable will be set such that *sys.argv[1:] == model_options*.

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: *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

BFGSFit	BFGS quasi-newton optimizer.
CheckpointMonitor	Periodically save fit state so that it can be resumed later.
ConsoleMonitor	Display fit progress on the console
DEFit	Classic Storn and Price differential evolution optimizer.
DreamFit	
DreamModel	DREAM wrapper for fit problems.
FitBase	FitBase defines the interface from bumps models to the various fitting engines available within bumps.
FitDriver	
LevenbergMarquardtFit	Levenberg-Marquardt optimizer.
MPFit	MPFit optimizer.
MonitorRunner	Adaptor which allows solvers to accept progress monitors.
MultiStart	Multi-start monte carlo fitter.
PSFit	Particle swarm optimizer.
PTFit	Parallel tempering optimizer.
RLFit	Random lines optimizer.
Resampler	
SimplexFit	Nelder-Mead simplex optimizer.
SnobFit	
StepMonitor	Collect information at every step of the fit and save it to a file.
fit	Simplified fit interface.
fit_result_summary	Return a JSON-serializable summary of the fit *results*.
get_fit_from_webview	Retrieve the current problem and fit result from webview.
help	
load_fit_from_export	Reload a bumps export directory.
load_fit_from_session	Reload the results from a bumps session file.

continues on next page

Table 10 – continued from previous page

load_session	Reload a session file from a saved hdf
parse_tolerance	
plot_convergence	Create a convergence plot from the simple fitter results.
register	Register a new fitter with bumps, if it is not already there.
save_best	Save the fit data, including parameter values, uncertainties and plots.
save_fit	Write a fit to a session file.
save_fit_result	Save the *fit_result_summary()* of *fit* to *filename* as JSON, conventionally <*model*>*-fit.json* in the export directory.
show_results	
show_table	
test_fit_result_summary	
test_fitters	Run the fit tests to make sure they work.

Interfaces to various optimizers.

class bumps.fitters.**BFGSFit**(*problem:* bumps.fitproblem.FitProblem)

Bases: *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 numerical differentiation.

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)
```

static h5dump(*group: Group*, *state: Any*) → None

Store fitter.state into the given HDF5 Group.

This will be restored by the corresponding h5load, then passed to the fitter to resume from its current state. This strategy is particularly useful for MCMC analysis where you may need more iterations for the chains to reach equilibrium. It is also the basis of checkpoint/restore operations for fitters such as de and amoeba which manage a population, though in those cases the best point seen so far may be good enough.

static h5load(*group: Group*) → Any

Load internal fit state from the group saved by h5dump. Note that this function will be responsible for migrating state from older versions to newer versions of the saved representation.

id = 'newton'

Short name for the fit method, used as –id on the command line.

classmethod max_steps(*problem:* bumps.fitproblem.FitProblem | *None* = *None*, *options: Dict[str, Any]* | *None* = *None*) → int

Return the maximum number of steps the fitter will take based on the given options.

Default implementation just returns the 'steps' option multiplied by 'starts',

name = 'Quasi-Newton BFGS'

Display name for the fit method

problem

The problem that is being fit.

settings = [('steps', 3000), ('ftol', 1e-06), ('xtol', 1e-12), ('starts', 1), ('jump', 0)]

Available fitting options and their default values.

solve(*monitors:* MonitorRunner, *mapper=None*, ***options*)

state = None

Internal fit state. If the state object has a draw method this should return a set of points from the posterior probability distribution for the fit.

class bumps.fitters.**CheckpointMonitor**(*checkpoint*, *progress=1800*)

Bases: *TimedUpdate*

Periodically save fit state so that it can be resumed later.

checkpoint: Callable[None, None] = 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*)

show_progress(*history*)

class bumps.fitters.**ConsoleMonitor**(*problem*, *progress=1*, *improvement=30*)

Bases: *TimedUpdate*

Display fit progress on the console

config_history(*history*)

Indicate which fields are needed by the monitor and for what duration.

final(*history:* History, *best: Dict[str, Any]*)

info(*message: str*)

show_improvement(*history*)

show_progress(*history*)

class bumps.fitters.**DEFit**(*problem:* bumps.fitproblem.FitProblem)

Bases: *FitBase*

Classic Storn and Price differential evolution optimizer.

static h5dump(*group: Group*, *state: Dict[str, Any]*)

Store fitter.state into the given HDF5 Group.

This will be restored by the corresponding h5load, then passed to the fitter to resume from its current state. This strategy is particularly useful for MCMC analysis where you may need more iterations for the chains to reach equilibrium. It is also the basis of checkpoint/restore operations for fitters such as de and amoeba which manage a population, though in those cases the best point seen so far may be good enough.

static h5load(*group: Group*) → Any

Load internal fit state from the group saved by h5dump. Note that this function will be responsible for migrating state from older versions to newer versions of the saved representation.

id = 'de'

Short name for the fit method, used as –id on the command line.

load(*input_path*)

classmethod max_steps(*problem:* bumps.fitproblem.FitProblem | *None = None*, *options: Dict[str, Any]* | *None = None*) → int

Return the maximum number of steps the fitter will take based on the given options.

Default implementation just returns the 'steps' option multiplied by 'starts',

name = 'Differential Evolution'

Display name for the fit method

problem

The problem that is being fit.

save(*output_path*)

settings = [('steps', 1000), ('pop', 10), ('CR', 0.9), ('F', 2.0), ('ftol', 1e-08), ('xtol', 1e-06)]

Available fitting options and their default values.

solve(*monitors:* MonitorRunner, *mapper=None*, ***options*)

state = None

Internal fit state. If the state object has a draw method this should return a set of points from the posterior probability distribution for the fit.

class bumps.fitters.**DreamFit**(*problem*)

Bases: *FitBase*

entropy(***kw*)

error_plot(*figfile*)

static h5dump(*group: Group*, *state:* MCMCDraw)

Store fitter.state into the given HDF5 Group.

This will be restored by the corresponding h5load, then passed to the fitter to resume from its current state. This strategy is particularly useful for MCMC analysis where you may need more iterations for the chains to reach equilibrium. It is also the basis of checkpoint/restore operations for fitters such as de and amoeba which manage a population, though in those cases the best point seen so far may be good enough.

static h5load(*group: Group*) → *MCMCDraw*

Load internal fit state from the group saved by h5dump. Note that this function will be responsible for migrating state from older versions to newer versions of the saved representation.

id = 'dream'

Short name for the fit method, used as –id on the command line.

load(*input_path*)

classmethod max_steps(*problem:* bumps.fitproblem.FitProblem, *options: Dict[str, Any]* | *None = None*) → int

Return the maximum number of steps the fitter will take based on the given options.

Default implementation just returns the 'steps' option multiplied by 'starts',

name = 'DREAM'

Display name for the fit method

plot(*output_path*)

problem

The problem that is being fit.

save(*output_path*)

settings = [('samples', 10000), ('burn', 100), ('pop', 10), ('init', 'eps'), ('thin', 1), ('alpha', 0.0), ('outliers', 'iqr'), ('trim', True), ('steps', 0)]

Available fitting options and their default values.

show()

solve(*monitors:* MonitorRunner, *mapper=None*, ***options*)

state = None

Internal fit state. If the state object has a draw method this should return a set of points from the posterior probability distribution for the fit.

stderr()

Approximate standard error as 1/2 the 68% interval fo 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: object

DREAM wrapper for fit problems. Implements dream.core.Model protocol.

bounds: ndarray[tuple[Any, ...], dtype[_ScalarT]]

labels: List[str]

map(*pop*)

class bumps.fitters.**FitBase**(*problem:* bumps.fitproblem.FitProblem)

Bases: 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 *name* attribute is the name of the optimizer. This is just a simple string.

The *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 *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 *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 *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 *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.

static h5dump(*group: Group*, *state: Any*) → None

Store fitter.state into the given HDF5 Group.

This will be restored by the corresponding h5load, then passed to the fitter to resume from its current state. This strategy is particularly useful for MCMC analysis where you may need more iterations for the chains to reach equilibrium. It is also the basis of checkpoint/restore operations for fitters such as de and amoeba which manage a population, though in those cases the best point seen so far may be good enough.

static h5load(*group: Group*) → Any

Load internal fit state from the group saved by h5dump. Note that this function will be responsible for migrating state from older versions to newer versions of the saved representation.

id: str

Short name for the fit method, used as –id on the command line.

classmethod max_steps(*problem:* bumps.fitproblem.FitProblem | *None = None*, *options: Dict[str, Any] | None = None*) → int

Return the maximum number of steps the fitter will take based on the given options.

Default implementation just returns the 'steps' option multiplied by 'starts',

name: str

Display name for the fit method

problem: *bumps.fitproblem.FitProblem*

The problem that is being fit.

settings: List[Tuple[str, Any]]

Available fitting options and their default values.

solve(*monitors:* MonitorRunner, *mapper=None*, ***options*)

state: Any = None

Internal fit state. If the state object has a draw method this should return a set of points from the posterior probability distribution for the fit.

class bumps.fitters.**FitDriver**(*fitclass=None*, *problem=None*, *monitors=None*, *abort_test=None*, *mapper=None*, *time=0.0*, ***options*)

Bases: `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*, *fit_state=None*)

Providing *fit_state* allows the fit to resume from a previous state. If None then the fit will be started from a clean state.

The *fitclass* object should provide static methods for *h5dump/h5load* for saving and loading the internal state of the fitter to a specific group in an hdf5 file. The result of *h5load(group)* can be passed as *fit_state* to resume a fit with whatever new options are provided. It is up to the fitter to decide how to interpret this. The state can be retrieved from state=driver.fitter.state at the end of the fit and saved using *h5dump(group, state)*.

resume (= resume_path / problem.name) is used by the pre-1.0 command line interface to provide the base path for the fit state files. For dream this can be replaced by the following:

```
fn, labels = getattr(problem, "derive_vars", (None, []))
fit_state = load_state(input_path, report=100, derived_vars=len(labels))
```

load(*input_path*)

plot(*output_path*, *view=None*)

save(*output_path*)

show()

show_cov()

show_entropy(*method=None*)

show_err()

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:* bumps.fitproblem.FitProblem)

Bases: *FitBase*

Levenberg-Marquardt optimizer.

cov()

static h5dump(*group: Group*, *state: Any*) → None

Store fitter.state into the given HDF5 Group.

This will be restored by the corresponding h5load, then passed to the fitter to resume from its current state. This strategy is particularly useful for MCMC analysis where you may need more iterations for the chains to reach equilibrium. It is also the basis of checkpoint/restore operations for fitters such as de and amoeba which manage a population, though in those cases the best point seen so far may be good enough.

static h5load(*group: Group*) → Any

Load internal fit state from the group saved by h5dump. Note that this function will be responsible for migrating state from older versions to newer versions of the saved representation.

id = 'scipy.leastsq'

Short name for the fit method, used as –id on the command line.

classmethod max_steps(*problem:* bumps.fitproblem.FitProblem | *None* = *None*, *options: Dict[str, Any]* | *None* = *None*) → int

Return the maximum number of steps the fitter will take based on the given options.

Default implementation just returns the 'steps' option multiplied by 'starts',

name = 'Levenberg-Marquardt (scipy.leastsq)'

Display name for the fit method

problem

The problem that is being fit.

settings = [('steps', 200), ('ftol', 1.5e-08), ('xtol', 1.5e-08)]

Available fitting options and their default values.

solve(*monitors:* MonitorRunner, *mapper=None*, ***options*)

state = None

Internal fit state. If the state object has a draw method this should return a set of points from the posterior probability distribution for the fit.

class bumps.fitters.**MPFit**(*problem:* bumps.fitproblem.FitProblem)

Bases: *FitBase*

MPFit optimizer.

static h5dump(*group: Group*, *state: Any*) → None

Store fitter.state into the given HDF5 Group.

This will be restored by the corresponding h5load, then passed to the fitter to resume from its current state. This strategy is particularly useful for MCMC analysis where you may need more iterations for the chains

to reach equilibrium. It is also the basis of checkpoint/restore operations for fitters such as de and amoeba which manage a population, though in those cases the best point seen so far may be good enough.

static h5load(*group: Group*) → Any

Load internal fit state from the group saved by h5dump. Note that this function will be responsible for migrating state from older versions to newer versions of the saved representation.

id = 'lm'

Short name for the fit method, used as –id on the command line.

classmethod max_steps(*problem:* bumps.fitproblem.FitProblem | *None = None*, *options: Dict[str, Any]* | *None = None*) → int

Return the maximum number of steps the fitter will take based on the given options.

Default implementation just returns the 'steps' option multiplied by 'starts',

name = 'Levenberg-Marquardt'

Display name for the fit method

problem

The problem that is being fit.

settings = [('steps', 200), ('ftol', 1e-10), ('xtol', 1e-10), ('starts', 1), ('jump', 0)]

Available fitting options and their default values.

solve(*monitors=None*, *mapper=None*, ***options*)

state = None

Internal fit state. If the state object has a draw method this should return a set of points from the posterior probability distribution for the fit.

class bumps.fitters.**MonitorRunner**(*monitors: List[*Monitor*]*, *problem*, *abort_test=None*, *max_time=0.0*)

Bases: object

Adaptor which allows solvers to accept progress monitors.

The stopping() method manages checks for abort and timeout.

final(*point: ndarray[tuple[Any, ...], dtype[_ScalarT]]*, *value: float*)

info(*message: str*)

stopping()

update(*step: int*, *point: ndarray[tuple[Any, ...], dtype[_ScalarT]]*, *value: float*, *population_points: ndarray[tuple[Any, ...], dtype[_ScalarT]]* | *None = None*, *population_values: ndarray[tuple[Any, ...], dtype[_ScalarT]]* | *None = None*)

class bumps.fitters.**MultiStart**(*fitter:* FitBase)

Bases: *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 jump radius is non-zero, then restart near the best fit, otherwise restart at random.

static h5dump(*group: Group*, *state: Any*) → None

Store fitter.state into the given HDF5 Group.

This will be restored by the corresponding h5load, then passed to the fitter to resume from its current state. This strategy is particularly useful for MCMC analysis where you may need more iterations for the chains to reach equilibrium. It is also the basis of checkpoint/restore operations for fitters such as de and amoeba which manage a population, though in those cases the best point seen so far may be good enough.

static h5load(*group: Group*) → Any

Load internal fit state from the group saved by h5dump. Note that this function will be responsible for migrating state from older versions to newer versions of the saved representation.

id

Short name for the fit method, used as –id on the command line.

classmethod max_steps(*problem:* bumps.fitproblem.FitProblem | *None = None*, *options: Dict[str, Any] | None = None*) → int

Return the maximum number of steps the fitter will take based on the given options.

Default implementation just returns the 'steps' option multiplied by 'starts',

name = 'Multistart Monte Carlo'

Display name for the fit method

problem

The problem that is being fit.

settings = [('starts', 100), ('jump', 0)]

Available fitting options and their default values.

solve(*monitors:* MonitorRunner, *mapper=None*, ***options*)

state = None

Internal fit state. If the state object has a draw method this should return a set of points from the posterior probability distribution for the fit.

class bumps.fitters.**PSFit**(*problem:* bumps.fitproblem.FitProblem)

Bases: *FitBase*

Particle swarm optimizer.

static h5dump(*group: Group*, *state: Any*) → None

Store fitter.state into the given HDF5 Group.

This will be restored by the corresponding h5load, then passed to the fitter to resume from its current state. This strategy is particularly useful for MCMC analysis where you may need more iterations for the chains to reach equilibrium. It is also the basis of checkpoint/restore operations for fitters such as de and amoeba which manage a population, though in those cases the best point seen so far may be good enough.

static h5load(*group: Group*) → Any

Load internal fit state from the group saved by h5dump. Note that this function will be responsible for migrating state from older versions to newer versions of the saved representation.

id = 'ps'

Short name for the fit method, used as –id on the command line.

classmethod max_steps(*problem:* bumps.fitproblem.FitProblem | *None* = *None*, *options: Dict[str, Any]* | *None* = *None*) → int

Return the maximum number of steps the fitter will take based on the given options.

Default implementation just returns the 'steps' option multiplied by 'starts',

name = 'Particle Swarm'

Display name for the fit method

problem

The problem that is being fit.

settings = [('steps', 3000), ('pop', 1)]

Available fitting options and their default values.

solve(*monitors:* MonitorRunner, *mapper=None*, ***options*)

state = None

Internal fit state. If the state object has a draw method this should return a set of points from the posterior probability distribution for the fit.

class bumps.fitters.**PTFit**(*problem:* bumps.fitproblem.FitProblem)

Bases: *FitBase*

Parallel tempering optimizer.

static h5dump(*group: Group*, *state: Any*) → None

Store fitter.state into the given HDF5 Group.

This will be restored by the corresponding h5load, then passed to the fitter to resume from its current state. This strategy is particularly useful for MCMC analysis where you may need more iterations for the chains to reach equilibrium. It is also the basis of checkpoint/restore operations for fitters such as de and amoeba which manage a population, though in those cases the best point seen so far may be good enough.

static h5load(*group: Group*) → Any

Load internal fit state from the group saved by h5dump. Note that this function will be responsible for migrating state from older versions to newer versions of the saved representation.

id = 'pt'

Short name for the fit method, used as –id on the command line.

classmethod max_steps(*problem:* bumps.fitproblem.FitProblem | *None* = *None*, *options: Dict[str, Any]* | *None* = *None*) → int

Return the maximum number of steps the fitter will take based on the given options.

Default implementation just returns the 'steps' option multiplied by 'starts',

name = 'Parallel Tempering'

Display name for the fit method

problem

The problem that is being fit.

settings = [('steps', 400), ('nT', 24), ('CR', 0.9), ('burn', 100), ('Tmin', 0.1), ('Tmax', 10.0)]

Available fitting options and their default values.

solve(*monitors:* MonitorRunner, *mapper=None*, ***options*)

state = None

Internal fit state. If the state object has a draw method this should return a set of points from the posterior probability distribution for the fit.

class bumps.fitters.**RLFit**(*problem:* bumps.fitproblem.FitProblem)

Bases: *FitBase*

Random lines optimizer.

static h5dump(*group: Group*, *state: Any*) → None

Store fitter.state into the given HDF5 Group.

This will be restored by the corresponding h5load, then passed to the fitter to resume from its current state. This strategy is particularly useful for MCMC analysis where you may need more iterations for the chains to reach equilibrium. It is also the basis of checkpoint/restore operations for fitters such as de and amoeba which manage a population, though in those cases the best point seen so far may be good enough.

static h5load(*group: Group*) → Any

Load internal fit state from the group saved by h5dump. Note that this function will be responsible for migrating state from older versions to newer versions of the saved representation.

id = 'rl'

Short name for the fit method, used as –id on the command line.

classmethod max_steps(*problem:* bumps.fitproblem.FitProblem | *None = None*, *options: Dict[str, Any] | None = None*) → int

Return the maximum number of steps the fitter will take based on the given options.

Default implementation just returns the 'steps' option multiplied by 'starts',

name = 'Random Lines'

Display name for the fit method

problem

The problem that is being fit.

settings = [('steps', 3000), ('pop', 0.5), ('CR', 0.9), ('starts', 20), ('jump', 0)]

Available fitting options and their default values.

solve(*monitors:* MonitorRunner, *mapper=None*, ***options*)

state = None

Internal fit state. If the state object has a draw method this should return a set of points from the posterior probability distribution for the fit.

class bumps.fitters.**Resampler**(*fitter*)

Bases: *FitBase*

static h5dump(*group: Group*, *state: Any*) → None

Store fitter.state into the given HDF5 Group.

This will be restored by the corresponding h5load, then passed to the fitter to resume from its current state. This strategy is particularly useful for MCMC analysis where you may need more iterations for the chains to reach equilibrium. It is also the basis of checkpoint/restore operations for fitters such as de and amoeba which manage a population, though in those cases the best point seen so far may be good enough.

static h5load(*group: Group*) → Any

Load internal fit state from the group saved by h5dump. Note that this function will be responsible for migrating state from older versions to newer versions of the saved representation.

id

Short name for the fit method, used as –id on the command line.

classmethod max_steps(*problem:* bumps.fitproblem.FitProblem | *None* = *None*, *options: Dict[str, Any]* | *None* = *None*) → int

Return the maximum number of steps the fitter will take based on the given options.

Default implementation just returns the 'steps' option multiplied by 'starts',

name

Display name for the fit method

problem

The problem that is being fit.

settings

Available fitting options and their default values.

solve(***options*)

state = None

Internal fit state. If the state object has a draw method this should return a set of points from the posterior probability distribution for the fit.

class bumps.fitters.**SimplexFit**(*problem:* bumps.fitproblem.FitProblem)

Bases: *FitBase*

Nelder-Mead simplex optimizer.

static h5dump(*group: Group*, *state: Any*) → None

Store fitter.state into the given HDF5 Group.

This will be restored by the corresponding h5load, then passed to the fitter to resume from its current state. This strategy is particularly useful for MCMC analysis where you may need more iterations for the chains to reach equilibrium. It is also the basis of checkpoint/restore operations for fitters such as de and amoeba which manage a population, though in those cases the best point seen so far may be good enough.

static h5load(*group: Group*) → Any

Load internal fit state from the group saved by h5dump. Note that this function will be responsible for migrating state from older versions to newer versions of the saved representation.

id = 'amoeba'

Short name for the fit method, used as –id on the command line.

classmethod max_steps(*problem:* bumps.fitproblem.FitProblem | *None* = *None*, *options: Dict[str, Any]* | *None* = *None*) → int

Return the maximum number of steps the fitter will take based on the given options.

Default implementation just returns the 'steps' option multiplied by 'starts',

name = 'Nelder-Mead Simplex'

Display name for the fit method

problem

The problem that is being fit.

settings = [('steps', 1000), ('radius', 0.15), ('xtol', 1e-06), ('ftol', 1e-08), ('starts', 1), ('jump', 0.01)]

Available fitting options and their default values.

solve(*monitors:* MonitorRunner, *mapper=None*, ***options*)

state = None

Internal fit state. If the state object has a draw method this should return a set of points from the posterior probability distribution for the fit.

class bumps.fitters.**SnobFit**(*problem:* bumps.fitproblem.FitProblem)

Bases: *FitBase*

static h5dump(*group: Group*, *state: Any*) → None

Store fitter.state into the given HDF5 Group.

This will be restored by the corresponding h5load, then passed to the fitter to resume from its current state. This strategy is particularly useful for MCMC analysis where you may need more iterations for the chains to reach equilibrium. It is also the basis of checkpoint/restore operations for fitters such as de and amoeba which manage a population, though in those cases the best point seen so far may be good enough.

static h5load(*group: Group*) → Any

Load internal fit state from the group saved by h5dump. Note that this function will be responsible for migrating state from older versions to newer versions of the saved representation.

id = 'snobfit'

Short name for the fit method, used as –id on the command line.

classmethod max_steps(*problem:* bumps.fitproblem.FitProblem | *None = None*, *options: Dict[str, Any] | None = None*) → int

Return the maximum number of steps the fitter will take based on the given options.

Default implementation just returns the 'steps' option multiplied by 'starts',

name = 'SNOBFIT'

Display name for the fit method

problem

The problem that is being fit.

settings = [('steps', 200)]

Available fitting options and their default values.

solve(*monitors:* MonitorRunner, *mapper=None*, ***options*)

state = None

Internal fit state. If the state object has a draw method this should return a set of points from the posterior probability distribution for the fit.

class bumps.fitters.**StepMonitor**(*problem*, *fid*, *fields=['step', 'time', 'value', 'point']*)

Bases: *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.

update(*history*)

bumps.fitters.**fit**(*problem*, *method: str = 'amoeba'*, *session: Path | str | None = None*, *resume: OptimizeResult | None = None*, *export: Path | str | None = None*, *name: str | None = None*, *verbose: bool = False*, *parallel: int = 1*, ***options*) → OptimizeResult

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.

The keyword argments mostly follow the argument names used in the bumps command line interface. Try use *method=method* rather than *fit=method*)

Returns a scipy *OptimizeResult* object containing *x* and *dx*, plus *labels* (the fitted parameter labels, matching the order of x and dx), *numpoints* (number of data points) and *chisq* (normalized chi-squared at the best point, at full precision). Some fitters also include a *result.state* object. For dream this can be used as *bumps.dream.views.plot_all(result.state)* to generate the uncertainty plots. Use *plot_convergence(result)* to show how the fit progressed. *result.convergence* is a vector of current value for each step, or an array with best and population quantiles, [best, 0%, 20%, 50%, 80%, 100%].

Note: All fits report *result.success=True* even if they did not converge. Similarly *result.status=0* and *result.message="successful termination"*.

If *session=path/file.h5* is provided, append the fit to the session history, or create a new session if the file doesn't exist.

If *resume=result* is provided, then attempt to resume the fit from the previous result.

If *export=path* is provided, generate the standard plots and export files to the specified directory.

You can set *name=basename* as the base file name for all files stored in the export directory, or use *name=problem.name* as the default.

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.

If *parallel=n* is provided, then run on *n* separate cpus. By default *parallel=1* to run on a single cpu. For slow functions set *parallel=0* to run on all cpus. You want to run on a single cpu if your function is already parallel (for example using multiprocessing or using gpu code), or if your function is so fast that the overhead of transfering data is higher than cost of *n* function calls.

bumps.fitters.**fit_result_summary**(*results: OptimizeResult*) → dict

Return a JSON-serializable summary of the fit *results*.

The summary contains every entry from *results* except the fitter state and the convergence history. This includes the scipy-convention entries (x, dx, fun, success, status, message, nit, ...) plus *labels* (the fitted parameter labels, matching the order of x and dx), *numpoints* (number of data points) and *chisq* (normalized chi-squared at the best point, at full precision). For a model with gaussian independent uncertainties the raw sum of squares is chisq*dof.

async bumps.fitters.**get_fit_from_webview**(*wait: bool = False*)

Retrieve the current problem and fit result from webview.

If *wait* then wait for any running fit to complete before fetching.

Returns the problem and an OptimizerResult similar to the simple fit() result

bumps.fitters.**help**(**shown*)

bumps.fitters.**load_fit_from_export**(*path: Path | str*, *modelfile: Path | str | None = None*, *args: list[str] | None = None*)

Reload a bumps export directory.

path is the path to the directory, or to a <model>.par file within that directory. Use the <model>.par file if you have multiple models exported to the same path.

If *modelfile* is provided then use it, otherwise use <model>.py in the current directory. That means you can change to the directory containing your model then run bumps with –reload-export=path without having to list <model>.py on the command line. This is handy if you have several variations saved to different filenames stored along with your data.

sys.argv is set to *args* before loading the model.

bumps.fitters.**load_fit_from_session**(*filename: Path | str*)

Reload the results from a bumps session file.

filename is a path to an HDF5 file.

Returns (*problem*, *results*) where *problem* contains the model executable and *results* is a scipy OptimizationResults object. We add *state* if available and *convergence* to the results, similar to simple fit.

Use *load_session(filename)* directly if you want more control.

bumps.fitters.**load_session**(*filename*)

Reload a session file from a saved hdf

bumps.fitters.**parse_tolerance**(*options*)

bumps.fitters.**plot_convergence**(*results*, *cutoff=0.25*, *ax=None*)

Create a convergence plot from the simple fitter results.

Note that this will not work with a webview FitResults object returned from a session file. Those will not have dof defined, and use fit_state instead of state.

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_best**(*fitdriver*, *problem=None*, *best=None*, *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.fitters.**save_fit**(*path: Path | str*, *problem*, *fit: OptimizeResult*, *label: str | None = None*)

Write a fit to a session file.

bumps.fitters.**save_fit_result**(*problem*, *fit*, *filename*)

Save the *fit_result_summary()* of *fit* to *filename* as JSON, conventionally *<model>-fit.json* in the export directory.

fit may be an `OptimizeResult` or a webview `FitResult`; the latter is converted to an `OptimizeResult` for the summary.

bumps.fitters.**show_results**(*problem*, *results: OptimizeResult*)

bumps.fitters.**show_table**(*problem*, *results*)

bumps.fitters.**test_fit_result_summary**()

bumps.fitters.**test_fitters**()

Run the fit tests to make sure they work.

4.11 history - Optimizer evaluation trace

History	Collection of traces.
Trace	Value trace.

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 defined using *History.provides()*. For example, the following adds traces for 'value' and 'point' to the history, and requires the value on the two previous cycles in order to do its work:

```
>>> from bumps.history import History
>>> h = History(value=2, point=0) # keep two values and zero points
```

Initially the history is empty:

```
>>> print(len(h.value))
0
```

After three updates we see that only two values are kept.

```
>>> h.update(value=2.6, 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
```

Since the required length of 'point' is zero no values are kept:

```
>>> print(h.point[0])
Traceback (most recent call last):
    ...
IndexError: point has not accumulated enough history
```

A history consumer can override this, and require a certain length of a trace. Then future values will be preserved:

```
>>> h.requires(point=1)
>>> h.update(value=0.25, point=[1,0.5,0.92])
>>> print(h.point[0])
[1, 0.5, 0.92]
```

Traces are independent of each other. A new trace can be added to the history and updated separately from the existing traces. This can be handy if there are separate sources of history though it may be difficult to keep the in sync. The following adds a 'step' to the existing history, initialized to 15, without changing 'value' or 'point':

```
>>> h.provides(step=2) # keep two steps
>>> h.update(step=15) # initialize step to 15
>>> print(h.step)
Trace step: 15
```

Traces may be used as accumulators, with the delta added to the existing value before being stored in the trace. For example:

```
>>> h.accumulate(step=1)
>>> print(h.step)
Trace step: 16, 15
```

Within bumps, history is used by monitors, with bumps.fitters.MonitorRunner managing updates to history and feeding them to the fit progress monitors.

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.

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.snapeshot() 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*)

put(*value*)

Add an item to the trace, shifting off from the beginning when the trace is full.

requires(*n*)

Set the trace length to be at least n.

restore(*state*)

Restore a trace from a captured snapshot.

Lists are converted to numpy arrays.

snapshot()

Capture state of the trace.

Numpy arrays are converted to lists so that the trace can be easily converted to json.

4.12 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 x 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.**cov_init**(*n: int*, *initial: ndarray[tuple[Any, ...], dtype[_ScalarT]]*, *bounds: ndarray[tuple[Any, ...], dtype[_ScalarT]]*, *use_point: bool = False*, *cov: ndarray[tuple[Any, ...], dtype[_ScalarT]] | None = None*, *dx: ndarray[tuple[Any, ...], dtype[_ScalarT]] | None = None*) → ndarray[tuple[Any, ...], dtype[_ScalarT]]

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.

bumps.initpop.**eps_init**(*n: int*, *initial: ndarray[tuple[Any, ...], dtype[_ScalarT]]*, *bounds: ndarray[tuple[Any, ...], dtype[_ScalarT]]*, *use_point: bool = False*, *eps: float = 1e-06*) → ndarray[tuple[Any, ...], dtype[_ScalarT]]

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.

bumps.initpop.**generate**(*problem:* FitProblem, *init: str = 'eps'*, *pop: int = 10*, *use_point: bool = True*, ***options: ...*) → NDArray

Population initializer.

problem is a fit problem with *getp* and *bounds* methods.

init is 'eps', 'cov', 'lhs' or 'random', indicating which initializer should be used.

pop is the population scale factor, generating *pop* individuals for each parameter in the fit. If *pop < 0*, generate a total of *-pop* individuals regardless of the number of parameters.

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.**lhs_init**(*n: int*, *initial: ndarray[tuple[Any, ...], dtype[_ScalarT]]*, *bounds: ndarray[tuple[Any, ...], dtype[_ScalarT]]*, *use_point: bool = False*) → ndarray[tuple[Any, ...], dtype[_ScalarT]]

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 implmeneted.

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.

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.

4.13 lsqerror - Least squares eorror analysis

chol_cov	Given the cholesky decomposition of the Hessian matrix H, compute the covariance matrix $C = H^{-1}$
chol_stderr	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.
comb	n choose r combination function
corr	Convert covariance matrix C to correlation matrix R^2.
demo_hessian	
demo_jacobian	
demo_stderr_hilbert	
demo_stderr_perturbed	
gradient	
hessian	Returns the derivative wrt to the fit parameters at point p.
hessian_cov	Given Hessian H, return the covariance matrix inv(H).
hilbert	Generate ill-conditioned Hilbert matrix of size n x n
hilbertinv	Analytical inverse for ill-conditioned Hilbert matrix of size n x n
jacobian	Returns the derivative wrt the fit parameters at point p.
jacobian_cov	Given Jacobian J, return the covariance matrix inv(J'J).
max_correlation	Return the maximum correlation coefficient for any pair of variables in correlation matrix Rsq.
perturbed_hessian	**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.
stderr	Return parameter uncertainty from the covariance matrix C.

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:

getp()	get the current value of the model
setp(p)	set a new value in the model
nllf(p)	negative log likelihood function
residuals(p)	residuals around its current value
bounds()	get the bounds on the parameter p [optional]

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.

hessian() computes the Hessian matrix H using numerical differentiation on nllf.

jacobian_cov() takes the Jacobian and computes the covariance matrix C. *hessian_cov()* takes the Hessian and computes the covariance matrix C.

corr() uses the off-diagonal elements of C to compute correlation coefficients R^2_{ij} between the parameters.

stderr() computes the uncertain σ_i from covariance matrix C, assuming that the C_{diag} contains σ_i^2, which should be the case for functions which are approximately linear near the minimum.

max_correlation() takes R^2 and returns the maximum correlation.

The user should be shown the uncertainty σ_i for each parameter, and if there are strong parameter correlations (e.g., $R^2_{\text{max}} > 0.2$), 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.

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).

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*)

bumps.lsqerror.**hessian**(*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 x n

bumps.lsqerror.**hilbertinv**(*n*)

Analytical inverse for ill-conditioned Hilbert matrix of size n x 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.

Note that the problem.residuals() method should not reuse memory for the returned value otherwise the derivative calculation (f(x+dx) - f(x))/dx will always be zero. The returned value need not be 1D, but it should be contiguous so that it can be reshaped to 1D without an extra copy. This will only be an issue for very large datasets.

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.

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 χ^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.

4.14 mapper - Parallel processing implementations

BaseMapper	
MPIMapper	
MPMapper	
SerialMapper	
ThreadPoolMapper	Thread-based parallel mapper using concurrent.futures.ThreadPoolExecutor.
can_pickle	Returns True if *problem* can be pickled.
cpu_id	Return the processor id for the currently running process.
nice	
pool_size	Get the number of cpus available for processing, or use the number provided.
setpriority	Set The Priority of a Windows Process.
show_performance	*timestamps* is a series of pairs (tstart, tstop) before and after the synchronous map call, with times in nanoseconds returned from time.perf_counter_ns().
using_mpi	

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.

Available mappers: - SerialMapper: Single-threaded execution - MPMapper: Multi-process execution using multiprocessing - ThreadPoolMapper: Multi-threaded execution using ThreadPoolExecutor - MPIMapper: MPI-based distributed execution across cluster nodes

Usage:

