FitBenchmarking Documentation

Release 0.1.dev1

STFC

Jan 26, 2022

CONTENTS

1 FitBenchmarking **3**
1.1 Table Of Contents 3

Bibliography **143**

Python Module Index **145**

Index **147**

CHAPTER

ONE

FITBENCHMARKING

FitBenchmarking is an open source tool for comparing different minimizers/fitting frameworks. FitBenchmarking is cross platform and we support Windows, Linux and Mac OS. For questions, feature requests or any other inquiries, please open an issue on GitHub, or send us an e-mail at support@fitbenchmarking.com.

- **Installation Instructions:** https://fitbenchmarking.readthedocs.io/en/v0.2.0-rc1/users/install_instructions/index.html
- **User Documentation & Example Usage:** https://fitbenchmarking.readthedocs.io/en/v0.2.0-rc1/users/index.html
- **Community Guidelines:** https://fitbenchmarking.readthedocs.io/en/v0.2.0-rc1/contributors/guidelines.html
- **Automated Tests:** Run via GitHub Actions, https://github.com/fitbenchmarking/fitbenchmarking/actions, and tests are documented at https://fitbenchmarking.readthedocs.io/en/v0.2.0-rc1/users/tests.html

The package is the result of a collaboration between STFC's Scientific Computing Department and ISIS Neutron and Muon Facility and the Diamond Light Source. We also would like to acknowledge support from:

- EU SINE2020 WP-10, which received funding from the European Union's Horizon2020 research and innovation programme under grant agreement No 654000.
- EPSRC Grant EP/M025179/1 Least Squares: Fit for the Future.
- The Ada Lovelace Centre (ALC). ALC is an integrated, cross-disciplinary data intensive science centre, for better exploitation of research carried out at our large scale National Facilities including the Diamond Light Source (DLS), the ISIS Neutron and Muon Facility, the Central Laser Facility (CLF) and the Culham Centre for Fusion Energy (CCFE).

1.1 Table Of Contents

1.1.1 FitBenchmarking Concept Documentation

Here we outline why we built the fitbenchmarking software, and how the software benchmarks minimizers with the goal of highlighting the best tool for different types of data.

Why is FitBenchmarking important?

Fitting a mathematical model to data is a fundamental task across all scientific disciplines. (At least) three groups of people have an interest in fitting software:

- **Scientists**, who want to know what is the best algorithm for fitting their model to data they might encounter, on their specific hardware;
- **Scientific software developers**, who want to know what is the state-of-the-art in fitting algorithms and implementations, what they should recommend as their default solver, and if they should implement a new method in their software; and
- **Mathematicians** and **numerical software developers**, who want to understand the types of problems on which current algorithms do not perform well, and to have a route to expose newly developed methods to users.

Representatives of each of these communities have got together to build FitBenchmarking. We hope this tool will help foster fruitful interactions and collaborations across the disciplines.

Example workflow

The black crosses on the plot below are data obtained from an experiment at the VESUVIO beamline at ISIS Neutron and Muon source:

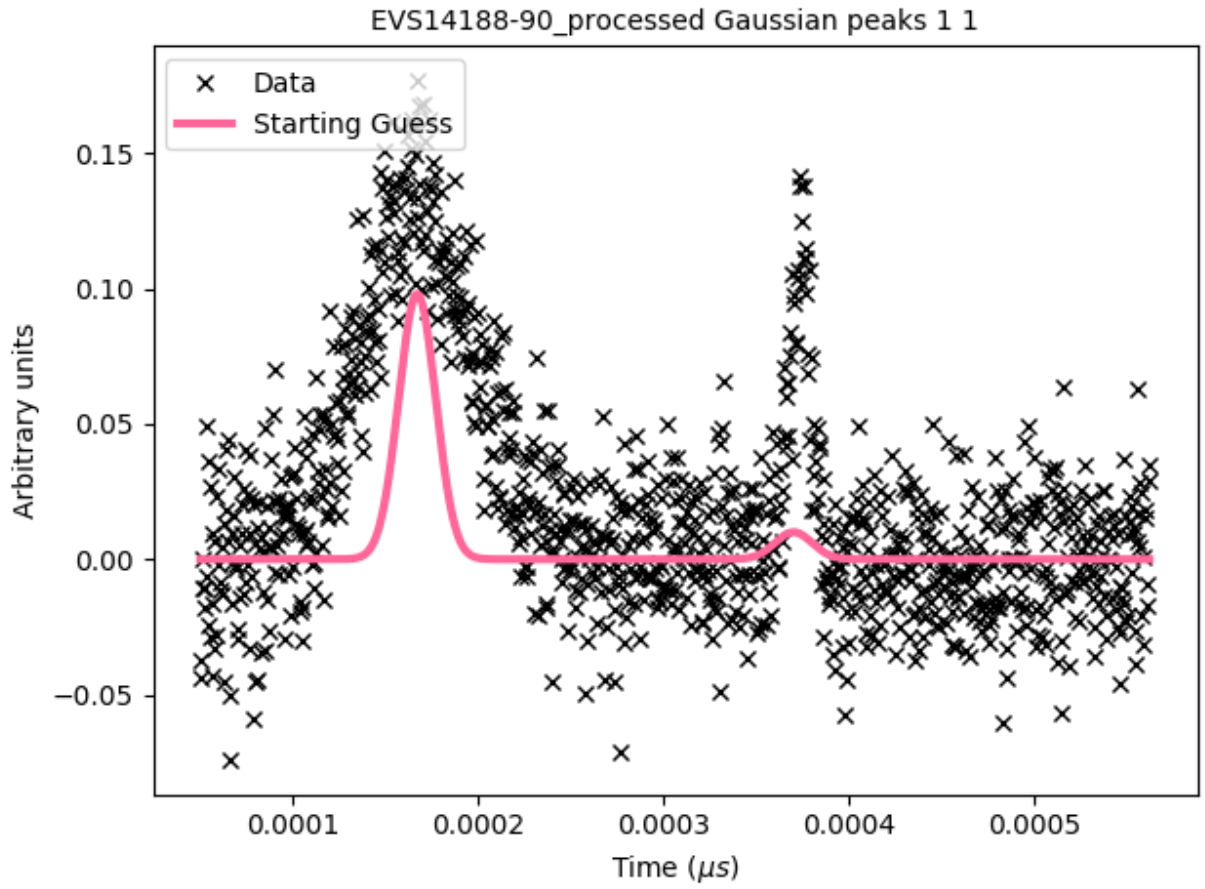


Fig. 1: VESUVIO experiment data

The scientist needs to interpret this data, and will typically use a data analysis package to help with this. Such packages are written by specialist scientific software developers, who are experts in analysing the kind of data produced by a given experiment; examples include Mantid, SasView, and Horace.

These packages include mathematical models, which depend on parameters, that can describe the data. We need to find values for the parameters in these models which best fit the data – for more background, see this Wikipedia article. The usual way this is done is by finding parameters that minimize the (weighted) squares of the error in the data, or χ^2 value. This is equivalent to formulating a nonlinear least-squares problem; specifically, given n data points (x_i, y_i) (the crosses in the figure above), together with estimates of the errors on the values of y_i, σ_i, we solve

$$\boldsymbol{\beta}^* = \arg\min_{\boldsymbol{\beta}} \underbrace{\sum_i \left(\frac{y_i - f(\boldsymbol{\beta}; x_i)}{\sigma_i} \right)^2}_{\chi^2(\boldsymbol{\beta})},$$

where $f(\boldsymbol{\beta}; x)$ is the model we're trying to fit, and $\boldsymbol{\beta}$ are the parameters we're trying to find.

Usually the scientist will supply a starting guess, $\boldsymbol{\beta}_0$ (the pink curve in the graph above), which describes where they think the solution might be. She then has to *choose which algorithm to use to fit the curve* from the selection available in the analysis software. Different algorithms may be more or less suited to a problem, depending on factors such as the architecture of the machine, the availability of first and second derivatives, the amount of data, the type of model used, etc.

Below we show the data overlayed by a blue curve, which is a model fitted using the implementation of the Levenberg-Marquardt algorithm from the GNU Scientific Library (`lmsder`). The algorithm claims to have found a local minimum with a Chi-squared error of 0.4771 in 1.9 seconds.

We also solved the nonlinear least squares problem using GSL's implementation of a Nelder-Mead simplex algorithm (`nmsimplex2`), which again claimed to solve the problem, this time in a faster 1.5 seconds. However, this time the Chi-squared error was 0.8505, and we plot the curve obtained in green below. The previous curve is in dotted-blue, for comparison.

By eye it is clear that the solution given by `lmsder` is better. As the volume of data increases, and we do more and more data analysis algorithmically, it is increasingly important that we have the best algorithm without needing to check it by eye.

FitBenchmarking will help the **scientist** make an informed choice by comparing runtime and accuracy of all available minimizers, on their specific hardware, on problems from their science area, which will ensure they are using the most appropriate minimizer.

FitBenchmarking will help the **scientific software developer** ensure that the most robust and quickest algorithms for the type of data analysis they support are available in their software.

FitBenchmarking will help **mathematicians** see what the state of the art is, and what kinds of data are problematic. It will give them access to real data, and will give a route for novel methods to quickly make it into production.

A workflow as described above plays a crucial role in the processing and analysis of data at large research facilities in tasks as diverse as instrument calibration, refinement of structures, and data analysis methods specific to different scientific techniques. FitBenchmarking will ensure that, across all areas that utilise least-squares fitting, scientists can be confident they are using the best tool for the job.

We discuss the specific FitBenchmarking paradigm in the Section *How does FitBenchmarking work?*

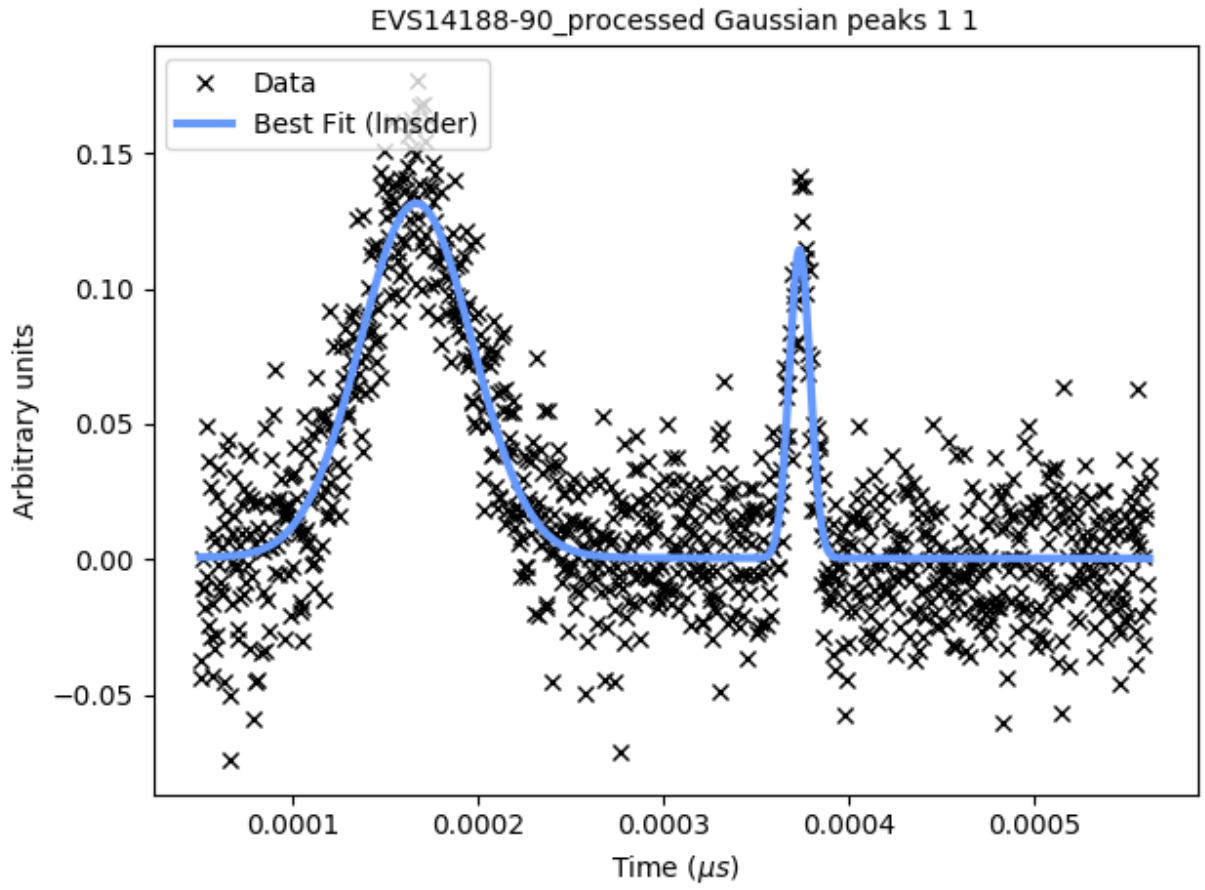


Fig. 2: GSL's `lmsder` (Levenberg-Marquardt) algorithm on the data

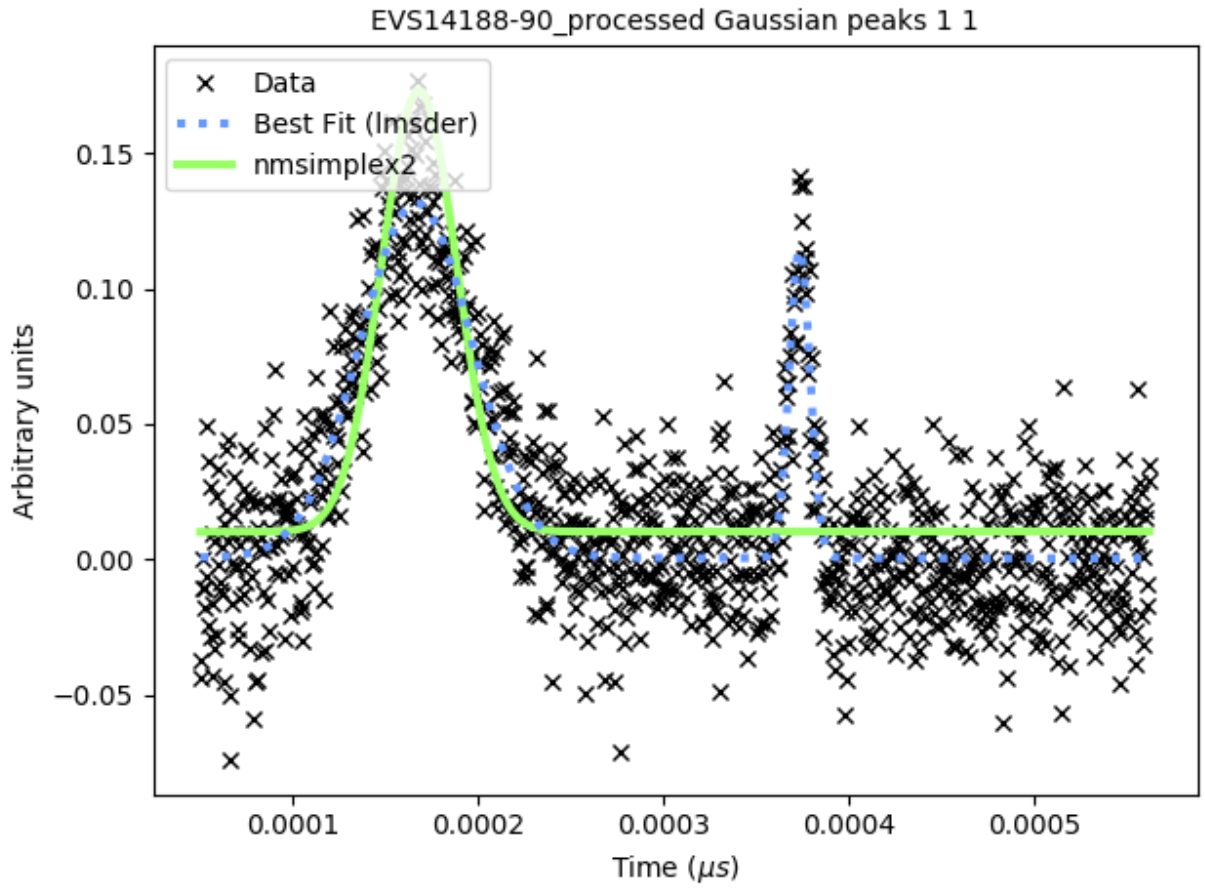
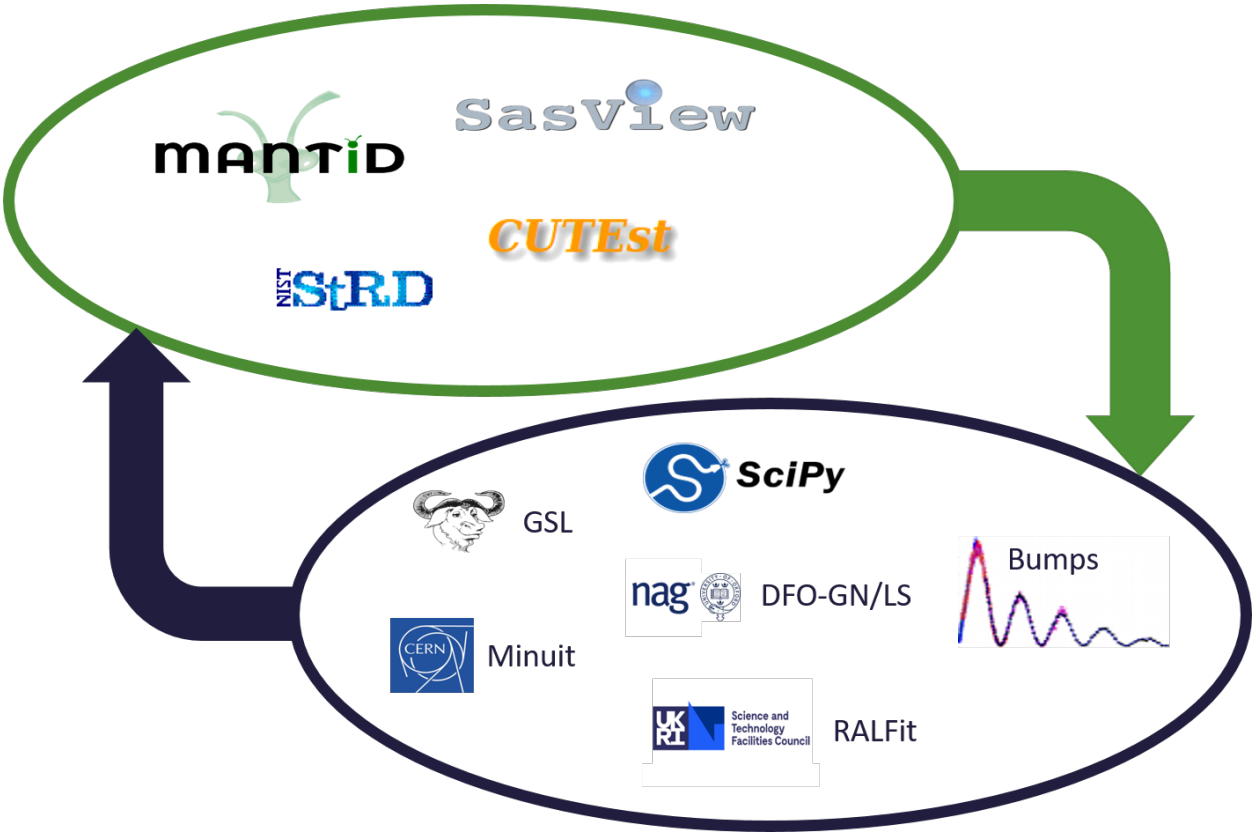


Fig. 3: GSL's `nmsimplex2` (Nelder-Mead Simplex) algorithm on the data

How does FitBenchmarking work?

FitBenchmarking takes data and models from real world applications and data analysis packages. It fits the data to the models by casting them as a nonlinear least-squares problem. We fit the data using a range of data fitting and nonlinear optimization software, and present comparisons on the accuracy and timings.



The Benchmarking Paradigm

FitBenchmarking can compare against any of the supported minmizers listed in *Minimizer Options*. We've also made it straightforward to add new software by following the instructions in *Adding Fitting Software* – the software just needs to be callable from Python.

Once you have chosen which minimizers you want to compare for a given problem, running FitBenchmarking will give you a comparison to indicate the minimizer that performs best.

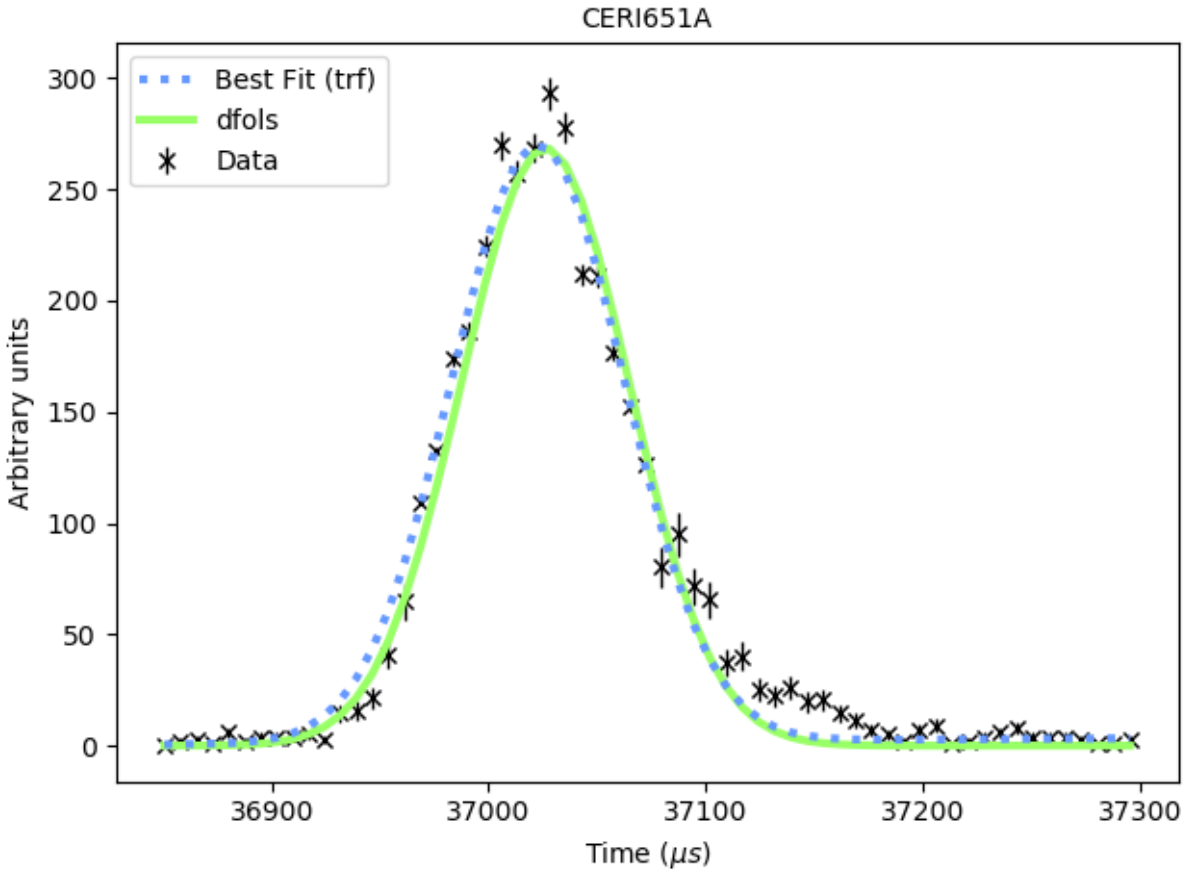
There are a number of options that you can pick to customize what your tests are comparing, or how they are run. A full list of these options, and how to select them, is given in the section *FitBenchmarking Options*.

FitBenchmarking creates tables, as given in the section *FitBenchmarking Output*, which show a comparison between the different minimizers available. An example of a table is:

This is the result of FitBenchmarking for a selection of software/minimizers and different problem definition types supported in FitBenchmarking. Both the raw chi squared values, and the values normalised with respect to the best minimizer per problem, are given. The problem names link to html pages that display plots of the data and the fit that was performed, together with initial and final values of the parameters. Here is an example of the final plot fit:

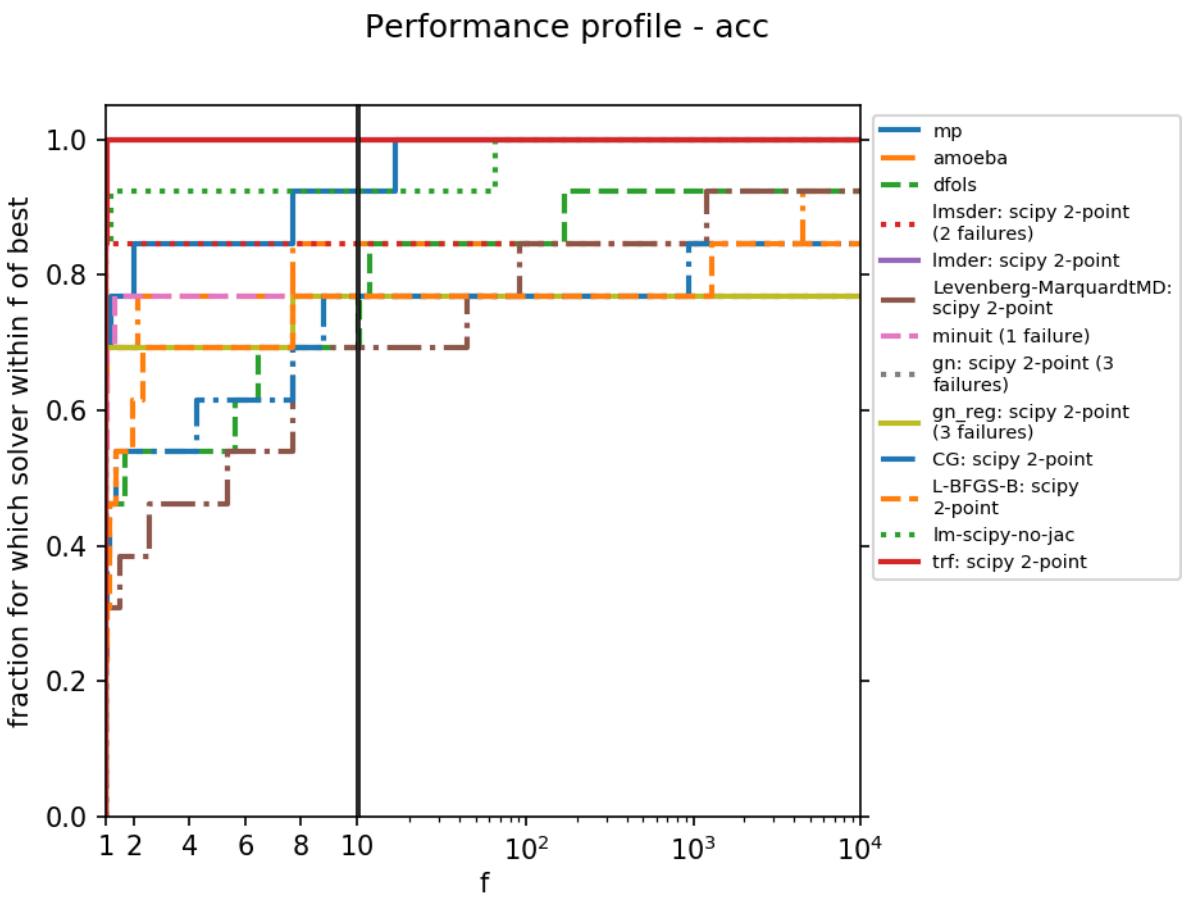
	WeightedNLLSCostFunc										
	scipy								scipy-ls		
	BFGS: scipy 2-point	CG: scipy 2-point	L-BFGS-B: scipy 2-point	Nelder-Mead	Newton-CG: scipy 2-point	Powell	SLSQP: scipy 2-point	TNC: scipy 2-point	dogbox: scipy 2-point	lm-scipy: scipy 2-point	trf: scipy 2-point
ENSO, Start 1	107.5 (1)	107.5 (1)	107.5 (1)	111.7 (1.039)[1]	110.2 (1.025)[2]	107.5 (1)	107.5 (1)	107.6 (1.001)[2]	107.5 (1)	107.5 (1)	107.5 (1)
ENSO, Start 2	107.5 (1)	107.5 (1)	107.5 (1)	107.5 (1)[1]	107.9 (1.004)[2]	107.6 (1)	107.5 (1)	107.5 (1)	107.5 (1)	107.5 (1)	107.5 (1)
Gauss3, Start 1	76.64 (1)[2]	77.14 (1.007)[2]	76.69 (1.001)[1]	76.64 (1)[1]	377.1 (4.92)	76.65 (1)	191.1 (2.494)[1]	76.64 (1)	76.64 (1)	76.64 (1)	76.64 (1)
Gauss3, Start 2	76.64 (1)	77.69 (1.014)[2]	81.89 (1.068)	76.64 (1)[1]	187.7 (2.45)	76.65 (1)	191.1 (2.494)[1]	76.7 (1.001)[2]	76.64 (1)	76.64 (1)	76.64 (1)
Hahn1, Start 1	1.61 (6.995)[2]	5.683 (24.68)[1]	3.333 (14.48)	4.987 (21.66)[1]	1136 (4933)	3.836 (16.66)	3356 (1.458e+04)	278.8 (1211)	2.941 (12.78)	6.217 (27)	0.2302 (1)
Hahn1, Start 2	2.998 (13.02)[2]	29.34 (127.5)[1]	1.07e+05 (4.648e+05)[2]	0.2304 (1.001)[1]	1.07e+05 (4.648e+05)[2]	6.284 (27.3)	3354 (1.457e+04)	1.07e+05 (4.648e+05)[2]	0.2302 (1)	0.2302 (1)	0.2302 (1)
Kirby2, Start 1	2.742 (1)[2]	5878 (2143)[2]	49.7 (18.12)	2.742 (1)	881.3 (321.4)	2.742 (1)	7307 (2664)	2.743 (1)	2.742 (1)	2.742 (1)	2.742 (1)
Kirby2, Start 2	2.742 (1)[2]	30.57 (11.15)[1]	30.44 (11.1)	2.742 (1)	72.33 (26.38)	2.742 (1)	4046 (1475)[2]	2.932 (1.069)[2]	2.742 (1)	2.742 (1)	2.742 (1)
Lanczos1, Start 1	3.321e-08 (2.201e+17)	1.305e-05 (8.652e+19)	1.383e-05 (9.169e+19)	0.0009938 (6.587e+21)	0.007201 (4.773e+22)[2]	0.06269 (4.156e+23)	0.0001257 (8.33e+20)	7.709e-05 (5.11e+20)	8.976e-06 (5.95e+19)	1.509e-25 (1)	1.545e-25 (1.024)
Lanczos1, Start 2	3.277e-07 (2.174e+18)	9.664e-07 (6.411e+18)	1.371e-05 (9.094e+19)	1.425e-05 (9.45e+19)[1]	0.005256 (3.487e+22)[2]	1.045e-05 (6.934e+19)	1.485e-05 (9.851e+19)	1.405e-05 (9.322e+19)	1.509e-25 (1.001)	1.507e-25 (1)	1.508e-25 (1)
Lanczos2, Start 1	3.39e-08 (1549)	1.333e-05 (6.091e+05)	1.373e-05 (6.275e+05)	0.0009937 (4.54e+07)	0.1435 (6.556e+09)[2]	0.06269 (2.865e+09)	0.0001254 (5.732e+06)	1.398e-05 (6.387e+05)	2.189e-11 (1)	2.189e-11 (1)	2.189e-11 (1)
Lanczos2, Start 2	3.27e-07 (1.494e+04)	1.69e-06 (7.72e+04)	1.375e-05 (6.281e+05)	1.425e-05 (6.51e+05)[1]	inf (inf)[3]	1.058e-05 (4.832e+05)	1.418e-05 (6.478e+05)	1.437e-05 (6.564e+05)	2.189e-11 (1)	2.189e-11 (1)	2.189e-11 (1)
MGH17, Start 1	1.623 (2.222e+04)	1.497 (2.049e+04)	1.497 (2.049e+04)	1.497 (2.049e+04)	1.497 (2.049e+04)[2]	1.496 (2.048e+04)	1.623 (2.222e+04)	1.497 (2.049e+04)	0.6454 (8833)[1]	7.306e-05 (1)	0.0001073 (1.469)[1]
MGH17, Start 2	7.306e-05 (1)	9.946e-05 (1.361)[1]	0.0001156 (1.582)	7.432e-05 (1.017)[1]	0.0005247 (7.182)	7.307e-05 (1)	7.511e-05 (1.028)	0.0001127 (1.543)	7.306e-05 (1)	7.306e-05 (1)	7.306e-05 (1)
Misra1c, Start 1	0.0008786 (1)	0.0008786 (1)[2]	0.1073 (122.1)	0.0008786 (1)	2.11 (2402)[2]	inf (inf)[3]	188.7 (2.148e+05)	0.1073 (122.1)	0.0008786 (1)	0.0008786 (1)	0.0008786 (1)
Misra1c, Start 2	0.0008786 (1)[2]	4.899 (5576)[2]	0.00491 (5.588)	0.0008786 (1)	0.2184 (248.6)[2]	inf (inf)[3]	188.4 (2.145e+05)	0.00491 (5.588)	0.0008786 (1)	0.0008786 (1)	0.0008786 (1)

Form	Parameters
CERI651A	f0=0.0, f1=0.0, f2=1.0, f3=3702.76, f4=26061.4, f5=38.7105, f6=37027.1



Performance Profile

With each test FitBenchmarking also produces a Dolan-Moré performance profile:



Fits are taken from all benchmarks, so if FitBenchmarking is run with `n` problems and `m` cost functions, the resulting profile plots will have `n*m` steps on the y-axis.

The solvers appearing in the top left corner may be considered the best performing on this test set. See Dolan and Moré (2001) for more information.

1.1.2 FitBenchmarking User Documentation

In these pages we describe how to install, run, and set options to use the FitBenchmarking software.

Installation

Fitbenchmarking will install all packages that can be installed through pip. This includes the minimizers from SciPy, bumps, DFO-GN/LS, Minuit, and also the SASModels package.

To enable Fitbenchmarking with the other supported software, you must install them with the external software instructions.

Installing FitBenchmarking and default fitting packages

We recommend using for running/installing Fitbenchmarking. The easiest way to install FitBenchmarking is by using the Python package manager, pip.

Installing via `pip`

FitBenchmarking can be installed via the command line by entering:

```
python -m pip install fitbenchmarking[bumps,DFO,gradient_free,minuit,SAS,numdifftools]
```

This will install the latest stable version of FitBenchmarking. For all available versions please visit the FitBenchmarking PyPI project. FitBenchmarking can also use additional software that cannot be installed using pip; please see *Installing External Software* for details.

Note: This install will include additional optional packages – see *Extra dependencies*. Any of the dependencies in the square brackets can be omitted, if required, and that package will not be available for Benchmarking, or will use the version of the package already on your system, if appropriate.

Installing from source

You may instead wish to install from source, e.g., to get the very latest version of the code that is still in development.

1. Download this repository or clone it using git: `git clone https://github.com/fitbenchmarking/fitbenchmarking.git`
2. Open up a terminal (command prompt) and go into the `fitbenchmarking` directory.
3. Once you are in the right directory, we recommend that you type

   ```
   python -m pip install .[bumps,DFO,gradient_free,minuit,SAS,numdifftools]
   ```

4. Additional software that cannot be installed via pip can also be used with FitBenchmarking. Follow the instructions at *Installing External Software*.

Extra dependencies

In addition to the external packages described at *Installing External Software*, some optional dependencies can be installed directly by FitBenchmarking. These are installed by issuing the commands

```
python -m pip install fitbenchmarking['option-1','option-2',...]
```

or

```
python -m pip install .['option-1','option-2',...]
```

where valid strings `option-x` are:

- `bumps`– installs the Bumps fitting package.
- `DFO` – installs the DFO-LS and DFO-GN fitting packages.
- `gradient_free` – installs the Gradient-Free-Optimizers fitting package
- `levmar` – installs the levmar fitting package. Note that the interface we use also requires BLAS and LAPLACK to be installed on the system, and calls to this minimizer will fail if these libraries are not present.
- `mantid` – installs the h5py and pyyaml modules.
- `matlab` – installs the dill module required to run matlab controllers in fitbenchmarking
- `minuit` – installs the Minuit fitting package.
- `SAS` – installs the Sasmodels fitting package and the tinycc module.
- `numdifftools` – installs the numdifftools numerical differentiation package.

Installing External Software

Fitbenchmarking will install all packages that are available through pip.

To enable Fitbenchmarking with the other supported software, they need to be installed and available on your machine. We give pointers outlining how to do this below, and you can find install scripts for Ubuntu 18.04 in the directory */build/<software>/*

CUTEst

CUTEst is used to parse SIF files in FitBenchmarking, and is called via the PyCUTEst interface.

Currently this is only supported for Mac and Linux, and can be installed by following the instructions outlined on the pycutest documentation

Please note that the `PYCUTEST_CACHE` environment variable must be set, and it must be in the `PYTHONPATH`.

GSL

GSL is used as a fitting software in FitBenchmarking, and is called via the pyGSL interface.

Install instructions can be found at the pyGSL docs. This package is also installable via pip, provided GSL is available on your system; see our example build script in *build/gsl*.

Note: pyGSL may not be installable with the latest versions of pip. We have found that 20.0.2 works for our tests.

Mantid

Mantid is used both as fitting software, and to parse data files.

Instructions on how to install Mantid for a range of systems are available at https://download.mantidproject.org/.

MATLAB

MATLAB is available to use as fitting software in FitBenchmarking, and is called via the MATLAB Engine API for Python.

To use this fitting software, both MATLAB and the MATLAB engine must be installed. Installation instructions for MATLAB are available at https://uk.mathworks.com/help/install/ug/install-products-with-internet-connection.html, and instructions for installing and setting up the MATLAB engine are here: https://uk.mathworks.com/help/matlab/matlab_external/install-the-matlab-engine-for-python.html

RALFit

RALFit is available to use as fitting software.

Instructions on how to build the python interface are at https://ralfit.readthedocs.io/projects/Python/en/latest/install.html

Running FitBenchmarking

Once installed, issuing the command

```
fitbenchmarking
```

will run the NIST average difficulty problem set on SciPy minmizers.

Running alternative problems

Other problems written in a *supported file format* can be analyzed with FitBenchmarking by passing the path using the `-p` or `--problem-sets` argument. Example problems can be downloaded from *Benchmark problems*, and they can also be found in the `fitbenchmarking/examples` directory of the code.

For example, to run the NIST low difficulty set from the base directory of the source, type into the terminal:

```
fitbenchmarking -p examples/benchmark_problems/NIST/low_difficulty
```

Changing the options

An options file can also be passed with the `-o` or `--options-file` argument. For example, the template file can be run by issuing the command

```
fitbenchmarking -o examples/options_template.ini \
-p examples/benchmark_problems/NIST/low_difficulty
```

Details about how the options file must be formatted are given in *FitBenchmarking Options*.

Changing the results directory

The default directory where the results are saved can be changed using the `-r` or `--results-dir` argument. The *results directory option* can also be changed in the options file.

```
fitbenchmarking -r new_results/
```

The default results directory is `fitbenchmarking_results`.

Optimization Algorithms

The different minimizers used in Fitbenchmarking implement various numerical optimization algorithms. These are catagorised by the algorithm type list, with options detailed below, along with advantages/disadvantages of each algorithm.

Derivative Free

Derivative Free methods do not compute the gradient of a function and so are often used to minimize problems with nondifferentiable functions. Some derivative free methods will attempt to approximate the gradient using a finite difference approach, another class of methods constructs a linear or quadratic model of the objective functions and uses a trust region approach to find the next iterate. Another widely used derivative free method is the Nelder-Mead simplex method. [Nocedal]

To select all minimizers in fitbenchmarking that use derivative free methods, use the algorithm type `deriv_free`.

Simplex (`simplex`)

Nelder-Mead is a simplex based algorithm, with a simplex S in $\mathbb{R}$ being defined as the convex hull of $n+1$ vertices $\{x_1, ..., x_{n+1}\}$.

In an iteration of the algorithm, the idea is to remove the vertex with the worst function value. It is then replaced with another point with a better value. An iteration consists of the following steps:

1. **Ordering** the vertices of S so that $f(x_1) \leq f(x_2) \leq ... \leq f(x_{n+1})$
2. Calculating the **centroid**, $\bar{x}$ of the best n points $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$
3. Carry out a **transformation** to compute the new simplex. Try to replace only the worst vertex x_{n+1} with a better point, which then becomes the new vertex. If this fails, the simplex is shrunk towards the best vertex x_1 and n new vertices are computed. The algorithm terminates when the simplex S is sufficiently small. [Singer]

Advantages:

- Method only requires 1 or 2 functions evaluations per iteration.

- Gives significant improvements in first few iterations - quick to produce satisfactory results.

Disadvantages:

- Stagnation can occur at non-optimal points, with large numbers of iterations leading to negligible improvement even when nowhere near a minimum.
- If numerical computation of function derivative can be trusted, then other algorithms are more robust.

[Singer]

Line Search Methods

In line search methods, each iteration is given by $x_{k+1} = x_k + \alpha_k p_k$, where p_k is the search direction and α_k is the step length.

The search direction is often of the form $p_k = -B_k^{-1} \nabla f_k$ where B_k is a symmetric and non-singular matrix. The form of p_k is dependent on algorithm choice.

The ideal step length would be $min_{\alpha>0} f(x_k + \alpha p_k)$ but this is generally too expensive to calculate. Instead an inexact line search condition such as the Wolfe Conditions can be used:

$$f(x_k + \alpha p_k) \leq f(x_k) + c_1 \alpha \nabla f_k^T p_k$$
$$f(x_k + \alpha_k p_k)^T p_k \geq c_2 \nabla f_k^T p_k$$

With $0 < c_1 < c_2 < 1$. Here, the first condition ensures that α_k gives a sufficient decrease in f, whilst the second condition rules out unacceptably short steps. [Nocedal]

Steepest Descent (`steepest_descent`)

Simple method where search direction p_k is set to be $-\nabla f_k$, i.e. the direction along which f decreases most rapidly.

Advantages:

- Low storage requirements
- Easy to compute

Disadvantages:

- Slow convergence for nonlinear problems

[Nocedal]

Conjugate Gradient (`conjugate_gradient`)

Conjugate Gradient methods have a faster convergence rate than Steepest Descent but avoid the high computational cost of methods where the inverse Hessian is calculated.

Given an iterate x_0, evaluate $f_0 = f(x_0), \nabla f_0 = \nabla f(x_0)$.

Set $p_0 \leftarrow -\nabla f_0, k \leftarrow 0$

Then while $\nabla f_k \neq 0$:

Carry out a line search to compute the next iterate, then evaluate ∇f_{k+1} and use this to determine the subsequent conjugate direction $p_{k+1} = -\nabla f(x_{k+1}) + \beta_k p_k$

Different variations of the Conjugate Gradient algorithm use different formulas for β_k, for example:

Fletcher-Reeves: $\beta_{k+1} = \frac{f_{k+1}^T \nabla f_{k+1}}{\nabla f_k^T \nabla f_k}$ Polak-Ribiere: $\beta_{k+1} = \frac{\nabla f_{k+1}^T (\nabla f_{k+1} - \nabla f_k)}{\|\nabla f_k\|^2}$

[Nocedal] [Poczos]

Advantages:

- Considered to be one of the best general purpose methods.
- Faster convergence rate compared to Steepest Descent and only requires evaluation of objective function and it's gradient - no matrix operations.

Disadvantages:

- For Fletcher-Reeves method it can be shown that if the method generates a bad direction and step, then the next direction and step are also likely to be bad. However, this is not the case with the Polak Ribiere method.
- Generally, the Polak Ribiere method is more efficient that the Fletcher-Reeves method but it has the disadvantage is requiring one more vector of storage.

[Nocedal]

BFGS (`bfgs`)

Most popular quasi-Newton method, which uses an approximate Hessian rather than the true Hessian which is used in a Newton line search method.

Starting with an initial Hessian approximation H_0 and starting point x_0:

While $\|\nabla f_k\| > \epsilon$:

Compute the search direction $p_k = -H_k \nabla f_k$

Then find next iterate x_{k+1} by performing a line search.

Next, define $s_k = x_{k+1} - x_k$ and $y_k = \nabla f_{k+1} - \nabla f_k$, then compute

$$H_{k+1} = (I - \rho_k s_k y_k^T) H_k (I - \rho_k y_k s_K^T) + \rho_k s_k s_k^T$$

with $\rho_k = \frac{1}{y_k^T s_k}$

Advantages:

- Superlinear rate of convergence
- Has self-correcting properties - if there is a bad estimate for H_k, then it will tend to correct itself within a few iterations.
- No need to compute the Jacobian or Hessian.

Disadvantages:

- Newton's method has quadratic convergence but this is lost with BFGS.

[Nocedal]

Gauss Newton (`gauss_newton`)

Modified Newton's method with line search. Instead of solving standard Newton equations

$$\nabla^2 f(x_k)p = -\nabla f(x_k),$$

solve the system

$$J_k^T J_k p_k^{GN} = -J_k^T r_k$$

(where J_k is the Jacobian) to obtain the search direction p_k^{GN}. The next iterate is then set as $x_{k+1} = x_k + p_k^{GN}$.

Here, the approximation of the Hessian $\nabla^2 f_k \approx J_k^T J_k$ has been made, which helps to save on computation time as second derivatives are not calculated.

Advantages:

- Calculation of second derivatives is not required.
- If residuals or their second order partial derivatives are small, then $J_k^T J_k$ is a close approximation to $\nabla^2 f_k$ and convergence of Gauss-Newton is fast.
- The search direction p_J^{GN} is always a descent direction as long as J_k has full rank and the gradient ∇f_k is nonzero.

Disadvantages:

- Without a good initial guess, or if the matrix $J_k^T J_k$ is ill-conditioned, the Gauss Newton Algorithm is very slow to converge to a solution.
- If relative residuals are large, then large amounts of information will be lost.
- J_k must be full rank.

[Nocedal] [Floater]

Trust Region

Trust region approach involves constructing a model function m_k that approximates the function f in the region, Δ, near the current point x_k. The model m_k is often a quadratic obtained by a Taylor Series expansion of the function around x_k.

$$m_k(p) = f_k + \nabla f(x_k)^T p + \frac{1}{2} p^T B_k p$$

where B_k is an approximation of the Hessian.

The subproblem to be solved at each iteration in order to find the step length is $\min_p m_k(p)$, subject to $\|p\| \leq \Delta_k$. [Nocedal]

To select all minimizers in fitbenchmarking that use a trust region approach, use the algorithm type `trust_region`.

Levenberg-Marquardt (`levenberg-Marquardt`)

Most widely used optimization algorithm, which uses the same Hessian approximation as Gauss-Newton but uses a trust region strategy instead of line search. As the Hessian approximation is the same as Gauss-Newton, convergence rate is similar.

For Levenberg-Marquardt, the model function m_k, is chosen to be

$$m_k(p) = \frac{1}{2}\|r_k\|^2 + p^T J_k^T r_k + \frac{1}{2} p^T J_k^T J_k p$$

So, for a spherical trust region, the subproblem to be solved at each iteration is $\min_p \frac{1}{2}\|J_k p + r_k\|^2$, subject to $\|p\| \leq \Delta_k$.

Levenberg-Marquardt uses a combination of gradient descent and Gauss-Newton method. When the solution p^{GN} lies inside of the trust region Δ, then p^{GN} also solves the sub-problem. Otherwise, the current iteration is far from the optimal value and so the search direction is determined using steepest descent, which performs better than Gauss-Newton when far from the minimum.

Advantages:

- Robust (more so than Gauss-Newton).
- Avoids the weakness with Gauss-Newton that Jacobian must be full rank.
- Fast to converge.
- Good initial guess not required.

Disadvantages:

- Similarly to Gauss-Newton, not good for large residual problems.
- Can be slow to converge if a problem has many parameters

[Nocedal] [Ranganathan]

Algorithm Types of Available Minimizers

Least Squares (`ls`):

```
{'bumps': ['lm-bumps', 'mp'],
 'dfo': ['dfogn', 'dfols'],
 'gradient_free': [],
 'gsl': ['lmsder', 'lmder'],
 'levmar': ['levmar'],
 'mantid': ['Levenberg-Marquardt',
            'Levenberg-MarquardtMD',
            'Trust Region',
            'FABADA'],
 'matlab': [],
 'matlab_curve': ['Trust-Region', 'Levenberg-Marquardt'],
 'matlab_opt': ['levenberg-marquardt', 'trust-region-reflective'],
 'matlab_stats': ['Levenberg-Marquardt'],
 'minuit': [],
 'ralfit': ['gn', 'hybrid', 'gn_reg', 'hybrid_reg'],
 'scipy': [None],
 'scipy_go': [None],
 'scipy_ls': ['lm-scipy', 'trf', 'dogbox']}
```

Deriv-Free (deriv_free):

```
{'bumps': ['amoeba', 'de'],
 'dfo': ['dfogn', 'dfols'],
 'gradient_free': ['HillClimbingOptimizer',
                   'RepulsingHillClimbingOptimizer',
                   'SimulatedAnnealingOptimizer',
                   'RandomSearchOptimizer',
                   'RandomRestartHillClimbingOptimizer',
                   'RandomAnnealingOptimizer',
                   'ParallelTemperingOptimizer',
                   'ParticleSwarmOptimizer',
                   'EvolutionStrategyOptimizer',
                   'BayesianOptimizer',
                   'TreeStructuredParzenEstimators',
                   'DecisionTreeOptimizer'],
 'gsl': ['nmsimplex', 'nmsimplex2'],
 'levmar': [],
 'mantid': ['Simplex', 'FABADA'],
 'matlab': ['Nelder-Mead Simplex'],
 'matlab_curve': [],
 'matlab_opt': [],
 'matlab_stats': [],
 'minuit': [],
 'ralfit': [],
 'scipy': ['Nelder-Mead', 'Powell'],
 'scipy_go': ['differential_evolution'],
 'scipy_ls': [None]}
```

General (general):

```
{'bumps': ['amoeba', 'newton', 'de'],
 'dfo': [],
 'gradient_free': ['HillClimbingOptimizer',
                   'RepulsingHillClimbingOptimizer',
                   'SimulatedAnnealingOptimizer',
                   'RandomSearchOptimizer',
                   'RandomRestartHillClimbingOptimizer',
                   'RandomAnnealingOptimizer',
                   'ParallelTemperingOptimizer',
                   'ParticleSwarmOptimizer',
                   'EvolutionStrategyOptimizer',
                   'BayesianOptimizer',
                   'TreeStructuredParzenEstimators',
                   'DecisionTreeOptimizer'],
 'gsl': ['nmsimplex',
         'nmsimplex2',
         'conjugate_pr',
         'conjugate_fr',
         'vector_bfgs',
         'vector_bfgs2',
         'steepest_descent'],
 'levmar': [],
 'mantid': ['BFGS',
```

(continues on next page)

(continued from previous page)

```
            'Conjugate gradient (Fletcher-Reeves imp.)',
            'Conjugate gradient (Polak-Ribiere imp.)',
            'Damped GaussNewton',
            'Simplex',
            'SteepestDescent'],
 'matlab': ['Nelder-Mead Simplex'],
 'matlab_curve': [],
 'matlab_opt': [],
 'matlab_stats': [],
 'minuit': ['minuit'],
 'ralfit': [],
 'scipy': ['Nelder-Mead',
           'Powell',
           'CG',
           'BFGS',
           'Newton-CG',
           'L-BFGS-B',
           'TNC',
           'SLSQP'],
 'scipy_go': ['differential_evolution', 'shgo', 'dual_annealing'],
 'scipy_ls': [None]}
```

Simplex (`simplex`):

```
{'bumps': ['amoeba'],
 'dfo': [],
 'gradient_free': [],
 'gsl': ['nmsimplex', 'nmsimplex2'],
 'levmar': [],
 'mantid': ['Simplex'],
 'matlab': ['Nelder-Mead Simplex'],
 'matlab_curve': [],
 'matlab_opt': [],
 'matlab_stats': [],
 'minuit': [],
 'ralfit': [],
 'scipy': ['Nelder-Mead'],
 'scipy_go': [],
 'scipy_ls': []}
```

Trust Region (`trust_region`):

```
{'bumps': ['lm-bumps'],
 'dfo': ['dfols', 'dfogn'],
 'gradient_free': [],
 'gsl': ['lmder', 'lmsder'],
 'levmar': ['levmar'],
 'mantid': ['Trust Region', 'Levenberg-Marquardt', 'Levenberg-MarquardtMD'],
 'matlab': [],
 'matlab_curve': ['Trust-Region', 'Levenberg-Marquardt'],
 'matlab_opt': ['levenberg-marquardt', 'trust-region-reflective'],
 'matlab_stats': ['Levenberg-Marquardt'],
```

(continues on next page)

(continued from previous page)

```
 'minuit': [],
 'ralfit': ['gn', 'hybrid'],
 'scipy': [],
 'scipy_go': [],
 'scipy_ls': ['lm-scipy', 'trf', 'dogbox']}
```

Levenberg-Marquardt (`levenberg-marquardt`):

```
{'bumps': ['lm-bumps'],
 'dfo': [],
 'gradient_free': [],
 'gsl': ['lmder', 'lmsder'],
 'levmar': ['levmar'],
 'mantid': ['Levenberg-Marquardt', 'Levenberg-MarquardtMD'],
 'matlab': [],
 'matlab_curve': ['Levenberg-Marquardt'],
 'matlab_opt': ['levenberg-marquardt'],
 'matlab_stats': ['Levenberg-Marquardt'],
 'minuit': [],
 'ralfit': ['gn', 'gn_reg'],
 'scipy': [],
 'scipy_go': [],
 'scipy_ls': ['lm-scipy']}
```

Gauss Newton (`gauss_newton`):

```
{'bumps': [],
 'dfo': ['dfogn'],
 'gradient_free': [],
 'gsl': [],
 'levmar': [],
 'mantid': ['Damped GaussNewton'],
 'matlab': [],
 'matlab_curve': [],
 'matlab_opt': [],
 'matlab_stats': [],
 'minuit': [],
 'ralfit': ['gn', 'gn_reg'],
 'scipy': [],
 'scipy_go': [],
 'scipy_ls': []}
```

BFGS (`bfgs`):

```
{'bumps': ['newton'],
 'dfo': [],
 'gradient_free': [],
 'gsl': ['vector_bfgs', 'vector_bfgs2'],
 'levmar': [],
 'mantid': ['BFGS'],
 'matlab': [],
 'matlab_curve': [],
 'matlab_opt': [],
```

(continues on next page)

(continued from previous page)

```
 'matlab_stats': [],
 'minuit': [],
 'ralfit': [],
 'scipy': ['BFGS', 'L-BFGS-B'],
 'scipy_go': [],
 'scipy_ls': []}
```

Conjugate Gradient (`conjugate_gradient`):

```
{'bumps': [],
 'dfo': [],
 'gradient_free': [],
 'gsl': ['conjugate_fr', 'conjugate_pr'],
 'levmar': [],
 'mantid': ['Conjugate gradient (Fletcher-Reeves imp.)',
            'Conjugate gradient (Polak-Ribiere imp.)'],
 'matlab': [],
 'matlab_curve': [],
 'matlab_opt': [],
 'matlab_stats': [],
 'minuit': [],
 'ralfit': [],
 'scipy': ['CG', 'Newton-CG', 'Powell'],
 'scipy_go': [],
 'scipy_ls': []}
```

Steepest Descent (`steepest_descent`):

```
{'bumps': [],
 'dfo': [],
 'gradient_free': [],
 'gsl': ['steepest_descent'],
 'levmar': [],
 'mantid': ['SteepestDescent'],
 'matlab': [],
 'matlab_curve': [],
 'matlab_opt': [],
 'matlab_stats': [],
 'minuit': [],
 'ralfit': [],
 'scipy': [],
 'scipy_go': [],
 'scipy_ls': []}
```

Global Optimization (`global_optimization`):

```
{'bumps': ['de'],
 'dfo': [],
 'gradient_free': ['HillClimbingOptimizer',
                   'RepulsingHillClimbingOptimizer',
                   'SimulatedAnnealingOptimizer',
                   'RandomSearchOptimizer',
                   'RandomRestartHillClimbingOptimizer',
```

(continues on next page)

(continued from previous page)

```
                  'RandomAnnealingOptimizer',
                  'ParallelTemperingOptimizer',
                  'ParticleSwarmOptimizer',
                  'EvolutionStrategyOptimizer',
                  'BayesianOptimizer',
                  'TreeStructuredParzenEstimators',
                  'DecisionTreeOptimizer'],
'gsl': [],
'levmar': [],
'mantid': ['FABADA'],
'matlab': [],
'matlab_curve': [],
'matlab_opt': [],
'matlab_stats': [],
'minuit': [],
'ralfit': [],
'scipy': [],
'scipy_go': ['differential_evolution', 'shgo', 'dual_annealing'],
'scipy_ls': []}
```

Cost functions

Fitbenchmarking supports multiple cost functions. These can be set via the `cost_func_type` option in *Fitting Options*.

Fitbenchmarking is designed to work with problems that have the form

$$\min_p F(r(x, y, p)).$$

The function $F(\cdot)$ is known as the cost function, while the function $r(x, u, p)$ is known as the *residual* of the cost function. The residual will generally be zero if the fit was perfect. Both of these quantities together define a cost function in FitBenchmarking.

The cost functions that are currently supported are:

- Non-linear least squares cost function

  class fitbenchmarking.cost_func.nlls_cost_func.**NLLSCostFunc**(*problem*)

  This defines the non-linear least squares cost function where, given a set of n data points (x_i, y_i), associated errors e_i, and a model function $f(x, p)$, we find the optimal parameters in the root least-squares sense by solving:

  $$\min_p \sum_{i=1}^{n} (y_i - f(x_i, p))^2$$

  where p is a vector of length m, and we start from a given initial guess for the optimal parameters. More information on non-linear least squares cost functions can be found here.

- Weighted non-linear least squares cost function

  class fitbenchmarking.cost_func.weighted_nlls_cost_func.**WeightedNLLSCostFunc**(*problem*)

  This defines the weighted non-linear least squares cost function where, given a set of n data points (x_i, y_i), associated errors e_i, and a model function $f(x, p)$, we find the optimal parameters in the root least-squares sense by solving:

  $$\min_p \sum_{i=1}^{n} \left(\frac{y_i - f(x_i, p)}{e_i} \right)^2$$

where p is a vector of length m, and we start from a given initial guess for the optimal parameters. More information on non-linear least squares cost functions can be found here.

- Hellinger non-linear least squares cost function

  class fitbenchmarking.cost_func.hellinger_nlls_cost_func.**HellingerNLLSCostFunc**(*problem*)

  This defines the Hellinger non-linear least squares cost function where, given a set of n data points (x_i, y_i), associated errors e_i, and a model function $f(x, p)$, we find the optimal parameters in the Hellinger least-squares sense by solving:

  $$\min_p \sum_{i=1}^{n} \left(\sqrt{y_i} - \sqrt{f(x_i, p)}\right)^2$$

  where p is a vector of length m, and we start from a given initial guess for the optimal parameters. More information on non-linear least squares cost functions can be found here and for the Hellinger distance measure see here.

- Poisson deviance cost function

  class fitbenchmarking.cost_func.poisson_cost_func.**PoissonCostFunc**(*problem*)

  This defines the Poisson deviance cost-function where, given the set of n data points (x_i, y_i), and a model function $f(x, p)$, we find the optimal parameters in the Poisson deviance sense by solving:

  $$\min_p \sum_{i=1}^{n} \left(y_i \left(\log y_i - \log f(x_i, p)\right) - \left(y_i - f(x_i, p)\right)\right)$$

  where p is a vector of length m, and we start from a given initial guess for the optimal parameters.

  This cost function is intended for positive values.

  This cost function is not a least squares problem and as such will not work with least squares minimizers. Please use *algorithm_type* to select *general* solvers. See options docs (*Fitting Options*) for information on how to do this.

FitBenchmarking Output

FitBenchmarking produces tables and reports called support pages as outputs. The links below give descriptions of these outputs.

Tables

Comparison Table

class fitbenchmarking.results_processing.compare_table.**CompareTable**(*results*, *best_results*, *options*, *group_dir*, *pp_locations*, *table_name*)

The combined results show the accuracy in the first line of the cell and the runtime on the second line of the cell.

Accuracy Table

class fitbenchmarking.results_processing.acc_table.**AccTable**(*results*, *best_results*, *options*, *group_dir*, *pp_locations*, *table_name*)

The accuracy results are calculated by evaluating the cost function with the fitted parameters.

Runtime Table

class fitbenchmarking.results_processing.runtime_table.**RuntimeTable**(*results*, *best_results*, *options*, *group_dir*, *pp_locations*, *table_name*)

The timing results are calculated from an average (over num_runs) using the timeit module in python. num_runs is set in *FitBenchmarking Options*.

Local Minimizer Table

class fitbenchmarking.results_processing.local_min_table.**LocalMinTable**(*results*, *best_results*, *options*, *group_dir*, *pp_locations*, *table_name*)

The local min results shows a `True` or `False` value together with $\frac{||J^T r||}{||r||}$. The `True` or `False` indicates whether the software finds a minimum with respect to the following criteria:

- $||r|| \leq$ RES_TOL,
- $||J^T r|| \leq$ GRAD_TOL,
- $\frac{||J^T r||}{||r||} \leq$ GRAD_TOL,

where J and r are the Jacobian and residual of $f(x, p)$, respectively. The tolerances can be found in the results object.

Display modes

The tables for `accuracy`, `runtime` and `compare` have three display modes:

```
{ 'abs': 'Absolute values are displayed in the table.',
  'both': 'Absolute and relative values are displayed in the table '
          'in the format ``abs (rel)``',
  'rel': 'Relative values are displayed in the table.'}
```

This can be set in the option file using the *Comparison Mode* option.

The *Local Minimizer Table* table is formatted differently, and doesn't use this convention.

Performance profile

Below the table there is a *Performance Profile*.

Support Pages

In each of the tables, a fitting_report for an individual result can be accessed by clicking on the associated table cell. Clicking the problem name at the start of a row will open a *Problem Summary Page* for the problem as a whole.

Fitting Report

The fitting report pages can be used to see more information about the problem and a given fit.

Each page represents a single fitting combination and is split into 2 sections.

Problem Outline

First is the initial problem. Here you will see information about the function being fit and the set of initial parameters used for the fitting. If plots are enabled (see *Make plots (make_plots)*), you will also see a scatter plot of the data to fit with a line of the initial fit given to the minimizer.

Fitting Results

The second section focusses on the results of the fitting. Here you will find the minimizer name and the final parameters for the fit found by the minimizer. A plot of the fit is also shown with an overlaid best fit from whichever minimizer was found to produce the smallest error.

Problem Summary Page

The problem summary page can be used to give an overview of the problem and solutions obtained.

Problem Outline

First is the initial problem. Here you will see information about the function being fit and the set of initial parameters used for the fitting. If plots are enabled (see *Make plots (make_plots)*), you will also see a scatter plot of the data to fit with a line of the initial fit given to the minimizer.

Comparison

The main plot on the page shows a comparison of all fits at once. This can be used to compare how cost functions perform for a problem accross all minimizers.

This uses colours to identify the cost function for each fit and shows all fits on a single graph. The best minimizer for each cost function is more pronounced on the plot.

This should not be used to identify the best individual fit, but can be a good indication of whether cost functions are biased to certain datapoints in the input.

Best Plots

The page ends with an expandable section for each cost function tested, which gives the parameter values and plot of the best fit obtained for that cost function.

Benchmark problems

To help choose between the different minimizers, we have made some curated problems available to use with FitBenchmarking. It is also straightforward to add custom data sets to the benchmark, if that is more appropriate; see *Problem Definition Files* for specifics of how to add additional problems in a supported file format.

Downloads

You can download a folder containing all examples here: `.zip` or `.tar.gz`

Individual problem sets are also available to download below.

We supply some standard nonlinear least-squares test problems in the form of the NIST nonlinear regression set and the relevant problems from the CUTEst problem set, together with some real-world data sets that have been extracted from Mantid and SASView usage examples and system tests. We've made it possible to extend this list by following the steps in *Adding Fitting Problem Definition Types*.

Each of the test problems contain:

- a data set consisting of points (x_i, y_i) (with optional errors on y_i, σ_i);
- a definition of the fitting function, $f(\boldsymbol{\beta}; x)$; and
- (at least) one set of initial values for the function parameters $\boldsymbol{\beta}_0$.

If a problem doesn't have observational errors (e.g., the NIST problem set), then FitBenchmarking can approximate errors by taking $\sigma_i = \sqrt{y_i}$. Alternatively, there is an option to disregard errors and solve the unweighted nonlinear least-squares problem, setting $\sigma_i = 1.0$ irrespective of what has been passed in with the problem data.

As we work with scientists in other areas, we will extend the problem suite to encompass new categories. The FitBenchmarking framework has been designed to make it easy to integrate new problem sets, and any additional data added to the framework can be tested with any and all of the available fitting methods.

Currently FitBenchmarking ships with data from the following sources:

CrystalField Data (Mantid)

Download `.zip` or `.tar.gz`

This folder (also found in *examples/benchmark_problems/CrystalField*) contains a test set for inelastic neutron scattering measurements of transitions between crystal field energy levels.

This problem has 8 parameters, and fits around 200 data points.

Warning: The external package Mantid must be installed to run this data set. See *Installing External Software* for details.

CUTEst (NIST files)

Download `.zip` or `.tar.gz`

This folder (also found in *examples/benchmark_problems/CUTEst*) contains several problems from the CUTEst continuous optimization testing environment which have been converted to the NIST format.

These problems all have 8 unknown parameters, and fit around 15 data points with the exception of `VESUVIOLS` which fits around 1000.

Data Assimilation

Download `.zip` or `.tar.gz`

This folder (also found in *examples/benchmark_problems/Data_Assimilation*) contains two examples using the data assimilation problem definition in fitbenchmarking. These examples follow the method set out in this paper.

These data files are synthetic and have been generated as an initial test of the minimizers. We plan to extend this with time series data which is more representative of the expectations for data assimilation in future updates.

These problems have either 2 or 3 unknown parameters, and fit either 100 or 1000 data points for `Simplified ANAC` and `Lorentz` problems respectively.

Powder Diffraction Data (SIF files)

Download `.zip` or `.tar.gz`

These problems (also found in the folder *examples/benchmark_problems/DIAMOND_SIF*) contain data from powder diffraction experiments. The data supplied comes from the I14 Hard X-Ray Nanoprobe beamline at the Diamond Light source, and has been supplied in the SIF format used by CUTEst.

These problems have either 66 or 99 unknown parameters, and fit around 5,000 data points.

> **Warning:** The external packages CUTEst and pycutest must be installed to run this data set. See *Installing External Software* for details.

MultiFit Data (Mantid)

Download `.zip` or `.tar.gz`

These problems (also found in the folder *examples/benchmark_problems/MultiFit*) contain data for testing the MultiFit functionality of Mantid. This contains a simple data set, on which two fits are done, and a calibration dataset from the MuSR spectrometer at ISIS, on which there are four fits available. See *The MultiFit documentation* for more details.

Basic Multifit has 3 unknown parameters, and fits 40 data points. MUSR62260 has 18 unknown parameters, and fits around 8000 data points.

> **Warning:** The external package Mantid must be installed to run this data set. See *Installing External Software* for details.
>
> This will also only work using the Mantid Minimizers.

Muon Data (Mantid)

Download `.zip` or `.tar.gz`

These problems (also found in the folder *examples/benchmark_problems/Muon*) contain data from Muon spectrometers. The data supplied comes from the HiFi and EMU instruments at STFC's ISIS Neutron and Muon source, and has been supplied in the format that Mantid uses to process the data.

These problems have between 5 and 13 unknown parameters, and fit around 1,000 data points.

> **Warning:** The external package Mantid must be installed to run this data set. See *Installing External Software* for details.

Neutron Data (Mantid)

Download `.zip` or `.tar.gz`

These problems (also found in the folder *examples/benchmark_problems/Neutron*) contain data from Neutron scattering experiments. The data supplied comes from the Engin-X, GEM, eVS, and WISH instruments at STFC's ISIS Neutron and Muon source, and has been supplied in the format that Mantid uses to process the data.