```
Mapper.start_worker(problem)
mapper = Mapper.start_mapper(problem, None, cpus)
result = mapper(points)
...
mapper = Mapper.start_mapper(problem, None, cpus)
result = mapper(points)
Mapper.stop_mapper()
```

class bumps.mapper.**BaseMapper**

Bases: `object`

has_problem = False

static start_mapper(*problem*, *modelargs=None*, *cpus=0*)

Called with the problem on a new fit.

static start_worker(*problem*)

Called with the problem to initialize the worker

static stop_mapper(*mapper=None*)

class bumps.mapper.**MPIMapper**

Bases: *BaseMapper*

has_problem = True

For MPIMapper only the worker is initialized with the fit problem.

static start_mapper(*problem*, *modelargs=None*, *cpus=0*)

Called with the problem on a new fit.

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 t problem is set to None, then exit the process and never

static stop_mapper(*mapper=None*)

timestamps = []

class bumps.mapper.**MPMapper**

Bases: *BaseMapper*

has_problem = False

manager = None

pool = None

problem = None

problem_id = 0

shared_pickled_problem = None

static start_mapper(*problem*, *modelargs=None*, *cpus=0*)

Called with the problem on a new fit.

static start_worker(*problem*)

Called with the problem to initialize the worker

static stop_mapper(*mapper=None*)

timestamps = []

class bumps.mapper.**SerialMapper**

Bases: *BaseMapper*

has_problem = False

static start_mapper(*problem*, *modelargs=None*, *cpus=0*)

Called with the problem on a new fit.

static start_worker(*problem*)

Called with the problem to initialize the worker

static stop_mapper(*mapper=None*)

timestamps = []

class bumps.mapper.**ThreadPoolMapper**

Bases: *BaseMapper*

Thread-based parallel mapper using concurrent.futures.ThreadPoolExecutor.

Each thread maintains its own copy of the problem object for independent calculations of nllf.

This mapper will only be efficient when using a free-threaded python interpreter (otherwise the GIL will prevent true parallelism).

has_problem = False

pool = None

problem_id = 0

static start_mapper(*problem*, *modelargs=None*, *cpus=0*)

Called with the problem on a new fit.

static start_worker(*problem*)

Called with the problem to initialize the worker

static stop_mapper(*mapper=None*)

timestamps = []

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.

bumps.mapper.**cpu_id**(*num_sockets=2*)

Return the processor id for the currently running process.

bumps.mapper.**nice**()

bumps.mapper.**pool_size**(*cpus=0*)

Get the number of cpus available for processing, or use the number provided.

On linux, use os.sched_getaffinity to count the number of cpus allocated to the process rather than multiprocessing.cpu_count to return all processors on the system. This allows us to restrict the amount of parallelism to the number of cpus allocated by slurm when running on a compute cluster with a partial node.

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.

bumps.mapper.**show_performance**(*timestamps*)

timestamps is a series of pairs (tstart, tstop) before and after the synchronous map call, with times in nanoseconds returned from time.perf_counter_ns(). Display the median time within *(tstop[k] - tstart[k])* and between *(tstart[k+1] = tstop[k])* map calls.

bumps.mapper.**using_mpi**()

4.15 monitor - Monitor fit progress

Monitor	Base class for monitors.
Logger	Keeps a record of all values for the desired fields.
TimedUpdate	Indicate progress every n seconds.

Progress monitors.

Process monitors accept a *bumps.history.History* object each cycle and perform some sort of work.

Monitors have a *Monitor.config_history()* method which calls *history.requires()* to set the amount of history it needs and a *Monitor.__call__* method which takes the updated history and generates the monitor output.

Most monitors are subclassed from *TimedUpdate* to set a minimum time between updates and to only show updates when there is an improvement. The *TimedUpdate* subclasses must override *TimedUpdate.show_progress()* and *TimedUpdate.show_improvement()* to control the output form. History must be updated with time, value, point and step. The bumps.fitters.MonitorRunner class manages history and updates.

class bumps.monitor.**Logger**(*fields=()*, *table=None*)

Bases: *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.

config_history(*history*)

Make sure history records each logged field.

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.

class bumps.monitor.**TimedUpdate**(*progress=60*, *improvement=5*)

Bases: *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 update only prints step number and improved value. Subclass TimedUpdate with replaced *show_progress()* and *show_improvement()* to trigger GUI updates or show parameter values.

config_history(*history*)

Indicate which fields are needed by the monitor and for what duration.

show_improvement(*history*)

show_progress(*history*)

4.16 mono - Freeform - Monotonic Spline

monospline	Monotonic cubic hermite interpolation.
hermite	Computes the cubic hermite polynomial $p(x_t)$.
count_inflections	Count the number of inflection points in a curve.
plot_inflections	Plot inflection points in a curve.

Monotonic spline modeling.

bumps.mono.**count_inflections**(*x*, *y*)

Count the number of inflection points in a curve.

bumps.mono.**hermite**(*x*, *y*, *m*, *xt*)

Computes the cubic hermite polynomial $p(x_t)$.

The polynomial goes through all points (x_i, y_i) with slope m_i at the point.

bumps.mono.**monospline**(*x*, *y*, *xt*)

Monotonic cubic hermite interpolation.

Returns $p(x_t)$ where $p(x_i) = y_i$ and $p(x) \leq p(x_i)$ 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

bumps.mono.**plot_inflections**(*x*, *y*)

Plot inflection points in a curve.

4.17 names - External interface

Exported names.

Usage:

import bumps.names as bp ... bp.FitProblem(experiment)

In model definition scripts, rather than importing symbols one by one, you can simply perform:

```
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 numpy array package
- *sys* for the python sys module

Math functions:

- math constants: *inf*, *pi*, *e*

- exponentials: *exp*, *log*, *log10*, *sqrt*
- *degrees*, *radians*
- radians trig: *sin*, *cos*, *tan*, *arcsin*, *arccos*, *arctan*, *arctan2*
- degrees trig: *sind*, *cosd*, *tand*, *arcsind*, *arccosd*, *arctand*, *arctan2d*
- hyperbolic trig: *sinh*, *cosh*, *tanh*, *arcsinh*, *arccosh*, *arctanh*
- *pmath* for *pmath.min()* and *pmath.max()*

Problem definition functions:

- *Parameter* for defining parameters
- *FreeVariables* for defining shared parameters
- ***Distribution*** **for indicating prior**
  probability for a model parameter
- *Curve* for defining models from functions
- *PoissonCurve* for modelling data with Poisson uncertainty
- *PDF* for fitting a probability distribution directly
- *FitProblem* for defining the fit.

Jupyter notebook functions:

- Simple fitter: *bumps.fitters.fit()*, *bumps.fitters.plot_convergence()*
- Webview server: *bumps.webview.server.webserver.start_bumps*, *bumps.webview.server.webserver.display_bumps*
- MCMC save/load: *bumps.fitters.load_session()*, *bumps.fitters.load_fit_from_session()*, *bumps.fitters.load_fit_from_export()*

4.18 options - Command line options processor

BumpsOpts	Option parser for bumps.
ChoiceList	
FIT_CONFIG	FitConfig singleton for the common case in which only one config is needed.
FitConfig	Fit settings configuration object.
ParseOpts	Options parser.
getopts	Process command line options.
parse_int	
yesno	

Option parser for bumps command line

class bumps.options.**BumpsOpts**(*args*)

Bases: *ParseOpts*

Option parser for bumps.

FLAGS = {'batch', 'chisq', 'cov', 'edit', 'err', 'i', 'mpi', 'multiprocessing-fork', 'noshow', 'overwrite', 'preview', 'profile', 'remote', 'shake', 'simrandom', 'simulate', 'staj', 'stepmon', 'time_model', 'worker'}

IMPLICIT_VALUES = {'entropy': 'llf', 'parallel': '0', 'resume': '-'}

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 = 1

PLOTTERS = ('linear', 'log', 'residuals')

TRANSPORTS = ('amqp', 'mp', 'mpi', 'celery')

```
USAGE = 'Usage:  bumps [options] modelfile [modelargs]\n\nThe modelfile is a Python
script (i.e., a series of Python commands)\nwhich sets up the data, the models, and
the fittable parameters.\nThe model arguments are available in the modelfile as
sys.argv[1:].\nModel arguments may not start with \'-\'.\n\nOptions:\n\n --preview\n
display model but do not perform a fitting operation\n --pars=filename or store
path\n initial parameter values; fit results are saved as path/<modelname>.par\n
--plot=log [linear|log|residuals]\n type of plot to display\n --trim=true\n trim any
remaining burn before displaying plots [dream only]\n --simulate\n simulate a
dataset using the initial problem parameters\n --simrandom\n simulate a dataset
using random problem parameters\n --shake\n set random parameters before fitting\n
--noise=5%\n percent noise to add to the simulated data\n --seed=integer\n random
number seed\n --err\n show uncertainty estimate from curvature at the minimum\n
--cov\n show the covariance matrix for the model when done\n
--entropy=gmm|mvn|wnn|llf\n compute entropy on posterior distribution [dream only]\n
--staj\n output staj file when done [Refl1D only]\n --edit\n start the gui\n
--view=linear|log\n one of the predefined problem views; reflectometry also has
fresnel,\n logfresnel, q4 and residuals\n\n --store=path\n output directory for
plots and models\n --overwrite\n if store already exists, replace it\n --resume=path
[dream]\n resume a fit from previous stored state; if path is \'-\' then use the\n
path given by --store, if it exists\n --parallel=n\n run fit using multiprocessing
for parallelism; use --parallel=0 for all cpus\n --mpi\n run fit using MPI for
parallelism (use command "mpirun -n cpus ...")\n --batch\n batch mode; save output
in .mon file and don\'t show plots after fit\n --noshow\n semi-batch; send output to
console but don\'t show plots after fit\n --time=inf\n run for a maximum number of
hours\n --checkpoint=0\n save fit state every n hours, or 0 for no checkpoints\n\n
--fit=amoeba [amoeba|de|dream|lm|newton|pt]\n fitting engine to use; see manual for
details\n --steps=0 [amoeba|de|dream|lm|newton|pt]\n number of fit iterations after
any burn-in time; use samples if steps=0\n --samples=1e4 [dream]\n set
steps=samples/(pop*#pars) so the target number of samples is drawn\n --xtol=1e-4
[de, amoeba]\n minimum population diameter\n --ftol=1e-4 [de, amoeba]\n minimum
population flatness\n --alpha=0.0 [dream]\n p-level for rejecting convergence; with
fewer samples use a stricter\n stopping condition, such as --alpha=0.01
--samples=20000\n --outliers=none [dream]\n name of test used for removing outlier
chains every N samples:\n none:  no outlier removal\n iqr:  use interquartile range
on likelihood\n grubbs:  use t-test on likelihood\n mahal:  use distance from
parameter values on the best chain\n --pop=10 [dream, de, rl, ps]\n population size
is pop times number of fitted parameters; if pop is\n negative, then set population
size to -pop.\n --burn=100 [dream, pt]\n number of burn-in iterations before
accumulating stats\n --thin=1 [dream]\n number of fit iterations between steps\n
--nT=25\n --Tmin=0.1\n --Tmax=10 [pt]\n temperatures vector; use a higher maximum
temperature and a larger\n nT if your fit is getting stuck in local minima\n
--CR=0.9 [de, rl, pt]\n crossover ratio for population mixing\n --starts=1 [newton,
rl, amoeba]\n number of times to run the fit from random starting points.\n
--near_best\n when running with multiple starts, restart from a point near the\n
last minimum rather than using a completely random starting point.\n --init=eps
[dream]\n population initialization method:\n eps:  ball around initial parameter
set\n lhs:  latin hypercube sampling\n cov:  normally distributed according to
covariance matrix\n random:  uniformly distributed within parameter ranges\n
--stepmon\n show details for each step\n --resynth=0\n run resynthesis error
analysis for n generations\n\n --time_model\n run the model --steps times in order
to estimate total run time.\n --profile\n run the python profiler on the model; use
--steps to run multiple\n models for better statistics\n --chisq\n print the model
description and chisq value and exit\n -m/-c/-p command\n run the python interpreter
with bumps on the path:\n m:  command is a module such as bumps.cli, run as
__main__\n c:  command is a python one-line command\n p:  command is the name of a
python script\n -i\n start the interactive interpreter\n -?/-h/--help\n display this
help\n"
```

VALUES = {'CR', 'F', 'Tmax', 'Tmin', 'alpha', 'burn', 'c', 'checkpoint', 'entropy', 'fit', 'ftol', 'init', 'jump', 'm', 'nT', 'near_best', 'noise', 'notify', 'outliers', 'p', 'parallel', 'pars', 'plot', 'pop', 'queue', 'radius', 'resume', 'resynth', 'samples', 'seed', 'starts', 'steps', 'stop', 'store', 'thin', 'time', 'transport', 'trim', 'view', 'xtol'}

alpha = 0.0

checkpoint = '0'

entropy = None

property fit

fit_config = <bumps.options.FitConfig object>

meshsteps = 40

noise = '5'

notify = ''

parallel = ''

pars = None

property plot

queue = None

resume = None

resynth = '0'

seed = ''

starts = '1'

store = None

time = 'inf'

property 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', '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: fitclasss} 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 their 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.

property selected_fitter

property selected_name

property 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 = {}

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*)

4.19 parameter - Optimization parameter definition

Alias	Parameter alias.
Calculation	A Parameter with a model-specific, calculated value.
Comparisons	comparison operators
Constant	Saved state for an unmodifiable value.
Constraint	Express inequality constraints between model elements
Expression	Parameter expression
FreeVariables	A collection of parameter sets for a group of models.
Function	**DEPRECATED**
HasParameters	
Normal	Normal distribution (Gaussian)
OperatorMixin	The set of operations that can be performed on parameter-like objects Parameter, Constant, Expression.
Operators	Operators that can be used to construct Expressions
Parameter	A parameter is a container for a symbolic value.
ParameterSet	A parameter that depends on the model.
Reference	Create an adaptor so that a model attribute can be treated as if it were a parameter.
SupportsPrior	Parameter mixin allowing the value to be restricted to a prior probability.
Uniform	Uniform distribution with hard boundaries
UniformSoftBounded	Uniform distribution with error-function PDF on boundaries
UserFunction	User-defined functions.
ValueProtocol	Values can be combined to form expressions Provide a suite of operators for creating parameter expressions.
Variable	Saved state for a random variable in the model.
acos	
acosd	Return the arc cosine (measured in in degrees) of x.
acosh	
arccosd	Return the arc cosine (measured in in degrees) of x.

continues on next page

Table 19 – continued from previous page

arcsind	Return the arc sine (measured in in degrees) of x.
arctan2d	Return the arc tangent (measured in in degrees) of y/x.
arctand	Return the arc tangent (measured in in degrees) of x.
asin	
asind	Return the arc sine (measured in in degrees) of x.
asinh	
atan	
atan2	
atan2d	Return the arc tangent (measured in in degrees) of y/x.
atand	Return the arc tangent (measured in in degrees) of x.
atanh	
copy_linked	make a copy of an object with parameters
cosd	Return the cosine of x (measured in in degrees).
current	
fittable	Return the list of fittable parameters in no paraticular order.
flatten	
format	Format parameter set for printing.
function	Convert a function into a delayed evaluator.
make_linked_copies	make a list of <num> copies of an object with parameters
max	
min	
priors	Return the list of parameters (fitted or computed) that have prior probabilities associated with them.
randomize	Set random values to the parameters in the parameter set, with values chosen according to the bounds.
sind	Return the sine of x (measured in in degrees).
substitute	Return structure a with values substituted for all parameters.
summarize	Return a stylized list of parameter names and values with range bars suitable for printing.
tag_all	
tand	Return the tangent of x (measured in in degrees).
test_operator	
to_dict	
unique	Return the unique set of parameters
untag_all	
varying	Return the list of fitted parameters in the model.