The size of these problems differ massively. The Engin-X calibration problems find 7 unknown parameters, and fit to 56-67 data points. The Engin-X vanadium problems find 4 unknown parameters, and fit to around 14,168 data points. The eVS problems find 8 unknown parameters, and fit to 1,025 data points. The GEM problem finds 105 unknown parameters, and fits to 1,314 data points. The WISH problems find 5 unknown parameters, and fit to 512 data points.

> **Warning:** The external package Mantid must be installed to run this data set. See *Installing External Software* for details.

NIST

Download `.zip` or `.tar.gz`

These problems (also found in the folder *examples/benchmark_problems/NIST*) contain data from the NIST Nonlinear Regression test set.

These problems are split into low, average and high difficulty. They have between 2 and 9 unknown parameters, and fit between 6 and 250 data points.

Poisson Data

Download `.zip` or `.tar.gz`

These problems (also found in the folder *examples/benchmark_problems/Poisson*) contain both simulated and real data measuring particle counts. The real data is ISIS muon data, and the simulated datasets have been made to represent counts using models provided by both Mantid and Bumps.

These problems have between 4 and 6 unknown parameters, and around 350, 800, and 2000 data points for simulated bumps, HIFI_160973, and simulated mantid respectively.

Warning: The external package Mantid must be installed to run this data set. See *Installing External Software* for details.

Small Angle Scattering (SASView)

Download `.zip` or `.tar.gz`

These problems (also found in the folder *examples/benchmark_problems/SAS_modelling/1D*) are two data sets from small angle scattering experiments. These are from fitting data to a cylinder, and have been supplied in the format that SASView uses to process the data.

These have 6 unknown parameters, and fit to either 20 or 54 data points.

Warning: The external package `sasmodels` must be installed to run this data set. See *Installing External Software* for details.

CUTEst (SIF files)

Download `.zip` or `.tar.gz`

This directory (also found in the folder *examples/benchmark_problems/SIF*) contain SIF files encoding least squares problems from the CUTEst continuous optimization testing environment.

These are from a wide range of applications. They have between 2 and 9 unknown parameters, and for the most part fit between 6 and 250 data points, although the *VESUVIO* examples (from the VESUVIO instrument at ISIS) have 1,025 data points (with 8 unknown parameters).

Warning: The external packages CUTEst and pycutest must be installed to run this data set. See *Installing External Software* for details.

SIF_GO

Download `.zip` or `.tar.gz`

This directory (also found in the folder *examples/benchmark_problems/SIF_GO*) contains SIF files encoding least squares problems from the CUTEst continuous optimization testing environment.

All of these problems have been modified, with finite bounds added for all parameters, making the problems appropriate for testing global optimization solvers. The bounds that have been added to each problem are the same as those used in SciPy's global optimization benchmark functions.

These problems have between 3 and 7 unknown parameters, and fit between 9 and 37 data points.

Warning: The external packages CUTEst and pycutest must be installed to run this data set. See *Installing External Software* for details.

Simple tests

Download `.zip` or `.tar.gz`

This folder (also found in *examples/benchmark_problems/simple_tests*) contains a number of simple tests with known, and easy to obtain, answers. We recommend that this is used to test any new minimizers that are added, and also that any new parsers reimplement these data sets and models (if possible).

These problems have 3 or 4 unknown parameters, and around 100 data points.

FitBenchmarking Options

The default behaviour of FitBenchmarking can be changed by supplying an options file. The default values of these options, and how to override them, are given in the pages below.

Fitting Options

Options that control the benchmarking process are set here.

Software (`software`)

Software is used to select the fitting software to benchmark, this should be a newline-separated list. Available options are:

- `bumps` (default software)
- `dfo` (default software)
- `gradient_free` (default software)
- `gsl` (external software – see *Installing External Software*)
- `levmar` (external software – see *Extra dependencies*)
- `mantid` (external software – see *Installing External Software*)
- `matlab` (external software – see *Installing External Software*)
- `matlab_curve` (external software – see *Installing External Software*)
- `matlab_opt` (external software – see *Installing External Software*)
- `matlab_stats` (external software – see *Installing External Software*)
- `minuit` (default software)
- `ralfit` (external software – see *Installing External Software*)
- `scipy` (default software)
- `scipy_ls` (default software)
- `scipy_go` (default software)

Default are `bumps`, `dfo`, `gradient_free`, `minuit`, `scipy`, `scipy_ls` and `scipy_go`

```
[FITTING]
software: bumps
          dfo
          minuit
          scipy
          scipy_ls
          scipy_go
```

Warning: Software must be listed to be here to be run. Any minimizers set in *Minimizer Options* will not be run if the software is not also present in this list.

Number of minimizer runs (`num_runs`)

Sets the number of runs to average each fit over.

Default is `5`

```
[FITTING]
num_runs: 5
```

Algorithm type (`algorithm_type`)

This is used to select what type of algorithm is used within a specific software. For a full list of available minimizers for each algorithm type, see *Algorithm Types of Available Minimizers*. The options are:

- `all` - all minimizers
- `ls` - least-squares fitting algorithms
- `deriv_free` - derivative free algorithms (these are algorithms that cannot use information about derivatives – e.g., the `Simplex` method in `Mantid`), see *Derivative Free*.
- `general` - minimizers which solve a generic *min f(x)*
- `simplex` - derivative free simplex based algorithms e.g. Nelder-Mead, see *Simplex*
- `trust_region` - algorithms which employ a trust region approach, see *Trust Region*
- `levenberg-marquardt` - minimizers that use the Levenberg Marquardt algorithm, see *Levenberg-Marquardt*.
- `gauss_newton` - minimizers that use the Gauss Newton algorithm, see *Gauss-Newton*
- `bfgs` - minimizers that use the BFGS algorithm, see *BFGS*
- `conjugate_gradient` - Conjugate Gradient algorithms, see *Conjugate Gradient*
- `steepest_descent` - Steepest Descent algorithms, see *Steepest Descent*
- `global_optimization` - Global Optimization algorithms

Default is `all`

```
[FITTING]
algorithm_type: all
```

Warning: Choosing an option other than `all` may deselect certain minimizers set in the options file

Jacobian method (`jac_method`)

This sets the Jacobian used. Choosing multiple options via a new line seperated list will result in all combinations being benchmarked. Current Jacobian methods are:

- `analytic` - uses the analytic Jacobian extracted from the fitting problem.
- `scipy` - uses *SciPy's finite difference Jacobian approximations*.
- `default` - uses the default derivative approximation implemented in the minimizer.
- `numdifftools` - uses the python package *numdifftools*.

Default is `default`

```
[FITTING]
jac_method: scipy
```

Warning: Currently analytic Jacobians are only available for problems that use the cutest and NIST parsers.

Hessian method (`hes_method`)

This sets the Hessian used. Choosing multiple options via a new line seperated list will result in all combinations being benchmarked. Current Hessian methods are:

- `default` - Hessian information is not passed to minimizers
- `analytic` - uses the analytic Hessian extracted from the fitting problem.
- `scipy` - uses *SciPy's finite difference approximations*.
- `numdifftools` - uses the python package *numdifftools*.

Default is `default`

```
[FITTING]
hes_method: default
```

Warning: Currently analytic Hessians are only available for problems that use the cutest and NIST parsers.

Cost function (cost_func_type)

This sets the cost functions to be used for the given data. Choosing multiple options via a new line seperated list will result in all combinations being benchmarked. Currently supported cost functions are:

- `nlls` - This sets the cost function to be non-weighted non-linear least squares, *NLLSCostFunc*.
- `weighted_nlls` - This sets the cost function to be weighted non-linear least squares, *WeightedNLLSCostFunc*.
- `hellinger_nlls` - This sets the cost function to be the Hellinger cost function, *HellingerNLLSCostFunc*.
- `poisson` - This sets the cost function to be the Poisson Deviation cost function, *PoissonCostFunc*.

Default is `weighted_nlls`

```
[FITTING]
cost_func_type: weighted_nlls
```

Maximum Runtime (max_runtime)

This sets the maximum runtime a minimizer has to solve one benchmark problem *num_runs* number of times, where *num_runs* is another option a user can set. If the minimizer is still running after the maximum time has elapsed, then this result will be skipped and FitBenchmarking will move on to the next minimizer / benchmark dataset combination. The main purpose of this option is to get to result tables quicker by limit the runtime.

max_runtime is set by specifying a number in unit of seconds. Please note that depending on platform the time specified with *max_runtime* may not match entirely with the absolute run-times specified in tables. Hence you may have to experiment a bit with this option to get the cutoff you want.

Default is 600 seconds

```
[FITTING]
max_runtime: 600
```

Minimizer Options

This section is used to declare the minimizers to use for each fitting software. If a fitting software has been selected in *Fitting Options* then a default set of minimizers for that solver will be run unless alternative minimizer options have been set. All minimizers for a software are included on the default list of minimizers unless otherwise stated.

Warning: Options set in this section will only have an effect if the related software is also set in *Fitting Options* (either explicitly, or as a default option).

Bumps (`bumps`)

Bumps is a set of data fitting (and Bayesian uncertainty analysis) routines. It came out of the University of Maryland and NIST as part of the DANSE (*Distributed Data Analysis of Neutron Scattering Experiments*) project.

FitBenchmarking currently supports the Bumps minimizers:

- Nelder-Mead Simplex (`amoeba`)
- Levenberg-Marquardt (`lm`)
- Quasi-Newton BFGS (`newton`)
- Differential Evolution (`de`)
- MINPACK (`mp`) This is a translation of *MINPACK* to Python.

Licence The main licence file for Bumps is here. Individual files have their own copyright and licence – if you plan to incorporate this in your own software you should first check that the licences used are compatible.

Links GitHub - bumps

The Bumps minimizers are set as follows:

```
[MINIMIZERS]
bumps: amoeba
       lm-bumps
       newton
       de
       mp
```

> **Warning:** The additional dependency Bumps must be installed for this to be available; See *Extra dependencies*.

Note: *de* is not included in the default list of minimizers for bumps. To run this solver, you must explicitly set the minimizer as seen above.

DFO (`dfo`)

There are two Derivative-Free Optimization packages, DFO-LS and DFO-GN. They are derivative free optimization solvers that were developed by Lindon Roberts at the University of Oxford, in conjunction with NAG. They are particularly well suited for solving noisy problems.

FitBenchmarking currently supports the DFO minimizers:

- Derivative-Free Optimizer for Least Squares (`dfols`)
- Derivative-Free Gauss-Newton Solver (`dfogn`)

Licence Both DFO-GN and DFO-LS are available under the GPL-3 licence. A proprietary licence is also available from NAG .

Links GitHub - DFO-GN GitHub - DFO-LS

The DFO minimizers are set as follows:

```
[MINIMIZERS]
dfo: dfols
     dfogn
```

Warning: Additional dependencies *DFO-GN* and *DFO-LS* must be installed for these to be available; See *Extra dependencies*.

Gradient-Free-Optimizers (`gradient_free`)

Gradient-Free-Optimizers are a collection of gradient-free methods capable of solving various optimization problems. Please note that Gradient-Free-Optimizers must be run with problems that have finite bounds on all parameters.

- Hill Climbing (`HillClimbingOptimizer`)
- Repulsing Hill Climbing (`RepulsingHillClimbingOptimizer`)
- Simulated Annealing (`SimulatedAnnealingOptimizer`)
- Random Search (`RandomSearchOptimizer`)
- Random Restart Hill Climbing (`RandomRestartHillClimbingOptimizer`)
- Random Annealing (`RandomAnnealingOptimizer`)
- Parallel Tempering (`ParallelTemperingOptimizer`)
- Particle Swarm (`ParticleSwarmOptimizer`)
- Evolution Strategy (`EvolutionStrategyOptimizer`)
- Bayesian (`BayesianOptimizer`)
- Tree Structured Parzen Estimators (`TreeStructuredParzenEstimators`)
- Decision Tree (`DecisionTreeOptimizer`)

Licence The Gradient-Free-Optimizers package is available under an MIT Licence .

The *gradient_free* minimizers are set as follows:

```
[MINIMIZERS]
gradient_free: HillClimbingOptimizer
               RepulsingHillClimbingOptimizer
               SimulatedAnnealingOptimizer
               RandomSearchOptimizer
               RandomRestartHillClimbingOptimizer
               RandomAnnealingOptimizer
               ParallelTemperingOptimizer
               ParticleSwarmOptimizer
               EvolutionStrategyOptimizer
               BayesianOptimizer
               TreeStructuredParzenEstimators
               DecisionTreeOptimizer
```

Warning: The additional dependency Gradient-Free-Optimizers must be installed for this to be available; See *Extra dependencies*.

Note: BayesianOptimizer, TreeStructuredParzenEstimators and DecisionTreeOptimizer may be slow running and so are not run by default when *gradient_free* software is selected. To run these minimizers you must explicity set them as seen above.

GSL (`gsl`)

The GNU Scientific Library is a numerical library that provides a wide range of mathematical routines. We call GSL using the pyGSL Python interface.

The GSL routines have a number of parameters that need to be chosen, often without default suggestions. We have taken the values as used by Mantid.

We provide implementations for the following packages in the multiminimize and multifit sections of the library:

- Levenberg-Marquardt (unscaled) (`lmder`)
- Levenberg-Marquardt (scaled) (`lmsder`)
- Nelder-Mead Simplex Algorithm (`nmsimplex`)
- Nelder-Mead Simplex Algorithm (version 2) (`nmsimplex2`)
- Polak-Ribiere Conjugate Gradient Algorithm (`conjugate_pr`)
- Fletcher-Reeves Conjugate-Gradient (`conjugate_fr`)
- The vector quasi-Newton BFGS method (`vector_bfgs`)
- The vector quasi-Newton BFGS method (version 2) (`vector_bfgs2`)
- Steepest Descent (`steepest_descent`)

Links SourceForge PyGSL

Licence The GNU Scientific Library is available under the GPL-3 licence .

The GSL minimizers are set as follows:

```
[MINIMIZERS]
gsl: lmsder
     lmder
     nmsimplex
     nmsimplex2
     conjugate_pr
     conjugate_fr
     vector_bfgs
     vector_bfgs2
     steepest_descent
```

Warning: The external packages GSL and pygsl must be installed to use these minimizers.

Mantid (mantid)

Mantid is a framework created to manipulate and analyze neutron scattering and muon spectroscopy data. It has support for a number of minimizers, most of which are from GSL.

- BFGS (BFGS)
- Conjugate gradient (Fletcher-Reeves) (Conjugate gradient (Fletcher-Reeves imp.))
- Conjugate gradient (Polak-Ribiere) (Conjugate gradient (Polak-Ribiere imp.))
- Damped GaussNewton (Damped GaussNewton)
- FABADA (FABADA)
- Levenberg-Marquardt algorithm (Levenberg-Marquardt)
- Levenberg-Marquardt MD (Levenberg-MarquardtMD) - An implementation of Levenberg-Marquardt intended for MD workspaces, where work is divided into chunks to achieve a greater efficiency for a large number of data points.
- Simplex (Simplex)
- SteepestDescent (SteepestDescent)
- Trust Region (Trust Region) - An implementation of one of the algorithms available in RALFit.

  Links GitHub - Mantid Mantid's Fitting Docs

Licence Mantid is available under the GPL-3 licence .

The Mantid minimizers are set as follows:

```
[MINIMIZERS]
mantid: BFGS
        Conjugate gradient (Fletcher-Reeves imp.)
        Conjugate gradient (Polak-Ribiere imp.)
        Damped GaussNewton
        FABADA
        Levenberg-Marquardt
        Levenberg-MarquardtMD
        Simplex
        SteepestDescent
        Trust Region
```

Warning: The external package Mantid must be installed to use these minimizers.

Levmar (levmar)

The levmar package which implements the Levenberg-Marquardt method for nonlinear least-squares. We interface via the python interface available on PyPI.

- Levenberg-Marquardt with supplied Jacobian (levmar) - the Levenberg-Marquardt method

Licence Levmar is available under the GPL-3 licence . A paid licence for proprietary commerical use is available from the author .

The *levmar* minimizer is set as follows:

```
[MINIMIZERS]
levmar: levmar
```

Warning: The additional dependency levmar must be installed for this to be available; See *Extra dependencies*. This package also requires the BLAS and LAPACK libraries to be present on the system.

Matlab (`matlab`)

We call the fminsearch function from MATLAB, using the MATLAB Engine API for Python.

- Nelder-Mead Simplex (`Nelder-Mead Simplex`)

Licence Matlab is a proprietary product .

The *matlab* minimizer is set as follows:

```
[MINIMIZERS]
matlab: Nelder-Mead Simplex
```

Warning: MATLAB must be installed for this to be available; See *Installing External Software*.

Matlab Curve Fitting Toolbox (`matlab_curve`)

We call the fit function from the MATLAB Curve Fitting Toolbox, using the MATLAB Engine API for Python.

- Levenberg-Marquardt (`Levenberg-Marquardt`)
- Trust-Region (`Trust-Region`)

Licence Matlab and the Curve Fitting Toolbox are both proprietary products .

The *matlab_curve* minimizers are set as follows:

```
[MINIMIZERS]
matlab_curve: Levenberg-Marquardt
              Trust-Region
```

Warning: MATLAB Curve Fitting Toolbox must be installed for this to be available; See *Installing External Software*.

Matlab Optimization Toolbox (`matlab_opt`)

We call the lsqcurvefit function from the MATLAB Optimization Toolbox, using the MATLAB Engine API for Python.

- Levenberg-Marquardt (`levenberg-marquardt`)
- Trust-Region-Reflective (`trust-region-reflective`)

Licence Matlab and the Optimization Toolbox are both proprietary products .

The *matlab_opt* minimizers are set as follows:

```
[MINIMIZERS]
matlab_opt: levenberg-marquardt
            trust-region-reflective
```

Warning: MATLAB Optimization Toolbox must be installed for this to be available; See *Installing External Software*.

Matlab Statistics Toolbox (`matlab_stats`)

We call the nlinfit function from the MATLAB Statistics Toolbox, using the MATLAB Engine API for Python.

- Levenberg-Marquardt (`Levenberg-Marquardt`)

Licence Matlab and the Statistics Toolbox are both proprietary products .

The *matlab_stats* minimizer is set as follows:

```
[MINIMIZERS]
matlab_stats: Levenberg-Marquardt
```

Warning: MATLAB Statistics Toolbox must be installed for this to be available; See *Installing External Software*.

Minuit (`minuit`)

CERN developed the Minuit 2 package to find the minimum value of a multi-parameter function, and also to compute the uncertainties. We interface via the python interface iminuit with support for the 2.x series.

- Minuit's MIGRAD (`minuit`)

Links Github - iminuit

Licence iminuit is released under the MIT licence, while Minuit 2 is LGPL v2 .

The Minuit minimizers are set as follows:

```
[MINIMIZERS]
minuit: minuit
```

Warning: The additional dependency Minuit must be installed for this to be available; See *Extra dependencies*.

RALFit (`ralfit`)

RALFit is a nonlinear least-squares solver, the development of which was funded by the EPSRC grant *Least-Squares: Fit for the Future*. RALFit is designed to be able to take advantage of higher order derivatives, although only first order derivatives are currently utilized in FitBenchmarking.

- Gauss-Newton, trust region method (`gn`)
- Hybrid Newton/Gauss-Newton, trust region method (`hybrid`)
- Gauss-Newton, regularization (`gn_reg`)
- Hybrid Newton/Gauss-Newton, regularization (`hybrid_reg`)

Links Github - RALFit. RALFit's Documentation on: Gauss-Newton/Hybrid models, the trust region method and The regularization method

Licence RALFit is available under a 3-clause BSD Licence

The RALFit minimizers are set as follows:

```
[MINIMIZERS]
ralfit: gn
        gn_reg
        hybrid
        hybrid_reg
```

> **Warning:** The external package RALFit must be installed to use these minimizers.

SciPy (`scipy`)

SciPy is the standard python package for mathematical software. In particular, we use the minimize solver for general minimization problems from the optimization chapter of SciPy's library. Currently we only use the algorithms that do not require Hessian information as inputs.

- Nelder-Mead algorithm (`Nelder-Mead`)
- Powell algorithm (`Powell`)
- Conjugate gradient algorithm (`CG`)
- BFGS algorithm (`BFGS`)
- Newton-CG algorithm (`Newton-CG`)
- L-BFGS-B algorithm (`L-BFGS-B`)
- Truncated Newton (TNC) algorithm (`TNC`)
- Sequential Least SQuares Programming (`SLSQP`)

Links Github - SciPy minimize

Licence Scipy is available under a 3-clause BSD Licence. Individual packages may have their own (compatible) licences, as listed here.

The SciPy minimizers are set as follows:

```
[MINIMIZERS]
scipy: Nelder-Mead
       Powell
       CG
       BFGS
       Newton-CG
       L-BFGS-B
       TNC
       SLSQP
```

SciPy LS (scipy_ls)

SciPy is the standard python package for mathematical software. In particular, we use the least_squares solver for Least-Squares minimization problems from the optimization chapter of SciPy's library.

- Levenberg-Marquardt with supplied Jacobian (`lm-scipy`) - a wrapper around MINPACK
- The Trust Region Reflective algorithm (`trf`)
- A dogleg algorithm with rectangular trust regions (`dogbox`)

Links Github - SciPy least_squares

Licence Scipy is available under a 3-clause BSD Licence. Individual packages many have their own (compatible) licences, as listed here.

The SciPy least squares minimizers are set as follows:

```
[MINIMIZERS]
scipy_ls: lm-scipy
          trf
          dogbox
```

SciPy GO (scipy_go)

SciPy is the standard python package for mathematical software. In particular, we use the Global Optimization solvers for global optimization problems from the optimization chapter of SciPy's library.

- Differential Evolution (derivative-free) (`differential_evolution`)
- Simplicial Homology Global Optimization (SHGO) (`shgo`)
- Dual Annealing (`dual_annealing`)

Links Github - SciPy optimization

Licence Scipy is available under a 3-clause BSD Licence. Individual packages may have their own (compatible) licences, as listed here.

The SciPy global optimization minimizers are set as follows:

```
[MINIMIZERS]
scipy_go: differential_evolution
          shgo
          dual_annealing
```

Note: The shgo solver is particularly slow running and should generally be avoided. As a result, this solver is not run by default when *scipy_go* software is selected. In order to run this minimizer, you must explicitly set it as above.

Jacobian Options

The Jacobian section allows you to control which methods for computing Jacobians the software uses.

Analytic (`analytic`)

Analytic Jacobians can only be used for specific *Problem Definition Files*. Currently the supported formats are cutest and NIST. The only option is:

- `default` - use the analytic derivative provided by a supported format.

Default is `default`

```
[JACOBIAN]
analytic: default
```

SciPy (`scipy`)

Calculates the Jacobian using the numerical Jacobian in SciPy, this uses `scipy.optimize._numdiff.approx_derivative`. The supported options are:

- `2-point` - use the first order accuracy forward or backward difference.
- `3-point` - use central difference in interior points and the second order accuracy forward or backward difference near the boundary.
- `cs` - use a complex-step finite difference scheme. This assumes that the user function is real-valued and can be analytically continued to the complex plane. Otherwise, produces bogus results.

Default is `2-point`

Licence SciPy is available under a 3-clause BSD Licence. Individual packages may have their own (compatible) licences, as listed here.

```
[JACOBIAN]
scipy: 2-point
```

Solver Default Jacobian (`default`)

This uses the approximation of the Jacobian that is used by default in the minimizer, and will vary between solvers. If the minimizer requires the user to pass a Jacobian, a warning will be printed to the screen and the *SciPy (scipy)* 2-point approximation will be used. The only option is:

- `default` - use the default derivative approximation provided by the software.

Default is `default`

```
[JACOBIAN]
default: default
```

Numdifftools (numdifftools)

Calculates the Jacobian using the python package `numdifftools`. We allow the user to change the method used, but other options (e.g, the step size generator and the order of the approximation) are set the defaults. The supported options are:

- `central` - central differencing. Almost as accurate as complex, but with no restriction on the type of function.
- `forward` - forward differencing.
- `backward` - backward differencing.
- `complex` - based on the complex-step derivative method of Lyness and Moler. Usually the most accurate, provided the function is analytic.
- `multicomplex` - extends complex method using multicomplex numbers. (see, e.g., Lantoine, Russell, Dargent (2012)).

Default is `central`.

Licence `numdifftools` is available under a 3-clause BSD Licence.

```
[JACOBIAN]
numdifftools: central
```

Hessian Options

The Hessian section allows you to control which methods for computing Hessians the software uses.

Analytic (analytic)

Analytic Hessians can only be used for specific *Problem Definition Files*. Currently the supported formats are cutest and NIST. The only option is:

- `default` - use the analytic derivative provided by a supported format.

Default is `default`

```
[HESSIAN]
analytic: default
```

SciPy (scipy)

Calculates the Hessian from the Jacobian using the finite differencing in SciPy, this uses `scipy.optimize._numdiff.approx_derivative`. The supported options are:

- `2-point` - use the first order accuracy forward or backward difference.
- `3-point` - use central difference in interior points and the second order accuracy forward or backward difference near the boundary.
- `cs` - use a complex-step finite difference scheme. This assumes that the user function is real-valued and can be analytically continued to the complex plane. Otherwise, produces bogus results.

Default is `2-point`

Licence SciPy is available under a 3-clause BSD Licence. Individual packages may have their own (compatible) licences, as listed here.

```
[HESSIAN]
scipy: 2-point
```

Default Hessian (`default`)

Hessian information is not passed to minimizers. The only option is:

- `default` - don't pass Hessian information to minimizers.

Default is `default`

```
[HESSIAN]
default: default
```

Numdifftools (`numdifftools`)

Calculates the Hessian from the Jacobian using the python package `numdifftools`. We allow the user to change the method used, but other options (e.g, the step size generator and the order of the approximation) are set to the defaults. The supported options are:

- `central` - central differencing. Almost as accurate as complex, but with no restriction on the type of function.
- `forward` - forward differencing.
- `backward` - backward differencing.
- `complex` - based on the complex-step derivative method of Lyness and Moler. Usually the most accurate, provided the function is analytic.
- `multicomplex` - extends complex method using multicomplex numbers. (see, e.g., Lantoine, Russell, Dargent (2012)).

Default is `central`.

Licence `numdifftools` is available under a 3-clause BSD Licence.

```
[HESSIAN]
numdifftools: central
```

Plotting Options

The plotting section contains options to control how results are presented.

Make plots (make_plots)

This allows the user to decide whether or not to create plots during runtime. Toggling this to False will be much faster on large data sets.

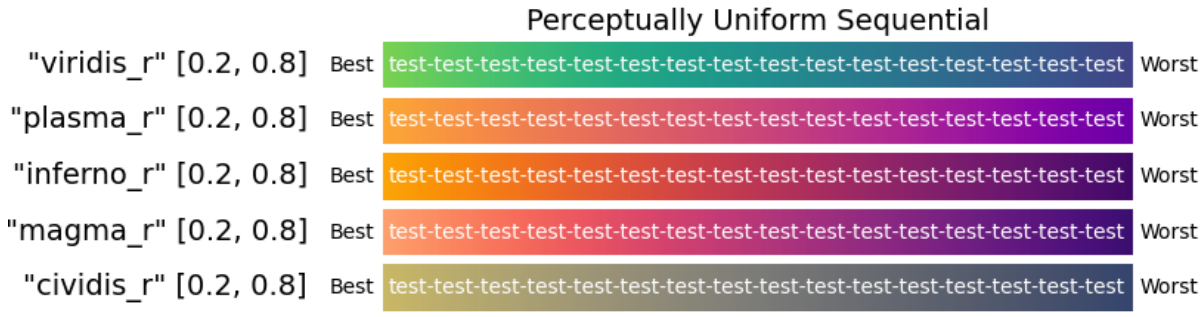
Default is `True` (`yes`/`no` can also be used)

```
[PLOTTING]
make_plots: yes
```

Colourmap (colour_map)

Specifies the name of the colourmap the user wishes to use, e.g. `magma`, `viridis`, `OrRd`. Options are:

- Any colourmap from the library in `matplotlib`, see the complete library here.
- Appending `_r` to the end of the name will reverse the colourmap.
- The following sequential colourmaps are recommended:



Default colourmap is `magma_r`

```
[PLOTTING]
colour_map: magma_r
```

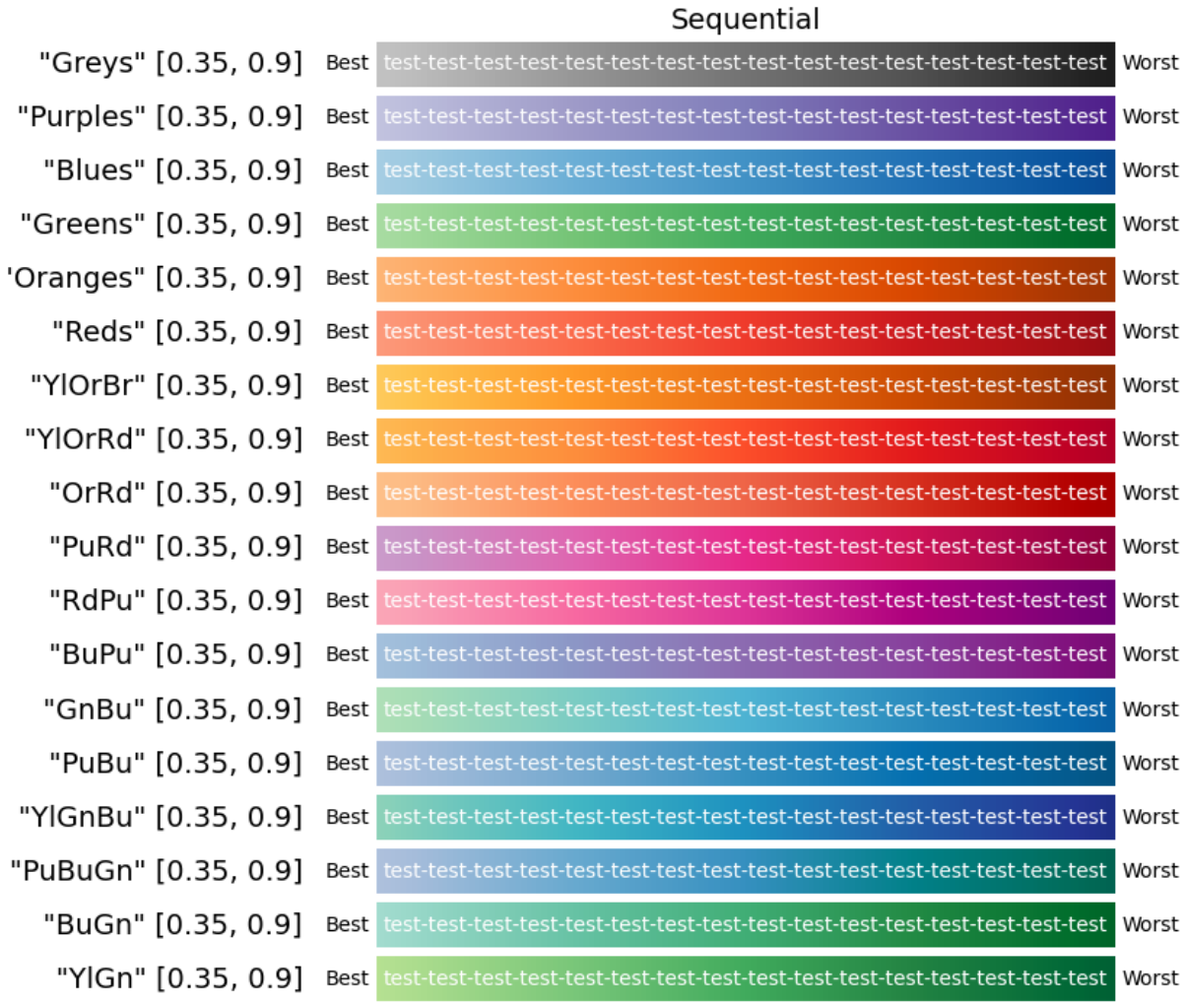
Colourmap Range (cmap_range)

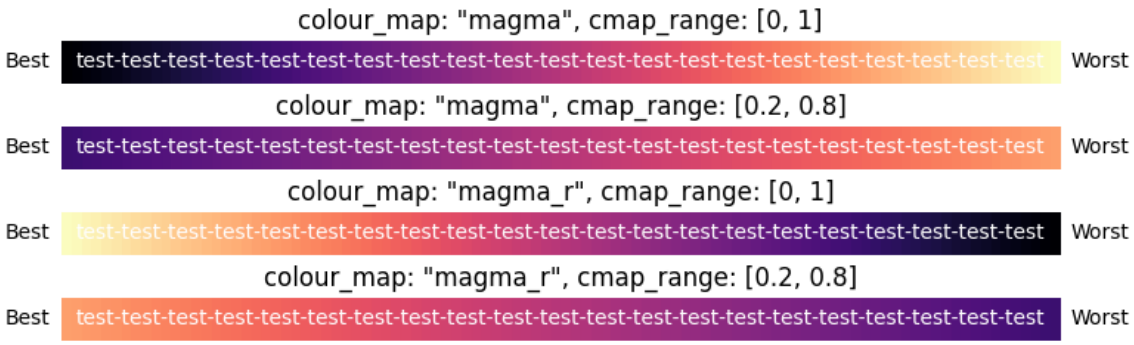
A two-element list used to specify the lower and upper limit of the chosen colourmap. Options are:

- `[lower_limit, upper_limit]` where limits consider the full colourscale limits to be 0 and 1, so any pair of values must fall within this range.
- Limits should be introduced to make the **white text** readable, see the following example.