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.

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.

parameters()

to_dict()

update()

class bumps.parameter.**Calculation**(*description: str = ''*, *function: Callable = None*)

Bases: *ValueProtocol*

A Parameter with a model-specific, calculated value. The function used to calculate this value should be well-documented in the description field, e.g. Stack.thickness: description = "a sum of the thicknesses of all layers in the stack"

arccos()

arccosh()

arcsin()

arcsinh()

arctan()

arctan2()

arctanh()

ceil()

cos()

cosh()

degrees()

description: str

exp()

expm1()

fittable: bool = False

fixed: bool = True

floor()

log()

log10()

log1p()

max()

min()

parameters() → List[*Parameter*]

radians()

rint()

round()

set_function(*function*)

sin()

sinh()

sqrt()

tan()

tanh()

trunc()

property value

class bumps.parameter.**Comparisons**(*value*)

Bases: Enum

comparison operators

ge = '>='

gt = '>'

le = '<='

lt = '<'

class bumps.parameter.**Constant**(*value: float*, *name: str | None = None*, *id: str = <factory>*)

Bases: *ValueProtocol*

Saved state for an unmodifiable value.

A constant is like a fixed parameter. You can't change it's value, set it equal to another parameter, or assign a prior distribution.

arccos()

arccosh()

arcsin()

arcsinh()

arctan()

arctan2()

arctanh()

ceil()

cos()

cosh()

degrees()

exp()

expm1()

fittable: bool = False

fixed: bool = True

floor()

id: str

log()

log10()

log1p()

max()

min()

name: str | None = None

parameters()

radians()

rint()

round()

sin()

sinh()

sqrt()

tan()

tanh()

trunc()

value: float

class bumps.parameter.**Constraint**(*a*, *b*, *op*)

Bases: object

Express inequality constraints between model elements

a: *Parameter* | *Expression* | *Calculation* | float

b: *Parameter* | *Expression* | *Calculation* | float

fixed = True

op: *Comparisons*

property satisfied

class bumps.parameter.**Expression**(*op: str | Operators | UserFunction*, *args*)

Bases: *ValueProtocol*

Parameter expression

arccos()

arccosh()

arcsin()

arcsinh()

arctan()

arctan2()

arctanh()

args: Sequence[*Expression* | *Parameter* | *Calculation* | float]

ceil()

cos()

cosh()

degrees()

exp()

expm1()

fittable: bool = False

fixed: bool = True

floor()

log()

log10()

log1p()

max()

min()

property name

op: *Operators* | *UserFunction*

parameters()

radians()

rint()

round()

sin()

sinh()

sqrt()

tan()

tanh()

trunc()

property value

class bumps.parameter.**FreeVariables**(*names=None*, *parametersets=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 variables 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.

Note that using FreeVariables within a fitproblem erases any caching that happens within the model. This means, for example, that the theory function will be calculated once for plotting and again for computing ^{2}. To avoid unnecessary cache clearing, use bool(freevars) to test if there are parameters to substitute.

get_model(*i*)

Get the parameters for model *i* as {reference: substitution}

isfree(*param*)

names: List[str]

parameters()

Return the set of free variables for all the models.

parametersets: Dict[str, *ParameterSet*]

set_model(*i*)

Set the reference parameters for model *i*.

to_dict()

class bumps.parameter.**Function**(*op*, **args*, ***kw*)

Bases: *ValueProtocol*

DEPRECATED

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()

arccosh()

arcsin()

arcsinh()

arctan()

arctan2()

arctanh()

args: Any | None

ceil()

cos()

cosh()

degrees()

exp()

expm1()

fittable: bool = False

fixed: bool = True

floor()

kw: Dict[Any, Any]

log()

log10()

log1p()

max()

min()

op: Callable[[...], float]

parameters()

radians()

rint()

round()

sin()

sinh()

sqrt()

tan()

tanh()

to_dict()

trunc()

property value

class bumps.parameter.**HasParameters**(**args*, ***kwargs*)

Bases: Protocol

parameters: Callable[[], List[*Parameter*] | Dict[str, *Parameter*]]

class bumps.parameter.**Normal**(*std: float*, *mean: float*)

Bases: object

Normal distribution (Gaussian)

mean: float

std: float

class bumps.parameter.**OperatorMixin**

Bases: object

The set of operations that can be performed on parameter-like objects Parameter, Constant, Expression.

These include: +, -, , /, //, *, abs, float, int

Also, numpy math functions: sin, cos, tan, ...

Much like abs(obj) => obj.__abs__(), np.sin(obj) => obj.sin()

arccos()

arccosh()

arcsin()

arcsinh()

arctan()

arctan2()

arctanh()

ceil()

cos()

cosh()

degrees()

exp()

expm1()

floor()

log()

log10()

log1p()

max()

min()

radians()

rint()

round()

sin()

sinh()

sqrt()

tan()

tanh()

trunc()

value: float

class bumps.parameter.**Operators**(*value*)

Bases: str, Enum

Operators that can be used to construct Expressions

abs = 'abs'

add = 'add'

arccos = 'arccos'

arccosh = 'arccosh'

arcsin = 'arcsin'

arcsinh = 'arcsinh'

arctan = 'arctan'

arctan2 = 'arctan2'

arctanh = 'arctanh'

capitalize()

Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower case.

casefold()

Return a version of the string suitable for caseless comparisons.

ceil = 'ceil'

center(*width*, *fillchar=' '*, */ (Positional-only parameter separator (PEP 570))*)

Return a centered string of length width.

Padding is done using the specified fill character (default is a space).

cos = 'cos'

cosh = 'cosh'

count(*sub*[, *start*[, *end*]]) → int

Return the number of non-overlapping occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

degrees = 'degrees'

encode(*encoding='utf-8'*, *errors='strict'*)

Encode the string using the codec registered for encoding.

encoding
The encoding in which to encode the string.

errors
The error handling scheme to use for encoding errors. The default is 'strict' meaning that encoding errors raise a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and 'xmlcharrefreplace' as well as any other name registered with codecs.register_error that can handle UnicodeEncodeErrors.

endswith(*suffix*[, *start*[, *end*]]) → bool

Return True if S ends with the specified suffix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. suffix can also be a tuple of strings to try.

exp = 'exp'

expandtabs(*tabsize=8*)

Return a copy where all tab characters are expanded using spaces.

If tabsize is not given, a tab size of 8 characters is assumed.

expm1 = 'expm1'

find(*sub*[, *start*[, *end*]]) → int

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

floor = 'floor'

floordiv = 'floordiv'

format(**args*, ***kwargs*) → str

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

format_map(*mapping*) → str

Return a formatted version of S, using substitutions from mapping. The substitutions are identified by braces ('{' and '}').

index(*sub*[, *start*[, *end*]]) → int

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

isalnum()

Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

isalpha()

Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

isascii()

Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

isdecimal()

Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

isdigit()

Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

isidentifier()

Return True if the string is a valid Python identifier, False otherwise.

Call keyword.iskeyword(s) to test whether string s is a reserved identifier, such as "def" or "class".

islower()

Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

isnumeric()

Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

isprintable()

Return True if the string is printable, False otherwise.

A string is printable if all of its characters are considered printable in repr() or if it is empty.

isspace()

Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

istitle()

Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

isupper()

Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

join(*iterable*, /)

Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: '.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'

ljust(*width*, *fillchar=' '*, /)

Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

log = 'log'

log10 = 'log10'

log1p = 'log1p'

lower()

Return a copy of the string converted to lowercase.

lstrip(*chars=None*, /)

Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

static maketrans()

Return a translation table usable for str.translate().

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

max = 'max'

min = 'min'

mul = 'mul'

neg = 'neg'

partition(*sep*, /)

Partition the string into three parts using the given separator.

This will search for the separator in the string. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing the original string and two empty strings.

pos = 'pos'

pow = 'pow'

radians = 'radians'

removeprefix(*prefix*, /)

Return a str with the given prefix string removed if present.

If the string starts with the prefix string, return string[len(prefix):]. Otherwise, return a copy of the original string.

removesuffix(*suffix*, /)

Return a str with the given suffix string removed if present.

If the string ends with the suffix string and that suffix is not empty, return string[:-len(suffix)]. Otherwise, return a copy of the original string.

replace(*old*, *new*, *count=-1*, /)

Return a copy with all occurrences of substring old replaced by new.

count
Maximum number of occurrences to replace. -1 (the default value) means replace all occurrences.

If the optional argument count is given, only the first count occurrences are replaced.

rfind(*sub*[, *start*[, *end*]]) → int

Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

rindex(*sub*[, *start*[, *end*]]) → int

Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

rint = 'rint'

rjust(*width*, *fillchar=' '*, /)

Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

round = 'round'

rpartition(*sep*, /)

Partition the string into three parts using the given separator.

This will search for the separator in the string, starting at the end. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing two empty strings and the original string.

rsplit(*sep=None*, *maxsplit=-1*)

Return a list of the substrings in the string, using sep as the separator string.

sep
The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including n r t f and spaces) and will discard empty strings from the result.

maxsplit
Maximum number of splits. -1 (the default value) means no limit.

Splitting starts at the end of the string and works to the front.

rstrip(*chars=None*, */*)

Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

sin = 'sin'

sinh = 'sinh'

split(*sep=None*, *maxsplit=-1*)

Return a list of the substrings in the string, using sep as the separator string.

sep
The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including n r t f and spaces) and will discard empty strings from the result.

maxsplit
Maximum number of splits. -1 (the default value) means no limit.

Splitting starts at the front of the string and works to the end.

Note, str.split() is mainly useful for data that has been intentionally delimited. With natural text that includes punctuation, consider using the regular expression module.

splitlines(*keepends=False*)

Return a list of the lines in the string, breaking at line boundaries.

Line breaks are not included in the resulting list unless keepends is given and true.

sqrt = 'sqrt'

startswith(*prefix*[, *start*[, *end*]]) → bool

Return True if S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.

strip(*chars=None*, */*)

Return a copy of the string with leading and trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

sub = 'sub'

swapcase()

Convert uppercase characters to lowercase and lowercase characters to uppercase.

tan = 'tan'

tanh = 'tanh'

title()

Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining cased characters have lower case.

translate(*table*, /)

Replace each character in the string using the given translation table.

table
Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.

The table must implement lookup/indexing via __getitem__, for instance a dictionary or list. If this operation raises LookupError, the character is left untouched. Characters mapped to None are deleted.

truediv = 'truediv'

trunc = 'trunc'

upper()

Return a copy of the string converted to uppercase.

zfill(*width*, /)

Pad a numeric string with zeros on the left, to fill a field of the given width.

The string is never truncated.

class bumps.parameter.**Parameter**(*value: float | Tuple[float, float] | None = None*, *slot:* Variable | Expression | Parameter | Calculation | *float | None = None*, *fixed: bool | None = None*, *name: str | None = None*, *id: str | None = None*, *limits: Tuple[float | Literal[None, '-inf'], float | Literal[None, 'inf']] | None = None*, *bounds: Tuple[float | Literal['-inf'], float | Literal['inf']] | None = None*, *distribution:* Uniform | Normal | UniformSoftBounded = *Uniform()*, *discrete: bool = False*, *tags: List[str] | None = None*)

Bases: *ValueProtocol*, *SupportsPrior*

A parameter is a container for a symbolic value.

Parameters have a prior probability, as set by a bounds constraint:

import numpy as np from scipy.stats.distributions import lognorm from bumps.parameter import Parameter

p = Parameter(3) p.pmp(10) # 3 +/- 10% uniform p.pmp(-5,10) # 3 in [2.85, 3.30] uniform p.pm(2) # 3 +/- 2 uniform p.pm(-1,2) # 3 in [2,5] uniform p.range(0,5) # 3 in [0,5] uniform p.dev(2) # 3 +/- 2 gaussian p.soft_range(2,5,2) # 3 in [2,5] uniform with gauss wings p.dev(2, limits=(0,6)) # 3 +/- 2 truncated gaussian p.pdf(lognorm(3, 1)) # lognormal centered on 3, width 1.

Parameters have hard limits on the possible values, dictated by the model. These bounds apply in addition to any other bounds.

Parameters can be constrained to be equal to another parameter or parameter expression:

a, b = Parameter(3), Parameter(4) p = Parameter(limits=(6, 10)) p.equals(a+b) assert p.nllf() == 0. # within the bounds a.value = 20 assert np.isinf(p.nllf()) # out of bounds

Constraints on the computed value follow from the constraints on the underlying parameters in addition to any hard limits on the parameter value given by the model.

Inputs

value can be a constant, a variable, an expression or a link to another parameter.

bounds are user-supplied limits on the parameter value within the model. If bounds are supplied then the parameter defaults to fittable.

distribution is one of Uniform, Normal or UniformSoftBounded classes

fixed is True if the parameter is fixed, even if bounds are supplied.

name is the label associated with the parameter in plots. The names need not be unique, but it will be confusing if there are duplicates. The name will usually correspond to the role of the parameter in the model. For models with sequences (e.g., layer numbers), try using a layer name (e.g., based on the material in the layer) rather than a layer number for parameters in that layer. This will make it easier for the user to associate the parameters displayed at the end of the the fit with the layer in the model. Also, when exploring the space of models, the parameter names will be preserved even if a new layer is introduced before the existing layers. That will allow the parameters from the previous fit to be easily used as a seed for the fit to the new model.

id must be a unique identifier associated with the parameter. This is used to link parameters on save and reload.

limits are hard limits on the parameter value within the model. Separate from the prior distribution on a random variable provided by the user, the hard limits are restrictions on the value imposed by the model. For example, the thickness of a layer must be zero or more.

discrete is True if the parameter value must be an integer.

tags are user-supplied labels that can be used to group parameters together for display or for setting values, and are used in the GUI for filtering parameters.

add_tag(*tag: str*)

arccos()

arccosh()

arcsin()

arcsinh()

arctan()

arctan2()

arctanh()

bounds: `Tuple[float | Literal['-inf'], float | Literal['inf']] | None = None`

static calculation(*obj:* Parameter | *None*, *name: str*, *function: Callable[[], float]*) → *Parameter*

Create a parameter to hold a value derived from the model. This can be used in parameter expressions, for example to constrain total thickness or to set the value in the next segment equal to the value at the end of a freeform segment.

Note that this function should be called in the __init__ or __post_init__ methods of the class where the parameter is defined, in order to bind the Calculation function to the newly created (or deserialized) Parameter before it is used.

If obj is a Parameter, use it - otherwise create a new Parameter obj with the given name.

Then create a Calculation object and attach the evaluator function to the Calculation, and put the Calculation in obj.slot

Returns obj: Parameter

ceil()

clip_set(*value*)

Set a new value for the parameter, clipping it to the bounds.

cos()

cosh()

classmethod default(*value: float | Tuple[float, float] |* Expression | Parameter | Calculation, ***kw*) → *Parameter*

Create a new parameter with the *value* and *kw* attributes. If value is already a parameter or expression, set it to that value.

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: bool = False

distribution: *Uniform* | *Normal* | *UniformSoftBounded*

equals(*expression:* Expression | Parameter | Calculation | *float*)

Set a parameter equal to another parameter or expression.

Use *unlink()* to convert from an expression to a variable.

exp()

expm1()

feasible()

Value is within the limits defined by the model

property fittable

bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

property fixed

bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

floor()

format()

Format the parameter, value and range as a string.

id: str

limits: Tuple[float | Literal['-inf'], float | Literal['inf']] = (-inf, inf)

log()

log10()

log1p()

max()

min()

name: str | None = None

nllf() → float

Return -log(P) for the current parameter value.

parameters()

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.

prior: *Unbounded* | *Bounded* | *BoundedAbove* | *BoundedBelow* | *BoundedNormal* | *SoftBounded* | *Normal* | None

radians()

randomize(*rng=None*)

Set a random value for the parameter.

range(*low*, *high*)

Allow the parameter to vary within the given range.

remove_tag(*tag: str | None = None*)

reset_prior()

residual() → float

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.

rint()

round()

set(*value*)

Set a new value for the parameter, ignoring the bounds.

sin()

sinh()

slot: *Variable* | *Expression* | *Parameter* | *Calculation* | **float**

soft_range(*low*, *high*, *std*)

Allow the parameter to vary within the given range, or with Gaussian probability, stray from the range.

sqrt()

tags: List[str]

tan()

tanh()

trunc()

unlink()

valid()

Return true if the parameter is within the valid range.

property value

class bumps.parameter.**ParameterSet**(*reference:* Parameter, *names: List[str] | None = None*, *parameterlist: List[*Parameter*] | None = None*)

Bases: object

A parameter that depends on the model.

get_model(*index*)

Get the reference and underlying model parameter for the nth model.

names: List[str] | None

property parameterlist: List[*Parameter*]

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.

reference: *Parameter*

set_model(*index*)

Set the underlying model parameter to the value of the nth model.

to_dict()

property values

class bumps.parameter.**Reference**(*obj*, *attr*, ***kw*)

Bases: *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.

NOTE: this class can not be serialized with a dataclass schema TODO: can sasmodels just use Parameter directly?

add_tag(*tag: str*)

arccos()

arccosh()

arcsin()

arcsinh()

arctan()

arctan2()

arctanh()

bounds: `Tuple[float | Literal['-inf'], float | Literal['inf']] | None = None`

static calculation(*obj:* Parameter | *None*, *name: str*, *function: Callable[[], float]*) → *Parameter*

Create a parameter to hold a value derived from the model. This can be used in parameter expressions, for example to constrain total thickness or to set the value in the next segment equal to the value at the end of a freeform segment.

Note that this function should be called in the __init__ or __post_init__ methods of the class where the parameter is defined, in order to bind the Calculation function to the newly created (or deserialized) Parameter before it is used.

If obj is a Parameter, use it - otherwise create a new Parameter obj with the given name.

Then create a Calculation object and attach the evaluator function to the Calculation, and put the Calculation in obj.slot

Returns obj: Parameter

ceil()

clip_set(*value*)

Set a new value for the parameter, clipping it to the bounds.

cos()

cosh()

classmethod default(*value: float | Tuple[float, float]* | Expression | Parameter | Calculation, ***kw*) → *Parameter*

Create a new parameter with the *value* and *kw* attributes. If value is already a parameter or expression, set it to that value.

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: bool = False

distribution: *Uniform* | *Normal* | *UniformSoftBounded*

equals(*expression:* Expression | Parameter | Calculation | *float*)

Set a parameter equal to another parameter or expression.

Use *unlink()* to convert from an expression to a variable.

exp()

expm1()

feasible()

Value is within the limits defined by the model

property fittable

bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

property fixed

bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

floor()

format()

Format the parameter, value and range as a string.

id: str

limits: Tuple[float | Literal['-inf'], float | Literal['inf']] = (-inf, inf)

log()

log10()

log1p()

max()

min()

name: str | None = None

nllf() → float

Return -log(P) for the current parameter value.

parameters()

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.

prior: *Unbounded* | *Bounded* | *BoundedAbove* | *BoundedBelow* | *BoundedNormal* | *SoftBounded* | *Normal* | None

radians()

randomize(*rng=None*)

Set a random value for the parameter.

range(*low*, *high*)

Allow the parameter to vary within the given range.

remove_tag(*tag: str | None = None*)

reset_prior()

residual() → float

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.

rint()

round()

set(*value*)

Set a new value for the parameter, ignoring the bounds.

sin()

sinh()

slot: *Variable* | *Expression* | *Parameter* | *Calculation* | **float**

soft_range(*low*, *high*, *std*)

Allow the parameter to vary within the given range, or with Gaussian probability, stray from the range.

sqrt()

tags: List[str]

tan()

tanh()

trunc()

unlink()

valid()

Return true if the parameter is within the valid range.

property value

class bumps.parameter.**SupportsPrior**

Bases: object

Parameter mixin allowing the value to be restricted to a prior probability.

Every prior has hard limits and bounds.

bounds: *Unbounded* | *Bounded* | *BoundedAbove* | *BoundedBelow* | *BoundedNormal* | *SoftBounded* | *Normal*

distribution: *Uniform* | *Normal* | *UniformSoftBounded*

limits: Tuple[float, float]

prior: *Unbounded* | *Bounded* | *BoundedAbove* | *BoundedBelow* | *BoundedNormal* | *SoftBounded* | *Normal* | None

reset_prior()

class bumps.parameter.**Uniform**

Bases: object

Uniform distribution with hard boundaries

class bumps.parameter.**UniformSoftBounded**(*std: float*)

Bases: object

Uniform distribution with error-function PDF on boundaries

std: float

class bumps.parameter.**UserFunction**(*fn: Callable*)

Bases: object

User-defined functions.

This is a helper class for the @function decorator, which treats the operator as one of the possible expression operators.

These won't be properly serialized/deserialized through the JSON schema unless the function is registered in advance. The schema will not include these functions as possible values even if registered, so a schema validator may fail on one of these functions.

name: str

class bumps.parameter.**ValueProtocol**

Bases: *OperatorMixin*

Values can be combined to form expressions Provide a suite of operators for creating parameter expressions.

arccos()

arccosh()

arcsin()

arcsinh()

arctan()

arctan2()

arctanh()

ceil()

cos()

cosh()

degrees()

exp()

expm1()

fittable: bool = False

fixed: bool = True

floor()

log()

log10()

log1p()

max()

min()

parameters() → List[*Parameter*]

radians()

rint()

round()

sin()

sinh()

sqrt()

tan()

tanh()

trunc()

value: float

class bumps.parameter.**Variable**(*value: float*)

Bases: *ValueProtocol*

Saved state for a random variable in the model.

arccos()

arccosh()

arcsin()

arcsinh()

arctan()

arctan2()

arctanh()

ceil()

cos()

cosh()

degrees()

exp()

expm1()

fittable: bool = False

fixed: bool = True

floor()

log()

log10()

log1p()

max()

min()

parameters()

radians()

rint()

round()

sin()

sinh()

sqrt()

tan()

tanh()

trunc()

value: float

bumps.parameter.**acos**(**args*)

bumps.parameter.**acosd**(**args:* Parameter | Expression | Calculation | *float*)

Return the arc cosine (measured in in degrees) of x.

bumps.parameter.**acosh**(**args*)

bumps.parameter.**arccosd**(**args:* Parameter | Expression | Calculation | *float*)

Return the arc cosine (measured in in degrees) of x.

bumps.parameter.**arcsind**(**args:* Parameter | Expression | Calculation | *float*)

Return the arc sine (measured in in degrees) of x.

bumps.parameter.**arctan2d**(**args:* Parameter | Expression | Calculation | *float*)

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**(**args:* Parameter | Expression | Calculation | *float*)

Return the arc tangent (measured in in degrees) of x.

bumps.parameter.**asin**(**args*)

bumps.parameter.**asind**(**args:* Parameter | Expression | Calculation | *float*)

Return the arc sine (measured in in degrees) of x.

bumps.parameter.**asinh**(**args*)

bumps.parameter.**atan**(**args*)

bumps.parameter.**atan2**(**args*)

bumps.parameter.**atan2d**(**args:* Parameter | Expression | Calculation | *float*)

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**(**args:* Parameter | Expression | Calculation | *float*)

Return the arc tangent (measured in in degrees) of x.

bumps.parameter.**atanh**(**args*)

bumps.parameter.**copy_linked**(*has_parameters:* HasParameters, *exclude: List[*Parameter*]* | *None = None*) → *HasParameters*

make a copy of an object with parameters

- then link all the parameters, except
- those in exclude

bumps.parameter.**cosd**(**args:* Parameter | Expression | Calculation | *float*)

Return the cosine of x (measured in in degrees).

bumps.parameter.**current**(*s: List[*Parameter*]*)

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=None*, *field=None*)

Format parameter set for printing.

Note that this only says how the parameters are arranged, not how they relate to each other.

Note that *freevars* here is the substitution dictionary returned by *FreeVars.get_model(i)*, not the FreeVars object itself.

bumps.parameter.**function**(*fn: Callable*)

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.

Wrapped functions are automatically added to bumps.pmath

bumps.parameter.**make_linked_copies**(*has_parameters:* HasParameters, *num: int = 1*, *exclude: List[*Parameter*]* | *None = None*) → List[*HasParameters*]

make a list of <num> copies of an object with parameters

- then link all the parameters, except
- those in exclude

bumps.parameter.**max**(*Parameter*)

bumps.parameter.**min**(*Parameter*)

bumps.parameter.**priors**(*s: List[*Parameter*]*) → List[*Parameter*]

Return the list of parameters (fitted or computed) that have prior probabilities associated with them. This may include parameters linked to expressions and parameters that are not fitted, but excludes parameters that are unbounded.

bumps.parameter.**randomize**(*s: List[*Parameter*]*)

Set random values to the parameters in the parameter set, with values chosen according to the bounds.

bumps.parameter.**sind**(**args:* Parameter | Expression | Calculation | *float*)

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.**tag_all**(*parameter_tree*, *tag*, *remove=False*)

bumps.parameter.**tand**(**args:* Parameter | Expression | Calculation | *float*)

Return the tangent of x (measured in in degrees).

bumps.parameter.**test_operator**()

bumps.parameter.**to_dict**(*p*)

bumps.parameter.**unique**(*s*) → List[*Parameter*]

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.**untag_all**(*parameter_tree*, *tag: str | None = None*)

bumps.parameter.**varying**(*s: List[*Parameter*]*) → List[*Parameter*]

Return the list of fitted parameters in the model.

This is the set of parameters that will vary during the fit.

4.20 partemp - Parallel tempering optimizer

parallel_tempering	Perform a MCMC walk using multiple temperatures in parallel.

Parallel tempering for continuous function optimization and uncertainty analysis.

The program performs Markov chain Monte Carlo exploration of a probability density function using a combination of random and differential evolution updates.

bumps.partemp.**parallel_tempering**(*nllf*, *p*, *bounds*, *T=None*, *steps=1000*, *CR=0.9*, *burn=1000*, *monitor=<function every_ten>*, *logfile=None*)

Perform a MCMC walk using multiple temperatures in parallel.

Parameters

nllf

[function(vector) -> 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.

p

[vector] Initial value

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.

4.21 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: *CovarianceMixin*

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()

cov(*x*)

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 has residuals available, 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)
```

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()

show_cov(*x*, *cov*)

show_err(*x*, *dx*)

Display the error approximation from the covariance matrix.

err is the standard deviation computed from the covariance matrix. It is available as *result.dx* from the simple fitter, or using:

```
from bumps import lsqerror

dx = lsqerror.stderr(problem.cov(x))
```

Warning: cost to compute cov grows as the cube of the number of parameters.

summarize()

class bumps.pdfwrapper.**PDF**(*fn*, *name=''*, *plotter=None*, *dof=1*, *pars=None*, ***kw*)

Bases: *CovarianceMixin*

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()

cov(*x*)

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 has residuals available, 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)
```

dof: int

fn: Callable

has_residuals = False

name: str

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.

pars: Dict[str, *Parameter*] = None

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.

plotter: Callable

residuals()

show_cov(*x*, *cov*)

show_err(*x*, *dx*)

Display the error approximation from the covariance matrix.

err is the standard deviation computed from the covariance matrix. It is available as *result.dx* from the simple fitter, or using:

```
from bumps import lsqerror

dx = lsqerror.stderr(problem.cov(x))
```

Warning: cost to compute cov grows as the cube of the number of parameters.

class bumps.pdfwrapper.**VectorPDF**(*fn*, *p*, *name=''*, *plot=None*, *dof=1*, *labels=None*, ***kw*)

Bases: *CovarianceMixin*

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()

cov(*x*)

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 has residuals available, 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)
```

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.

show_cov(*x*, *cov*)

show_err(*x*, *dx*)

Display the error approximation from the covariance matrix.

err is the standard deviation computed from the covariance matrix. It is available as *result.dx* from the simple fitter, or using:

```
from bumps import lsqerror

dx = lsqerror.stderr(problem.cov(x))
```

Warning: cost to compute cov grows as the cube of the number of parameters.

4.22 plotutil - Plotting utilities

config_matplotlib	Setup matplotlib to use a particular backend.
auto_shift	Return a y-offset coordinate transform for the current axes.
coordinated_colors	Return a set of coordinated colors as c['base|light|dark'].
dhsv	Modify color on hsv scale.
next_color	Return the next color in the plot color cycle.
plot_quantiles	Plot quantile curves for a set of lines.
form_quantiles	Return quantiles and values for a list of confidence intervals.

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:

```
from matplotlib import pyplot as plt
from bumps.plotutil import auto_shift
trans = auto_shift(plt.gca())
plot(x, y, trans=trans)
```

bumps.plotutil.**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 matplotlib. To allow this, modules should not import matplotlib 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.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 clipped to [0,1] after changing. The hue scale wraps between red to violet.

Example

Make sea green 10% darker:

```
>>> 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.**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.

bumps.plotutil.**next_color**(*axes=None*)

Return the next color in the plot color cycle.

Example:

```
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*, *axes=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.

4.23 plugin - Domain branding

new_model	Return a new empty model or None.
load_model	Return a model stored within a file.
calc_errors	Gather data needed to display uncertainty in the model and the data.
show_errors	Display the model with uncertainty on the current figure.
data_view	Panel factory for the data tab in the GUI.
model_view	Panel factory for the model tab in the GUI.
migrate_serialized	Migrate serialized model to the current version.

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:

```
if __name__ == "__main__":
    import multiprocessing
    multiprocessing.freeze_support()

    from bumps.plugin import install_plugin
    from bumps.cli import main as bumps_main
    import mypackage.plugin
    install_plugin(mypackage.plugin)
    bumps_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 only one plugin at a time is supported.

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.**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.**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.**migrate_serialized**(*model_dict*)

Migrate serialized model to the current version.

This function is called when loading a model from a file. It is used to update the model to the current version of the plugin.

bumps.plugin.**model_view**()

Panel factory for the model tab in the GUI.

Return None if not present.

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.**show_errors**(*errs*, *fig=None*, *save=None*)

Display the model with uncertainty on the current figure.

errs is the data returned from calc_errs.

4.24 pmath - Parametric versions of standard functions

min	
max	
cosd	Return the cosine of x (measured in in degrees).
sind	Return the sine of x (measured in in degrees).
tand	Return the tangent of x (measured in in degrees).
arccosd	Return the arc cosine (measured in in degrees) of x.
arcsind	Return the arc sine (measured in in degrees) of x.
arctand	Return the arc tangent (measured in in degrees) of x.
arctan2d	Return the arc tangent (measured in in degrees) of y/x.
asin	
acos	
atan	
atan2	
asind	Return the arc sine (measured in in degrees) of x.
acosd	Return the arc cosine (measured in in degrees) of x.
atand	Return the arc tangent (measured in in degrees) of x.
atan2d	Return the arc tangent (measured in in degrees) of y/x.
asinh	
acosh	
atanh	

Standard math functions for parameter expressions.

These functions delay evaluation of the expression until the value is requested, thus allowing the value to change when the parameter is updated. Most numpy functions will do this automatically, so nothing special is needed.