Default for `magma` is `[0.2, 0.8]` (suitability depends on colourmap)

```
[PLOTTING]
colour_map: magma_r
cmap_range: [0.2, 0.8]
```


Sequential
"Greys" [0.35, 0.9]
"Purples" [0.35, 0.9]
"Blues" [0.35, 0.9]
"Greens" [0.35, 0.9]
'Oranges" [0.35, 0.9]
"Reds" [0.35, 0.9]
"YlOrBr" [0.35, 0.9]
"YlOrRd" [0.35, 0.9]
"OrRd" [0.35, 0.9]
"PuRd" [0.35, 0.9]
"RdPu" [0.35, 0.9]
"BuPu" [0.35, 0.9]
"GnBu" [0.35, 0.9]
"PuBu" [0.35, 0.9]
"YlGnBu" [0.35, 0.9]
"PuBuGn" [0.35, 0.9]
"BuGn" [0.35, 0.9]
"YlGn" [0.35, 0.9]
Best
Worst



Colour Upper Limit (`colour_ulim`)

Controls how relatively poorly a minimizer has to perform in order to receive the *worst* colour. For example, a value of 100 would mean that any performance greater than or equal to 100 times worse than the best minimizer would receive the worst colour. This ensures that colour scale is not compromised by especially poor relative results. Options are:

- Any float between 1 and `np.inf`
- Recommended value `100`

Default is `100`

```
[PLOTTING]
colour_map: magma_r
cmap_range: [0.2, 0.8]
colour_ulim: 100
```

Comparison mode (`comparison_mode`)

This selects the mode for displaying values in the resulting table options are `abs`, `rel`, `both`:

- `abs` indicates that the absolute values should be displayed
- `rel` indicates that the values should all be relative to the best result
- `both` will show data in the form "abs (rel)"

Default is `both`

```
[PLOTTING]
comparison_mode: both
```

Table type (table_type)

This selects the types of tables to be produced in FitBenchmarking. Options are:

- acc indicates that the resulting table should contain the chi squared values for each of the minimizers.
- runtime indicates that the resulting table should contain the runtime values for each of the minimizers.
- compare indicates that the resulting table should contain both the chi squared value and runtime value for each of the minimizers. The tables produced have the chi squared values on the top line of the cell and the runtime on the bottom line of the cell.
- local_min indicates that the resulting table should return true if a local minimum was found, or false otherwise. The value of $\frac{||J^T r||}{||r||}$ for those parameters is also returned. The output looks like {bool} (norm_value), and the colouring is red for false and cream for true. This option is only meaningful for least-squares cost functions.

Default is acc, runtime, compare, and local_min.

```
[PLOTTING]
table_type: acc
            runtime
            compare
            local_min
```

Output Options

The output section contains options to control how results are outputted.

Results directory (results_dir)

This is used to select where the output should be saved. If the *results directory command line argument* is provided, this option is overridden.

Default is fitbenchmarking_results

```
[OUTPUT]
results_dir: fitbenchmarking_results
```

Logging Options

The logging section contains options to control how fitbenchmarking logs information.

Logging file name (file_name)

This specifies the file path to write the logs to.

Default is fitbenchmarking.log

```
[LOGGING]
file_name: fitbenchmarking.log
```

Logging append (append)

This specifies whether to log in append mode or not. If append mode is active, the log file will be extended with each subsequent run, otherwise the log will be cleared after each run.

Default is `False` (`yes`/`no` can also be used)

```
[LOGGING]
append: no
```

Logging level (level)

This specifies the minimum level of logging to display on console during runtime. Options are (from most logging to least):

- `NOTSET`
- `DEBUG`
- `INFO`
- `WARNING`
- `ERROR`
- `CRITICAL`

Default is `INFO`

```
[LOGGING]
level: INFO
```

Logging external output (external_output)

This selects the amount of information displayed from third-parties. There are 3 options:

- `display`: Print information from third-parties to the stdout stream during a run.
- `log_only`: Print information to the log file but not the stdout stream.
- `debug`: Do not intercept third-party use of output streams.

Default is `log_only`

```
[LOGGING]
append: log_only
```

The options file must be a `.ini` formatted file (see here). Some example files can be found in the `examples` folder of the source, which is also available to download at *Benchmark problems*.

Problem Definition Files

In FitBenchmarking, problems can be defined using several file formats. The `examples/benchmark_problems` directory holds a collection of these that can be used for reference.

More information on the supported formats can be found on the following pages.

CUTEst File Format

The CUTEst file format in FitBenchmarking is a slight modification of the SIF format. Specifically, the data points, errors, and the number of variables must be defined in such a way to allow FitBenchmarking to access this data; see below. In FitBenchmarking, all SIF files are assumed to be CUTEst problems.

These problems are a subset of the problems in the CUTEr/st Test Problem Set, which may have been adapted to work with FitBenchmarking.

The SIF file format is very powerful, and CUTEst will work with arbitrary variable names, however for FitBenchmarking, these must match a set of expected variable names.

Licence This file format needs PyCUTEst and the packages ARCHDefs, CUTEst and SIFDECODE to be installed. PyCUTEst is available under the GPL-3 licence. ARCHDEFS, CUTEst and SIFDECODE are available under an LGPL (v2.1 or later) licence.

Modifications to the SIF format for FitBenchmarking problems

In order for FitBenchmarking to access the data, the SIF files must be written using the following conventions.

Defining Data

Data should be defined using the format:

```
RE X<idx>        <val_x>
RE Y<idx>        <val_y>
RE E<idx>        <val_error>
```

where <idx> is the index of the data point, and <val_x>, <val_y>, and <val_error> are the values attributed to it.

Usually, <idx> will range from 1 to <num_x>, with that defined as:

```
IE M             <num_x>
```

If <idx> does not start at 1, the following lines can be used to specify the range:

```
IE MLOWER        <min_idx>
IE MUPPER        <max_idx>
```

Defining Variables

For the free variables in functions, we use the convention:

```
IE N              <num_vars>
```

This is used to tell FitBenchmarking how many degrees of freedom we need to fit. In some cases variables will be vectors, and the number of degrees of freedom will be greater, most problems use `NVEC` as a convention to input the number of vectors.

Support for Bounds

Parameter ranges can be added to SIF files using the BOUNDS indicator card.

Currently in Fitbenchmarking, problems with parameter ranges can be handled by SciPy, Bumps, Minuit, Mantid, Matlab Optimization Toolbox, DFO, Levmar and RALFit fitting software. Please note that the following Mantid minimizers currently throw an exception when parameter ranges are used: BFGS, Conjugate gradient (Fletcher-Reeves imp.), Conjugate gradient (Polak-Ribiere imp.) and SteepestDescent.

Native File Format

In FitBenchmarking, the native file format is used to read IVP, Mantid, and SASView problems.

In this format, data is separated from the function. This allows running the same dataset against multiple different models to assess which is the most appropriate.

Examples of native problems are:

```
# Fitbenchmark Problem
software = 'ivp'
name = 'Lorentz'
description = 'A simple lorentz system for testing the ivp parser. Exact results should␣
↪be 10, 28, 8/3.'
input_file = 'lorentz3d.txt'
function = 'module=functions/lorentz,func=lorentz3d,step=0.1,sigma=11,r=30,b=3'
```

```
# FitBenchmark Problem#
software = 'Mantid'
name = 'HIFI 113856'
description = 'An example of (full) detector calibration for the HIFI instrument'
input_file = 'HIFIgrouped_113856.txt'
function = 'name=FlatBackground,A0=0;name=DynamicKuboToyabe,BinWidth=0.
↪05000000000000003,Asym=0.2,Delta=0.2,Field=0,Nu=0.1'
fit_ranges = {'x': [0.1, 16]}
```

```
# FitBenchmark Problem
software = 'SASView'
name = '1D cylinder (synthetic neutron) IV0'
description = 'A first iteration synthetic dataset generated for the 1D cylinder SASView␣
↪model in the fashion of neutron small angle scattering experiments. Generated on Fri␣
↪May 28 10:31:19 2021.'
input_file = '1D_cylinder_20_400_nosmearing_neutron_synth.txt'
```

(continues on next page)

(continued from previous page)

```
function = 'name=cylinder,radius=35.0,length=350.0,background=0.0,scale=1.0,sld=4.0,sld_
↪solvent=1.0'
```

These examples show the basic structure in which the file starts with a comment indicating it is a FitBenchmark problem followed by key-value pairs. Available keys are described below:

software Either 'IVP', 'Mantid', or 'SasView' (case insensitive).

This defines whether to use an IVP format, Mantid, or SasView to generate the model. The 'Mantid' software also supports Mantid's MultiFit functionality, which requires the parameters listed here to be defined slightly differently. More information can be found in *Native File Format (Mantid MultiFit)*.

Licence Mantid is available under a GPL-3 Licence. The component of SasView we use is SasModels, which is available under a BSD 3-clause licence.

name The name of the problem.

This will be used as a unique reference so should not match other names in the dataset. A sanitised version of this name will also be used in filenames with commas stripped out and spaces replaced by underscores.

description A description of the dataset.

This will be displayed in the Problem Summary Pages and Fitting Reports produced by a benchmark.

input_file The name of a file containing the data to fit.

The file must be in a subdirectory named `data_files`, and should have the form:

```
header

x11 [x12 [x13 ...]] y11 [y12 [y13 ...]] [e11 [e12 ...]]
x21 [x22 [x23 ...]] y21 [y22 [y23 ...]] [e21 [e22 ...]]
...
```

Mantid uses the convention of `# X Y E` as the header and SASView uses the convention `<X> <Y> <E>`, although neither of these are enforced. The error column is optional in this format.

If the data contains multiple inputs or outputs, the header must be written in one of the above conventions with the labels as "x", "y", or "e" followed by a number. An example of this can be seen in `examples/benchmark_problems/Data_Assimilation/data_files/lorentz.txt`

function This defines the function that will be used as a model for the fitting.

Inside FitBenchmarking, this is passed on to the specified software and, as such, the format is specific to the package we wish to use, as described below.

IVP

The IVP parser allows a user to define `f` in the following equation:

$$x' = f(t, x, *args)$$

To do this we use a python module to define the function. As in the above formula, the function can take the following arguments:

- *t* (float): The time to evaluate at
- *x* (np.array): A value for x to evaluate at
- **args* (floats): The parameters to fit

To link to this function we use a function string with the following parameters:

- *module*: The path to the module
- *func*: The name of the function within the module
- *step*: The time step that the input data uses (currently only fixed steps are supported - if you need varying time steps please raise an issue on our GitHub)
- **args*: Starting values for the parameters

Mantid

A Mantid function consists of one or more base functions separated by a semicolon. This allows for a powerful way of describing problems, which may have multiple components such as more than one Gaussian and a linear background.

To use one of the base functions in Mantid, please see the list available here.

Note: Any non-standard arguments (e.g. ties, constraints, fixes, ...) will only work with Mantid fitting software. Using other minimizers to fit these problems will result in the non-standard arguments being ignored.

SASView

SASView functions can be any of these.

fit_ranges This specifies the region to be fit.

It takes the form shown in the example, where the first number is the minimum in the range and the second is the maximum.

parameter_ranges An optional setting which specifies upper and lower bounds for parameters in the problem.

Similarly to `fit_ranges`, it takes the form where the first number is the minimum in the range and the second is the maximum.

Currently in Fitbenchmarking, problems with *parameter_ranges* can be handled by SciPy, Bumps, Minuit, Mantid, Matlab Optimization Toolbox, DFO, Levmar and RALFit fitting software. Please note that the following Mantid minimizers currently throw an exception when *parameter_ranges* are used: BFGS, Conjugate gradient (Fletcher-Reeves imp.), Conjugate gradient (Polak-Ribiere imp.) and SteepestDescent.

Native File Format (Mantid MultiFit)

As part of the Mantid parsing we also offer limited support for Mantid's MultiFit functionality.

Here we outline how to use Mantid's MultiFit with FitBenchmarking, in which some options differ from the standard *Native File Format*.

Warning: Due to the way Mantid uses ties (a central feature of MultiFit), MultiFit problems can only be used with Mantid minimizers.

In this format, data is separated from the function. This allows running the same dataset against multiple different models to assess which is the most appropriate.

An example of a multifit problem is:

```
# FitBenchmark Problem
software = 'Mantid'
name = 'MUSR62260'
description = 'Calibration data for mu SR intrument. Run 62260.'
input_file = ['MUSR62260_bkwd.txt','MUSR62260_bottom.txt','MUSR62260_fwd.txt','MUSR62260_
↪top.txt']
```

(continues on next page)

(continued from previous page)

```
function = 'name=FlatBackground,A0=0; name=GausOsc,A=0.2,Sigma=0.2,Frequency=1,Phi=0'
ties = ['f1.Sigma', 'f1.Frequency']
fit_ranges = [{'x': [0.1, 15.0]}, {'x': [0.1, 15.0]}, {'x': [0.1, 15.0]}, {'x': [0.1, 15.
↪0]}]
```

Below we outline the differences between this and the *Native File Format*.

software Must be *Mantid*.

name As in *Native File Format*.

description As in *Native File Format*.

input_file As in *Native File Format*, but you must pass in a list of data files (see above example).

function As in *Native File Format*.

When fitting, this function will be used for each of the `input_files` given simultaneously.

ties This entry is used to define global variables by tieing a variable across input files.

Each string in the list should reference a parameter in the function using Mantid's convention of `f<i>.<name>` where `i` is the position of the function in the function string, and `name` is the global parameter.

For example to run a fit which has a shared background and peak height, the function and ties fields might look like:

```
function='name=LinearBackground, A0=0, A1=0; name=Gaussian, Height=0.01,␣
↪PeakCentre=0.00037, Sigma=1e-05'
ties=['f0.A0', 'f0.A1', 'f1.Height']
```

fit_ranges As in *Native File Format*.

NIST Format

The NIST file format is based on the nonlinear regression problems found at the NIST Standard Reference Database. Documentation and background of these problems can be found here.

We note that FitBenchmarking recognizes the NIST file type by checking the first line of the file starts with *# NIST/ITL StRD*.

Detecting problem file type

FitBenchmarking detects which parser to use in two ways:

- For the CUTEst file format we check that the extension of the data file is *sif*
- For native and NIST file formats we check the first line of the file
  - *# FitBenchmark Problem* corresponds to the native format
  - *# NIST/ITL StRD* corresponds to the NIST format

FitBenchmarking Tests

The tests for FitBenchmarking require `pytest>=3.6`. We have split the tests into three categories:

- `default`: denotes tests involving `pip` installable *software packages*,
- `all`: in addition to `default`, also runs tests on *external packages*, with the exception of matlab.
- `matlab`: Runs tests for matlab fitting software. Please note that these tests can currently only be run locally through pytest.

Unit tests

Each module directory in FitBenchmarking (e.g. `controllers`) contains a test folder which has the `unit` tests for that module. One can run the tests for a module by:

```
pytest fitbenchmarking/<MODULE_DIR> --test-type <TEST_TYPE>
```

where <TEST_TYPE> is either `default` or `all`. If `--test-type` argument is not given the default is `all`

System tests

System tests can be found in the `systests` directory in FitBenchmarking. As with the unit tests, these can be run via:

```
pytest fitbenchmarking/systests --test-type <TEST_TYPE>
```

Warning: The files in the expected results subdirectory of the `systests` directory are generated to check consistency in our automated tests via GitHub Actions. They might not pass on your local operating system due to, for example, different software package versions being installed.

GitHub Actions tests

The scripts that are used for our automated tests via GitHub Actions are located in the `ci` folder. These give an example of how to run both the unit and system tests within FitBenchmarking.

Known Issues

This page is used to detail any known issues or unexpected behaviour within the software.

Problem-Format/Software Combinations

When comparing minimizer options from one software package (e.g., comparing all *scipy_ls* minimizers), we are not aware of any issues. However, the general problem of comparing minimizers from multiple software packages, and with different problem-formats, on truly equal terms is harder to achieve.

The following list details all cases where we are aware of a possible bias:

- **Using native FitBenchmarking problems with the Mantid software and fitting using Mantid.**

  With Mantid data, the function evaluation is slightly faster for Mantid minimizers than for all other minimizers. You should account for this when interpreting the results obtained in this case.

- **Using non-scalar ties in native FitBenchmarking problems with the Mantid software.**

  Mantid allows parameters to be tied to expressions - e.g. X0=5.0 or X0=X1*2. While scalar ties are now supported for all minimizers the more complicated expressions are not supported. If you need this feature please get in touch with the development team with your use case.

- **Running Mantid problems with Matlab fitting software.**

  To run problems with Matlab fitting software through FitBenchmarking, within the Matlab Controller the dynamically created *cost_func.eval_model* function is serialized and then loaded in the Matlab Engine workspace. However for Mantid problems, this function is not picklable resulting in the problem being skipped over.

In all cases, the stopping criterion of each minimizer is set to the default value. An experienced user can change this.

Specific Problem/Minimizer Combinations

- **CrystalField Example with Mantid - DampedGaussNewton Minimizer.**

  With this combination, GSL is known to crash during Mantid's fitting. This causes python to exit without completing any remaining runs or generating output files. More information may be available via the issue on Mantid's github page.

1.1.3 Extending FitBenchmarking Documentation

FitBenchmarking is designed to be easily extendable to add new features to the software. Below we outline instructions for doing this.

Adding Fitting Problem Definition Types

The problem definition types we currently support are listed in the page *Problem Definition Files*.

To add a new fitting problem type, the parser name must be derived from the file to be parsed. For current file formats by including it as the first line in the file. e.g `# Fitbenchmark Problem` or `NIST/ITL StRD`, or by checking the file extension.

To add a new fitting problem definition type, complete the following steps:

1. Give the format a name (`<format_name>`). This should be a single word or string of alphanumeric characters, and must be unique ignoring case.
2. Create a parser in the `fitbenchmarking/parsing` directory. This parser must satisfy the following:
   - The filename should be of the form `"<format_name>_parser.py"`
   - The parser must be a subclass of the base parser, *`Parser`*
   - The parser must implement `parse(self)` method which takes only `self` and returns a populated *`FittingProblem`*

   Note: File opening and closing is handled automatically.
3. If the format is unable to accommodate the current convention of starting with the `<format_name>`, you will need to edit *`ParserFactory`*. This should be done in such a way that the type is inferred from the file.
4. Document the parser (see *Problem Definition Files*), being sure to include any licencing information.

5. Create the files to test the new parser. Automated tests are run against the parsers in FitBenchmarking, which work by using test files in `fitbenchmarking/parsing/tests/<format_name>`. In the `test_parsers.generate_test_cases()` function, one needs to add the new parser's name to the variable `formats`, based on whether or not the parser is `pip` installable. There are 2 types of test files needed:

   - **Generic tests**: `fitbenchmarking/parsing/tests/expected/` contains two files, `basic.json` and `start_end_x.json`. You must write two input files in the new file format, which will be parsed using the new parser to check that the entries in the generated fitting problem match the values expected. These must be called `basic.<ext>`, `start_end_x.<ext>`, where `<ext>` is the extension of the new file format, and they must be placed in `fitbenchmarking/parsing/tests/<format_name>/`.
   - **Function tests**: A file named `function_evaluations.json` must also be provided in `fitbenchmarking/parsing/tests/<format_name>/`, which tests that the function evaluation behaves as expected. This file must be in json format and contain a string of the form:

     ```
     {"file_name1": [[[x11,x12,...,x1n], [param11, param12,...,param1m], [result11,
     ↪result12,...,result1n]],
                    [[x21,x22,...,x2n], [param21, param22,...,param2m], [result21,
     ↪result22,...,result2n]],
                    ...],
     {"file_name2": [...],
      ...}
     ```

     The test will then parse the files `file_name<x>` in turn evaluate the function at the given `xx` values and `params`. If the result is not suitably close to the specified value the test will fail.
   - **Jacobian tests**: *If the parser you add has analytic Jacobian information*, then in `test_parsers.py` add `<format_name>` to the `JACOBIAN_ENABLED_PARSERS` global variable. Then add a file `jacobian_evaluations.json` to `fitbenchmarking/parsing/tests/<format_name>/`, which tests that the Jacobian evaluation behaves as expected. This file should have the same file structure as *function_evaluations.json*, and works in a similar way.
   - **Hessian tests**: *If the parser you add has analytic Hessian information*, then in `test_parsers.py` add `<format_name>` to the `HESSIAN_ENABLED_PARSERS` global variable. Then add a file `hessian_evaluations.json` to `fitbenchmarking/parsing/tests/<format_name>/`, which tests that the Hessian evaluation behaves as expected. This file should have the same file structure as *function_evaluations.json*, and works in a similar way.
   - **Integration tests**: Add an example to the directory `fitbenchmarking/mock_problems/all_parser_set/`. This will be used to verify that the problem can be run by scipy, and that accuracy results do not change unexpectedly in future updates. If the software used for the new parser is pip-installable, and the installation is done via FitBenchmarking's `setup.py`, then add the same example to `fitbenchmarking/mock_problems/default_parsers/`.

     As part of this, the `systests/expected_results/all_parsers.txt` file, and if necessary the `systests/expected_results/default_parsers.txt` file, will need to be updated. This is done by running the systests:

     ```
     pytest fitbenchmarking/systests
     ```

     and then checking that the only difference between the results table and the expected value is the new problem, and updating the expected file with the result.

6. Verify that your tests have been found and are successful by running *pytest -vv fitbenchmarking/parsing/tests/test_parsers.py*

Once the new parser is added, please add some examples that use this problem definition type following the instructions at *Adding More Data*.

Adding More Data

We encourage users to contribute more data sets to FitBenchmarking; the more data we have available to test against, the more useful FitBenchmarking is. First, please ensure that there is a *parser* available that can read your dataset, and if necessary follow the *instructions to add this functionality* to FitBenchmarking. Once this is done, follow the steps below and add to a pull request to make the data available to others.

1. Create a directory that contains:
   - the data sets to be included
   - a file *META.txt* containing metadata about the dataset.
   - a subfolder *data_files* which contains any supplemental data

   needed by the data parser. We particularly encourage analytic derivative information, if available.
2. Update the *Benchmark problems* page to include a description of the dataset. As well as information about the source of the data, this should include:
   - information about how many parameters and how many data points

   are to be fitted in the dataset
   - details of any external software that needs to be installed to load these

   datasets.
3. Create *zip* and *tar.gz* archives of these directories, and pass along to one of the core developers to put on the webspace. They will pass you a location of the dataset to include in the description page, and update the folder containing all examples to contain your data set.
4. If the data is to be distributed with the GitHub source, add the directory to the *examples/benchmark_problems* folder and commit to the repository. Please note that the maximum size of a file supported by GitHub is 100MB, and so datasets with files larger than this cannot be added to the repository. Also, GitHub recommends that the size of a Git repository is not more than 1GB.

Adding new cost functions

This section describes how to add cost functions to benchmarking in FitBenchmarking

In order to add a new cost function, <cost_func>, you will need to:

1. Create `fitbenchmarking/cost_func/<cost_func>_cost_func.py`, which contains a new subclass of *CostFunc*. Then implement the methods:
   - **abstract** CostFunc.**eval_cost**()
     Evaluate the cost function
     Parameters **params** (*list*) – The parameters to calculate residuals for
     Returns evaluated cost function
     Return type float
   - **abstract** CostFunc.**jac_res**()
     Uses the Jacobian of the model to evaluate the Jacobian of the cost function residual, $\nabla_p r(x, y, p)$, at the given parameters.
     Parameters **params** (*list*) – The parameters at which to calculate Jacobians
     Returns evaluated Jacobian of the residual at each x, y pair
     Return type a list of 1D numpy arrays
   - **abstract** CostFunc.**jac_cost**()
     Uses the Jacobian of the model to evaluate the Jacobian of the cost function, $\nabla_p F(r(x, y, p))$, at the given parameters.

Parameters **params** (*list*) – The parameters at which to calculate Jacobians
Returns evaluated Jacobian of the cost function
Return type 1D numpy array

- **abstract** CostFunc.**hes_res**()
  Uses the Hessian of the model to evaluate the Hessian of the cost function residual, $\nabla_p^2 r(x, y, p)$, at the given parameters.
  Parameters **params** (*list*) – The parameters at which to calculate Hessians
  Returns evaluated Hessian and Jacobian of the residual at
  each x, y pair :rtype: tuple(list of 2D numpy arrays, list of 1D numpy arrays)
- **abstract** CostFunc.**hes_cost**()
  Uses the Hessian of the model to evaluate the Hessian of the cost function, $\nabla_p^2 F(r(x, y, p))$, at the given parameters.
  Parameters **params** (*list*) – The parameters at which to calculate Hessians
  Returns evaluated Hessian of the cost function
  Return type 2D numpy array

2. Document the available cost functions by:

- adding <cost_func> to the `cost_func_type` option in *Fitting Options*.
- updating any example files in the `examples` directory
- adding the new cost function to the *Cost functions* user docs.

3. Create tests for the cost function in `fitbenchmarking/cost_func/tests/test_cost_func.py`.

The `FittingProblem` and `CostFunc` classes

When adding new cost functions, you will find it helpful to make use of the following members of the *FittingProblem* class.

class fitbenchmarking.parsing.fitting_problem.**FittingProblem**(*options*)
Definition of a fitting problem, which will be populated by a parser from a problem definition file.

Onces populated, this should include the data, the function and any other additional requirements from the data.

data_e
numpy array The errors or weights

data_x
numpy array The x-data

data_y
numpy array The y-data

eval_model(*params*, ***kwargs*)
Function evaluation method

Parameters **params** (*list*) – parameter value(s)

Returns data values evaluated from the function of the problem

Return type numpy array

You will also find it useful to implement the subclass members of *CostFunc*, *Jacobian* and *Hessian*.

class fitbenchmarking.cost_func.base_cost_func.**CostFunc**(*problem*)
Base class for the cost functions.

abstract eval_cost(*params*, ***kwargs*)
Evaluate the cost function

Parameters **params** (*list*) – The parameters to calculate residuals for

Returns evaluated cost function

Return type float

abstract hes_cost(*params*, ***kwargs*)
Uses the Hessian of the model to evaluate the Hessian of the cost function, $\nabla_p^2 F(r(x, y, p))$, at the given parameters.

Parameters **params** (*list*) – The parameters at which to calculate Hessians

Returns evaluated Hessian of the cost function

Return type 2D numpy array

abstract hes_res(*params*, ***kwargs*)
Uses the Hessian of the model to evaluate the Hessian of the cost function residual, $\nabla_p^2 r(x, y, p)$, at the given parameters.

Parameters **params** (*list*) – The parameters at which to calculate Hessians

Returns evaluated Hessian and Jacobian of the residual at

each x, y pair :rtype: tuple(list of 2D numpy arrays, list of 1D numpy arrays)

abstract jac_cost(*params*, ***kwargs*)
Uses the Jacobian of the model to evaluate the Jacobian of the cost function, $\nabla_p F(r(x, y, p))$, at the given parameters.

Parameters **params** (*list*) – The parameters at which to calculate Jacobians

Returns evaluated Jacobian of the cost function

Return type 1D numpy array

abstract jac_res(*params*, ***kwargs*)
Uses the Jacobian of the model to evaluate the Jacobian of the cost function residual, $\nabla_p r(x, y, p)$, at the given parameters.

Parameters **params** (*list*) – The parameters at which to calculate Jacobians

Returns evaluated Jacobian of the residual at each x, y pair

Return type a list of 1D numpy arrays

class fitbenchmarking.jacobian.base_jacobian.**Jacobian**(*problem*)
Base class for Jacobian.

abstract eval(*params*, ***kwargs*)
Evaluates Jacobian of the model, $\nabla_p f(x, p)$, at the point given by the parameters.

Parameters **params** (*list*) – The parameter values at which to evaluate the Jacobian

Returns Computed Jacobian

Return type numpy array

class fitbenchmarking.hessian.base_hessian.**Hessian**(*problem*, *jacobian*)
Base class for Hessian.

abstract eval(*params*, ***kwargs*)
Evaluates Hessian of the model

Parameters **params** (*list*) – The parameter values to find the Hessian at

Returns Computed Hessian

Return type 3D numpy array

Adding new Hessians

This section describes how to add further methods to approximate the Hessian within FitBenchmarking

In order to add a new Hessian evaluation method, <hes_method>, you will need to:

1. Create `fitbenchmarking/hessian/<hes_method>_hessian.py`, which contains a new subclass of `hessian`. Then implement the method:

   - **abstract** `Hessian.eval()`
     Evaluates Hessian of the model
     Parameters **params** (*list*) – The parameter values to find the Hessian at
     Returns Computed Hessian
     Return type 3D numpy array

   The method is set sequentially within *loop_over_hessians()* by using the `method` attribute of the class.

2. Enable the new method as an option in *Fitting Options*, following the instructions in *Adding new Options*. Specifically:

   - Amend the `VALID_FITTING` dictionary so that the element associated with the `hes_method` key contains the new <hes_method>.

3. Document the available Hessians by:

- adding to the list of available `method` options under `hes_method` in *Fitting Options*.
- updating any example files in the `examples` directory

4. Create tests for the Hessian evaluation in `fitbenchmarking/hessian/tests/test_hessians.py`.

The `FittingProblem` and `Jacobian`

When adding new Hessian, you will find it helpful to make use of the following members of the *FittingProblem* and subclasses of *Jacobian*:

class fitbenchmarking.parsing.fitting_problem.**FittingProblem**(*options*)
Definition of a fitting problem, which will be populated by a parser from a problem definition file.

Onces populated, this should include the data, the function and any other additional requirements from the data.

data_e
numpy array The errors or weights

data_x
numpy array The x-data

data_y
numpy array The y-data

eval_model(*params*, ***kwargs*)
Function evaluation method

Parameters **params** (*list*) – parameter value(s)

Returns data values evaluated from the function of the problem

Return type numpy array

class fitbenchmarking.jacobian.base_jacobian.**Jacobian**(*problem*)

Base class for Jacobian.

abstract eval(*params*, ***kwargs*)

Evaluates Jacobian of the model, $\nabla_p f(x, p)$, at the point given by the parameters.

Parameters **params** (*list*) – The parameter values at which to evaluate the Jacobian

Returns Computed Jacobian

Return type numpy array

Adding new Jacobians

FitBenchmarking allows the use of custom methods for evaluating the Jacobian of the model. This section describes how to add further methods within FitBenchmarking

In order to add a new Jacobian evaluation method, <jac_method>, you will need to:

1. Create `fitbenchmarking/jacobian/<jac_method>_jacobian.py`, which contains a new subclass of *Jacobian*. Then implement the method:

   - **abstract** Jacobian.**eval**()

     Evaluates Jacobian of the model, $\nabla_p f(x, p)$, at the point given by the parameters.

     Parameters **params** (*list*) – The parameter values at which to evaluate the Jacobian
     Returns Computed Jacobian
     Return type numpy array

   The numerical method is set sequentially within *loop_over_jacobians()* by using the `method` attribute of the class.

2. Enable the new method as an option in *Fitting Options*, following the instructions in *Adding new Options*. Specifically:

   - Amend the `VALID_FITTING` dictionary so that the element associated with the `jac_method` key contains the new <jac_method>.
   - Extend the `VALID_JACOBIAN` dictionary to have a new key <jac_method>, with the associated element being a list of valid options for this Jacobian.
   - Extend the `DEFAULT_JACOBIAN` dictionary to have a new key <jac_method>, with the associated element being a subset of the valid options added in `VALID_JACOBIAN` in the previous step.
   - Amend the file `fitbenchmarking/utils/tests/test_options_jacobian.py` to include tests for the new options.

3. Document the available Jacobians by:

- adding a list of available `method` options to the docs for *Jacobian Options*, and including licencing information, if appropriate.
- updating any example files in the `examples` directory

4. Create tests for the Jacobian evaluation in `fitbenchmarking/jacobian/tests/test_jacobians.py`.

The FittingProblem class

When adding new Jacobian, you will find it helpful to make use of the following members of the *FittingProblem* class:

class fitbenchmarking.parsing.fitting_problem.**FittingProblem**(*options*)
Definition of a fitting problem, which will be populated by a parser from a problem definition file.

Onces populated, this should include the data, the function and any other additional requirements from the data.

data_e
numpy array The errors or weights

data_x
numpy array The x-data

data_y
numpy array The y-data

eval_model(*params*, ***kwargs*)
Function evaluation method

Parameters **params** (*list*) – parameter value(s)

Returns data values evaluated from the function of the problem

Return type numpy array

Adding Fitting Software

Controllers are used to interface FitBenchmarking with the various fitting packages. Controllers are responsible for converting the problem into a format that the fitting software can use, and converting the result back to a standardised format (numpy arrays). As well as this, the controller must be written so that the fitting is separated from the preparation wherever possible in order to give accurate timings for the fitting. Supported controllers are found in `fitbenchmarking/controllers/`.

In order to add a new controller, you will need to:

1. Give the software a name <software_name>. This will be used by users when selecting this software.
2. Create `fitbenchmarking/controllers/<software_name>_controller.py` which contains a new subclass of *Controller*.

Note: Please note that if the fitting package being added uses Matlab, then the new controller should also inherit from the mixin class *MatlabMixin*.

class fitbenchmarking.controllers.matlab_mixin.**MatlabMixin**(*cost_func*)
Mixin class for matlab fitting software controllers

static py_to_mat(*func*, *eng*)
Function that serializes a python function and then loads it into the Matlab engine workspace

---

The new controller should implement four functions, as well as initializing the dictionary `algorithm_check`:

- Controller.**algorithm_check = {'all': [], 'bfgs': [], 'conjugate_gradient': [], 'deriv_free': [], 'gauss_newton': [], 'general': [], 'global_optimization': [], 'levenberg-marquardt': [], 'ls': [], 'simplex': [], 'steepest_descent': [], 'trust_region': []}**
Within the controller class, you must initialize a dictionary, `algorithm_check`, such that the

keys are given by:
- `all` - all minimizers
- `ls` - least-squares fitting algorithms
- `deriv_free` - derivative free algorithms (these are algorithms that cannot use information about derivatives – e.g., the `Simplex` method in `Mantid`)
- `general` - minimizers which solve a generic *min f(x)*
- `simplex` - derivative free simplex based algorithms e.g. Nelder-Mead
- `trust_region` - algorithms which emply a trust region approach
- `levenberg-marquardt` - minimizers that use the Levenberg-Marquardt algorithm
- `gauss_newton` - minimizers that use the Gauss Newton algorithm
- `bfgs` - minimizers that use the BFGS algorithm
- `conjugate_gradient` - Conjugate Gradient algorithms
- `steepest_descent` - Steepest Descent algorithms
- `global_optimization` - Global Optimization algorithms

The **values** of the dictionary are given as a list of minimizers for that specific controller that fit into each of the above categories. See for example the `GSL` controller.

- `Controller.__init__()`
  Initialise anything that is needed specifically for the software, do any work that can be done without knowledge of the minimizer to use, or function to fit, and call `super(<software_name>Controller, self).__init__(problem)` (the base class's `__init__` implementation).
  Parameters **cost_func** (subclass of *CostFunc*) – Cost function object selected from options.

- **abstract** `Controller.setup()`
  Setup the specifics of the fitting.

  Anything needed for "fit" that can only be done after knowing the minimizer to use and the function to fit should be done here. Any variables needed should be saved to self (as class attributes).

  If a solver supports bounded problems, then this is where *value_ranges* should be set up for that specific solver. The default format is a list of tuples containing the lower and upper bounds for each parameter e.g. [(p1_lb, p2_ub), (p2_lb, p2_ub),...]

- **abstract** `Controller.fit()`
  Run the fitting.

  This will be timed so should include only what is needed to fit the data.

- **abstract** `Controller.cleanup()`
  Retrieve the result as a numpy array and store results.

  Convert the fitted parameters into a numpy array, saved to `self.final_params`, and store the error flag as `self.flag`.

  The flag corresponds to the following messages:

By default, a controller does not accept Jacobian or Hessian information. If the controller being added can use hessians and/or jacobians, then the following controller attributes should be set:

- `Controller.jacobian_enabled_solvers = []`
  Within the controller class, you must define the list `jacobian_enabled_solvers` if any of the minimizers for the specific software are able to use jacobian information.
  – `jacobian_enabled_solvers`: a list of minimizers in a specific software that allow Jacobian information to be passed into the fitting algorithm

- `Controller.hessian_enabled_solvers = []`
  Within the controller class, you must define the list `hessian_enabled_solvers` if

any of the minimizers for the specific software are able to use hessian information.
– `hessian_enabled_solvers`: a list of minimizers in a specific
software that allow Hessian information to be passed into the fitting algorithm

3. Add the new software to the default options, following the instructions in *Adding new Options*.

Your new software is now fully hooked in with FitBenchmarking, and you can compare it with the current software. You are encouraged to contribute this to the repository so that other can use this package. To do this need to follow our *Coding Standards* and our *Git Workflow*, and you'll also need to

4. Document the available minimizers (see *Fitting Options*, *Minimizer Options*), including licencing information, if appropriate. Note: make sure that you use `<software_name>` in these places so that the software links in the HTML tables link correctly to the documentation. Add the software to `examples/all_software.ini`.

   You should also ensure that the available minimizers are catagorised correctly in `self.algorithm_check` using the *algorithm type* options. Please refer to the *Optimization Algorithms* page for more information about each algorithm type.

5. Create tests for the software in `fitbenchmarking/controllers/tests/test_controllers.py`. If the package is `pip` installable then add the tests to the `DefaultControllerTests` class and if not add to the `ExternalControllerTests` class. Unless the new controller is more complicated than the currently available controllers, this can be done by following the example of the others.

6. If the software is deterministic, add the software to the regression tests in `fitbenchmarking/systests/test_regression.py`.

7. If *pip* installable add to `install_requires` in `setup.py` and add to the installation step in `.github/workflows/release.yml`. If not, document the installation procedure in *Installing External Software* and update the `FullInstall` Docker Container – the main developers will help you with this.

Note: For ease of maintenance, please add new controllers to a list of software in alphabetical order.

The `FittingProblem`, `CostFunc` and `Jacobian` classes

When adding new minimizers, you will find it helpful to make use of the following members of the *FittingProblem*, subclasses of *CostFunc* and subclasses of *Jacobian* classes:

class fitbenchmarking.parsing.fitting_problem.**FittingProblem**(*options*)
Definition of a fitting problem, which will be populated by a parser from a problem definition file.

Onces populated, this should include the data, the function and any other additional requirements from the data.

data_e
numpy array The errors or weights

data_x
numpy array The x-data

data_y
numpy array The y-data

eval_model(*params*, ***kwargs*)
Function evaluation method

Parameters **params** (*list*) – parameter value(s)

Returns data values evaluated from the function of the problem

Return type numpy array

set_value_ranges(*value_ranges*)
Function to format parameter bounds before passing to controllers, so self.value_ranges is a list of tuples, which contain lower and upper bounds (lb,ub) for each parameter in the problem

Parameters **value_ranges** (*dict*) –

dictionary of bounded parameter names with lower and upper bound values e.g.

`{p1_name: [p1_min, p1_max], ...}`

class fitbenchmarking.cost_func.base_cost_func.**CostFunc**(*problem*)
Base class for the cost functions.

abstract eval_cost(*params*, ***kwargs*)
Evaluate the cost function

Parameters **params** (*list*) – The parameters to calculate residuals for

Returns evaluated cost function

Return type float

class fitbenchmarking.jacobian.base_jacobian.**Jacobian**(*problem*)
Base class for Jacobian.

abstract eval(*params*, ***kwargs*)
Evaluates Jacobian of the model, $\nabla_p f(x, p)$, at the point given by the parameters.

Parameters **params** (*list*) – The parameter values at which to evaluate the Jacobian

Returns Computed Jacobian

Return type numpy array

Adding new Options

Default options are set by the class *Options*, which is defined in the file *fitbenchmarking/utils/options.py*.

The options used can be changed using an *.ini* formatted file (see here). *FitBenchmarking Options* gives examples of how this is currently implemented in FitBenchmarking.

To add a new option to one of the five sections `FITTING`, `MINIMIZERS`, `JACOBIAN`, `PLOTTING` and `LOGGING`, follow the steps below. We'll illustrate the steps using <SECTION>, which could be any of the sections above.

1. Amend the dictionary `DEFAULT_<SECTION>` in *Options* to include any new default options.
2. If the option amended is to be checked for validity, add accepted option values to the `VALID_<SECTION>` dictionary in *Options*.
3. Using the *read_value()* function, add your new option to the class, following the examples already in *Options*. The syntax of this function is:

   Options.**read_value**(*func*, *option*)
   Helper function which loads in the value

   Parameters

   - **func** (*callable*) – configparser function
   - **option** (*str*) – option to be read for file

   Returns value of the option

   Return type list/str/int/bool

4. Add tests in the following way:

- Each of the sections has it's own test file, for example, `test_option_fitting` has tests for the `FITTING` section.
- Add default tests to the class called `<SECTION>OptionTests`.
- Add user defined tests to the class called `User<SECTION>OptionTests`. These should check that the user added option is valid and raise an `OptionsError` if not.

5. Add relevant documentation for the new option in *FitBenchmarking Options*.

Adding new Sections is also possible. To do this you'll need to extend `VALID_SECTIONS` with the new section, and follow the same structure as the other SECTIONS.

Amending FitBenchmarking Outputs

Here we describe how to add ways of displaying results obtained by FitBenchmarking.

Adding further Tables

The tables that are currently supported are listed in *FitBenchmarking Output*. In order to add a new table, you will need to:

1. Give the table a name `<table_name>`. This will be used by users when selecting this output from FitBenchmarking.
2. Create `fitbenchmarking/results_processing/<table_name>_table.py` which contains a new subclass of *Table*. The main functions to change are:

   - **abstract** Table.**get_value**(*result*)
     Gets the main value to be reported in the tables for a given result

     If more than one value is returned please note that the first value will be used in the default colour handling.
     Parameters **result** (FittingResult) – The result to generate the values for.
     Returns The value to convert to a string for the tables
     Return type tuple(float)

   - Table.**display_str**(*value*)
     Converts a value generated by *get_value()* into a string respresentation to be used in the tables. Base class implementation takes the relative and absolute values and uses `self.output_string_type` as a template for the string format. This can be overridden to adequately display the results.
     Parameters **value** (*tuple*) – Relative and absolute values
     Returns string representation of the value for display in the table.
     Return type str

   Additional functions that may need to be overridden are:

   - **static** Table.**get_error_str**(*result*, *error_template='[{}]'*)
     Get the error string for a result based on error_template This can be overridden if tables require different error formatting.
     Parameters **result** (FittingResult) – The result to get the error string for
     Returns A string representation of the error
     Return type str

   - Table.**get_link_str**(*result*)
     Get the link as a string for the result. This can be overridden if tables require different links.
     Parameters **result** (FittingResult) – The result to get the link for
     Returns The link to go to when the cell is selected

Return type string

- **static** Table.**vals_to_colour**(*vals*, *cmap*, *cmap_range*, *log_ulim*)
  Converts an array of values to a list of hexadecimal colour strings using logarithmic sampling from a matplotlib colourmap according to relative value.
  Parameters
  - **vals** (*list[float]*) – values in the range [0, 1] to convert to colour strings
  - **cmap** (*matplotlib colourmap object*) – matplotlib colourmap
  - **cmap_range** (*list[float], 2 elements*) – values in range [0, 1] for colourmap cropping
  - **log_ulim** (*float*) – log10 of worst shading cutoff value

  Returns colours as hex strings for each input value
  Return type list[str]

3. Extend the `table_type` option in `PLOTTING` following the instructions in *Adding new Options*.
4. Document the new table class is by setting the docstring to be the description of the table, and add to *FitBenchmarking Output*.
5. Create tests for the table in `fitbenchmarking/results_processing/tests/test_tables.py`. This is done by generating, ahead of time using the results problems constructed in `fitbenchmarking/results_processing/tests/test_tables.generate_mock_results`, both a HTML and text table output as the expected result and adding the new table name to the global variable `SORTED_TABLE_NAMES`. This will automatically run the comparison tests for the tables.

HTML/CSS Templates

In FitBenchmarking, templates are used to generate all html output files, and can be found in the *fitbenchmarking/templates* directory.

HTML Templates

HTML templates allow for easily adaptable outputs. In the simple case of rearranging the page or making static changes (changes that don't vary with results), this can be done by editing the template, and it will appear in every subsequent HTML page.

For more complicated changes such as adding information that is page dependent, we use jinja. Jinja allows code to be added to templates which can contain conditional blocks, replicated blocks, or substitutions for values from python.

Changes to what information is displayed in the page will usually involve editing the python source code to ensure you pass the appropriate values to jinja. In practice the amount of effort required to do this can range from one line of code to many depending on the object to be added.

CSS Templates

The CSS files contained in the *templates* directory are used to format the HTML pages.

If you wish to change the style of the output results (e.g. to match a website), this is where they can be changed.

1.1.4 FitBenchmarking Contributor Documentation

Thank you for being a contributor to the FitBenchmarking project. Here you will find all you need in order to get started.

Install Instructions for Contributors

Please see below for the recommended install instructions for new contributors:

1. Before installing, it is recommended that you create an empty virtual environment. For more information about installing packages using virtual environments please see here.
2. To ensure that you are working with the latest version of the code, you should install Fitbenchmarking from source. For details on how to do this, please see *Installing from source*.

   Note: The *editable* flag should be used when installing from source, like `pip install -e ...`, so that changes made to the cloned code are immediately reflected in the imported package.

   ---

3. So that you are able to run all tests locally, all extra dependencies and external software should be installed. Please see *Extra dependencies* and *Installing External Software* for instructions on how to do this.

   Note: Please note that if you are using WSL, Matlab is likely to be difficult to install so it is recommended that you use an alternative operating system if you are working on Matlab related issues.

   Additionally, the tests run through Github do not currently include Matlab fitting software, so please ensure tests are run locally before submitting code.

   ---

4. You should check that the versions of packages such as pylint are up to date with those used listed in requirements.txt. This can be done by running the command `pip install -r requirements.txt` from within the `fitbenchmarking` directory.

Coding Standards

All code submitted must meet certain standards, outlined below, before it can be merged into the master branch. It is the contributor's job to ensure that the following is satisfied, and the reviewer's role to check that these guidelines have been followed.

The workflow to be used for submitting new code/issues is described in *Git Workflow*.

Linting

All pull requests should be PEP 8 compliant. We suggest running code through flake8 and pylint before submitting to check for this.

Documentation

Any new code will be accepted only if the documentation, written in sphinx and found in *docs/*, has been updated accordingly, and the docstrings in the code have been updated where necessary.

Testing

All tests should pass before submitting code. Tests are written using pytest.

The following should be checked before any code is merged:

- Function: Does the change do what it's supposed to?
- Tests: Does it pass? Is there adequate coverage for new code?
- Style: Is the coding style consistent? Is anything overly confusing?
- Documentation: Is there a suitable change to documentation for this change?

Logging

Code should use the logging in `utils.log`. This uses Python's built in logging module, and should be used in place of any print statements to ensure that persistent logs are kept after runs.

Git Workflow

Issues

All new work should start with a new GitHub issue being filed. This should clearly explain what the change to the code will do. There are templates for *Bug report*, *Documentation*, *Feature request* and *Test* issues on GitHub, and you can also open a blank issue if none of these work.

If issues help meet a piece of work agreed with our funders, it is linked to the appropriate Milestone in GitHub.

Adding new code

The first step in adding new code is to create a branch, where the work will be done. Branches should be named according to the convention *<nnn>-description_of_work*, where *<nnn>* is the issue number.

Please ensure our *Coding Standards* are adhered to throughout the branch.

When you think your new code is ready to be merged into the codebase, you should open a pull request (PR) to master. The description should contain the words *Fixes #<nnn>*, where *<nnn>* is the issue number; this will ensure the issue is closed when the code is merged into master. At this point the automated tests will trigger, and you can see if the code passes on an independent system.

Sometimes it is desirable to open a PR when the code is not quite ready to be merged. This is a good idea, for example, if you want to get an early opinion on a coding decision. If this is the case, you should mark the PR as a *draft* on GitHub.

Once the work is ready to be reviewed, you may want to assign a reviewer, if you think someone would be well suited to review this change. It is worth messaging them on, for example, Slack, as well as requesting their review on GitHub.

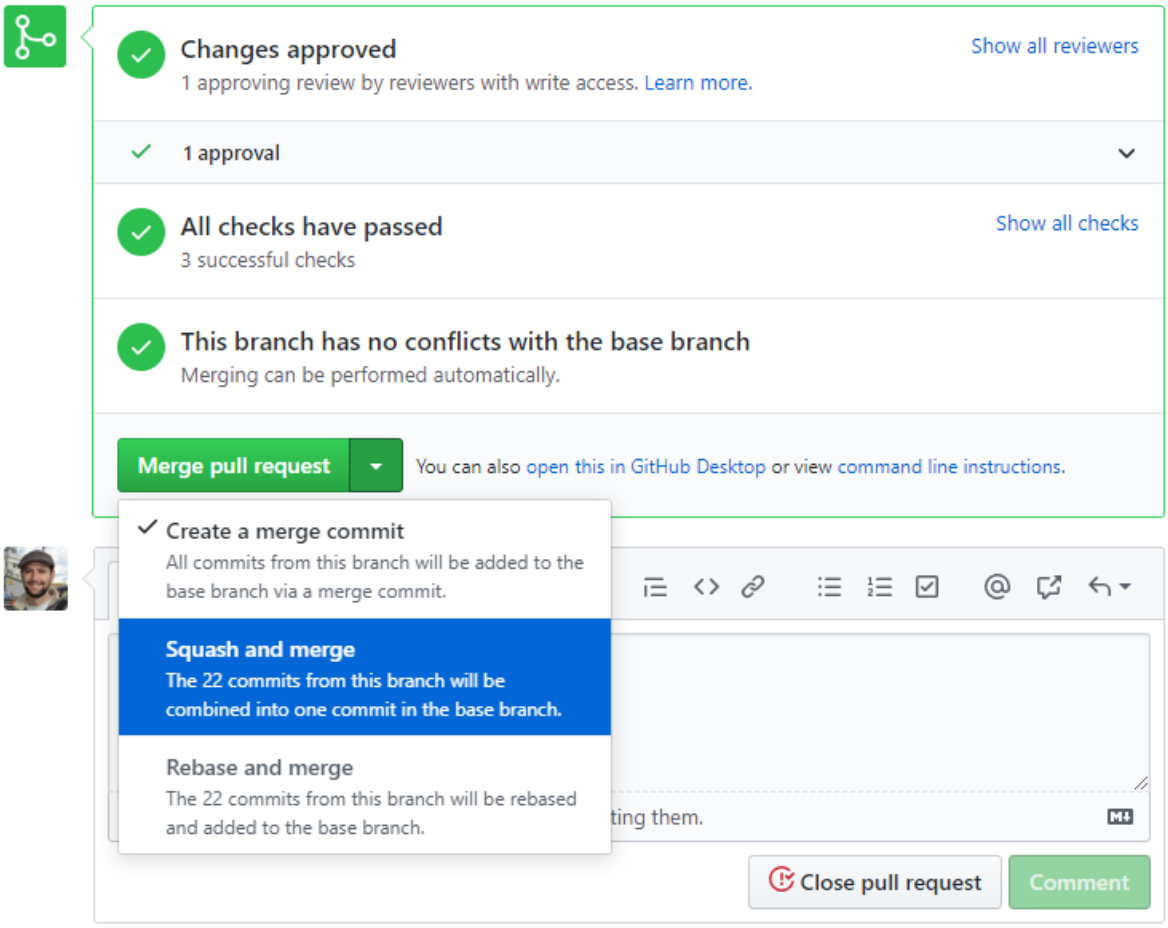
Release branches

Branches named *release-** are protected branches; code must be approved by a reviewer before being added to them, and automated tests will be run on PRs to these branches. If code is to be included in the release, it must be pulled into this branch from master.

Release branches should have the format *release-major.minor.x*, starting from *release-0.1.x*. When the code is released, we will tag that commit with a version number *v0.1.0*. Any hotfixes will increment *x* by one, and a new tag will be created accordingly. If at some point we don't want to provide hot-fixes to a given minor release, then the corresponding release branch may be deleted.

All changes must be initially merged into master. There is a *backport-candidate* label, which must be put on PRs that in addition must be merged into the release branch.

The recommended mechanism for merging PRs labeled with *backport-candidate* into master is to use the *Squash and merge* commit option:



After such a PR (with label *backport-candidate*) has been merged into master, it must then subsequently be merged into the release branch as soon as possible. It is the responsibility of the person merging such PRs to also perform this merge into the release branch.

This can be done using git cherry pick:

```
git checkout release-x.x.x
git cherry-pick -x <commit-id>
git push
```

If you didn't do a squash merge, you will have to cherry pick each commit in the PR that this being backported separately.

If you encounter problems with cherry picking into release branch please don't hesitate to speak to an experienced member of the FitBenchmarking team.

Creating a release

In order to create a new release for FitBenchmarking, there are a few manual steps. These have been streamlined as much as possible.

First checkout the branch to create the release from. Releases should only be made from a *release-x-x* branch, not a development branch or master.

From the root of the repo run the "ci/prep_and_tag_release.sh" script with the new version number. The version number will be rejected if it is not of the expected form. We expect a "v" followed by the major, minor, and patch numbers, and an optional numbered label to mark the type of release.

Possible labels are:

- -beta (release for testing)
- -rc (release candidate)

This script will create a new commit with the docs and testing links updated, tag it, and revert the change in a second commit so that the links point back to the latest versions.

These commits will need to be pushed to github.

Finally, you will need to create a release on github. This can be done by navigating to the releases page, selecting new release and typing in the tag that was given to the release (it should tell you the tag exists at this point!).

For example, For a first beta version of release 0.1.0, one would run:

```
git checkout release-0.1.x
ci/prep_and_tag_release.sh v0.1.0-beta1
git push origin release-0.1.x

<And make the release on GitHub>
```

Adding New Datasets

Users or developers are encouraged to add new data sets following the instructions *here*. Someone in SCD's Computational Mathematics Group must make this publically available by:

- adding the *zip* and *tar.gz* archives to *powell:/var/www/html/numerical-www/fitbenchmarking/*
- adding the datasets to the master *examples.zip* and *examples.tar.gz* folders, and updating the versions on *powell*

Please note that the maximum file size allowed by GitHub is 100MB, and the total repository size is recommended to be kept below 1GB. Please bear this in mind when advising users whether or not they should also add their data to the *examples/benchmark_problems* directory of FitBenchmarking.

Repository Structure

At the root of the repository there are six directories:

- build
- ci
- Docker
- docs
- examples
- fitbenchmarking

Build (`build`)

This directory contains scripts to allow for installing packages such as Mantid through setuptools.

CI (`ci`)

We use GitHub Actions to run our Continuous Integration tests. The specific tests run are defined in a series of Bash scripts, which are stored in this folder.

Docker (`Docker`)

The continuous integration process on Github Actions currently run on a Docker container, and this directory holds the Dockerfiles. The Docker containers are hosted on Dockerhub.

`BasicInstall` holds the Dockerfile that is pushed to the repository `fitbenchmarking/fitbenchmarking-deps`, the latest of which should have the tag `latest`. This contains a basic Ubuntu install, with just the minimal infrastructure needed to run the tests.

`FullInstall` holds the Dockerfile that is pushed to the repository `fitbenchmarking/fitbenchmarking-extras`, the latest of which should have the tag `latest`. This is built on top of the basic container, and includes optional third party software that FitBenchmarking can work with.

The versions on Docker Hub can be updated from a connected account by issuing the commands:

```
docker build --tag fitbenchmarking-<type>:<tag>
docker tag fitbenchmarking-<type>:<tag> fitbenchmarking/fitbenchmarking-<type>:<tag>
docker push fitbenchmarking/fitbenchmarking-<type>:<tag>
```

where <type> is, e.g., `deps` or `extras`, and <tag> is, e.g., `latest`.

Documentation (`docs`)

The documentation for FitBenchmarking is stored in this folder under `source`. A local copy of the documentation can be build using `make html` in the `build` directory.

Examples (`examples`)

Examples is used to store sample problem files and options files.

A collection of problem files can be found organised into datasets within the `examples/benchmark_problems/` directory.

An options template file and a prewritten options file to run a full set of minimizers is also made available in the `examples/` directory.

FitBenchmarking Package (`fitbenchmarking`)

The main FitBenchmarking package is split across several directories with the intention that it is easily extensible. The majority of these directories are source code, with exceptions being Templates, Mock Problems, and System Tests.

Each file that contains source code will have a directory inside it called `tests`, which contains all of the tests for that section of the code.

Benchmark Problems (`benchmark_problems`)

This is a copy of the NIST benchmark problems from *examples/benchmark_problems*. These are the default problems that are run if no problem is passed to the `fitbenchmarking` command, and is copied here so that it is distributed with the package when installed using, say, *pip*.

CLI (`cli`)

The CLI directory is used to define all of the entry points into the software. Currently this is just the main *fitbenchmarking* command.

Controllers (`controllers`)

In FitBenchmarking, controllers are used to interface with third party minimizers.

The controllers directory holds a base controller class (*`Controller`*) and all its subclasses, each of which of which interfaces with a different fitting package. The controllers currently implemented are described in *Fitting Options* and *Minimizer Options*.

New controllers can be added by following the instructions in *Adding Fitting Software*.

Core (`core`)

This directory holds all code central to FitBenchmarking. For example, this manages calling the correct parser and controller, as well as compiling the results into a data object.

Hessian (`Hessian`)

This directory holds the `Hessian` class, and subclasses, which are used by the controllers to approximate second derivatives. Currently available options are described in *Fitting Options*, and new Hessians can be added by following the instructions in *Adding new Hessians*.

Jacobian (`jacobian`)

This directory holds the `Jacobian` class, and subclasses, which are used by the controllers to approximate derivatives. Currently available options are described in *Jacobian Options*, and new numerical Jacobians can be added by following the instructions in *Adding new Jacobians*.

Mock Problems (`mock_problems`)

The mock problems are used in some tests where full problem files are required. These are here so that the examples can be moved without breaking the tests.

Parsing (`parsing`)

The parsers read raw data into a format that FitBenchmarking can use. This directory holds a base parser, `Parser` and all its subclasses. Each subclass implements a parser for a specific file format. Information about existing parsers can be found in *Problem Definition Files*, and see *Adding Fitting Problem Definition Types* for instructions on extending these.

Results Processing (`results_processing`)

All files that are used to generate output are stored here. This includes index pages, text/html tables, plots, and support pages. Information about the tables we provide can be found in *FitBenchmarking Output*, and instructions on how to add further tables and change the formatting of the displayed information can be found in *Amending FitBenchmarking Outputs*.

System Tests (`systests`)

FitBenchmarking runs regression tests to check that the accuracy results do not change with updates to the code. These tests run fitbenchmarking against a subset of problems (in subdirectories of */fitbenchmarking/mock_problems/*), and compares the text output with that stored in */fitbenchmarking/systests/expected_results/*.

Templates (`templates`)

Files in Templates are used to create the resulting html pages, and are a combination of css, html, and python files. The python files in this directory are scripts to update the css and html assets. Instructions on updating these can be found in *HTML/CSS Templates*.

Utils (`utils`)

This directory contains utility functions that do not fit into the above sections. This includes the `Options` class (see *Adding new Options* to extend) and `FittingResult` class, as well as functions for logging and directory creation.

fitbenchmarking package

Subpackages

fitbenchmarking.cli package

Submodules

fitbenchmarking.cli.exception_handler module

This file holds an exception handler decorator that should wrap all cli functions to provide cleaner output.

`fitbenchmarking.cli.exception_handler.`**`exception_handler`**`(f)`
Decorator to simplify handling exceptions within FitBenchmarking This will strip off any 'debug' inputs.

Parameters **f** (*python function*) – The function to wrap

fitbenchmarking.cli.main module

This is the main entry point into the FitBenchmarking software package. For more information on usage type fitbenchmarking –help or for more general information, see the online docs at docs.fitbenchmarking.com.

`fitbenchmarking.cli.main.`**`get_parser`**`()`
Creates and returns a parser for the args.

Returns configured argument parser

Return type argparse.ArgParser

`fitbenchmarking.cli.main.`**`main`**`()`
Entry point to be exposed as the *fitbenchmarking* command.

`fitbenchmarking.cli.main.`**`run`**`(problem_sets, results_directory, options_file='', debug=False)`
Run benchmarking for the problems sets and options file given. Opens a webbrowser to the results_index after fitting.

Parameters

- **problem_sets** (*list of str*) – The paths to directories containing problem_sets
- **results_directory** (*str*) – The directory to store the resulting files in
- **options_file** (*str, optional*) – The path to an options file, defaults to ''
- **debug** (*bool*) – Enable debugging output

Module contents

fitbenchmarking.controllers package

Submodules

fitbenchmarking.controllers.base_controller module

Implements the base class for the fitting software controllers.

class fitbenchmarking.controllers.base_controller.**Controller**(*cost_func*)
Bases: object

Base class for all fitting software controllers. These controllers are intended to be the only interface into the fitting software, and should do so by implementing the abstract classes defined here.

VALID_FLAGS = [0, 1, 2, 3, 4, 5, 6, 7]

algorithm_check = {'all': [], 'bfgs': [], 'conjugate_gradient': [], 'deriv_free': [], 'gauss_newton': [], 'general': [], 'global_optimization': [], 'levenberg-marquardt': [], 'ls': [], 'simplex': [], 'steepest_descent': [], 'trust_region': []}

Within the controller class, you must initialize a dictionary, `algorithm_check`, such that the **keys** are given by:

- `all` - all minimizers
- `ls` - least-squares fitting algorithms
- `deriv_free` - derivative free algorithms (these are algorithms that cannot use information about derivatives – e.g., the `Simplex` method in `Mantid`)
- `general` - minimizers which solve a generic *min f(x)*
- `simplex` - derivative free simplex based algorithms e.g. Nelder-Mead
- `trust_region` - algorithms which emply a trust region approach
- `levenberg-marquardt` - minimizers that use the Levenberg-Marquardt algorithm
- `gauss_newton` - minimizers that use the Gauss Newton algorithm
- `bfgs` - minimizers that use the BFGS algorithm
- `conjugate_gradient` - Conjugate Gradient algorithms
- `steepest_descent` - Steepest Descent algorithms
- `global_optimization` - Global Optimization algorithms

The **values** of the dictionary are given as a list of minimizers for that specific controller that fit into each of the above categories. See for example the `GSL` controller.

check_attributes()
A helper function which checks all required attributes are set in software controllers

check_bounds_respected()
Check whether the selected minimizer has respected parameter bounds

check_minimizer_bounds(*minimizer*)
Helper function which checks whether the selected minimizer from the options (options.minimizer) supports problems with parameter bounds

Parameters **minimizer** (*str*) – string of minimizers selected from the options

abstract cleanup()
Retrieve the result as a numpy array and store results.

Convert the fitted parameters into a numpy array, saved to `self.final_params`, and store the error flag as `self.flag`.

The flag corresponds to the following messages:

controller_name = None
A name to be used in tables. If this is set to None it will be inferred from the class name.

eval_chisq(*params*, *x=None*, *y=None*, *e=None*)
Computes the chisq value

Parameters

- **params** (*list*) – The parameters to calculate residuals for
- **x** (*numpy array, optional*) – x data points, defaults to self.data_x
- **y** (*numpy array, optional*) – y data points, defaults to self.data_y
- **e** (*numpy array, optional*) – error at each data point, defaults to self.data_e

Returns The sum of squares of residuals for the datapoints at the given parameters

Return type numpy array

execute()
Starts and stops the timer used to check if the fit reaches the 'max_runtime'. In the middle, it calls self.fit().

abstract fit()
Run the fitting.

This will be timed so should include only what is needed to fit the data.

property flag

0: *Successfully converged*
1: *Software reported maximum number of iterations exceeded*
2: *Software run but didn't converge to solution*
3: *Software raised an exception*
4: *Solver doesn't support bounded problems*
5: *Solution doesn't respect parameter bounds*
6: *Solver has exceeded maximum allowed runtime*
7: *Validation of the provided options failed*

hessian_enabled_solvers = []
Within the controller class, you must define the list `hessian_enabled_solvers` if any of the minimizers for the specific software are able to use hessian information.

- `hessian_enabled_solvers`: a list of minimizers in a specific

software that allow Hessian information to be passed into the fitting algorithm

incompatible_problems = []
A list of incompatible problem formats for this controller.

jacobian_enabled_solvers = []
Within the controller class, you must define the list `jacobian_enabled_solvers` if any of the minimizers for the specific software are able to use jacobian information.

- `jacobian_enabled_solvers`: a list of minimizers in a specific

software that allow Jacobian information to be passed into the fitting algorithm

prepare()
Check that function and minimizer have been set. If both have been set, run self.setup().

record_alg_type(*minimizer*, *algorithm_type*)
Helper function which records the algorithm types of the selected minimizer that match those chosen in options

Parameters

- **minimizer** (*str*) – string of minimizers selected from the options
- **algorithm_type** (*list*) – the algorithm type selected from the options

abstract setup()
Setup the specifics of the fitting.

Anything needed for "fit" that can only be done after knowing the minimizer to use and the function to fit should be done here. Any variables needed should be saved to self (as class attributes).

If a solver supports bounded problems, then this is where *value_ranges* should be set up for that specific solver. The default format is a list of tuples containing the lower and upper bounds for each parameter e.g. [(p1_lb, p2_ub), (p2_lb, p2_ub),...]

property software
Return the name of the software.

This assumes the class is named '<software>Controller'

validate() → None
Validates that the provided options are compatible with each other. If there are some invalid options, the relevant exception is raised.

validate_minimizer(*minimizer*, *algorithm_type*)
Helper function which checks that the selected minimizer from the options (options.minimizer) exists and whether the minimizer is in self.algorithm_check[options.algorithm_type] (this is a list set in the controller)

Parameters

- **minimizer** (*str*) – string of minimizers selected from the options
- **algorithm_type** (*list*) – the algorithm type selected from the options

fitbenchmarking.controllers.bumps_controller module

Implements a controller for the Bumps fitting software.

class fitbenchmarking.controllers.bumps_controller.**BumpsController**(*cost_func*)
Bases: *fitbenchmarking.controllers.base_controller.Controller*

Controller for the Bumps fitting software.

Sasview requires a model to fit. Setup creates a model with the correct function.

algorithm_check = {'all': ['amoeba', 'lm-bumps', 'newton', 'de', 'mp'], 'bfgs': ['newton'], 'conjugate_gradient': [], 'deriv_free': ['amoeba', 'de'], 'gauss_newton': [], 'general': ['amoeba', 'newton', 'de'], 'global_optimization': ['de'], 'levenberg-marquardt': ['lm-bumps'], 'ls': ['lm-bumps', 'mp'], 'simplex': ['amoeba'], 'steepest_descent': [], 'trust_region': ['lm-bumps']}

Within the controller class, you must initialize a dictionary, `algorithm_check`, such that the **keys** are given by:

- `all` - all minimizers
- `ls` - least-squares fitting algorithms
- `deriv_free` - derivative free algorithms (these are algorithms that cannot use information about derivatives – e.g., the `Simplex` method in `Mantid`)
- `general` - minimizers which solve a generic *min f(x)*
- `simplex` - derivative free simplex based algorithms e.g. Nelder-Mead
- `trust_region` - algorithms which emply a trust region approach
- `levenberg-marquardt` - minimizers that use the Levenberg-Marquardt algorithm
- `gauss_newton` - minimizers that use the Gauss Newton algorithm
- `bfgs` - minimizers that use the BFGS algorithm
- `conjugate_gradient` - Conjugate Gradient algorithms
- `steepest_descent` - Steepest Descent algorithms
- `global_optimization` - Global Optimization algorithms

The **values** of the dictionary are given as a list of minimizers for that specific controller that fit into each of the above categories. See for example the `GSL` controller.

cleanup()

Convert the result to a numpy array and populate the variables results will be read from.

fit()

Run problem with Bumps.

setup()

Setup problem ready to run with Bumps.

Creates a FitProblem for calling in the fit() function of Bumps

fitbenchmarking.controllers.controller_factory module

This file contains a factory implementation for the controllers. This is used to manage the imports and reduce effort in adding new controllers.

class fitbenchmarking.controllers.controller_factory.**ControllerFactory**

Bases: `object`

A factory for creating software controllers. This has the capability to select the correct controller, import it, and generate an instance of it. Controllers generated from this must be a subclass of base_controller.Controller

static create_controller(*software*)

Create a controller that matches the required software.