- *degrees*, *radians*, *min*, *max*
- powers: *exp*, *log*, *log10*, *sqrt*
- radians trig: *sin*, *cos*, *tan*, *arcsin*, *arccos*, *arctan*, *arctan2*
- degrees trig: *sind*, *cosd*, *tand*, *arcsind*, *arccosd*, *arctand*, *arctan2d*
- hyperbolic trig: *sinh*, *cosh*, *tanh*, *arcsinh*, *arccosh*, *arctanh*

arc-function aliases (*asin*, *asinh*, ...) are added to pmath for convenience.

bumps.pmath.**acos**(**args*)

bumps.pmath.**acosd**(**args:* Parameter | Expression | Calculation | *float*)
Return the arc cosine (measured in in degrees) of x.

bumps.pmath.**acosh**(**args*)

bumps.pmath.**arccosd**(**args:* Parameter | Expression | Calculation | *float*)
Return the arc cosine (measured in in degrees) of x.

bumps.pmath.**arcsind**(**args:* Parameter | Expression | Calculation | *float*)
Return the arc sine (measured in in degrees) of x.

bumps.pmath.**arctan2d**(**args:* Parameter | Expression | Calculation | *float*)
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.pmath.**arctand**(**args:* Parameter | Expression | Calculation | *float*)
Return the arc tangent (measured in in degrees) of x.

bumps.pmath.**asin**(**args*)

bumps.pmath.**asind**(**args:* Parameter | Expression | Calculation | *float*)
Return the arc sine (measured in in degrees) of x.

bumps.pmath.**asinh**(**args*)

bumps.pmath.**atan**(**args*)

bumps.pmath.**atan2**(**args*)

bumps.pmath.**atan2d**(**args:* Parameter | Expression | Calculation | *float*)
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.pmath.**atand**(**args:* Parameter | Expression | Calculation | *float*)
Return the arc tangent (measured in in degrees) of x.

bumps.pmath.**atanh**(**args*)

bumps.pmath.**cosd**(**args:* Parameter | Expression | Calculation | *float*)
Return the cosine of x (measured in in degrees).

bumps.pmath.**max**(*Parameter*)

bumps.pmath.**min**(*Parameter*)

bumps.pmath.**sind**(**args:* Parameter | Expression | Calculation | *float*)
Return the sine of x (measured in in degrees).

bumps.pmath.**tand**(**args:* Parameter | Expression | Calculation | *float*)
Return the tangent of x (measured in in degrees).

4.25 pymcfit - Wrapper for pyMC models

PyMCProblem	

Bumps wrapper for PyMC models.

class bumps.pymcfit.**PyMCProblem**(*input*)

Bases: object

bounds()

chisq()

chisq_str()

getp()

labels()

model_reset()

nllf(*pvec=None*)

plot(*p=None*, *fignum=None*, *figfile=None*)

randomize(*N=None*)

setp(*values*)

show()

summarize()

4.26 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:

```
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                        # todo. see what default value is the best
macheps = eps
eta = eps
maxstep = 100
gradtol = 1e-6
```

(continues on next page)

(continued from previous page)

```
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"%(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*, *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

gradtol 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

monitor(x,fx,step) is called every iteration so that a user interface function can monitor the progress of the fit. Return False if the user requests stop. 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

4.27 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.

Sahin, I. (2013). Minimization over randomly selected lines. An International Journal Of Optimization And Control: Theories & Applications (IJOCTA), 3(2), 111-119. http://dx.doi.org/10.11121/ijocta.01.2013.00167

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(step, x, fx, k) is called each iteration using *monitor(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 population size

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.**random_lines**(*cfo*, *NP*, *CR=0.9*, *epsilon=1e-10*, *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 population size

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.

4.28 simplex - Nelder-Mead simplex optimizer (amoeba)

simplex	Minimize a function using Nelder-Mead downhill simplex algorithm.

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 Nealder-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: `fopt = func(xopt)`.

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.

4.29 util - Miscellaneous functions

kbhit	Check whether a key has been pressed on the console.
profile	Profile a function called with the given arguments.
pushdir	Change directories for the duration of a with statement.
push_seed	Set the seed value for the random number generator.
redirect_console	Console output redirection context
format_uncertainty	Opinionated formatting of mean and standard deviation.

Miscellaneous utility functions.

bumps.util.**format_uncertainty**(*mean*, *std*)
Opinionated formatting of mean and standard deviation.

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.**push_seed**(*seed=None*)
Bases: `object`

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:

```
>>> 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:

```
>>> 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:

```
>>> 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:

```
>>> 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.**pushdir**(*path*)

Bases: `object`

Change directories for the duration of a with statement.

Example

Show that the original directory is restored:

```
>>> 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.**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:

```
>>> from bumps.util import redirect_console
>>> print("hello")
hello
>>> with redirect_console("redirect_out.log"):
...     print("captured")
>>> print("hello")
hello
>>> print(open("redirect_out.log").read()[:-1])
captured
>>> import os; os.unlink("redirect_out.log")
```

Output can also be captured to a string:

```
>>> with redirect_console() as fid:
...     print("captured to string")
...     captured_string = fid.getvalue()
>>> print(captured_string.strip())
captured to string
```

4.30 wsolve - Weighted linear and polynomial solver with uncertainty

wsolve	Given a linear system $y = Ax + \delta y$, estimates x and δx.
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 $Ax = y$.

continues on next page

Table 30 – continued from previous page

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:

s.x	solution
s.std	uncertainty estimate assuming no correlation
s.rnorm	residual norm
s.DoF	degrees of freedom
s.cov	covariance matrix
s.ci(p)	confidence intervals at point p
s.pi(p)	prediction intervals at point p
s(p)	predicted value at point p

4.30.1 Example

Weighted system:

```
>>> 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(f"{a:0.2f} +/- {b:0.2f}" for a,b in zip(s.x[:, 0],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.

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 $1-\sigma$ standand 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 $n-p$:

$$w^T x \pm t_{n-p}^{\alpha/2} \sqrt{\text{var}(w)}$$

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:

$$\text{var}(w) = \sigma^2 (1 + w^T (A^T A)^{-1} w)$$

DoF

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 $1-\sigma$ confidence interval

Confidence intervals are sometimes expressed as $1-\alpha$ values, where $\alpha = \text{erfc}(\sigma/\sqrt{2})$.

property cov

covariance matrix [inv(A'A); O(n^3)]

property 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

2-norm of the residuals $||y - Ax||_2$

property std

solution standard deviation [sqrt(var); O(n^2)]

property var

solution variance [diag(cov); O(n^2)]

x

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
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
polynomial coefficients

property cov
covariance matrix

Note that the ones column will be absent if *origin* is True.

degree
polynomial degree

der(*x*)
Evaluate the polynomial derivative at x.

origin
True if polynomial goes through the origin

property 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*)

rnorm
2-norm of the residuals $||y - Ax||_2$

property std
solution standard deviation

property var
solution variance

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*.

bumps.wsolve.**wsolve**(*A*, *y*, *dy=1*, *rcond=1e-12*)
Given a linear system $y = Ax + \delta y$, estimates x and δx.

A is an n x m array of measurement points.

y is an n x k array or vector of length n of measured values at *A*.

dy is a scalar or an n x 1 array of uncertainties in the values at *A*.

Returns *LinearModel*.

bounds	Parameter bounds and prior probabilities.
bspline	BSpline calculator.
cheby	Freeform modeling with Chebyshev polynomials.
cli	Bumps command line interface.
curve	Build a bumps model from a function and data.
data	Data handling utilities.
errplot	Estimate model uncertainty from random sample.
fitproblem	Interface between the models and the fitters.
fitservice	Fit job definition for the distributed job queue.
fitters	Interfaces to various optimizers.
history	Log of progress through a computation.
initpop	Population initialization strategies.
lsqerror	Least squares error analysis.
mapper	Parallel and serial mapper implementations.
monitor	Progress monitors.
mono	Monotonic spline modeling.
names	Exported names.
options	Option parser for bumps command line
parameter	Fitting parameter objects.
partemp	Parallel tempering for continuous function optimization and uncertainty analysis.
pdfwrapper	Build a bumps model from a function.
plotutil	Pylab plotting utilities.
plugin	Bumps plugin architecture.
pmath	Standard math functions for parameter expressions.
pymcfit	Bumps wrapper for PyMC models.
quasinewton	BFGS quasi-newton optimizer.
random_lines	Random Lines Algorithm finds the optimal minimum of a function.
simplex	Downhill simplex optimizer.
util	Miscellaneous utility functions.
wsolve	Weighted linear and polynomial solver with uncertainty.

CHAPTER

FIVE

REFERENCE: BUMPS.DREAM

5.1 acr - A C Rencher normal outlier test

ACR	Return critical value for test of single multivariate normal outlier using the Mahalanobis distance metric.

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:

```
>>> 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.
atrujo@uabc.mx
```

Copyright. August 20, 2006.

To cite this file, this would be an appropriate format:

```
Trujillo-Ortiz, A., R. Hernandez-Walls, A. Castro-Perez and K. Barba-Rojo.
(2006). *ACR:Upper percentiles critical value for test of single
multivariate  normal outlier.* A MATLAB file. [WWW document].  URL
http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=12161
```

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:

[1] Barnett, V. and Lewis, T. (1978), Outliers on Statistical Data.
New-York:John Wiley & Sons.

[2] Rencher, A. C. (2002), Methods of Multivariate Analysis. 2nd. ed.
New-Jersey:John Wiley & Sons. Chapter 13 (pp. 408-450).

[3] Wilks, S. S. (1963), Multivariate Statistical Outliers. Sankhya,
Series A, 25: 407-426.

[4] Yang, S. S. and Lee, Y. (1987), Identification of a Multivariate
Outlier. Presented at the Annual Meeting of the American Statistical Association, San Francisco, August 1987.

`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

make_bounds_handler	Return a bounds object which can update the bounds.
Bounds	Base class for all times of bounds objects.
ReflectBounds	Reflect parameter values into bounded region
ClipBounds	Clip values to bounded region
FoldBounds	Wrap values into the bounded region
RandomBounds	Randomize values into the bounded region
IgnoreBounds	Leave values outside the bounded region

Bounds handling.

Use bounds(low, high, 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 apply(x) method which transforms the point *x*.

`class` `bumps.dream.bounds.`**`Bounds`**

Bases: `object`

Base class for all times of bounds objects.

`static apply`(*minn*, *maxn*, *pop*)

Force pop (population) values within bounds

`c_interface: Callable[[int, int, Any, Any, Any], None] = None`

`high: ndarray[tuple[Any, ...], dtype[_ScalarT]] = None`

`low: ndarray[tuple[Any, ...], dtype[_ScalarT]] = None`

`class` `bumps.dream.bounds.`**`ClipBounds`**(*low*, *high*)

Bases: *Bounds*

Clip values to bounded region

`static apply`(*minn*, *maxn*, *pop*)

Force pop (population) values within bounds

c_interface = None

high = None

low = None

class bumps.dream.bounds.**FoldBounds**(*low*, *high*)

Bases: *Bounds*

Wrap values into the bounded region

static apply(*minn*, *maxn*, *pop*)

Force pop (population) values within bounds

c_interface = None

high = None

low = None

class bumps.dream.bounds.**IgnoreBounds**(*low=None*, *high=None*)

Bases: *Bounds*

Leave values outside the bounded region

static apply(*minn*, *maxn*, *pop*)

Force pop (population) values within bounds

c_interface = None

high = None

low = None

class bumps.dream.bounds.**RandomBounds**(*low*, *high*)

Bases: *Bounds*

Randomize values into the bounded region

static apply(*minn*, *maxn*, *pop*)

Force pop (population) values within bounds

c_interface = None

high = None

low = None

class bumps.dream.bounds.**ReflectBounds**(*low*, *high*)

Bases: *Bounds*

Reflect parameter values into bounded region

static apply(*minn*, *maxn*, *pop*)

Update pop so all values lie within bounds

c_interface = None

high = None

low = None

bumps.dream.bounds.**make_bounds_handler**(*bounds*, *style='reflect'*)

Return a bounds object which can update the bounds.

Bounds handling *style* name is one of:

```
reflect:   reflect off the boundary
clip:      stop at the boundary
fold:      wrap values to the other side of the boundary
randomize: move to a random point in the bounds
none:      ignore the bounds
```

With semi-infinite intervals folding and randomizing aren't well defined, and reflection is used instead.

With finite intervals the 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.

5.3 convergence - Convergence tests

burn_point	Determines the point at which the MCMC chain seems to have converged, using a Kolmogorov-Smirnov sliding window test.
ks_converged	Return True if the MCMC has converged according to the K-S window test.

5.3.1 Convergence diagnostics

The function *burn_point()* returns the point within the MCMC chain at which the chain can be said to have converged, or -1 if the log probabilities are still improving throughout the chain.

bumps.dream.convergence.**burn_point**(*state:* MCMCDraw, *trials: int = 5*, *density: float = 0.6*, *alpha: float = 0.01*, *samples: int = 1000*) → int

Determines the point at which the MCMC chain seems to have converged, using a Kolmogorov-Smirnov sliding window test.

state contains the MCMC chain information.

trials is the number of times to run the K-S test.

density is the proportion of samples to select from the window. Prefer lower density from larger number of samples so the sets chosen for the K-S test have fewer duplicates. For density=0.1 about 5% of samples will be duplicates. For density=0.6 about 25% will be duplicates.

alpha is the significance level for the test. With smaller alpha values the K-S test is less likely to reject the current window when testing against the tail of the distribution, and so the fit will end earlier, with more samples after the burn point.

samples is the size of the sample window. If the window is too big the test will falsly end burn when the start of the window is still converging. If the window is too small the test will take a long time, and will start to show effects of autocorrelation (efficient MCMC samplers move slowly across the posterior probability space, showing short term autocorrelation between samples.) A minimum of 10 generations and a maximum of 1/2 the generations will be used.

Returns the index of the burn points, or -1 if no good burn point is found.

Kolmogorov-Smirnov sliding window

Detects convergence by comparing the distribution of $\log(p)$ values in a small window at the start of the chains and the values in a section at the end of the chain using a Kolmogorov-Smirnov test.

bumps.dream.convergence.**ks_converged**(*state:* MCMCDraw, *trials: int = 5*, *density: float = 0.6*, *alpha: float = 0.01*, *samples: int = 1000*) → bool

Return True if the MCMC has converged according to the K-S window test.

Since we are only looking at the distribution of logp values, and not the individual points, we should be relatively stable regardless of the properties of the sampler. The main reason for failure will be "stuck" fits which have not had a chance to jump to a lower minimum.

state contains the MCMC chain information.

trials is the number of times to run the K-S test.

density is the proportion of samples to select from the window. Prefer lower density from larger number of samples so the sets chosen for the K-S test have fewer duplicates. For density=0.1 about 5% of samples will be duplicates. For density=0.6 about 25% will be duplicates.

alpha is the significance level for the test. With smaller alpha values the K-S test is less likely to reject the current window when testing against the tail of the distribution, and so the fit will end earlier, with more samples after the burn point.

samples is the size of the sample window. If the window is too big the test will falsly end burn when the start of the window is still converging. If the window is too small the test will take a long time, and will start to show effects of autocorrelation (efficient MCMC samplers move slowly across the posterior probability space, showing short term autocorrelation between samples.) A minimum of 10 generations and a maximum of 1/2 the generations will be used.

There is a strong interaction between density, alpha, samples and trials. If the K-S test has too many points (=density*samples), it will often reject simply because the different portions of the Markov chain are not identical (Markov chains can have short range correlations yet still have the full chain as a representative draw from the posterior distribution) unless alpha is reduced. With fewer points, the estimated K-S statistic will have more variance, and so more trials will be needed to avoid spurious accept/reject decisions.

5.4 core - DREAM core

Dream	Data structure containing the details of the running DREAM analysis code.
Model	Dream model interface definition.

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:

Vrugt, J.A., C.J.F. ter Braak, M.P. Clark, J.M. Hyman, and B.A. Robinson,
Treatment of input uncertainty in hydrologic modeling: Doing hydrology backward with Markov chain Monte Carlo simulation, Water Resources Research, 44, W00B09, 2008. doi:10.1029/2007WR006720

Vrugt, J.A., C.J.F. ter Braak, C.G.H. Diks, D. Higdon, B.A. Robinson,
and J.M. Hyman, *Accelerating Markov chain Monte Carlo simulation by differential evolution with self-adaptive randomized subspace sampling*, International Journal of Nonlinear Sciences and Numerical Simulation, 10(3), 271-288, 2009.

Vrugt, J.A., C.J.F. ter Braak, H.V. Gupta, and B.A. Robinson,
Equifinality of formal (DREAM) and informal (GLUE) Bayesian approaches in hydrologic modeling, Stochastic Environmental Research and Risk Assessment, 1-16, 2009, In Press. doi:10.1007/s00477-008-0274-y

For more information please read:

Ter Braak, C.J.F.,
A Markov Chain Monte Carlo version of the genetic algorithm Differential Evolution: easy Bayesian computing for real parameter spaces, Stat. Comput., 16, 239 - 249, 2006. doi:10.1007/s11222-006-8769-1

Vrugt, J.A., H.V. Gupta, W. Bouten and S. Sorooshian,
A Shuffled Complex Evolution Metropolis algorithm for optimization and uncertainty assessment of hydrologic model parameters, Water Resour. Res., 39 (8), 1201, 2003. doi:10.1029/2002WR001642

Ter Braak, C.J.F., and J.A. Vrugt,
Differential Evolution Markov Chain with snooker updater and fewer chains, Statistics and Computing, 2008. doi:10.1007/s11222-008-9104-9

Vrugt, J.A., C.J.F. ter Braak, and J.M. Hyman,
Differential evolution adaptive Metropolis with snooker update and sampling from past states, SIAM journal on Optimization, 2009.

Vrugt, J.A., C.J.F. ter Braak, and J.M. Hyman,
Parallel Markov chain Monte Carlo simulation on distributed computing networks using multi-try Metropolis with sampling from past states, SIAM journal on Scientific Computing, 2009.

G. **Schoups, and J.A. Vrugt,**
A formal likelihood function for Bayesian inference of hydrologic models with correlated, heteroscedastic and non-Gaussian errors, Water Resources Research, 2010, In Press.

G. **Schoups, J.A. Vrugt, F. Fenicia, and N.C. van de Giesen,**
Inaccurate numerical solution of hydrologic models corrupts efficiency and robustness of MCMC simulation, Water Resources Research, 2010, In Press.

- 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.

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 Complete changelog on github.com/bumps/bumps

class bumps.dream.core.**Dream**(***kw*)

Bases: `object`

Data structure containing the details of the running DREAM analysis code.

CR: *Crossover* = None

CR_spacing: str = 'linear'

DE_eps: float = 0.05

DE_noise: float = 1e-06

DE_pairs: int = 3

DE_snooker_rate: float = 0.1

DE_steps: int = 10

DR_scale: float = 1

alpha: float = 0.01

convergence criteria

bounds_style: str = 'reflect'

burn: int = 0

draws: int = 100000

goalseek_interval: int = 2000000000.0

goalseek_minburn: int = 1000

goalseek_optimizer = None

model: *Model* = None

outlier_test = 'none'

population: ndarray[tuple[Any, ...], dtype[_ScalarT]] = None

sample(*state:* MCMCDraw | *None* = *None*, *abort_test=<function Dream.<lambda>>*)

Pull the requisite number of samples from the distribution

state: *MCMCDraw* = None

thinning: int = 1

use_delayed_rejection: bool = False

class bumps.dream.core.**Model**(**args*, ***kwargs*)

Bases: Protocol

Dream model interface definition.

bounds: Sequence[Sequence[float]]

Bounds for each parameter as a pair of sequences of the same length as the labels.

labels: List[str]

Labels for all the parameters

map(*pop: ndarray[tuple[Any, ...], dtype[_ScalarT]]*) → ndarray[tuple[Any, ...], dtype[_ScalarT]]

Function which takes an array of [k x n] and returns an array of n where k is the number of parameters. TODO: check if

5.5 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.

R()

plot(*title=None*, *fig=None*)

Plot the correlation histograms on the specified figure

5.6 crossover - Adaptive crossover support

Crossover	Fixed weight crossover ratios.
BaseAdaptiveCrossover	Adapted weight crossover ratios.
AdaptiveCrossover	Adapted weight crossover ratios.
LogAdaptiveCrossover	Adapted weight crossover ratios, log-spaced.

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.

5.6.1 Usage

1. Traditional differential evolution:

```
crossover = Crossover(CR=CR)
```

2. Weighted crossover ratios:

```
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:

```
crossover = AdaptiveCrossover(N)
```

The CRs are set to *[1/N, 2/N, … 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.

5.6.2 Compatibility Notes

For *Extra.pCR == 'Update'* in the matlab interface use:

```
CR = AdaptiveCrossover(Ncr=MCMCPar.nCR)
```

For *Extra.pCR != 'Update'* in the matlab interface use:

```
CR = Crossover(CR=[1./Ncr], pCR=[1])
```

class bumps.dream.crossover.**AdaptiveCrossover**(*N*)

Bases: *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: ndarray[tuple[Any, ...], dtype[_ScalarT]] = None

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: ndarray[tuple[Any, ...], dtype[_ScalarT]] = None

class bumps.dream.crossover.**Crossover**(*CR: ndarray[tuple[Any, ...], dtype[_ScalarT]]*, *weight: ndarray[tuple[Any, ...], dtype[_ScalarT]]*)

Bases: object

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.

CR: ndarray[tuple[Any, ...], dtype[_ScalarT]]

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: ndarray[tuple[Any, ...], dtype[_ScalarT]]

class bumps.dream.crossover.**LogAdaptiveCrossover**(*dim*, *N=4.5*)

Bases: *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: ndarray[tuple[Any, ...], dtype[_ScalarT]] = None

5.7 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

Nchain is the number of simultaneous changes that are running.

pop is an array of shape [Npop x Nvar] providing the active points used to generate the next proposal for each chain. This may be larger than the Nchains if the caller is using ancestor generations for active population. The current population is assumed to be the first *Nchain* rows of *pip*..

CR is an array of [Ncrossover x 2] crossover ratios with weights. The crossover ratio is the probability of selecting a particular dimension when generating the difference vector. The weights are used to select the crossover ratio. The weights are adjusted dynamically during the fit based on the acceptance rate of points generated with each crossover ratio.

max_pairs determines the maximum number of pairs which contribute to the differential evolution step. The number of pairs is chosen at random, with the difference vectors between the pairs averaged when creating the DE step.

eps determines the jitter added to the DE step.

snooker_rate determines the probability of using the snooker stepper. Otherwise use DE stepper 80% of the time, or apply the difference between pairs the other 20% of the time.

scale=1 scales the difference vector (constant, not stochastic)

noise=1e-6 adds random noise to the non-zero components of the difference vector. This noise is relative rather than absolute to allow for parameter values far from 1.0. Noise is also scaled by *scale*.

5.8 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.[1]

There are many other entropy calculations implemented within this file,[2][3][4][5] 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.

The following is an informal summary of the results from different algorithms applied to DREAM output:

```
from .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.wnn_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.wnn_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.wnn_entropy(points)
print("Entropy from wnn: %s"%str(S_wnn))

# Try wnn with gmm ... still no good
with T(): S_wnn, Serr_wnn = entropy.wnn_entropy(drawn.points, n_est=20000, gmm=20)
print("Entropy from wnn with gmm: %s"%str(S_wnn))
```

(continues on next page)

[1] Kramer, A., Hasenauer, J., Allgower, F., Radde, N., 2010. Computation of the posterior entropy in a Bayesian framework for parameter estimation in biological networks, in: 2010 IEEE International Conference on Control Applications (CCA). Presented at the 2010 IEEE International Conference on Control Applications (CCA), pp. 493-498. doi:10.1109/CCA.2010.5611198

[2] **Trujillo-Ortiz, A. and R. Hernandez-Walls. (2003). Mskekur: Mardia's** multivariate skewness and kurtosis coefficients and its hypotheses testing. A MATLAB file. [WWW document]. http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=3519

[3] **Mardia, K. V. (1970), Measures of multivariate skewnees and kurtosis with** applications. Biometrika, 57(3):519-530.

[4] **Mardia, K. V. (1974), Applications of some measures of multivariate skewness** and kurtosis for testing normality and robustness studies. Sankhy A, 36:115-128

[5] **Stevens, J. (1992), Applied Multivariate Statistics for Social Sciences.** 2nd. ed. New-Jersey:Lawrance Erlbaum Associates Publishers. pp. 247-248.

(continued from previous page)

```
# Try pure gmm ... pretty good
with T(): S_gmm, Serr_gmm = entropy.gmm_entropy(drawn.points, n_est=10000)
print("Entropy from 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))
```

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

bumps.dream.entropy.**cov_entropy**(*C*)

Entropy estimate from covariance matrix C

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_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.**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 $n_{\mathrm{est=n}}$

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.

Berrett, T. B., Samworth, R.J., Yuan, M., 2016. Efficient multivariate entropy estimation via k-nearest neighbour distances. DOI:10.1214/18-AOS1688 https://arxiv.org/abs/1606.00304

5.9 exppow - Exponential power density parameter calculator

exppow_pars	Return w(B) and c(B) for the exponential power density:

Exponential power density parameter calculator.

bumps.dream.exppow.**exppow_pars**(*B*)

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 -> -1: uniform
```

[1] Thiemann, M., M. Trosser, H. Gupta, and S. Sorooshian (2001). *Bayesian recursive parameter estimation for hydrologic models*, Water Resour. Res. 37(10) 2521-2535.

5.10 gelman - R-statistic convergence test

gelman	Calculates the R-statistic convergence diagnostic

Convergence test statistic from Gelman and Rubin, 1992.[1]

[1] Gelman, Andrew, and Donald B. Rubin.
"Inference from Iterative Simulation Using Multiple Sequences." Statistical Science 7, no. 4 (November 1, 1992): 457-72. https://doi.org/10.2307/2246093.

bumps.dream.gelman.**gelman**(*sequences*, *portion=0.5*)

Calculates the R-statistic convergence diagnostic

For more information please refer to: Gelman, A. and D.R. Rubin, 1992. Inference from Iterative Simulation Using Multiple Sequences, Statistical Science, Volume 7, Issue 4, 457-472. doi:10.1214/ss/1177011136

5.11 geweke - Geweke convergence test

geweke	Calculates the Geweke convergence diagnostic

Convergence test statistic from Gelman and Rubin, 1992.

bumps.dream.geweke.**geweke**(*sequences*, *portion=0.25*)

Calculates the Geweke convergence diagnostic

Refer to:

pymc-devs.github.com/pymc/modelchecking.html#informal-methods support.sas.com/documentation/cdl/en/statug/63033/HTML/default/viewer.htm#statug_introbayes_sect008.html

5.12 initpop - Population initialization routines

lhs_init	Latin Hypercube Sampling
cov_init	Initialize *N* sets of random variables from a gaussian model.

Population initialization routines.

To start the analysis an initial population is required. This will be an array of size M x 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. lhs_init(N, bounds) returns a latin hypercube sampling, which tests every parameter at each of N levels.

2. cov_init(N, x, cov) returns a Gaussian sample along the ellipse defined by the covariance matrix, cov. Covariance defaults to diag(dx) if dx is provided as a parameter, or to I if it is not.

Additional options are random box: rand(M, N) or random scatter: randn(M, N).

bumps.dream.initpop.**cov_init**(*N*, *x*, *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, x=x, N=20)
```

bumps.dream.initpop.**lhs_init**(*N*, *bounds*)

Latin Hypercube Sampling

Returns an array whose columns each have *N* samples from equally spaced bins between *bounds=(xmin, xmax)* for the column. DREAM bounds objects, with bounds.low and bounds.high can be used as well.

Note: Indefinite ranges are not supported.

5.13 ksmirnov - Kolmogorov-Smirnov test for MCMC convergence

ksmirnov	Kolmogorov-Smirnov test of similarity between the empirical distribution at the start and at the end of the chain.

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

continues on next page

Table 15 – continued from previous page

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:

```
x_new[k] = x[k]     if U > alpha
         = x_old[k] if U <= alpha
```

where alpha is p/p_old and accept is U > alpha.

Returns x_new, logp_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

MCMCModel	MCMC model abstract base class.
Density	Construct an MCMC model from a probablility density function.
LogDensity	Construct an MCMC model from a log probablility density function.
Simulation	Construct an MCMC model from a simulation function.
MVNormal	multivariate normal negative log likelihood function
Mixture	Create a mixture model from a list of weighted density models.

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:

```
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*:

```
M = Density(f, bounds=M_bounds)
```

If your model f computes the log probability density, use *LogDensity*:

```
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*:

```
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:

```
M = MVNormal(mu, sigma)
```

Mixture models:

```
M = Mixture(M1, w1, M2, w2, ...)
```

For more complex functions, you can subclass MCMCModel:

```
class Model(MCMCModel):
    def __init__(self, ..., bounds=None, ...):
        ...
        self.bounds = bounds
        ...
    def nnlf(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:

```
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:

```
model = Density(f, bounds=M_bounds)
```

Option 2. Model computes simulation, data has known 1-sigma uncertainty:

```
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:

```
model = Simulation(f, data=Measurement.MeasData, bounds=M_bounds,
                   gamma = MCMCPar.Gamma)
```

Option 4. Model directly computes log posterior density:

```
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.

class bumps.dream.model.**Density**(*f*, *bounds=None*, *labels=None*)

Bases: *MCMCModel*

Construct an MCMC model from a probablility density function.

f is the density function

bounds: **ndarray[tuple[Any, ...], dtype[_ScalarT]] = None**

labels: **List[str] = None**

map(*pop*)

nllf(*x*)

plot(*x*)

class bumps.dream.model.**LogDensity**(*f*, *bounds=None*, *labels=None*)

Bases: *MCMCModel*

Construct an MCMC model from a log probablility density function.

f is the log density function

bounds: **ndarray[tuple[Any, ...], dtype[_ScalarT]] = None**

labels: **List[str] = None**

map(*pop*)

nllf(*x*)

plot(*x*)

class bumps.dream.model.**MCMCModel**

Bases: object

MCMC 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: **ndarray[tuple[Any, ...], dtype[_ScalarT]] = None**

labels: **List[str] = None**

map(*pop*)

abstractmethod nllf(*x*)

plot(*x*)

class bumps.dream.model.**MVNormal**(*mu*, *sigma*)

Bases: *MCMCModel*

multivariate normal negative log likelihood function

bounds: ndarray[tuple[Any, ...], dtype[_ScalarT]] = None

labels: List[str] = None

map(*pop*)

nllf(*x*)

plot(*x*)

class bumps.dream.model.**Mixture**(**args*)

Bases: *MCMCModel*

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: ndarray[tuple[Any, ...], dtype[_ScalarT]] = None

labels: List[str] = None

map(*pop*)

nllf(*x*)

plot(*x*)

class bumps.dream.model.**Simulation**(*f=None*, *bounds=None*, *data=None*, *sigma=1*, *gamma=0*, *labels=None*)

Bases: *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) = w(G)/S exp(-c(G) |v/S|^(2/(1+G)))
```

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: ndarray[tuple[Any, ...], dtype[_ScalarT]] = None

labels: List[str] = None

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 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 parcoord - Parallel coordinates plot

best_in_bin	Find the index of the minimum *value* in each bin for the data in *x*.
parallel_coordinates	Produce a parallel coordinates plot.
plot	Plot parallel coordinates from a draw from the distribution.
scale	Returns *x* with values scaled to [0, 1].

bumps.dream.parcoord.**best_in_bin**(*x*, *value*, *bins=50*, *range=None*, *keep_empty=False*)

Find the index of the minimum *value* in each bin for the data in *x*.

x are the coordinates to be binned.

value are the objective to be minimized in each bin, such as chisq. There is one value for each x coordinate.

bins are the number of bins within *range* or an array of bin edges if the bins are not uniform.

range is the data range for the bins. The default is (x.min(), x.max()).

keep_empty is True if empty bins will return an index of -1. When False (the default), empty bins are removed.

Returns indices of the elements which are the minimum within each bin. This list may be shorter than the number of bins if *keep_empty* is False.

The algorithm works by assigning a bin number to each point then adding an offset in [0, 1) according to the scaled *value*. These point values are then sorted, and searched by bin number. The returned index will correspond to the first value in each bin, and therefore, the best value in that bin. From this the index into the original list can be returned.

bumps.dream.parcoord.**parallel_coordinates**(*data*, *labels=None*, *value=None*, *value_label=''*)

Produce a parallel coordinates plot.

data is a set of points to draw, one row per point.

labels is the label to assign to the dimensions, one per column in *data*.

value is an optional value to use to color the different lines. This could be the value of the control variable, or some additional dimension such as overall quality for the individual points.

value_label is the label to put on the colorbar for the line colors.

bumps.dream.parcoord.**plot**(*draw*, *nlines=150*, *control_var=None*)

Plot parallel coordinates from a draw from the distribution.

draw is the draw from the sample

nlines is the number of lines to draw

If *control_var* is provided, then the best value from each bin in the histogram for that variable will be chosen as a line to draw on the coordination plot. This will will produce a roughly equally spaced set of coordination points for that variable. If *control_var* is None, then lines will be picked at random.

bumps.dream.parcoord.**scale**(*x*, *axis=None*)

Returns *x* with values scaled to [0, 1].

If *axis* is not None, then scale values within each axis independently, otherwise use the min/max value across all dimensions.

5.19 state - Sampling history for MCMC

MCMCDraw	Initialization parameters:
Draw	A set of sample points from an MCMC draw with restrictions applied.
dream_load	Load the saved DREAM state.
h5load	
h5dump	

Sampling history for MCMC.