Parameters **software** (*string*) – The name of the software to create a controller for

Returns Controller class for the problem

Return type fitbenchmarking.fitting.base_controller.Controller subclass

fitbenchmarking.controllers.dfo_controller module

Implements a controller for DFO-GN http://people.maths.ox.ac.uk/robertsl/dfogn/

class fitbenchmarking.controllers.dfo_controller.**DFOController**(*cost_func*)

Bases: *fitbenchmarking.controllers.base_controller.Controller*

Controller for the DFO-{GN/LS} fitting software.

algorithm_check = {'all': ['dfogn', 'dfols'], 'bfgs': [], 'conjugate_gradient': [], 'deriv_free': ['dfogn', 'dfols'], 'gauss_newton': ['dfogn'], 'general': [], 'global_optimization': [], 'levenberg-marquardt': [], 'ls': ['dfogn', 'dfols'], 'simplex': [], 'steepest_descent': [], 'trust_region': ['dfols', 'dfogn']}

Within the controller class, you must initialize a dictionary, `algorithm_check`, such that the **keys** are given by:

- `all` - all minimizers
- `ls` - least-squares fitting algorithms
- `deriv_free` - derivative free algorithms (these are algorithms that cannot use information about derivatives – e.g., the `Simplex` method in `Mantid`)
- `general` - minimizers which solve a generic *min f(x)*
- `simplex` - derivative free simplex based algorithms e.g. Nelder-Mead
- `trust_region` - algorithms which emply a trust region approach
- `levenberg-marquardt` - minimizers that use the Levenberg-Marquardt algorithm
- `gauss_newton` - minimizers that use the Gauss Newton algorithm
- `bfgs` - minimizers that use the BFGS algorithm
- `conjugate_gradient` - Conjugate Gradient algorithms
- `steepest_descent` - Steepest Descent algorithms
- `global_optimization` - Global Optimization algorithms

The **values** of the dictionary are given as a list of minimizers for that specific controller that fit into each of the above categories. See for example the `GSL` controller.

cleanup()

Convert the result to a numpy array and populate the variables results will be read from.

fit()

Run problem with DFO.

setup()

Setup for DFO

fitbenchmarking.controllers.gradient_free_controller module

Implements a controller for Gradient Free Optimizers

class fitbenchmarking.controllers.gradient_free_controller.**GradientFreeController**(*cost_func*)
Bases: *fitbenchmarking.controllers.base_controller.Controller*

Controller for the Gradient Free Optimizers fitting software.

algorithm_check = {'all': ['HillClimbingOptimizer', 'RepulsingHillClimbingOptimizer', 'SimulatedAnnealingOptimizer', 'RandomSearchOptimizer', 'RandomRestartHillClimbingOptimizer', 'RandomAnnealingOptimizer', 'ParallelTemperingOptimizer', 'ParticleSwarmOptimizer', 'EvolutionStrategyOptimizer', 'BayesianOptimizer', 'TreeStructuredParzenEstimators', 'DecisionTreeOptimizer'], 'bfgs': [], 'conjugate_gradient': [], 'deriv_free': ['HillClimbingOptimizer', 'RepulsingHillClimbingOptimizer', 'SimulatedAnnealingOptimizer', 'RandomSearchOptimizer', 'RandomRestartHillClimbingOptimizer', 'RandomAnnealingOptimizer', 'ParallelTemperingOptimizer', 'ParticleSwarmOptimizer', 'EvolutionStrategyOptimizer', 'BayesianOptimizer', 'TreeStructuredParzenEstimators', 'DecisionTreeOptimizer'], 'gauss_newton': [], 'general': ['HillClimbingOptimizer', 'RepulsingHillClimbingOptimizer', 'SimulatedAnnealingOptimizer', 'RandomSearchOptimizer', 'RandomRestartHillClimbingOptimizer', 'RandomAnnealingOptimizer', 'ParallelTemperingOptimizer', 'ParticleSwarmOptimizer', 'EvolutionStrategyOptimizer', 'BayesianOptimizer', 'TreeStructuredParzenEstimators', 'DecisionTreeOptimizer'], 'global_optimization': ['HillClimbingOptimizer', 'RepulsingHillClimbingOptimizer', 'SimulatedAnnealingOptimizer', 'RandomSearchOptimizer', 'RandomRestartHillClimbingOptimizer', 'RandomAnnealingOptimizer', 'ParallelTemperingOptimizer', 'ParticleSwarmOptimizer', 'EvolutionStrategyOptimizer', 'BayesianOptimizer', 'TreeStructuredParzenEstimators', 'DecisionTreeOptimizer'], 'levenberg-marquardt': [], 'ls': [], 'simplex': [], 'steepest_descent': [], 'trust_region': []}

Within the controller class, you must initialize a dictionary, `algorithm_check`, such that the **keys** are given by:

- `all` - all minimizers
- `ls` - least-squares fitting algorithms
- `deriv_free` - derivative free algorithms (these are algorithms that cannot use information about derivatives – e.g., the `Simplex` method in `Mantid`)
- `general` - minimizers which solve a generic *min f(x)*
- `simplex` - derivative free simplex based algorithms e.g. Nelder-Mead
- `trust_region` - algorithms which emply a trust region approach
- `levenberg-marquardt` - minimizers that use the Levenberg-Marquardt algorithm
- `gauss_newton` - minimizers that use the Gauss Newton algorithm
- `bfgs` - minimizers that use the BFGS algorithm
- `conjugate_gradient` - Conjugate Gradient algorithms
- `steepest_descent` - Steepest Descent algorithms
- `global_optimization` - Global Optimization algorithms

The **values** of the dictionary are given as a list of minimizers for that specific controller that fit into each of the above categories. See for example the `GSL` controller.

cleanup()
Convert the result to a numpy array and populate the variables results will be read from.

controller_name = 'gradient_free'
A name to be used in tables. If this is set to None it will be inferred from the class name.

fit()
Run problem with Gradient Free Optimizers

setup()
Setup for Gradient Free Optimizers

fitbenchmarking.controllers.gsl_controller module

Implements a controller for GSL https://www.gnu.org/software/gsl/ using the pyGSL python interface https://sourceforge.net/projects/pygsl/

class fitbenchmarking.controllers.gsl_controller.**GSLController**(*cost_func*)
Bases: *fitbenchmarking.controllers.base_controller.Controller*

Controller for the GSL fitting software

algorithm_check = {'all': ['lmsder', 'lmder', 'nmsimplex', 'nmsimplex2', 'conjugate_pr', 'conjugate_fr', 'vector_bfgs', 'vector_bfgs2', 'steepest_descent'], 'bfgs': ['vector_bfgs', 'vector_bfgs2'], 'conjugate_gradient': ['conjugate_fr', 'conjugate_pr'], 'deriv_free': ['nmsimplex', 'nmsimplex2'], 'gauss_newton': [], 'general': ['nmsimplex', 'nmsimplex2', 'conjugate_pr', 'conjugate_fr', 'vector_bfgs', 'vector_bfgs2', 'steepest_descent'], 'global_optimization': [], 'levenberg-marquardt': ['lmder', 'lmsder'], 'ls': ['lmsder', 'lmder'], 'simplex': ['nmsimplex', 'nmsimplex2'], 'steepest_descent': ['steepest_descent'], 'trust_region': ['lmder', 'lmsder']}
Within the controller class, you must initialize a dictionary, `algorithm_check`, such that the **keys** are given by:

- `all` - all minimizers
- `ls` - least-squares fitting algorithms
- `deriv_free` - derivative free algorithms (these are algorithms that cannot use information about derivatives – e.g., the `Simplex` method in `Mantid`)
- `general` - minimizers which solve a generic *min f(x)*
- `simplex` - derivative free simplex based algorithms e.g. Nelder-Mead
- `trust_region` - algorithms which emply a trust region approach
- `levenberg-marquardt` - minimizers that use the Levenberg-Marquardt algorithm
- `gauss_newton` - minimizers that use the Gauss Newton algorithm
- `bfgs` - minimizers that use the BFGS algorithm
- `conjugate_gradient` - Conjugate Gradient algorithms
- `steepest_descent` - Steepest Descent algorithms
- `global_optimization` - Global Optimization algorithms

The **values** of the dictionary are given as a list of minimizers for that specific controller that fit into each of the above categories. See for example the `GSL` controller.

cleanup()
Convert the result to a numpy array and populate the variables results will be read from

fit()
Run problem with GSL

jacobian_enabled_solvers = ['lmsder', 'lmder', 'conjugate_pr', 'conjugate_fr', 'vector_bfgs', 'vector_bfgs2', 'steepest_descent']
Within the controller class, you must define the list `jacobian_enabled_solvers` if any of the minimizers for the specific software are able to use jacobian information.

- `jacobian_enabled_solvers`: a list of minimizers in a specific

software that allow Jacobian information to be passed into the fitting algorithm

setup()
Setup for GSL

fitbenchmarking.controllers.levmar_controller module

Implements a controller for the levmar fitting software. http://www.ics.forth.gr/~lourakis/levmar/ via the python interface https://pypi.org/project/levmar/

class fitbenchmarking.controllers.levmar_controller.**LevmarController**(*cost_func*)
Bases: *fitbenchmarking.controllers.base_controller.Controller*

Controller for the levmar fitting software

algorithm_check = {'all': ['levmar'], 'bfgs': [], 'conjugate_gradient': [], 'deriv_free': [], 'gauss_newton': [], 'general': [], 'global_optimization': [], 'levenberg-marquardt': ['levmar'], 'ls': ['levmar'], 'simplex': [], 'steepest_descent': [], 'trust_region': ['levmar']}
Within the controller class, you must initialize a dictionary, `algorithm_check`, such that the **keys** are given by:

- `all` - all minimizers
- `ls` - least-squares fitting algorithms
- `deriv_free` - derivative free algorithms (these are algorithms that cannot use information about derivatives – e.g., the `Simplex` method in `Mantid`)
- `general` - minimizers which solve a generic *min f(x)*
- `simplex` - derivative free simplex based algorithms e.g. Nelder-Mead
- `trust_region` - algorithms which emply a trust region approach
- `levenberg-marquardt` - minimizers that use the Levenberg-Marquardt algorithm
- `gauss_newton` - minimizers that use the Gauss Newton algorithm
- `bfgs` - minimizers that use the BFGS algorithm
- `conjugate_gradient` - Conjugate Gradient algorithms
- `steepest_descent` - Steepest Descent algorithms
- `global_optimization` - Global Optimization algorithms

The **values** of the dictionary are given as a list of minimizers for that specific controller that fit into each of the above categories. See for example the `GSL` controller.

cleanup()
Convert the result to a numpy array and populate the variables results will be read from.

fit()
run problem with levmar

jacobian_enabled_solvers = ['levmar']
Within the controller class, you must define the list `jacobian_enabled_solvers` if any of the minimizers for the specific software are able to use jacobian information.

- `jacobian_enabled_solvers`: a list of minimizers in a specific

software that allow Jacobian information to be passed into the fitting algorithm

setup()
Setup problem ready to be run with levmar

fitbenchmarking.controllers.mantid_controller module

Implements a controller for the Mantid fitting software.

class fitbenchmarking.controllers.mantid_controller.**MantidController**(*cost_func*)
Bases: *fitbenchmarking.controllers.base_controller.Controller*

Controller for the Mantid fitting software.

Mantid requires subscribing a custom function in a predefined format, so this controller creates that in setup.

COST_FUNCTION_MAP = {'nlls': 'Unweighted least squares', 'poisson': 'Poisson', 'weighted_nlls': 'Least squares'}
A map from fitbenchmarking cost functions to mantid ones.

algorithm_check = {'all': ['BFGS', 'Conjugate gradient (Fletcher-Reeves imp.)', 'Conjugate gradient (Polak-Ribiere imp.)', 'Damped GaussNewton', 'Levenberg-Marquardt', 'Levenberg-MarquardtMD', 'Simplex', 'SteepestDescent', 'Trust Region', 'FABADA'], 'bfgs': ['BFGS'], 'conjugate_gradient': ['Conjugate gradient (Fletcher-Reeves imp.)', 'Conjugate gradient (Polak-Ribiere imp.)'], 'deriv_free': ['Simplex', 'FABADA'], 'gauss_newton': ['Damped GaussNewton'], 'general': ['BFGS', 'Conjugate gradient (Fletcher-Reeves imp.)', 'Conjugate gradient (Polak-Ribiere imp.)', 'Damped GaussNewton', 'Simplex', 'SteepestDescent'], 'global_optimization': ['FABADA'], 'levenberg-marquardt': ['Levenberg-Marquardt', 'Levenberg-MarquardtMD'], 'ls': ['Levenberg-Marquardt', 'Levenberg-MarquardtMD', 'Trust Region', 'FABADA'], 'simplex': ['Simplex'], 'steepest_descent': ['SteepestDescent'], 'trust_region': ['Trust Region', 'Levenberg-Marquardt', 'Levenberg-MarquardtMD']}
Within the controller class, you must initialize a dictionary, `algorithm_check`, such that the **keys** are given by:

- `all` - all minimizers
- `ls` - least-squares fitting algorithms
- `deriv_free` - derivative free algorithms (these are algorithms that cannot use information about derivatives – e.g., the `Simplex` method in `Mantid`)
- `general` - minimizers which solve a generic *min f(x)*
- `simplex` - derivative free simplex based algorithms e.g. Nelder-Mead
- `trust_region` - algorithms which emply a trust region approach

- `levenberg-marquardt` - minimizers that use the Levenberg-Marquardt algorithm
- `gauss_newton` - minimizers that use the Gauss Newton algorithm
- `bfgs` - minimizers that use the BFGS algorithm
- `conjugate_gradient` - Conjugate Gradient algorithms
- `steepest_descent` - Steepest Descent algorithms
- `global_optimization` - Global Optimization algorithms

The **values** of the dictionary are given as a list of minimizers for that specific controller that fit into each of the above categories. See for example the `GSL` controller.

cleanup()
Convert the result to a numpy array and populate the variables results will be read from.

eval_chisq(*params*, *x=None*, *y=None*, *e=None*)
Computes the chisq value. If multi-fit inputs will be lists and this will return a list of chi squared of params[i], x[i], y[i], and e[i].

Parameters

- **params** (*list of float or list of list of float*) – The parameters to calculate residuals for
- **x** (*numpy array or list of numpy arrays, optional*) – x data points, defaults to self.data_x
- **y** (*numpy array or list of numpy arrays, optional*) – y data points, defaults to self.data_y
- **e** (*numpy array or list of numpy arrays, optional*) – error at each data point, defaults to self.data_e

Returns The sum of squares of residuals for the datapoints at the given parameters

Return type numpy array

fit()
Run problem with Mantid.

jacobian_enabled_solvers = ['BFGS', 'Conjugate gradient (Fletcher-Reeves imp.)', 'Conjugate gradient (Polak-Ribiere imp.)', 'Damped GaussNewton', 'Levenberg-Marquardt', 'Levenberg-MarquardtMD', 'SteepestDescent', 'Trust Region']
Within the controller class, you must define the list `jacobian_enabled_solvers` if any of the minimizers for the specific software are able to use jacobian information.

- `jacobian_enabled_solvers`: a list of minimizers in a specific

software that allow Jacobian information to be passed into the fitting algorithm

setup()
Setup problem ready to run with Mantid.

Adds a custom function to Mantid for calling in fit().

fitbenchmarking.controllers.matlab_controller module

Implements a controller for MATLAB

class fitbenchmarking.controllers.matlab_controller.**MatlabController**(*cost_func*)

Bases: *fitbenchmarking.controllers.matlab_mixin.MatlabMixin*, *fitbenchmarking.controllers.base_controller.Controller*

Controller for MATLAB fitting (fminsearch)

algorithm_check = {'all': ['Nelder-Mead Simplex'], 'bfgs': [], 'conjugate_gradient': [], 'deriv_free': ['Nelder-Mead Simplex'], 'gauss_newton': [], 'general': ['Nelder-Mead Simplex'], 'global_optimization': [], 'levenberg-marquardt': [], 'ls': [], 'simplex': ['Nelder-Mead Simplex'], 'steepest_descent': [], 'trust_region': []}

Within the controller class, you must initialize a dictionary, `algorithm_check`, such that the **keys** are given by:

- `all` - all minimizers
- `ls` - least-squares fitting algorithms
- `deriv_free` - derivative free algorithms (these are algorithms that cannot use information about derivatives – e.g., the `Simplex` method in `Mantid`)
- `general` - minimizers which solve a generic *min f(x)*
- `simplex` - derivative free simplex based algorithms e.g. Nelder-Mead
- `trust_region` - algorithms which emply a trust region approach
- `levenberg-marquardt` - minimizers that use the Levenberg-Marquardt algorithm
- `gauss_newton` - minimizers that use the Gauss Newton algorithm
- `bfgs` - minimizers that use the BFGS algorithm
- `conjugate_gradient` - Conjugate Gradient algorithms
- `steepest_descent` - Steepest Descent algorithms
- `global_optimization` - Global Optimization algorithms

The **values** of the dictionary are given as a list of minimizers for that specific controller that fit into each of the above categories. See for example the `GSL` controller.

cleanup()

Convert the result to a numpy array and populate the variables results will be read from.

fit()

Run problem with Matlab

incompatible_problems = ['mantid']

A list of incompatible problem formats for this controller.

setup()

Setup for Matlab fitting

fitbenchmarking.controllers.matlab_curve_controller module

Implements a controller for MATLAB Curve Fitting Toolbox

class fitbenchmarking.controllers.matlab_curve_controller.**MatlabCurveController**(*cost_func*)

Bases: *fitbenchmarking.controllers.matlab_mixin.MatlabMixin*, *fitbenchmarking.controllers.base_controller.Controller*

Controller for MATLAB Curve Fitting Toolbox fitting (fit)

algorithm_check = {'all': ['Trust-Region', 'Levenberg-Marquardt'], 'bfgs': [], 'conjugate_gradient': [], 'deriv_free': [], 'gauss_newton': [], 'general': [], 'global_optimization': [], 'levenberg-marquardt': ['Levenberg-Marquardt'], 'ls': ['Trust-Region', 'Levenberg-Marquardt'], 'simplex': [], 'steepest_descent': [], 'trust_region': ['Trust-Region', 'Levenberg-Marquardt']}

Within the controller class, you must initialize a dictionary, `algorithm_check`, such that the **keys** are given by:

- `all` - all minimizers
- `ls` - least-squares fitting algorithms
- `deriv_free` - derivative free algorithms (these are algorithms that cannot use information about derivatives – e.g., the `Simplex` method in `Mantid`)
- `general` - minimizers which solve a generic *min f(x)*
- `simplex` - derivative free simplex based algorithms e.g. Nelder-Mead
- `trust_region` - algorithms which emply a trust region approach
- `levenberg-marquardt` - minimizers that use the Levenberg-Marquardt algorithm
- `gauss_newton` - minimizers that use the Gauss Newton algorithm
- `bfgs` - minimizers that use the BFGS algorithm
- `conjugate_gradient` - Conjugate Gradient algorithms
- `steepest_descent` - Steepest Descent algorithms
- `global_optimization` - Global Optimization algorithms

The **values** of the dictionary are given as a list of minimizers for that specific controller that fit into each of the above categories. See for example the `GSL` controller.

cleanup()

Convert the result to a numpy array and populate the variables results will be read from.

controller_name = 'matlab_curve'

A name to be used in tables. If this is set to None it will be inferred from the class name.

fit()

Run problem with Matlab

incompatible_problems = ['mantid']

A list of incompatible problem formats for this controller.

setup()

Setup for Matlab fitting

fitbenchmarking.controllers.matlab_mixin module

Implements mixin class for the matlab fitting software controllers.

class fitbenchmarking.controllers.matlab_mixin.**MatlabMixin**(*cost_func*)
Bases: object

Mixin class for matlab fitting software controllers

static py_to_mat(*func*, *eng*)
Function that serializes a python function and then loads it into the Matlab engine workspace

fitbenchmarking.controllers.matlab_opt_controller module

Implements a controller for MATLAB Optimization Toolbox

class fitbenchmarking.controllers.matlab_opt_controller.**MatlabOptController**(*cost_func*)
Bases: *fitbenchmarking.controllers.matlab_mixin.MatlabMixin*, *fitbenchmarking.controllers.base_controller.Controller*

Controller for MATLAB Optimization Toolbox, implementing lsqcurvefit

algorithm_check = {'all': ['levenberg-marquardt', 'trust-region-reflective'], 'bfgs': [], 'conjugate_gradient': [], 'deriv_free': [], 'gauss_newton': [], 'general': [], 'global_optimization': [], 'levenberg-marquardt': ['levenberg-marquardt'], 'ls': ['levenberg-marquardt', 'trust-region-reflective'], 'simplex': [], 'steepest_descent': [], 'trust_region': ['levenberg-marquardt', 'trust-region-reflective']}
Within the controller class, you must initialize a dictionary, `algorithm_check`, such that the **keys** are given by:

- `all` - all minimizers
- `ls` - least-squares fitting algorithms
- `deriv_free` - derivative free algorithms (these are algorithms that cannot use information about derivatives – e.g., the `Simplex` method in `Mantid`)
- `general` - minimizers which solve a generic *min f(x)*
- `simplex` - derivative free simplex based algorithms e.g. Nelder-Mead
- `trust_region` - algorithms which emply a trust region approach
- `levenberg-marquardt` - minimizers that use the Levenberg-Marquardt algorithm
- `gauss_newton` - minimizers that use the Gauss Newton algorithm
- `bfgs` - minimizers that use the BFGS algorithm
- `conjugate_gradient` - Conjugate Gradient algorithms
- `steepest_descent` - Steepest Descent algorithms
- `global_optimization` - Global Optimization algorithms

The **values** of the dictionary are given as a list of minimizers for that specific controller that fit into each of the above categories. See for example the `GSL` controller.

cleanup()
Convert the result to a numpy array and populate the variables results will be read from.

controller_name = 'matlab_opt'
A name to be used in tables. If this is set to None it will be inferred from the class name.

fit()
Run problem with Matlab Optimization Toolbox

incompatible_problems = ['mantid']
A list of incompatible problem formats for this controller.

jacobian_enabled_solvers = ['levenberg-marquardt', 'trust-region-reflective']
Within the controller class, you must define the list `jacobian_enabled_solvers` if any of the minimizers for the specific software are able to use jacobian information.

- `jacobian_enabled_solvers`: a list of minimizers in a specific

software that allow Jacobian information to be passed into the fitting algorithm

setup()
Setup for Matlab Optimization Toolbox fitting

fitbenchmarking.controllers.matlab_stats_controller module

Implements a controller for MATLAB Statistics Toolbox

class fitbenchmarking.controllers.matlab_stats_controller.**MatlabStatsController**(*cost_func*)
Bases: *fitbenchmarking.controllers.matlab_mixin.MatlabMixin*, *fitbenchmarking.controllers.base_controller.Controller*

Controller for MATLAB Statistics Toolbox fitting (nlinfit)

algorithm_check = {'all': ['Levenberg-Marquardt'], 'bfgs': [], 'conjugate_gradient': [], 'deriv_free': [], 'gauss_newton': [], 'general': [], 'global_optimization': [], 'levenberg-marquardt': ['Levenberg-Marquardt'], 'ls': ['Levenberg-Marquardt'], 'simplex': [], 'steepest_descent': [], 'trust_region': ['Levenberg-Marquardt']}
Within the controller class, you must initialize a dictionary, `algorithm_check`, such that the **keys** are given by:

- `all` - all minimizers
- `ls` - least-squares fitting algorithms
- `deriv_free` - derivative free algorithms (these are algorithms that cannot use information about derivatives – e.g., the `Simplex` method in `Mantid`)
- `general` - minimizers which solve a generic *min f(x)*
- `simplex` - derivative free simplex based algorithms e.g. Nelder-Mead
- `trust_region` - algorithms which emply a trust region approach
- `levenberg-marquardt` - minimizers that use the Levenberg-Marquardt algorithm
- `gauss_newton` - minimizers that use the Gauss Newton algorithm
- `bfgs` - minimizers that use the BFGS algorithm
- `conjugate_gradient` - Conjugate Gradient algorithms
- `steepest_descent` - Steepest Descent algorithms
- `global_optimization` - Global Optimization algorithms

The **values** of the dictionary are given as a list of minimizers for that specific controller that fit into each of the above categories. See for example the GSL controller.

cleanup()
Convert the result to a numpy array and populate the variables results will be read from.

controller_name = 'matlab_stats'
A name to be used in tables. If this is set to None it will be inferred from the class name.

fit()
Run problem with Matlab Statistics Toolbox

incompatible_problems = ['mantid']
A list of incompatible problem formats for this controller.

setup()
Setup for Matlab fitting

fitbenchmarking.controllers.minuit_controller module

Implements a controller for the CERN package Minuit https://seal.web.cern.ch/seal/snapshot/work-packages/mathlibs/minuit/ using the iminuit python interface http://iminuit.readthedocs.org

class fitbenchmarking.controllers.minuit_controller.**MinuitController**(*cost_func*)
Bases: *fitbenchmarking.controllers.base_controller.Controller*

Controller for the Minuit fitting software

algorithm_check = {'all': ['minuit'], 'bfgs': [], 'conjugate_gradient': [], 'deriv_free': [], 'gauss_newton': [], 'general': ['minuit'], 'global_optimization': [], 'levenberg-marquardt': [], 'ls': [], 'simplex': [], 'steepest_descent': [], 'trust_region': []}
Within the controller class, you must initialize a dictionary, algorithm_check, such that the **keys** are given by:

- all - all minimizers
- ls - least-squares fitting algorithms
- deriv_free - derivative free algorithms (these are algorithms that cannot use information about derivatives – e.g., the Simplex method in Mantid)
- general - minimizers which solve a generic *min f(x)*
- simplex - derivative free simplex based algorithms e.g. Nelder-Mead
- trust_region - algorithms which emply a trust region approach
- levenberg-marquardt - minimizers that use the Levenberg-Marquardt algorithm
- gauss_newton - minimizers that use the Gauss Newton algorithm
- bfgs - minimizers that use the BFGS algorithm
- conjugate_gradient - Conjugate Gradient algorithms
- steepest_descent - Steepest Descent algorithms
- global_optimization - Global Optimization algorithms

The **values** of the dictionary are given as a list of minimizers for that specific controller that fit into each of the above categories. See for example the GSL controller.

cleanup()
Convert the result to a numpy array and populate the variables results will be read from

fit()
Run problem with Minuit

setup()
Setup for Minuit

fitbenchmarking.controllers.ralfit_controller module

Implements a controller for RALFit https://github.com/ralna/RALFit

class fitbenchmarking.controllers.ralfit_controller.**RALFitController**(*cost_func*)
Bases: *fitbenchmarking.controllers.base_controller.Controller*

Controller for the RALFit fitting software.

algorithm_check = {'all': ['gn', 'hybrid', 'gn_reg', 'hybrid_reg'], 'bfgs': [], 'conjugate_gradient': [], 'deriv_free': [], 'gauss_newton': ['gn', 'gn_reg'], 'general': [], 'global_optimization': [], 'levenberg-marquardt': ['gn', 'gn_reg'], 'ls': ['gn', 'hybrid', 'gn_reg', 'hybrid_reg'], 'simplex': [], 'steepest_descent': [], 'trust_region': ['gn', 'hybrid']}
Within the controller class, you must initialize a dictionary, `algorithm_check`, such that the **keys** are given by:

- `all` - all minimizers
- `ls` - least-squares fitting algorithms
- `deriv_free` - derivative free algorithms (these are algorithms that cannot use information about derivatives – e.g., the `Simplex` method in `Mantid`)
- `general` - minimizers which solve a generic *min f(x)*
- `simplex` - derivative free simplex based algorithms e.g. Nelder-Mead
- `trust_region` - algorithms which emply a trust region approach
- `levenberg-marquardt` - minimizers that use the Levenberg-Marquardt algorithm
- `gauss_newton` - minimizers that use the Gauss Newton algorithm
- `bfgs` - minimizers that use the BFGS algorithm
- `conjugate_gradient` - Conjugate Gradient algorithms
- `steepest_descent` - Steepest Descent algorithms
- `global_optimization` - Global Optimization algorithms

The **values** of the dictionary are given as a list of minimizers for that specific controller that fit into each of the above categories. See for example the `GSL` controller.

cleanup()
Convert the result to a numpy array and populate the variables results will be read from.

fit()
Run problem with RALFit.

hes_eval(*params*, *r*)

Function to ensure correct inputs and outputs are used for the RALFit hessian evaluation

param params parameters

type params numpy array

param r residuals, required by RALFit to be passed for hessian evaluation

type r numpy array

return hessian 2nd order term: sum_{i=1}^m r_i

abla^2 r_i

rtype numpy array

`hessian_enabled_solvers = ['hybrid', 'hybrid_reg']`
Within the controller class, you must define the list `hessian_enabled_solvers` if any of the minimizers for the specific software are able to use hessian information.

- `hessian_enabled_solvers`: a list of minimizers in a specific

software that allow Hessian information to be passed into the fitting algorithm

`jacobian_enabled_solvers = ['gn', 'hybrid', 'gn_reg', 'hybrid_reg']`
Within the controller class, you must define the list `jacobian_enabled_solvers` if any of the minimizers for the specific software are able to use jacobian information.

- `jacobian_enabled_solvers`: a list of minimizers in a specific

software that allow Jacobian information to be passed into the fitting algorithm

`setup()`
Setup for RALFit

fitbenchmarking.controllers.scipy_controller module

Implements a controller for the scipy fitting software. In particular, here for the scipy minimize solver for general minimization problems.

`class fitbenchmarking.controllers.scipy_controller.ScipyController(cost_func)`
Bases: *fitbenchmarking.controllers.base_controller.Controller*

Controller for the Scipy fitting software.

`algorithm_check = {'all': ['Nelder-Mead', 'Powell', 'CG', 'BFGS', 'Newton-CG', 'L-BFGS-B', 'TNC', 'SLSQP'], 'bfgs': ['BFGS', 'L-BFGS-B'], 'conjugate_gradient': ['CG', 'Newton-CG', 'Powell'], 'deriv_free': ['Nelder-Mead', 'Powell'], 'gauss_newton': [], 'general': ['Nelder-Mead', 'Powell', 'CG', 'BFGS', 'Newton-CG', 'L-BFGS-B', 'TNC', 'SLSQP'], 'global_optimization': [], 'levenberg-marquardt': [], 'ls': [None], 'simplex': ['Nelder-Mead'], 'steepest_descent': [], 'trust_region': []}`
Within the controller class, you must initialize a dictionary, `algorithm_check`, such that the **keys** are given by:

- `all` - all minimizers
- `ls` - least-squares fitting algorithms
- `deriv_free` - derivative free algorithms (these are algorithms that cannot use information about derivatives – e.g., the `Simplex` method in `Mantid`)
- `general` - minimizers which solve a generic *min f(x)*

- `simplex` - derivative free simplex based algorithms e.g. Nelder-Mead
- `trust_region` - algorithms which emply a trust region approach
- `levenberg-marquardt` - minimizers that use the Levenberg-Marquardt algorithm
- `gauss_newton` - minimizers that use the Gauss Newton algorithm
- `bfgs` - minimizers that use the BFGS algorithm
- `conjugate_gradient` - Conjugate Gradient algorithms
- `steepest_descent` - Steepest Descent algorithms
- `global_optimization` - Global Optimization algorithms

The **values** of the dictionary are given as a list of minimizers for that specific controller that fit into each of the above categories. See for example the `GSL` controller.

cleanup()
Convert the result to a numpy array and populate the variables results will be read from.

fit()
Run problem with Scipy.

hessian_enabled_solvers = ['Newton-CG']
Within the controller class, you must define the list `hessian_enabled_solvers` if any of the minimizers for the specific software are able to use hessian information.

- `hessian_enabled_solvers`: a list of minimizers in a specific

software that allow Hessian information to be passed into the fitting algorithm

jacobian_enabled_solvers = ['CG', 'BFGS', 'Newton-CG', 'L-BFGS-B', 'TNC', 'SLSQP']
Within the controller class, you must define the list `jacobian_enabled_solvers` if any of the minimizers for the specific software are able to use jacobian information.

- `jacobian_enabled_solvers`: a list of minimizers in a specific

software that allow Jacobian information to be passed into the fitting algorithm

setup()
Setup problem ready to be run with SciPy

fitbenchmarking.controllers.scipy_go_controller module

Implements a controller for the scipy fitting software. In particular, here for the scipy minimize solver for general minimization problems.

class fitbenchmarking.controllers.scipy_go_controller.**ScipyGOController**(*cost_func*)
Bases: *fitbenchmarking.controllers.base_controller.Controller*

Controller for the Scipy fitting software.

algorithm_check = {'all': ['differential_evolution', 'shgo', 'dual_annealing'], 'bfgs': [], 'conjugate_gradient': [], 'deriv_free': ['differential_evolution'], 'gauss_newton': [], 'general': ['differential_evolution', 'shgo', 'dual_annealing'], 'global_optimization': ['differential_evolution', 'shgo', 'dual_annealing'], 'levenberg-marquardt': [], 'ls': [None], 'simplex': [], 'steepest_descent': [], 'trust_region': []}
Within the controller class, you must initialize a dictionary, `algorithm_check`, such that the **keys** are given by:

- `all` - all minimizers
- `ls` - least-squares fitting algorithms
- `deriv_free` - derivative free algorithms (these are algorithms that cannot use information about derivatives – e.g., the `Simplex` method in `Mantid`)
- `general` - minimizers which solve a generic *min f(x)*
- `simplex` - derivative free simplex based algorithms e.g. Nelder-Mead
- `trust_region` - algorithms which emply a trust region approach
- `levenberg-marquardt` - minimizers that use the Levenberg-Marquardt algorithm
- `gauss_newton` - minimizers that use the Gauss Newton algorithm
- `bfgs` - minimizers that use the BFGS algorithm
- `conjugate_gradient` - Conjugate Gradient algorithms
- `steepest_descent` - Steepest Descent algorithms
- `global_optimization` - Global Optimization algorithms

The **values** of the dictionary are given as a list of minimizers for that specific controller that fit into each of the above categories. See for example the `GSL` controller.

cleanup()
Convert the result to a numpy array and populate the variables results will be read from.

controller_name = 'scipy_go'
A name to be used in tables. If this is set to None it will be inferred from the class name.

fit()
Run problem with Scipy GO.

jacobian_enabled_solvers = ['shgo', 'dual_annealing']
Within the controller class, you must define the list `jacobian_enabled_solvers` if any of the minimizers for the specific software are able to use jacobian information.

- `jacobian_enabled_solvers`: a list of minimizers in a specific

software that allow Jacobian information to be passed into the fitting algorithm

setup()
Setup problem ready to be run with SciPy GO

fitbenchmarking.controllers.scipy_ls_controller module

Implements a controller for the scipy ls fitting software. In particular, for the scipy least_squares solver.

class fitbenchmarking.controllers.scipy_ls_controller.**ScipyLSController**(*cost_func*)
Bases: *fitbenchmarking.controllers.base_controller.Controller*

Controller for the Scipy Least-Squares fitting software.

algorithm_check = {'all': ['lm-scipy', 'trf', 'dogbox'], 'bfgs': [], 'conjugate_gradient': [], 'deriv_free': [None], 'gauss_newton': [], 'general': [None], 'global_optimization': [], 'levenberg-marquardt': ['lm-scipy'], 'ls': ['lm-scipy', 'trf', 'dogbox'], 'simplex': [], 'steepest_descent': [], 'trust_region': ['lm-scipy', 'trf', 'dogbox']}
Within the controller class, you must initialize a dictionary, `algorithm_check`, such that the **keys** are given by:

- `all` - all minimizers
- `ls` - least-squares fitting algorithms
- `deriv_free` - derivative free algorithms (these are algorithms that cannot use information about derivatives – e.g., the `Simplex` method in `Mantid`)
- `general` - minimizers which solve a generic *min f(x)*
- `simplex` - derivative free simplex based algorithms e.g. Nelder-Mead
- `trust_region` - algorithms which emply a trust region approach
- `levenberg-marquardt` - minimizers that use the Levenberg-Marquardt algorithm
- `gauss_newton` - minimizers that use the Gauss Newton algorithm
- `bfgs` - minimizers that use the BFGS algorithm
- `conjugate_gradient` - Conjugate Gradient algorithms
- `steepest_descent` - Steepest Descent algorithms
- `global_optimization` - Global Optimization algorithms

The **values** of the dictionary are given as a list of minimizers for that specific controller that fit into each of the above categories. See for example the `GSL` controller.

cleanup()
Convert the result to a numpy array and populate the variables results will be read from.

controller_name = 'scipy_ls'
A name to be used in tables. If this is set to None it will be inferred from the class name.

fit()
Run problem with Scipy LS.

jacobian_enabled_solvers = ['lm-scipy', 'trf', 'dogbox']
Within the controller class, you must define the list `jacobian_enabled_solvers` if any of the minimizers for the specific software are able to use jacobian information.

- `jacobian_enabled_solvers`: a list of minimizers in a specific

software that allow Jacobian information to be passed into the fitting algorithm

setup()
Setup problem ready to be run with SciPy LS

Module contents

fitbenchmarking.core package

Submodules

fitbenchmarking.core.fitting_benchmarking module

Main module of the tool, this holds the master function that calls lower level functions to fit and benchmark a set of problems for a certain fitting software.

fitbenchmarking.core.fitting_benchmarking.**benchmark**(*options*, *data_dir*)
Gather the user input and list of paths. Call benchmarking on these. The benchmarking structure is:

```
loop_over_benchmark_problems()
    loop_over_starting_values()
        loop_over_software()
            loop_over_minimizers()
                loop_over_jacobians()
                    loop_over_hessians()
```

Parameters

- **options** (fitbenchmarking.utils.options.Options) – dictionary containing software used in fitting the problem, list of minimizers and location of json file contain minimizers
- **data_dir** – full path of a directory that holds a group of problem definition files

Returns all results, problems where all fitting failed, minimizers that were unselected due to algorithm_type

Return type list[fibenchmarking.utils.fitbm_result.FittingResult], list[str], dict[str, list[str]]

fitbenchmarking.core.fitting_benchmarking.**loop_over_benchmark_problems**(*problem_group*, *options*)

Loops over benchmark problems

Parameters

- **problem_group** (*list*) – locations of the benchmark problem files
- **options** (fitbenchmarking.utils.options.Options) – FitBenchmarking options for current run

Returns all results, problems where all fitting failed, and minimizers that were unselected due to algorithm_type

Return type list[fibenchmarking.utils.fitbm_result.FittingResult], list[str], dict[str, list[str]]

fitbenchmarking.core.fitting_benchmarking.**loop_over_cost_function**(*problem*, *options*, *start_values_index*, *grabbed_output*)

Run benchmarking for each cost function given in options.

Parameters

- **problem** (fitbenchmarking.parsing.fitting_problem.FittingProblem) – The problem to run fitting on
- **options** (fitbenchmarking.utils.options.Options) – FitBenchmarking options for current run
- **start_values_index** (*int*) – Integer that selects the starting values when datasets have multiple ones.
- **grabbed_output** (fitbenchmarking.utils.output_grabber.OutputGrabber) – Object that removes third part output from console

Returns all results, and minimizers that were unselected due to algorithm_type

Return type list[fibenchmarking.utils.fitbm_result.FittingResult], dict[str, list[str]]

fitbenchmarking.core.fitting_benchmarking.**loop_over_fitting_software**(*cost_func*, *options*, *start_values_index*, *grabbed_output*)

Loops over fitting software selected in the options

Parameters

- **cost_func** (*CostFunction*) – a cost_func object containing information used in fitting
- **options** (fitbenchmarking.utils.options.Options) – FitBenchmarking options for current run
- **start_values_index** (*int*) – Integer that selects the starting values when datasets have multiple ones.
- **grabbed_output** (fitbenchmarking.utils.output_grabber.OutputGrabber) – Object that removes third part output from console

Returns all results, and minimizers that were unselected due to algorithm_type

Return type list[fibenchmarking.utils.fitbm_result.FittingResult], dict[str, list[str]]

fitbenchmarking.core.fitting_benchmarking.**loop_over_hessians**(*controller*, *options*, *grabbed_output*)

Loops over Hessians set from the options file

Parameters

- **controller** (*Object derived from BaseSoftwareController*) – The software controller for the fitting
- **options** (fitbenchmarking.utils.options.Options) – FitBenchmarking options for current run
- **grabbed_output** (fitbenchmarking.utils.output_grabber.OutputGrabber) – Object that removes third part output from console

Returns a FittingResult for each run

Return type list[fibenchmarking.utils.fitbm_result.FittingResult],

fitbenchmarking.core.fitting_benchmarking.**loop_over_jacobians**(*controller*, *options*, *grabbed_output*)

Loops over Jacobians set from the options file

Parameters

- **controller** (*Object derived from BaseSoftwareController*) – The software controller for the fitting
- **options** (fitbenchmarking.utils.options.Options) – FitBenchmarking options for current run
- **grabbed_output** (fitbenchmarking.utils.output_grabber.OutputGrabber) – Object that removes third part output from console

Returns a FittingResult for each run.

Return type list[fibenchmarking.utils.fitbm_result.FittingResult]

fitbenchmarking.core.fitting_benchmarking.**loop_over_minimizers**(*controller*, *minimizers*, *options*, *grabbed_output*)

Loops over minimizers in fitting software

Parameters

- **controller** (*Object derived from BaseSoftwareController*) – The software controller for the fitting
- **minimizers** (*list*) – array of minimizers used in fitting
- **options** (fitbenchmarking.utils.options.Options) – FitBenchmarking options for current run
- **grabbed_output** (fitbenchmarking.utils.output_grabber.OutputGrabber) – Object that removes third part output from console

Returns all results, and minimizers that were unselected due to algorithm_type

Return type list[fibenchmarking.utils.fitbm_result.FittingResult], list[str])

fitbenchmarking.core.fitting_benchmarking.**loop_over_starting_values**(*problem*, *options*, *grabbed_output*)

Loops over starting values from the fitting problem.

Parameters

- **problem** (fitbenchmarking.parsing.fitting_problem.FittingProblem) – The problem to benchmark on
- **options** (fitbenchmarking.utils.options.Options) – FitBenchmarking options for current run
- **grabbed_output** (fitbenchmarking.utils.output_grabber.OutputGrabber) – Object that removes third party output from console

Returns all results, problems where all fitting failed, and minimizers that were unselected due to algorithm_type

Return type list[fibenchmarking.utils.fitbm_result.FittingResult], list[str], dict[str, list[str]]

fitbenchmarking.core.fitting_benchmarking.**perform_fit**(*controller*, *options*, *grabbed_output*)

Performs a fit using the provided controller and its data. It will be run a number of times specified by num_runs.

Parameters

- **controller** (*Object derived from BaseSoftwareController*) – The software controller for the fitting
- **options** (fitbenchmarking.utils.options.Options) – The user options for the benchmark.
- **grabbed_output** (fitbenchmarking.utils.output_grabber.OutputGrabber) – Object that removes third part output from console

Returns The chi squared and runtime of the fit.

Return type tuple(float, float)

fitbenchmarking.core.results_output module

Functions that create the tables, support pages, figures, and indexes.

fitbenchmarking.core.results_output.**create_directories**(*options*, *group_name*)
Create the directory structure ready to store the results

Parameters

- **options** (fitbenchmarking.utils.options.Options) – The options used in the fitting problem and plotting
- **group_name** (*str*) – name of the problem group

Returns paths to the top level group results, support pages, and figures directories

Return type (str, str, str)

fitbenchmarking.core.results_output.**create_plots**(*options*, *results*, *best_results*, *figures_dir*)
Create a plot for each result and store in the figures directory

Parameters

- **options** (fitbenchmarking.utils.options.Options) – The options used in the fitting problem and plotting
- **results** (*list of list of fitbenchmarking.utils.fitbm_result.FittingResult*) – results nested array of objects
- **best_results** (*list[dict[str: fitbenchmarking.utils.fitbm_result.FittingResult]]*) – best result for each problem seperated by cost function
- **figures_dir** (*str*) – Path to directory to store the figures in

fitbenchmarking.core.results_output.**create_problem_level_index**(*options*, *table_names*, *group_name*, *group_dir*, *table_descriptions*)
Generates problem level index page.

Parameters

- **options** (fitbenchmarking.utils.options.Options) – The options used in the fitting problem and plotting
- **table_names** (*list*) – list of table names
- **group_name** (*str*) – name of the problem group
- **group_dir** (*str*) – Path to the directory where the index should be stored
- **table_descriptions** (*dict*) – dictionary containing descriptions of the tables and the comparison mode

fitbenchmarking.core.results_output.**preprocess_data**(*results: list[*FittingResult*]*)
Generate a dictionary of results lists sorted into the correct order with rows and columns as the key and list elements respectively.

This is used to create HTML and txt tables. This is stored in self.sorted_results

Parameters **results** (*list[*fitbenchmarking.utils.fitbm_result.FittingResult*]*) – The results to process

Returns The best result grouped by row and category (cost function), The sorted results grouped by row and category

Return type dict[str, dict[str, *utils.fitbm_result.FittingResult*]], dict[str, dict[str, list[*utils.fitbm_result.FittingResult*]]]

fitbenchmarking.core.results_output.**save_results**(*options*, *results*, *group_name*, *failed_problems*, *unselected_minimizers*)

Create all results files and store them. Result files are plots, support pages, tables, and index pages.

Parameters

- **options** (fitbenchmarking.utils.options.Options) – The options used in the fitting problem and plotting
- **results** (*list[*fitbenchmarking.utils.fitbm_result.FittingResult*]*) – The results from fitting
- **group_name** (*str*) – name of the problem group
- **failed_problems** (*list*) – list of failed problems to be reported in the html output

Params unselected_minimizers Dictionary containing unselected minimizers based on the algorithm_type option

Returns Path to directory of group results

Return type str

Module contents

fitbenchmarking.cost_func package

Submodules

fitbenchmarking.cost_func.base_cost_func module

Implements the base class for the cost function class.

class fitbenchmarking.cost_func.base_cost_func.**CostFunc**(*problem*)

Bases: object

Base class for the cost functions.

abstract eval_cost(*params*, ***kwargs*)

Evaluate the cost function

Parameters **params** (*list*) – The parameters to calculate residuals for

Returns evaluated cost function

Return type float

abstract hes_cost(*params*, ***kwargs*)

Uses the Hessian of the model to evaluate the Hessian of the cost function, $\nabla_p^2 F(r(x, y, p))$, at the given parameters.

Parameters **params** (*list*) – The parameters at which to calculate Hessians

Returns evaluated Hessian of the cost function

Return type 2D numpy array

abstract hes_res(*params*, ***kwargs*)
Uses the Hessian of the model to evaluate the Hessian of the cost function residual, $\nabla_p^2 r(x, y, p)$, at the given parameters.

Parameters **params** (*list*) – The parameters at which to calculate Hessians

Returns evaluated Hessian and Jacobian of the residual at

each x, y pair :rtype: tuple(list of 2D numpy arrays, list of 1D numpy arrays)

abstract jac_cost(*params*, ***kwargs*)
Uses the Jacobian of the model to evaluate the Jacobian of the cost function, $\nabla_p F(r(x, y, p))$, at the given parameters.

Parameters **params** (*list*) – The parameters at which to calculate Jacobians

Returns evaluated Jacobian of the cost function

Return type 1D numpy array

abstract jac_res(*params*, ***kwargs*)
Uses the Jacobian of the model to evaluate the Jacobian of the cost function residual, $\nabla_p r(x, y, p)$, at the given parameters.

Parameters **params** (*list*) – The parameters at which to calculate Jacobians

Returns evaluated Jacobian of the residual at each x, y pair

Return type a list of 1D numpy arrays

validate_algorithm_type(*algorithm_check*, *minimizer*)
Helper function which checks that the algorithm type of the selected minimizer from the options (options.minimizer) is incompatible with the selected cost function

Parameters **algorithm_check** – dictionary object containing algorithm

types and minimizers for selected software :type algorithm_check: dict :param minimizer: string of minimizers selected from the options :type minimizer: str

fitbenchmarking.cost_func.cost_func_factory module

This file contains a factory implementation for the available cost functions within FitBenchmarking.

fitbenchmarking.cost_func.cost_func_factory.**create_cost_func**(*cost_func_type*)
Create a cost function class class.

Parameters **cost_func_type** (*str*) – Type of cost function selected from options

Returns Cost function class for the problem

Return type fitbenchmarking.cost_func.base_cost_func.CostFunc subclass

fitbenchmarking.cost_func.hellinger_nlls_cost_func module

Implements the root non-linear least squares cost function

class fitbenchmarking.cost_func.hellinger_nlls_cost_func.**HellingerNLLSCostFunc**(*problem*)
Bases: *fitbenchmarking.cost_func.nlls_base_cost_func.BaseNLLSCostFunc*

This defines the Hellinger non-linear least squares cost function where, given a set of n data points (x_i, y_i), associated errors e_i, and a model function $f(x, p)$, we find the optimal parameters in the Hellinger least-squares sense by solving:

$$\min_p \sum_{i=1}^{n} \left(\sqrt{y_i} - \sqrt{f(x_i, p)}\right)^2$$

where p is a vector of length m, and we start from a given initial guess for the optimal parameters. More information on non-linear least squares cost functions can be found here and for the Hellinger distance measure see here.

eval_r(*params*, ***kwargs*)
Calculate the residuals, $\sqrt{y_i} - \sqrt{f(x_i, p)}$

Parameters **params** (*list*) – The parameters, p, to calculate residuals for

Returns The residuals for the datapoints at the given parameters

Return type numpy array

hes_res(*params*, ***kwargs*)
Uses the Hessian of the model to evaluate the Hessian of the cost function residual, $\nabla_p^2 r(x, y, p)$, at the given parameters.

Parameters **params** (*list*) – The parameters at which to calculate Hessians

Returns evaluated Hessian and Jacobian of the residual at

each x, y pair :rtype: tuple(list of 2D numpy arrays, list of 1D numpy arrays)

jac_res(*params*, ***kwargs*)
Uses the Jacobian of the model to evaluate the Jacobian of the cost function residual, $\nabla_p r(x, y, p)$, at the given parameters.

Parameters **params** (*list*) – The parameters at which to calculate Jacobians

Returns evaluated Jacobian of the residual at each x, y pair

Return type a list of 1D numpy arrays

fitbenchmarking.cost_func.nlls_base_cost_func module

Implements the base non-linear least squares cost function

class fitbenchmarking.cost_func.nlls_base_cost_func.**BaseNLLSCostFunc**(*problem*)
Bases: *fitbenchmarking.cost_func.base_cost_func.CostFunc*

This defines a base cost function for objectives of the type

$$\min_p \sum_{i=1}^{n} r(y_i, x_i, p)^2$$

where p is a vector of length m, and we start from a given initial guess for the optimal parameters.

eval_cost(*params*, ***kwargs*)
Evaluate the square of the L2 norm of the residuals, $\sum_i r(x_i, y_i, p)^2$ at the given parameters

Parameters **params** (*list*) – The parameters, p, to calculate residuals for

Returns The sum of squares of residuals for the datapoints at the given parameters

Return type numpy array

abstract eval_r(*params*, ***kwargs*)
Calculate residuals used in Least-Squares problems

Parameters **params** (*list*) – The parameters to calculate residuals for

Returns The residuals for the datapoints at the given parameters

Return type numpy array

hes_cost(*params*, ***kwargs*)
Uses the Hessian of the model to evaluate the Hessian of the cost function, $\nabla_p^2 F(r(x, y, p))$, at the given parameters.

Parameters **params** (*list*) – The parameters at which to calculate Hessians

Returns evaluated Hessian of the cost function

Return type 2D numpy array

jac_cost(*params*, ***kwargs*)
Uses the Jacobian of the model to evaluate the Jacobian of the cost function, $\nabla_p F(r(x, y, p))$, at the given parameters. :param params: The parameters at which to calculate Jacobians :type params: list :return: evaluated Jacobian of the cost function :rtype: 1D numpy array

fitbenchmarking.cost_func.nlls_cost_func module

Implements the non-weighted non-linear least squares cost function

class fitbenchmarking.cost_func.nlls_cost_func.**NLLSCostFunc**(*problem*)
Bases: *fitbenchmarking.cost_func.nlls_base_cost_func.BaseNLLSCostFunc*

This defines the non-linear least squares cost function where, given a set of n data points (x_i, y_i), associated errors e_i, and a model function $f(x, p)$, we find the optimal parameters in the root least-squares sense by solving:

$$\min_p \sum_{i=1}^{n} (y_i - f(x_i, p))^2$$

where p is a vector of length m, and we start from a given initial guess for the optimal parameters. More information on non-linear least squares cost functions can be found here.

eval_r(*params*, ***kwargs*)
Calculate the residuals, $y_i - f(x_i, p)$

Parameters **params** (*list*) – The parameters, p, to calculate residuals for

Returns The residuals for the datapoints at the given parameters

Return type numpy array

hes_res(*params*, ***kwargs*)
Uses the Hessian of the model to evaluate the Hessian of the cost function residual, $\nabla_p^2 r(x, y, p)$, at the given parameters.

Parameters **params** (*list*) – The parameters at which to calculate Hessians

Returns evaluated Hessian and Jacobian of the residual at

each x, y pair :rtype: tuple(list of 2D numpy arrays, list of 1D numpy arrays)

jac_res(*params*, ***kwargs*)
Uses the Jacobian of the model to evaluate the Jacobian of the cost function residual, $\nabla_p r(x, y, p)$, at the given parameters.

Parameters **params** (*list*) – The parameters at which to calculate Jacobians

Returns evaluated Jacobian of the residual at each x, y pair

Return type a list of 1D numpy arrays

fitbenchmarking.cost_func.poisson_cost_func module

Implements a Poisson deviance cost function based on Mantid's: https://docs.mantidproject.org/nightly/fitting/fitcostfunctions/Poisson.html

class fitbenchmarking.cost_func.poisson_cost_func.**PoissonCostFunc**(*problem*)
Bases: *fitbenchmarking.cost_func.base_cost_func.CostFunc*

This defines the Poisson deviance cost-function where, given the set of n data points (x_i, y_i), and a model function $f(x, p)$, we find the optimal parameters in the Poisson deviance sense by solving:

$$\min_p \sum_{i=1}^{n} \left(y_i \left(\log y_i - \log f(x_i, p)\right) - \left(y_i - f(x_i, p)\right)\right)$$

where p is a vector of length m, and we start from a given initial guess for the optimal parameters.

This cost function is intended for positive values.

This cost function is not a least squares problem and as such will not work with least squares minimizers. Please use *algorithm_type* to select *general* solvers. See options docs (*Fitting Options*) for information on how to do this.

eval_cost(*params*, ***kwargs*)
Evaluate the Poisson deviance cost function

Parameters

- **params** (*list*) – The parameters to calculate residuals for
- **x** (*np.array (optional)*) – The x values to evaluate at. Default is self.problem.data_x
- **y** (*np.array (optional)*) – The y values to evaluate at. Default is self.problem.data_y

Returns evaluated cost function

Return type float

hes_cost(*params*, ***kwargs*)
Uses the Hessian of the model to evaluate the Hessian of the cost function, $\nabla_p^2 F(r(x, y, p))$, at the given parameters.

Parameters **params** (*list*) – The parameters at which to calculate Hessians

Returns evaluated Hessian of the cost function

Return type 2D numpy array

hes_res(*params*, ***kwargs*)
Uses the Hessian of the model to evaluate the Hessian of the cost function residual, $\nabla_p^2 r(x, y, p)$, at the given parameters.

Parameters **params** (*list*) – The parameters at which to calculate Hessians

Returns evaluated Hessian and Jacobian of the residual at

each x, y pair :rtype: tuple(list of 2D numpy arrays, list of 1D numpy arrays)

jac_cost(*params*, ***kwargs*)
Uses the Jacobian of the model to evaluate the Jacobian of the cost function, $\nabla_p F(r(x, y, p))$, at the given parameters. :param params: The parameters at which to calculate Jacobians :type params: list :return: evaluated Jacobian of the cost function :rtype: 1D numpy array

jac_res(*params*, ***kwargs*)
Uses the Jacobian of the model to evaluate the Jacobian of the cost function residual, $\nabla_p r(x, y, p)$, at the given parameters.

Parameters **params** (*list*) – The parameters at which to calculate Jacobians

Returns evaluated Jacobian of the residual at each x, y pair

Return type a list of 1D numpy arrays

fitbenchmarking.cost_func.weighted_nlls_cost_func module

Implements the weighted non-linear least squares cost function

class fitbenchmarking.cost_func.weighted_nlls_cost_func.**WeightedNLLSCostFunc**(*problem*)
Bases: *fitbenchmarking.cost_func.nlls_base_cost_func.BaseNLLSCostFunc*

This defines the weighted non-linear least squares cost function where, given a set of n data points (x_i, y_i), associated errors e_i, and a model function $f(x, p)$, we find the optimal parameters in the root least-squares sense by solving:

$$\min_p \sum_{i=1}^{n} \left(\frac{y_i - f(x_i, p)}{e_i} \right)^2$$

where p is a vector of length m, and we start from a given initial guess for the optimal parameters. More information on non-linear least squares cost functions can be found here.

eval_r(*params*, ***kwargs*)
Calculate the residuals, $\frac{y_i - f(x_i, p)}{e_i}$

Parameters **params** (*list*) – The parameters, p, to calculate residuals for

Returns The residuals for the data points at the given parameters

Return type numpy array

hes_res(*params*, ***kwargs*)
Uses the Hessian of the model to evaluate the Hessian of the cost function residual, $\nabla_p^2 r(x, y, p)$, at the given parameters.

Parameters **params** (*list*) – The parameters at which to calculate Hessians

Returns evaluated Hessian and Jacobian of the residual at

each x, y pair :rtype: tuple(list of 2D numpy arrays, list of 1D numpy arrays)

jac_res(*params*, ***kwargs*)
Uses the Jacobian of the model to evaluate the Jacobian of the cost function residual, $\nabla_p r(x, y, p)$, at the given parameters.

Parameters **params** (*list*) – The parameters at which to calculate Jacobians

Returns evaluated Jacobian of the residual at each x, y pair

Return type a list of 1D numpy arrays

Module contents

fitbenchmarking.hessian package

Submodules

fitbenchmarking.hessian.analytic_hessian module

Module which acts as an analytic Hessian calculator

class fitbenchmarking.hessian.analytic_hessian.**Analytic**(*problem*, *jacobian*)

Bases: *fitbenchmarking.hessian.base_hessian.Hessian*

Class to apply an analytic Hessian

eval(*params*, ***kwargs*)

Evaluates Hessian of problem.eval_model, returning the value

abla^2_p f(x, p)

param params The parameter values to find the Hessian at

type params list

return Approximation of the Hessian

rtype 3D numpy array

name() → str

Get a name for the current status of the jacobian.

Returns A unique name for this jacobian/method combination

Return type str

fitbenchmarking.hessian.base_hessian module

Implements the base class for the Hessian.

class fitbenchmarking.hessian.base_hessian.**Hessian**(*problem*, *jacobian*)

Bases: `object`

Base class for Hessian.

INCOMPATIBLE_PROBLEMS = {}

abstract eval(*params*, ***kwargs*)

Evaluates Hessian of the model

Parameters **params** (*list*) – The parameter values to find the Hessian at

Returns Computed Hessian

Return type 3D numpy array

property method
Utility function to get the numerical method

Returns the names of the parameters

Return type list of str

name() → str
Get a name for the current status of the hessian.

Returns A unique name for this hessian/method combination

Return type str

fitbenchmarking.hessian.hessian_factory module

This file contains a factory implementation for the Hessians. This is used to manage the imports and reduce effort in adding new Hessian methods.

fitbenchmarking.hessian.hessian_factory.**create_hessian**(*hes_method*)
Create a Hessian class.

Parameters **hes_method** (*str*) – Type of Hessian selected from options

Returns Controller class for the problem

Return type fitbenchmarking.hessian.base_controller.Hessian subclass

fitbenchmarking.hessian.numdifftools_hessian module

fitbenchmarking.hessian.scipy_hessian module

Module which calculates SciPy finite difference approximations

class fitbenchmarking.hessian.scipy_hessian.**Scipy**(*problem*, *jacobian*)
Bases: *fitbenchmarking.hessian.base_hessian.Hessian*

Implements SciPy finite difference approximations to the derivative

INCOMPATIBLE_PROBLEMS = {'cs': ['mantid']}

eval(*params*, ***kwargs*)

Evaluates Hessian of problem.eval_model, returning the value

abla^2_p f(x, p)

param params The parameter values to find the Hessian at

type params list

return Approximation of the Hessian

rtype 3D numpy array

Module contents

fitbenchmarking.jacobian package

Submodules

fitbenchmarking.jacobian.analytic_jacobian module

Module which acts as a analytic Jacobian calculator

class fitbenchmarking.jacobian.analytic_jacobian.**Analytic**(*problem*)
Bases: *fitbenchmarking.jacobian.base_jacobian.Jacobian*

Class to apply an analytical Jacobian

eval(*params*, ***kwargs*)
Evaluates Jacobian of problem.eval_model

Parameters **params** (*list*) – The parameter values to find the Jacobian at

Returns Approximation of the Jacobian

Return type numpy array

name() → str
Get a name for the current status of the jacobian.

Returns A unique name for this jacobian/method combination

Return type str

fitbenchmarking.jacobian.base_jacobian module

Implements the base class for the Jacobian.

class fitbenchmarking.jacobian.base_jacobian.**Jacobian**(*problem*)
Bases: object

Base class for Jacobian.

INCOMPATIBLE_PROBLEMS = {}

abstract eval(*params*, ***kwargs*)
Evaluates Jacobian of the model, $\nabla_p f(x, p)$, at the point given by the parameters.

Parameters **params** (*list*) – The parameter values at which to evaluate the Jacobian

Returns Computed Jacobian

Return type numpy array

property method
Utility function to get the numerical method

Returns the names of the parameters

Return type list of str

name() → str
Get a name for the current status of the jacobian.

Returns A unique name for this jacobian/method combination

Return type str

fitbenchmarking.jacobian.default_jacobian module

Module which uses the minimizer's default jacobian

class fitbenchmarking.jacobian.default_jacobian.**Default**(*problem*)
Bases: *fitbenchmarking.jacobian.scipy_jacobian.Scipy*

Use the minimizer's jacobian/derivative approximation

eval(*params*, ***kwargs*)
Evaluates Jacobian of problem.eval_model

Parameters **params** (*list*) – The parameter values to find the Jacobian at

Returns Approximation of the Jacobian

Return type numpy array

name() → str
Get a name for the current status of the jacobian.

Returns A unique name for this jacobian/method combination

Return type str

fitbenchmarking.jacobian.jacobian_factory module

This file contains a factory implementation for the Jacobians. This is used to manage the imports and reduce effort in adding new Jacobian methods.

fitbenchmarking.jacobian.jacobian_factory.**create_jacobian**(*jac_method*)
Create a Jacobian class.

Parameters **jac_method** (*str*) – Type of Jacobian selected from options

Returns Controller class for the problem

Return type fitbenchmarking.jacobian.base_controller.Jacobian subclass

fitbenchmarking.jacobian.numdifftools_jacobian module

fitbenchmarking.jacobian.scipy_jacobian module

Module which calculates SciPy finite difference approximations

class fitbenchmarking.jacobian.scipy_jacobian.**Scipy**(*problem*)
Bases: *fitbenchmarking.jacobian.base_jacobian.Jacobian*

Implements SciPy finite difference approximations to the derivative

INCOMPATIBLE_PROBLEMS = {'cs': ['mantid']}

eval(*params*, ***kwargs*)
Evaluates Jacobian of problem.eval_model

Parameters **params** (*list*) – The parameter values to find the Jacobian at

Returns Approximation of the Jacobian

Return type numpy array

Module contents

fitbenchmarking.parsing package

Submodules

fitbenchmarking.parsing.base_parser module

Implements the base Parser as a Context Manager.

class fitbenchmarking.parsing.base_parser.**Parser**(*filename*, *options*)
Bases: object

Base abstract class for a parser. Further parsers should inherit from this and override the abstract parse() method.

abstract parse()
Parse the file into a FittingProblem.

Returns The parsed problem

Return type *FittingProblem*

fitbenchmarking.parsing.cutest_parser module

This file calls the pycutest interface for SIF data

class fitbenchmarking.parsing.cutest_parser.**CutestParser**(*filename*, *options*)
Bases: *fitbenchmarking.parsing.base_parser.Parser*

Use the pycutest interface to parse SIF files

This parser has some quirks. Due to the complex nature of SIF files, we utilise pycutest to generate the function. This function returns the residual *r(f,x,y,e) := (f(x) - y)/e* for some parameters x. For consistency we require the value to be *f(x, p)*.

To avoid having to convert back from the residual in the evaluation, we store the data separately and set y to 0.0, and e to 1.0 for all datapoints.

We then accomodate for the missing x argument by caching x against an associated function *f_x(p)*.

As such the function is defined by: *f(x, p) := r(f_x,p,0,1)*

If a new x is passed, we write a new file and parse it to generate a function. We define x to be new if `not np.isclose(x, cached_x)` for each `cached_x` that has already been stored.

parse()
Get data into a Fitting Problem via cutest.

Returns The fully parsed fitting problem

Return type *fitbenchmarking.parsing.fitting_problem.FittingProblem*

fitbenchmarking.parsing.fitbenchmark_parser module

This file implements a parser for the Fitbenchmark data format.

class fitbenchmarking.parsing.fitbenchmark_parser.**FitbenchmarkParser**(*filename*, *options*)
Bases: *fitbenchmarking.parsing.base_parser.Parser*

Parser for the native FitBenchmarking problem definition (FitBenchmark) file.

parse() → *fitbenchmarking.parsing.fitting_problem.FittingProblem*
Parse the Fitbenchmark problem file into a Fitting Problem.

Returns The fully parsed fitting problem

Return type *fitbenchmarking.parsing.fitting_problem.FittingProblem*

fitbenchmarking.parsing.fitting_problem module

Implements the FittingProblem class, this will be the object that inputs are parsed into before being passed to the controllers

class fitbenchmarking.parsing.fitting_problem.**FittingProblem**(*options*)
Bases: `object`

Definition of a fitting problem, which will be populated by a parser from a problem definition file.

Onces populated, this should include the data, the function and any other additional requirements from the data.

additional_info
dict Container for software specific information. This should be avoided if possible.

correct_data()
Strip data that overruns the start and end x_range, and approximate errors if not given. Modifications happen on member variables.

data_e
numpy array The errors or weights

data_x
numpy array The x-data

data_y
numpy array The y-data

description
string Description of the fitting problem

end_x
float The end of the range to fit model data over (if different from entire range) (/float/)

equation
string Equation (function or model) to fit against data

eval_model(*params*, ***kwargs*)
Function evaluation method

Parameters **params** (*list*) – parameter value(s)

Returns data values evaluated from the function of the problem

Return type numpy array

format
string Name of the problem definition type (e.g., 'cutest')

function
Callable function

get_function_params(*params*)
Return the function definition in a string format for output

Parameters **params** (*list*) – The parameters to use in the function string

Returns Representation of the function example format: 'b1 * (b2+x) | b1=-2.0, b2=50.0'

Return type string

hessian
Callable function for the Hessian

jacobian
Callable function for the Jacobian

multifit
bool Used to check if a problem is using multifit.

multivariate
bool Whether the function has been wrapped to reduce the dimension of x on function calls

name
string Name (title) of the fitting problem

property param_names
Utility function to get the parameter names

Returns the names of the parameters

Return type list of str

set_value_ranges(*value_ranges*)
Function to format parameter bounds before passing to controllers, so self.value_ranges is a list of tuples, which contain lower and upper bounds (lb,ub) for each parameter in the problem

Parameters **value_ranges** (*dict*) –

dictionary of bounded parameter names with lower and upper bound values e.g.

`{p1_name: [p1_min, p1_max], ...}`

sorted_index
numpy array The index for sorting the data (used in plotting)

start_x
float The start of the range to fit model data over (if different from entire range)

starting_values: list
list of dict Starting values of the fitting parameters

e.g. `[{p1_name: p1_val1, p2_name: p2_val1, ...}, {p1_name: p1_val2, ...}, ...]`

value_ranges
list Smallest and largest values of interest in the data

e.g. `[(p1_min, p1_max), (p2_min, p2_max),...]`

verify()
Basic check that minimal set of attributes have been set.

Raise FittingProblemError if object is not properly initialised.

fitbenchmarking.parsing.fitting_problem.**correct_data**(*x*, *y*, *e*, *startx*, *endx*, *use_errors*)
Strip data that overruns the start and end x_range, and approximate errors if not given.

Parameters

- **x** (*np.array*) – x data
- **y** (*np.array*) – y data
- **e** (*np.array*) – error data
- **startx** (*float*) – minimum x value
- **endx** (*float*) – maximum x value
- **use_errors** (*bool*) – whether errors should be added if not present

fitbenchmarking.parsing.ivp_parser module

This file implements a parser for the IVP data format.

class fitbenchmarking.parsing.ivp_parser.**IVPParser**(*filename*, *options*)
Bases: *fitbenchmarking.parsing.fitbenchmark_parser.FitbenchmarkParser*

Parser for a IVP problem definition file.

fitbenchmarking.parsing.mantid_parser module

This file implements a parser for the Mantid data format.

class fitbenchmarking.parsing.mantid_parser.**MantidParser**(*filename*, *options*)
Bases: *fitbenchmarking.parsing.fitbenchmark_parser.FitbenchmarkParser*

Parser for a Mantid problem definition file.

fitbenchmarking.parsing.nist_data_functions module

Functions that prepare the function specified in NIST problem definition file into the right format for SciPy.

fitbenchmarking.parsing.nist_data_functions.**format_function_scipy**(*function*)
Formats the function string such that it is scipy-ready.