MCMC keeps track of a number of things during sampling.

The results may be queried as follows:

```
draws, generation, thinning, total_generations
sample(condition) returns draws, points, logp
gen_logp()        returns genid, logp
acceptance_rate() returns draws, AR
traces()          returns genid, chains
CR_weight()       returns draws, CR_weight
best()            returns best_x, best_logp
outliers()        returns outliers
show()/save(file)/load(file)
```

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. Due to tests for partially full circular buffers throughout the code (state.generation < state.Ngen) we are resetting generation to the size of the stored history on resume, and setting the generation_offset to the start of the history. If you need the number of generations across resume then use total_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.**Draw**(*state:* MCMCDraw, *portion: float = 1.0*, *vars: List[str | int] | None = None*, *exclude: List[str | int] | None = None*, *selection: Dict[int | str, Tuple[float, float]] | None = None*, *thin: int = 1*, *outliers: bool = False*, *derived: Callable | None = None*)

Bases: `object`

A set of sample points from an MCMC draw with restrictions applied.

Inputs

state is the *MCMCDraw* containing all points.

portion is the fraction of points to use for the sample, starting from the ends of the Markov chains.

vars is the list of vars that are active. Variables can be 0-origin integers or strings matching a variable label. Match uses standard glob rules, with * matching any sequence, *?* matching any character, *[abc]* matching one of *abc*, and *[!abc]* not matching one of *abc*.

exclude all labels that match a pattern in the list. Matching rules are the same as for *vars*. If a variable matches both *vars* and *exclude* then it is included.

selection restricts the points returned to a specific region of the parameter space using {var: (low, high)}. The var can be a 0-origin integer or label as for *vars*. It can also be *logp* if restricting by log likelihood. If a label matches multiple patterns then choose the first restriction (this may change to all that match in future).

thin is the amount of thinning to apply, using every nth point in the chain. This reduces autocorrelation in the chains and reduces artificial spikes in the marginal distributions. (The distributions will still be spiky, but the spikiness will be appropriate for the number of points in the bins.)

Attributes

points[n,k] are the set of sampled points, where n is the number of points and k is the number of parameters. The earlier points come from earlier generations, but this is an accident of the implementation and should not be relied on. n is usually the number of chains times the number of saved generations, but this will change depending on *portion*, *selection*, *thin* and *outliers* parameters. k is usually the number of fitted parameters, but the derived and active parameter lists will adjust this.

Nvar is the number of dimensions per point in the draw after including derived parameters and excluding nuisance parameters.

weights are the weights associated with each point. Since the chains are run at a single temperature the weights will all be 1.

logp[n] is the negative log likelihuud for each sampled point.

labels are the labels for the variables remaining after applying derived variables and restricting to the requested *vars* list.

integers is an array of flags, one per parameter* indicating whether the parameter is float (0) or int (1=floor). The values in the chain are managed as floats, and it is the responsibility of the likelihood function to transform them to integers. Models which use round, trunc or ceil rather than float are not yet supported.[1]

state is the original MCMCDraw.

[1] Note that discrete parameter handling is likely to change in the future. The derivative based optimizers in particular will need to know that the minimum step size for discrete parameters is one when computing the partial derivative. If bumps converts the parameters to integers before calling the likelihood function then we will only need a discrete flag since we will always use the *floor()* function.

Nvar: int

get_argsort_indices(*var: int*)

Sort var by value. Unlike argsort this caches the results for reuse.

integers: ndarray[tuple[Any, ...], dtype[_ScalarT]]

labels: List[str]

logp: ndarray[tuple[Any, ...], dtype[_ScalarT]]

points: ndarray[tuple[Any, ...], dtype[_ScalarT]]

portion: float

selection: Dict[int | str, Tuple[float, float]] | None

state: *MCMCDraw*

thin: int

title: str | None

vars: List[str | int] | None

weights: ndarray[tuple[Any, ...], dtype[_ScalarT]]

class bumps.dream.state.**MCMCDraw**(*Ngen: int*, *Nthin: int*, *Nupdate: int*, *Nvar: int*, *Npop: int*, *Ncr: int*, *thinning: int*)

Bases: object

Initialization parameters:

Ngen is the number of generations to store. These are retrievable through *gen_logp()* and *acceptance_rate()* methods. Note that this is not the same as the length of the saved chains.

Nthin is the number of thinned generations to store. These are retrievable through the *draw()* and *traces()* methods.

Nupdate is the number of crossover ratio sets to save. The DREAM algorithm runs in batches of steps, with various checks such as convergence and crossover ratio updates after each batch. The results are retrievable through the *CR_weight()* method.

Nvar is the number of parameters stored in each point.

Npop is the number of running chains. Note that unlike the –pop command line option, this is not the number of chains per parameter.

Ncr is the number of crossover ratios. DREAM maintains a fixed set of crossover ratios, choosing amongst them at each DE step. Depending on the success rate of the steps, it will adapt the weights after every batch of updates.

thinning is the number of update generations to skip between saves to the MC chains. Additional thinning can be done when drawing samples from the saved points.

Attributes and properties:

Ngen, *Nthin*, *Nupdate*, *Nvar*, *Npop*, *Ncr* gives the size of the buffers stored in state. If the buffers are not yet full, the returned sizes for the *traces()*, *gen_logp()*, *acceptance_rate()*, *CR_weight()* and *draw()* methods will be smaller.

generation is the number of generations in the current fit run. This is reset to zero each time the fit is resumed.

draws is the number of draws in the current fit run. This is reset to zero each time the fit is resumed. Use *total_generations* time *Npop* if you want the number of draws including all previous runs.

total_generations is the total number of generations for the fit across all fit runs.

title a title for the plot

labels a list of names for each parameter

thinning amount of thinning between the gen_logp buffer and the traces buffer. DREAM stores every nth generation starting at generation n (1-origin). It is possible to change thinning on resume, which will change the short range correlation in the traces, however these changes are not recorded. Generation number on the trace plot will be incorrect until DREAM burns through the entire buffer.

portion gives the fraction of each chain to use for statistical plots. A value can be automatically determined by calling *state.portion = state.trim_portion()*, or supplied by the user to various functions that work with the chains. The value should be between 0.0 and 1.0, where 1.0 means the entire chain is used for statistical plots.

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:

```
draw, weight = state.CR_weight()
plot(draw, weight)
```

See *crossover* for details.

property Ncr

property Ngen

property Npop

property Nsamples

property Nthin

property Nupdate

property Nvar

Number of parameters in the fit

acceptance_rate(*portion: float | None = None*)

Return the iteration number and the acceptance rate for that iteration.

For example, to plot the acceptance rate over time:

```
genid, AR = state.acceptance_rate()
plot(genid, AR)
```

V1.0.2: Now returns (genid, AR) rather than (num_draws, 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: Callable[[ndarray[tuple[Any, ...], dtype[_ScalarT]]], ndarray[tuple[Any, ...], dtype[_ScalarT]]]*, *labels: List[str] | None = None*)

*** DEPRECATED ***

Like set_derived_vars but operating in place, modifying the points in the history.

draw(*portion: float | None = None*, *vars: List[int] | None = None*, *exclude: List[int] | None = None*, *selection: Dict[int | str, Tuple[float, float]] | None = None*, *thin: int = 1*, *outliers: bool = False*, *derived: Callable | None = None*)

Return a sample from the posterior distribution.

portion is the portion of each chain to use

vars is the list of vars that are active. Variables can be 0-origin integers or strings matching a variable label. Match uses standard glob rules, with * matching any sequence, *?* matching any character, *[abc]* matching one of *abc*, and *[!abc]* not matching one of *abc*.

selection sets the range each parameter in the returned distribution, using {var: (low, high)}. If *var* matches multiple labels, then use the first restriction only. Missing variables use the full range.

thin takes every nth item.

outliers is True if outlier chains should be included (default False).

To plot the distribution for parameter p1:

```
draw = state.draw()
hist(draw.points[:, 0])
```

To plot the interdependence of p1 and p2:

```
draw = state.sample()
plot(draw.points[:, 0], draw.points[:, 1], '.')
```

entropy(*vars: List[int] | None = None*, *portion: float | None = None*, *selection: Dict[int | str, Tuple[float, float]] | None = None*, *n_est: int = 10000*, *thin: int | None = None*, *method: str | None = None*)

Return entropy estimate and uncertainty from an MCMC draw.

portion is the portion of each chain to use (uses self.portion if None).

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(*portion: float | None = None*)

Compute the Gelman and Rubin R-statistic for the Markov chains.

gen_logp(*portion: float | None = None*, *outliers: bool = 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:

```
genid, logp = state.gen_logp()
plot(genid, logp)
```

portion is the amount to trim from the head of the chain, or None to use the stored burn point.

If *outliers* is True, then return all chains, not just good chains.

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.

property labels

Labels associated with each parameter in a point. This doesn't include the labels for derived quantities. For these you will need to query state.draw().labels.

logp(*full: bool = False*)

Return the cumulative number of draws and the log likelihoods for each chain.

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.

** Deprecated ** Use state.gen_logp() instead.

logp_slice(*n: int*)

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: str = 'IQR'*, *portion: float | None = None*)

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', 'mahal', 'grubbs', or 'none'.

portion indicates what portion of the samples should be included in the outlier test. If None, then the stored portion is used.

min_slice(*n: int*)

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: ndarray[tuple[Any, ...], dtype[_ScalarT]]*, *logp: ndarray[tuple[Any, ...], dtype[_ScalarT]]*, *test: str = '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: int*, *Nthin: int*, *Nupdate: int*, *Nvar: int*, *Npop: int*, *Ncr: int*, *thinning: int*)

sample(***kw*)

Return a sample from the posterior distribution.

Deprecated use *draw()* instead.

save(*filename: Path | str*)

set_derived_vars(*fn: Callable[[ndarray[tuple[Any, ...], dtype[_ScalarT]]], ndarray[tuple[Any, ...], dtype[_ScalarT]]]*, *labels: List[str]*)

Define derived variables from the sample. When calling draw() it will add columns for the derived variables to each 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]:

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

set_integer_vars(*labels: List[str]*)

Indicate tha variables should be considered integer variables when computing statistics.

set_visible_vars(*labels: List[str]*)

show(*portion: float | None = None*, *figfile: str | Path | None = None*)

show_labels()

List of parameters in the state, with the parameter number for each. Use this to help with the inputs to state.draw().

stable_best()

Return True if the at least one full cycle of the circular buffer has passed since the best logp was first observed.

title = None

property total_generations

traces(*portion: float | None = None*, *thin: int = 1*, *outliers: bool = False*)

Grab traces for trace plot.

portion gives the starting point of the trace, skipping over the initial frames that may not have converged.

thin is the amount of additional thinning to apply on the traces.

outliers (default False) is True if bad chains should be included in the trace.

Returns *generation* [Ngen] and *chains* [Ngen x Nchain x Nvar]

trim_index(*portion: float | None = None*, *generation: int | None = None*)

Returns the generation index corresponding to the trim portion. The returned generation is relative to the start of the sampling, even if resume has been called multiple times to extend the burn or the sample size.

The optional *generation* parameter is needed in case we have a state object that is out of date with respect to the number of generations seen so far. This can happen when the state is transferred to the user interface thread much less frequently than the step monitor.

trim_portion()

Estimate the point at which the Markov chains have reached equilibrium, returning it as a portion of the currently stored length.

If not converged, then trim the first half of the samples.

bumps.dream.state.**dream_load**(*store*)

Load the saved DREAM state.

store is the path to the stored state, either as an HDF5 history file or as a bumps export directory. If the directory contains multiple exports then use the path to the .par file within the directory.

See also: h5load, load_state

bumps.dream.state.**h5dump**(*group: Group*, *state:* MCMCDraw)

bumps.dream.state.**h5load**(*group: Group*)

5.20 stats - Statistics helper functions

VarStats	
var_stats	
format_uncertainty	Opinionated formatting of mean and standard deviation.
format_num	
format_vars	
parse_var	Parse a line returned by format_vars back into the statistics for the variable on that line.
stats	Find mean and standard deviation of a set of weighted samples.
credible_interval	Find the credible interval covering the portion *ci* of the data.
shortest_credible_interval	Find the credible interval covering the portion *ci* of the data.

Statistics helper functions.

class bumps.dream.stats.**VarStats**(*label: str*, *index: int*, *p95: tuple[float, float]*, *p68: tuple[float, float]*, *median: float*, *mean: float*, *std: float*, *best: float*, *integer: bool = False*)

Bases: object

best: float

index: int

integer: bool = False

label: str

mean: float

median: float

property name

p68: tuple[float, float]

p68_range: tuple[float, float]

p95: tuple[float, float]

p95_range: tuple[float, float]

std: float

bumps.dream.stats.**credible_interval**(*x*, *ci*, *weights=None*, *x_sort_index=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 *[[x1_low, x1_high], [l2_low, x2_high], ...]* where *[xi_low, xi_high]* are the starting and ending values for credible interval *i*.

This function is faster if the inputs are already sorted.

bumps.dream.stats.**format_num**(*x*, *place*)

bumps.dream.stats.**format_uncertainty**(*mean*, *std*)

Opinionated formatting of mean and standard deviation.

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.**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=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.

bumps.dream.stats.**stats**(*x*, *weights=None*, *x_sort_index=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.**var_stats**(*draw*, *vars=None*)

5.21 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)

5.22 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.**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.

bumps.dream.util.**draw**(*k*, *n*)

Select k things from a pool of n without replacement.

5.23 varplot - Plot histograms for indiviual parameters

var_plot_size	
plot_vars	Plot parameter histograms in a grid on the figure.
plot_var	

Build layout for histogram plots

bumps.dream.varplot.**plot_var**(*draw*, *vstats*, *var*, *cbar*, *axes=None*, *nbins=30*, *full=False*)

bumps.dream.varplot.**plot_vars**(*draw*, *all_vstats*, *fig=None*, *nbins: int = 30*, *full: bool = False*)

Plot parameter histograms in a grid on the figure. Each bar on the histogram is shaded according the nllf of points in that bar, sorted to make a color gradient. The colorbar showing the nllf range is shared across all plots.

If *fig* is not provided and new figure will be created.

nbins controls the number of bars on each histogram.

Use *full=True* to plot the histogram across the full range of sample values. By default *full=False* and only the central 95% range is histogrammed.

The plots can be unreadable when labels overwrite each other. You can adjust the layout by setting H_SPACE, V_SPACE, T_MARGIN, B_MARGIN, L_MARGIN, R_MARGIN (see pyplot.subplots_adjust), TILE_W, TILE_H (for aspect ratio of the subplots) and CBAR_WIDTH (it's complicated). This will affect every subsequent plot.

bumps.dream.varplot.**var_plot_size**(*n*)

5.24 views - MCMC plotting methods

plot_all	
plot_corr	Plot kernel density estimate of the parameter correlation.
plot_corrmatrix	
plot_trace	
plot_logp	
format_vars	

MCMC plotting methods.

bumps.dream.views.**format_vars**(*all_vstats*)

bumps.dream.views.**plot_all**(*state:* MCMCDraw | Draw, *portion: float | None = None*, *figfile=None*)

bumps.dream.views.**plot_corr**(*draw*, *vars=(0, 1)*)

Plot kernel density estimate of the parameter correlation.

vars is the pair of parameters to plot (from *draw.labels*, 0-origin).

bumps.dream.views.**plot_corrmatrix**(*draw*, *nbins=50*, *vstats=None*, *full=False*, *fig=None*)

bumps.dream.views.**plot_logp**(*state:* MCMCDraw, *portion: float | None = None*, *outliers: bool = False*)

bumps.dream.views.**plot_trace**(*state:* MCMCDraw, *var: int = 0*, *portion: float | None = None*, *axes=None*, *fig=None*)

acr	ACR upper percentiles critical value for test of single multivariate normal outlier.
bounds	Bounds handling.
convergence	Convergence diagnostics
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.
gelman	Convergence test statistic from Gelman and Rubin, 1992.[1]
geweke	Convergence test statistic from Gelman and Rubin, 1992.
initpop	Population initialization routines.
ksmirnov	Kolmogorov-Smirnov test for MCMC convergence.
mahal	Mahalanobis distance calculator

continues on next page

Table 25 – continued from previous page

metropolis	MCMC step acceptance test.
model	MCMC model types
outliers	Chain outlier tests.
parcoord	
state	Sampling history for MCMC.
stats	Statistics helper functions.
tile	Split a rectangle into n panes.
util	Miscellaneous utilities.
varplot	Build layout for histogram plots
views	MCMC plotting methods.

PYTHON MODULE INDEX

b

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z