Parameters **function** (*str*) – The function to be formatted

Returns The formatted function

Return type str

fitbenchmarking.parsing.nist_data_functions.**is_int**(*value*)
Checks to see if a value is an integer or not

Parameters **value** (*str*) – String representation of an equation

Returns Whether or not value is an int

Return type bool

fitbenchmarking.parsing.nist_data_functions.**is_safe**(*func_str*)
Verifies that a string is safe to be passed to exec in the context of an equation.

Parameters **func_str** (*string*) – The function to be checked

Returns Whether the string is of the expected format for an equation

Return type bool

fitbenchmarking.parsing.nist_data_functions.**nist_func_definition**(*function*, *param_names*)
Processing a function plus different set of starting values as specified in the NIST problem definition file into a callable

Parameters

- **function** (*str*) – function string as defined in a NIST problem definition file
- **param_names** (*list*) – names of the parameters in the function

Returns callable function

Return type callable

fitbenchmarking.parsing.nist_data_functions.**nist_hessian_definition**(*hessian*, *param_names*)
Processing a Hessian into a callable

Parameters

- **hessian** (*str*) – Hessian string as defined in the data files for the corresponding NIST problem definition file
- **param_names** (*list*) – names of the parameters in the function

Returns callable function

Return type callable

fitbenchmarking.parsing.nist_data_functions.**nist_jacobian_definition**(*jacobian*, *param_names*)
Processing a Jacobian plus different set of starting values as specified in the NIST problem definition file into a callable

Parameters

- **jacobian** (*str*) – Jacobian string as defined in the data files for the corresponding NIST problem definition file
- **param_names** (*list*) – names of the parameters in the function

Returns callable function

Return type callable

fitbenchmarking.parsing.nist_parser module

This file implements a parser for the NIST style data format.

class fitbenchmarking.parsing.nist_parser.**NISTParser**(*filename*, *options*)
Bases: *fitbenchmarking.parsing.base_parser.Parser*

Parser for the NIST problem definition file.

parse()
Parse the NIST problem file into a Fitting Problem.

Returns The fully parsed fitting problem

Return type *fitbenchmarking.parsing.fitting_problem.FittingProblem*

fitbenchmarking.parsing.parser_factory module

This file contains a factory implementation for the parsers. This is used to manage the imports and reduce effort in adding new parsers.

class fitbenchmarking.parsing.parser_factory.**ParserFactory**

Bases: object

A factory for creating parsers. This has the capability to select the correct parser, import it, and generate an instance of it. Parsers generated from this must be a subclass of base_parser.Parser

static create_parser(*filename*)

Inspect the input and create a parser that matches the required file.

Parameters **filename** (*string*) – The path to the file to be parsed

Returns Parser for the problem

Return type fitbenchmarking.parsing.base_parser.Parser subclass

fitbenchmarking.parsing.parser_factory.**parse_problem_file**(*prob_file*, *options*)

Loads the problem file into a fitting problem using the correct parser.

Parameters

- **prob_file** (*string*) – path to the problem file
- **options** (fitbenchmarking.utils.options.Options) – all the information specified by the user

Returns problem object with fitting information

Return type *fitbenchmarking.parsing.fitting_problem.FittingProblem*

fitbenchmarking.parsing.sasview_parser module

Module contents

fitbenchmarking.results_processing package

Submodules

fitbenchmarking.results_processing.acc_table module

Accuracy table

class fitbenchmarking.results_processing.acc_table.**AccTable**(*results*, *best_results*, *options*, *group_dir*, *pp_locations*, *table_name*)

Bases: *fitbenchmarking.results_processing.base_table.Table*

The accuracy results are calculated by evaluating the cost function with the fitted parameters.

get_value(*result*)

Gets the main value to be reported in the tables for a given result

Note that the first value (relative chi_sq) will be used in the default colour handling.

Parameters **result** (FittingResult) – The result to generate the values for.

Returns The normalised chi sq with respect to the smallest chi_sq value and absolute chi_sq for the result.

Return type tuple(float, float)

fitbenchmarking.results_processing.base_table module

Implements the base class for the tables.

class fitbenchmarking.results_processing.base_table.**Table**(*results*, *best_results*, *options*, *group_dir*, *pp_locations*, *table_name*)

Bases: object

Base class for the FitBenchmarking HTML and text output tables.

When inheriting from this, it may be useful to override the following functions as required:

- get_value
- display_str
- get_error_str
- get_link_str

create_pandas_data_frame(*html=False*)

Creates a pandas data frame of results

Parameters **html** (*bool. defaults to False*) – Whether to make the dataframe for html or plain text

Returns DataFrame with string representations of results

Return type pandas.DataFrame

create_results_dict()

Generate a dictionary of results lists with rows and columns as the key and list elements respectively. This is used to create HTML and txt tables. This is stored in self.sorted_results

display_str(*value*)

Converts a value generated by *get_value()* into a string respresentation to be used in the tables. Base class implementation takes the relative and absolute values and uses `self.output_string_type` as a template for the string format. This can be overridden to adequately display the results.

Parameters **value** (*tuple*) – Relative and absolute values

Returns string representation of the value for display in the table.

Return type str

property file_path

Getter function for the path to the table

Returns path to table

Return type str

get_colour_df(*like_df=None*)

Generate a dataframe of colours to add to the html rendering.

If like_df is passed this will use the column and row indexes of that dataframe.

Parameters **like_df** (*pandas.DataFrame*) – The dataframe to copy headings from. Defaults to None.

Returns A dataframe with colourings as strings

Return type pandas.DataFrame

`get_colours_for_row`(*results*)
Get the colours as strings for the given results in the table. The base class implementation, for example, uses the first value from self.get_value and `colour_map`, `colour_ulim` and `cmap_range` within *Options*.

Parameters **result** (*list[fitbenchmarking.utils.fitbm_result.FittingResult]*) – Results to get the colours for.

Returns The colour to use for each cell in the list

Return type list[str]

`get_description`(*html_description*)
Generates table description from class docstrings and converts them into html

Parameters **html_description** (*dict*) – Dictionary containing table descriptions

Returns Dictionary containing table descriptions

Return type dict

`static get_error_str`(*result*, *error_template='[{}]'*)
Get the error string for a result based on error_template This can be overridden if tables require different error formatting.

Parameters **result** (FittingResult) – The result to get the error string for

Returns A string representation of the error

Return type str

`get_link_str`(*result*)
Get the link as a string for the result. This can be overridden if tables require different links.

Parameters **result** (FittingResult) – The result to get the link for

Returns The link to go to when the cell is selected

Return type string

`get_str_dict`(*html=False*)
Create a dictionary with the table values as strings for display.

Returns The dictionary of strings for the table

Return type dict[list[str]]

`get_str_result`(*result*, *html=False*)
Given a single result, generate the string to display in this table. The html flag can be used to switch between a plain text and html format.

This is intended to be easily extensible by overriding the following functions:

- get_value
- display_str
- get_error_str
- get_link_str

If you find yourself overriding this, please consider if changes could be made to allow future tables to benefit.

Parameters

- **result** (*fitbenchmarking.utils.ftibm_result.FittingResult*) – The result to generate a string for
- **html** (*bool*) – Flag to control whether to generate a html string or plain text. Defaults to False.

Returns The string representation.

Return type str

abstract get_value(*result*)
Gets the main value to be reported in the tables for a given result

If more than one value is returned please note that the first value will be used in the default colour handling.

Parameters **result** (FittingResult) – The result to generate the values for.

Returns The value to convert to a string for the tables

Return type tuple(float)

minimizer_dropdown_html() → str
Generates the HTML for a dropdown checklist of minimizers.

Returns HTML for a dropdown checklist of minimizers.

Return type str

problem_dropdown_html() → str
Generates the HTML for a dropdown checklist of problem sets.

Returns HTML for a dropdown checklist of problem sets.

Return type str

save_colourbar(*fig_dir*, *n_divs=100*, *sz_in=(3, 0.8)*) → str
Generates a png of a labelled colourbar using matplotlib.

Parameters

- **fig_dir** (*str*) – path to figures directory
- **n_divs** (*int*) – number of divisions of shading in colourbar
- **sz_in** (*list[float] - 2 elements*) – dimensions of png in inches [width, height]

Returns The relative path to the colourbar image.

Return type str

property table_title
Getter function for table name if self._table_title is None

Returns name of table

Return type str

to_html()
Generate a html version of the table.

Returns HTML table output

Return type str

to_txt()
Generate a plain text version of the table

Returns Plain text table output

Return type str

static vals_to_colour(*vals*, *cmap*, *cmap_range*, *log_ulim*)

Converts an array of values to a list of hexadecimal colour strings using logarithmic sampling from a matplotlib colourmap according to relative value.

Parameters

- **vals** (*list[float]*) – values in the range [0, 1] to convert to colour strings
- **cmap** (*matplotlib colourmap object*) – matplotlib colourmap
- **cmap_range** (*list[float], 2 elements*) – values in range [0, 1] for colourmap cropping
- **log_ulim** (*float*) – log10 of worst shading cutoff value

Returns colours as hex strings for each input value

Return type list[str]

fitbenchmarking.results_processing.compare_table module

compare table

class fitbenchmarking.results_processing.compare_table.**CompareTable**(*results*, *best_results*, *options*, *group_dir*, *pp_locations*, *table_name*)

Bases: *fitbenchmarking.results_processing.base_table.Table*

The combined results show the accuracy in the first line of the cell and the runtime on the second line of the cell.

display_str(*value*)

Combine the accuracy and runtime values into a string representation.

Parameters **value** (*list[list[float]]*) – Relative and absolute values for accuracy and runtime [[acc_rel, runtime_rel], [acc_abs, runtime_abs]]

Returns string representation of the value for display in the table.

Return type str

get_value(*result*)

Gets the main value to be reported in the tables for a given result

Note that the first value (relative chi_sq) will be used in the default colour handling.

Parameters **result** (FittingResult) – The result to generate the values for.

Returns The normalised chi sq and runtime with respect to the smallest chi_sq and runtime respectively as well as the absolute values for both chi_sq and runtime for the result. [[acc_rel, runtime_rel], [acc_abs, runtime_abs]]

Return type list[list[float]]

vals_to_colour(*vals*, **args*)

Override vals_to_colour to allow it to run for both accuracy and runtime.

Parameters **vals** (*list[list[float, float]]*) – The relative values to get the colours for

Returns The colours for the values

Return type list[list[str]]

fitbenchmarking.results_processing.fitting_report module

Set up and build the fitting reports for various types of problems.

fitbenchmarking.results_processing.fitting_report.**create**(*results*, *support_pages_dir*, *options*)
Iterate through problem results and create a fitting report html page for each.

Parameters

- **results** (*list[FittingResult]*) – results object
- **support_pages_dir** (*str*) – directory in which the results are saved
- **options** (fitbenchmarking.utils.options.Options) – The options used in the fitting problem and plotting

fitbenchmarking.results_processing.fitting_report.**create_prob_group**(*result*, *support_pages_dir*, *options*)
Creates a fitting report containing figures and other details about the fit for a problem. A link to the fitting report is stored in the results object.

Parameters

- **result** (fitbenchmarking.utils.fitbm_result.FittingResult) – The result for a specific benchmark problem-minimizer-etc combination
- **support_pages_dir** (*str*) – directory to store the support pages in
- **options** (fitbenchmarking.utils.options.Options) – The options used in the fitting problem and plotting

fitbenchmarking.results_processing.fitting_report.**get_figure_paths**(*result*)
Get the paths to the figures used in the support page.

Parameters **result** (*fitbenchmarking.utils.fitbm_result.FittingProblem*) – The result to get the figures for

Returns the paths to the required figures

Return type tuple(str, str)

fitbenchmarking.results_processing.local_min_table module

compare table

class fitbenchmarking.results_processing.local_min_table.**LocalMinTable**(*results*, *best_results*, *options*, *group_dir*, *pp_locations*, *table_name*)
Bases: *fitbenchmarking.results_processing.base_table.Table*

The local min results shows a `True` or `False` value together with $\frac{||J^T r||}{||r||}$. The `True` or `False` indicates whether the software finds a minimum with respect to the following criteria:

- $||r|| \leq$ RES_TOL,
- $||J^T r|| \leq$ GRAD_TOL,
- $\frac{||J^T r||}{||r||} \leq$ GRAD_TOL,

where J and r are the Jacobian and residual of $f(x, p)$, respectively. The tolerances can be found in the results object.

display_str(*value*)

Combine the boolean value from variable local_min with the normalised residual

Parameters **value** (*bool, float*) – Whether the minimizer found a local minimizer and the $\frac{||J^T r||}{||r||}$ value

Returns string representation of the value for display in the table.

Return type str

get_value(*result*)

Gets the main value to be reported in the tables for a given result

Note that the first value (relative chi_sq) will be used in the default colour handling.

Parameters **result** (FittingResult) – The result to generate the values for.

Returns Whether the minimizer found a local minimizer (under the tests specified above) and $\frac{||J^T r||}{||r||}$

Return type bool, float

save_colourbar(*fig_dir*, *n_divs=2*, *sz_in=None*) → str

Override default save_colourbar as there are only 2 possible divisions of the colour map (true or false).

Parameters

- **fig_dir** (*str*) – path to figures directory
- **n_divs** (*int, Fixed to 2*) – **Unused** number of divisions of shading in colourbar
- **sz_in** (*list[float] - 2 elements*) – dimensions of png in inches [width, height]

Returns The relative path to the colourbar image.

Return type str

static vals_to_colour(*vals*, *cmap*, *cmap_range*, *log_ulim*)

Converts an array of values to a list of hexadecimal colour strings using sampling from a matplotlib colourmap according to whether a minimum was found.

Set to the bottom of the range if minimum was found, otherwise set to the top of the range.

Parameters

- **vals** (*list[float]*) – values in the range [0, 1] to convert to colour strings
- **cmap** (*matplotlib colourmap object*) – matplotlib colourmap
- **cmap_range** (*list[float], 2 elements*) – values in range [0, 1] for colourmap cropping
- **log_ulim** (*float*) – **Unused** log10 of worst shading cutoff value

Returns colours as hex strings for each input value

Return type list[str]

fitbenchmarking.results_processing.performance_profiler module

Set up performance profiles for both accuracy and runtime tables

fitbenchmarking.results_processing.performance_profiler.**create_plot**(*ax*, *step_values: list[np.ndarray]*, *solvers: list[str]*)

Function to draw the profile on a matplotlib axis

Parameters

- **ax** (an *.axes.SubplotBase* subclass of *~.axes.Axes* (or a subclass of *~.axes.Axes*)) – A matplotlib axis to be filled
- **step_values** (*list of np.array[float]*) – a sorted list of the values of the metric being profiled
- **solvers** (*list of strings*) – A list of the labels for the different solvers

fitbenchmarking.results_processing.performance_profiler.**plot**(*acc*, *runtime*, *fig_dir*)

Function that generates profiler plots

Parameters

- **acc** (*dict*) – acc dictionary containing number of occurrences
- **runtime** (*dict*) – runtime dictionary containing number of occurrences
- **fig_dir** (*str*) – path to directory containing the figures

Returns path to acc and runtime profile graphs

Return type tuple(str, str)

fitbenchmarking.results_processing.performance_profiler.**prepare_profile_data**(*results*)

Helper function which generates acc and runtime dictionaries which contain the values for each minimizer.

Parameters **results** (*dict[str, dict[str, list[*utils.fitbm_result.FittingResult*]]]*) – The sorted results grouped by row and category

Returns dictionary containing number of occurrences

Return type tuple(dict, dict)

fitbenchmarking.results_processing.performance_profiler.**profile**(*results*, *fig_dir*)

Function that generates profiler plots

Parameters

- **results** (*dict[str, dict[str, list[*utils.fitbm_result.FittingResult*]]]*) – The sorted results grouped by row and category
- **fig_dir** (*str*) – path to directory containing the figures

Returns path to acc and runtime profile graphs

Return type tuple(str, str)

fitbenchmarking.results_processing.plots module

Higher level functions that are used for plotting the fit plot and a starting guess plot.

class fitbenchmarking.results_processing.plots.**Plot**(*best_result*, *options*, *figures_dir*)
Bases: object

Class providing plotting functionality.

best_fit_plot_options = {'color': '#6699ff', 'linestyle': ':', 'linewidth': 3, 'marker': '', 'zorder': 3}

data_plot_options = {'color': 'black', 'label': 'Data', 'linestyle': '', 'linewidth': 1, 'marker': 'x', 'zorder': 0}

fit_plot_options = {'color': '#99ff66', 'linestyle': '-', 'linewidth': 3, 'marker': '', 'zorder': 2}

format_plot()
Performs post plot processing to annotate the plot correctly

ini_guess_plot_options = {'color': '#ff6699', 'label': 'Starting Guess', 'linestyle': '-', 'linewidth': 3, 'marker': '', 'zorder': 1}

plot_best(*result*)
Plots the fit along with the data using the "best_fit" style and saves to a file

Parameters **result** (FittingResult) – The result to plot

Returns path to the saved file

Return type str

plot_data(*errors*, *plot_options*, *x=None*, *y=None*)
Plots the data given

Parameters

- **errors** (*bool*) – whether fit minimizer uses errors
- **plot_options** (*dict*) – Values for style of the data to plot, for example color and zorder
- **x** (*np.array*) – x values to be plotted
- **y** (*np.array*) – y values to be plotted

plot_fit(*result*)
Updates self.line to show the fit using the passed in params. If self.line is empty it will create a new line. Stores the plot in a file

Parameters **result** (FittingResult) – The result to plot

Returns path to the saved file

Return type str

plot_initial_guess()
Plots the initial guess along with the data and stores in a file

Returns path to the saved file

Return type str

classmethod plot_summary(*categories*, *title*, *options*, *figures_dir*)
Create a comparison plot showing all fits from the results with the best for each category highlighted.

Parameters

- **categories** (*dict[str, list[FittingResults]]*) – The results to plot sorted into colour groups
- **title** (*str*) – A title for the graph
- **options** (utils.options.Options) – The options for the run
- **figures_dir** (*str*) – The directory to save the figures in

Returns The path to the new plot

Return type str

```
summary_best_plot_options = {'linestyle': '-', 'linewidth': 2, 'marker': '',
'zorder': 2}

summary_plot_options = {'alpha': 0.5, 'linestyle': '-', 'linewidth': 1, 'marker':
'', 'zorder': 1}
```

fitbenchmarking.results_processing.problem_summary_page module

Create the summary pages for the best minimizers.

fitbenchmarking.results_processing.problem_summary_page.**create**(*results*, *best_results*, *support_pages_dir*, *figures_dir*, *options*)

Create the problem summary pages.

Parameters

- **results** (*dict[str, dict[str, list[FittingResult]]]*) – The results to create summary pages for
- **best_results** (*dict[str, dict[str, FittingResult]]*) – The best result from each row and category
- **support_pages_dir** (*str*) – directory in which the results are to be saved
- **figures_dir** (*str*) – The directory where figures are stored.
- **options** (fitbenchmarking.utils.options.Options) – The options used in the fitting problem and plotting

fitbenchmarking.results_processing.runtime_table module

Runtime table

class fitbenchmarking.results_processing.runtime_table.**RuntimeTable**(*results*, *best_results*, *options*, *group_dir*, *pp_locations*, *table_name*)

Bases: *fitbenchmarking.results_processing.base_table.Table*

The timing results are calculated from an average (over num_runs) using the timeit module in python. num_runs is set in *FitBenchmarking Options*.

get_value(*result*)

Gets the main value to be reported in the tables for a given result

Note that the first value (relative runtime) will be used in the default colour handling.

Parameters **result** (FittingResult) – The result to generate the values for.

Returns The normalised runtime with respect to the smallest runtime and absolute runtime for the result.

Return type tuple(float, float)

fitbenchmarking.results_processing.tables module

Set up and build the results tables.

fitbenchmarking.results_processing.tables.**create_results_tables**(*options*, *results*, *best_results*, *group_dir*, *fig_dir*, *pp_locations*, *failed_problems*, *unselected_minimzers*)

Saves the results of the fitting to html/txt tables.

Parameters

- **options** (fitbenchmarking.utils.options.Options) – The options used in the fitting problem and plotting
- **results** (*dict[str, dict[str, list[utils.fitbm_result.FittingResult]]]*) – Results grouped by row and category (for colouring)
- **best_results** (*dict[str, dict[str, utils.fitbm_result.FittingResult]]*) – The best results from each row/category
- **group_dir** (*str*) – path to the directory where group results should be stored
- **fig_dir** (*str*) – path to the directory where figures should be stored
- **pp_locations** (*tuple(str,str)*) – tuple containing the locations of the performance profiles (acc then runtime)
- **failed_problems** (*list*) – list of failed problems to be reported in the html output

Params unselected_minimzers Dictionary containing unselected minimizers based on the algorithm_type option

Returns filepaths to each table e.g {'acc': <acc-table-filename>, 'runtime': ...} and dictionary of table descriptions

Return type tuple(dict, dict)

fitbenchmarking.results_processing.tables.**generate_table**(*results*, *best_results*, *options*, *group_dir*, *fig_dir*, *pp_locations*, *table_name*, *suffix*)

Generate html/txt tables.

Parameters

- **results** (*dict[str, dict[str, list[utils.fitbm_result.FittingResult]]]*) – Results grouped by row and category (for colouring)
- **best_results** (*dict[str, dict[str, utils.fitbm_result.FittingResult]]*) – The best results from each row/category
- **options** (fitbenchmarking.utils.options.Options) – The options used in the fitting problem and plotting
- **group_dir** (*str*) – path to the directory where group results should be stored
- **fig_dir** (*str*) – path to the directory where figures should be stored

- **pp_locations** (*tuple(str,str)*) – tuple containing the locations of the performance profiles (acc then runtime)
- **table_name** (*str*) – name of the table
- **suffix** (*str*) – table suffix

Returns (Table object, Dict of HTML strings for table and dropdowns,

text string of table, path to colourbar) :rtype: tuple(Table object, dict{str: str}, str, str)

fitbenchmarking.results_processing.tables.**load_table**(*table*)
Create and return table object.

Parameters **table** (*string*) – The name of the table to create a table for

Returns Table class for the problem

Return type fitbenchmarking/results_processing/tables.Table subclass

Module contents

fitbenchmarking.utils package

Submodules

fitbenchmarking.utils.create_dirs module

Utility functions for creating/deleting directories.

fitbenchmarking.utils.create_dirs.**css**(*results_dir*)
Creates a local css directory inside the results directory.

Parameters **support_pages_dir** (*str*) – path to the results directory

Returns path to the local css directory

Return type str

fitbenchmarking.utils.create_dirs.**del_contents_of_dir**(*directory*)
Delete contents of a directory, including other directories.

Parameters **directory** (*str*) – the target directory

fitbenchmarking.utils.create_dirs.**figures**(*support_pages_dir*)
Creates the figures directory inside the support_pages directory.

Parameters **support_pages_dir** (*str*) – path to the support pages directory

Returns path to the figures directory

Return type str

fitbenchmarking.utils.create_dirs.**group_results**(*results_dir*, *group_name*)
Creates the results directory for a specific group. e.g. fitbenchmarking/results/Neutron/

Parameters

- **results_dir** (*str*) – path to directory that holds all the results
- **group_name** (*str*) – name of the problem group

Returns path to folder group specific results dir

Return type str

fitbenchmarking.utils.create_dirs.**results**(*results_dir*)
Creates the results folder in the working directory.

Parameters **results_dir** (*str*) – path to the results directory, results dir name

Returns proper path to the results directory

Return type str

fitbenchmarking.utils.create_dirs.**support_pages**(*group_results_dir*)
Creates the support_pages directory in the group results directory.

Parameters **group_results_dir** (*str*) – path to the group results directory

Returns path to the figures directory

Return type str

fitbenchmarking.utils.debug module

Functions used for debugging and printing information in a readable format.

fitbenchmarking.utils.debug.**get_printable_table**(*class_name: str*, *class_info: dict*) → str
Creates and returns a string displaying the class info in a format that is easily read in a table.

Parameters

- **class_name** (*str*) – The name of a class owning the data.
- **class_info** (*dict{str: variant}*) – The data in the class to display.

Returns The class info in a readable format.

Return type str

fitbenchmarking.utils.exceptions module

This file holds all FitBenchmarking exceptions, organised by exception id

exception fitbenchmarking.utils.exceptions.**ControllerAttributeError**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates an issue with the attributes within a controller

class_message = 'Error in the controller attributes.'

error_code = 7

exception fitbenchmarking.utils.exceptions.**CostFuncError**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates a problem with the cost function class.

class_message = 'FitBenchmarking ran with no results'

error_code = 16

exception fitbenchmarking.utils.exceptions.**FilepathTooLongError**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates the filepath to save a file to is too long.

class_message = 'The filepath for saving a file is too long.'

error_code = 25

exception fitbenchmarking.utils.exceptions.**FitBenchmarkException**(*message=''*)
Bases: Exception

The base class for all FitBenchmarking exceptions

To define a new exception, inherrit from this and override the _class_message

class_message = 'An unknown exception occurred.'

error_code = 1

exception fitbenchmarking.utils.exceptions.**FittingProblemError**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates a problem with the fitting problem.

class_message = 'Fitting Problem raised and exception.'

error_code = 10

exception fitbenchmarking.utils.exceptions.**IncompatibleHessianError**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.ValidationException*

Indicates that the selected Hessian method is not compatible with selected options/problem set

class_message = 'The provided Hessian method cannot be used with the selected options or problem set.'

error_code = 26

exception fitbenchmarking.utils.exceptions.**IncompatibleJacobianError**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.ValidationException*

Indicates that the selected jacobian method is not compatible with selected options/problem set

class_message = 'The provided Jacobian method cannot be used with the selected options or problem set.'

error_code = 24

exception fitbenchmarking.utils.exceptions.**IncompatibleMinimizerError**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates that the selected minimizer is not compatible with selected options/problem set

class_message = 'Minimizer cannot be used with selected options/problem set'

error_code = 17

exception fitbenchmarking.utils.exceptions.**IncompatibleProblemError**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.ValidationException*

Indicates that the selected problem is not compatible with the selected options.

class_message = 'The selected software can not be used with the given problem.'

error_code = 27

exception fitbenchmarking.utils.exceptions.**IncompatibleTableError**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates that selected cost function and table are not compatible

class_message = 'The table type selected is not compatible with the selected cost function'

error_code = 18

exception fitbenchmarking.utils.exceptions.IncorrectBoundsError(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates that *parameter_ranges* have been set incorrectly

class_message = 'Bounds for this problem are unable to be set, so this problem will be skipped.'

error_code = 19

exception fitbenchmarking.utils.exceptions.MaxRuntimeError(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates a minimizer has taken too long to run

class_message = 'Minimizer runtime exceeded maximum runtime'

error_code = 23

exception fitbenchmarking.utils.exceptions.MissingBoundsError(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates that *parameter_ranges* have not been set but are required

class_message = 'Bounds on all parameters are required to use this software.'

error_code = 20

exception fitbenchmarking.utils.exceptions.MissingSoftwareError(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates that the requirements for a software package are not available.

class_message = 'Missing dependencies for fit.'

error_code = 5

exception fitbenchmarking.utils.exceptions.NoAnalyticJacobian(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates when no Jacobian data files can be found

class_message = 'Could not find Jacobian data files'

error_code = 12

exception fitbenchmarking.utils.exceptions.NoControllerError(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates a controller could not be found

class_message = 'Could not find controller.'

error_code = 6

exception fitbenchmarking.utils.exceptions.NoDataError(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates that no data could be found.

class_message = 'No data found.'

error_code = 8

exception fitbenchmarking.utils.exceptions.**NoHessianError**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicated a problem with Hessian import

class_message = 'Could not find Hessian class'

error_code = 22

exception fitbenchmarking.utils.exceptions.**NoJacobianError**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates a problem with the Jacobian import.

class_message = 'Could not find Jacobian class'

error_code = 11

exception fitbenchmarking.utils.exceptions.**NoParserError**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates a parser could not be found.

class_message = 'Could not find parser.'

error_code = 4

exception fitbenchmarking.utils.exceptions.**NoResultsError**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates a problem with the fitting problem.

class_message = 'FitBenchmarking ran with no results'

error_code = 14

exception fitbenchmarking.utils.exceptions.**OptionsError**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates an error during processing options.

class_message = 'Failed to process options.'

error_code = 2

exception fitbenchmarking.utils.exceptions.**ParsingError**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates an error during parsing.

class_message = 'Could not parse problem.'

error_code = 3

exception fitbenchmarking.utils.exceptions.**PlottingError**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates an error during plotting results

class_message = 'An error occurred during plotting.'

error_code = 21

exception fitbenchmarking.utils.exceptions.**UnknownMinimizerError**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates that the controller does not support a given minimizer given the current "algorithm_type" option set.

class_message = 'Minimizer cannot be run with Controller with current "algorithm_type" option set.'

error_code = 9

exception fitbenchmarking.utils.exceptions.**UnknownTableError**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates a problem with the fitting problem.

class_message = 'Set table option could not be found'

error_code = 13

exception fitbenchmarking.utils.exceptions.**UnsupportedMinimizerError**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates that the controller does not support a given minimizer.

class_message = 'FitBenchmarking ran with no results'

error_code = 15

exception fitbenchmarking.utils.exceptions.**ValidationException**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

This should be subclassed to indicate any validation errors that preclude a combination from running.

class_message = 'An error occured while verifying controller.'

fitbenchmarking.utils.fitbm_result module

FitBenchmarking results object

class fitbenchmarking.utils.fitbm_result.**FittingResult**(*options*, *cost_func*, *jac*, *hess*, *initial_params*, *params*, *name=None*, *chi_sq=None*, *runtime=None*, *software=None*, *minimizer=None*, *error_flag=None*, *algorithm_type=None*, *dataset_id=None*)
Bases: `object`

Minimal definition of a class to hold results from a fitting problem test.

modified_minimizer_name(*with_software: bool = False*) → str
Get a minimizer name which contains jacobian and hessian information. Optionally also include the software.

Parameters **with_software** (*bool, optional*) – Add software to the name, defaults to False

Returns A name for the result combination

Return type str

property norm_acc
Getting function for norm_acc attribute

Returns normalised accuracy value

Return type float

property norm_runtime
Getting function for norm_runtime attribute

Returns normalised runtime value

Return type float

sanitised_min_name(*with_software=False*)
Sanitise the modified minimizer name into one which can be used as a filename.

Returns sanitised name

Return type str

property sanitised_name
Sanitise the problem name into one which can be used as a filename.

Returns sanitised name

Return type str

fitbenchmarking.utils.log module

Utility functions to support logging for the fitbenchmarking project.

fitbenchmarking.utils.log.**get_logger**(*name='fitbenchmarking'*)
Get the unique logger for the given name. This is a straight pass through but will be more intutive for people who have not used python logging.

Parameters **name** (*str, optional*) – Name of the logger to use, defaults to 'fitbenchmarking'

Returns The named logger

Return type logging.Logger

fitbenchmarking.utils.log.**setup_logger**(*log_file='./fitbenchmarking.log'*, *name='fitbenchmarking'*, *append=False*, *level='INFO'*)
Define the location and style of the log file.

Parameters

- **log_file** (*str, optional*) – path to the log file, defaults to './fitbenchmarking.log'
- **name** (*str, optional*) – The name of the logger to run the setup for, defaults to fitbenchmarking
- **append** (*bool, optional*) – Whether to append to the log or create a new one, defaults to False
- **level** (*str, optional*) – The level of error to print, defaults to 'INFO'

fitbenchmarking.utils.misc module

Miscellaneous functions and utilities used in fitting benchmarking.

fitbenchmarking.utils.misc.**get_css**(*options*, *working_directory*)
Returns the path of the local css folder

Parameters **working_directory** (*string*) – location of current directory

Returns A dictionary containing relative links to the local css directory

Return type dict of strings

fitbenchmarking.utils.misc.**get_js**(*options*, *working_directory*)
Returns the path of the local js folder

Parameters **working_directory** (*string*) – location of current directory

Returns A dictionary containing relative links to the local js directory

Return type dict of strings

fitbenchmarking.utils.misc.**get_problem_files**(*data_dir*)

Gets all the problem definition files from the specified problem set directory.

Parameters **data_dir** (*str*) – directory containing the problems

Returns array containing of paths to the problems e.g. In NIST we would have [low_difficulty/file1.txt, . . . , . . .]

Return type list of str

fitbenchmarking.utils.options module

This file will handle all interaction with the options configuration file.

class fitbenchmarking.utils.options.**Options**(*file_name=None*, *results_directory: str = ''*)

Bases: object

An options class to store and handle all options for fitbenchmarking

```
DEFAULTS = {'FITTING': {'algorithm_type': ['all'], 'cost_func_type':
['weighted_nlls'], 'hes_method': ['default'], 'jac_method': ['scipy'],
'max_runtime': 600, 'num_runs': 5, 'software': ['scipy', 'scipy_ls']}, 'HESSIAN':
{'analytic': ['default'], 'default': ['default'], 'numdifftools': ['central'],
'scipy': ['2-point']}, 'JACOBIAN': {'analytic': ['default'], 'default':
['default'], 'numdifftools': ['central'], 'scipy': ['2-point']}, 'LOGGING':
{'append': False, 'external_output': 'log_only', 'file_name':
'fitbenchmarking.log', 'level': 'INFO'}, 'MINIMIZERS': {'bumps': ['amoeba',
'lm-bumps', 'newton', 'mp'], 'dfo': ['dfogn', 'dfols'], 'gradient_free':
['HillClimbingOptimizer', 'RepulsingHillClimbingOptimizer',
'SimulatedAnnealingOptimizer', 'RandomSearchOptimizer',
'RandomRestartHillClimbingOptimizer', 'RandomAnnealingOptimizer',
'ParallelTemperingOptimizer', 'ParticleSwarmOptimizer',
'EvolutionStrategyOptimizer'], 'gsl': ['lmsder', 'lmder', 'nmsimplex',
'nmsimplex2', 'conjugate_pr', 'conjugate_fr', 'vector_bfgs', 'vector_bfgs2',
'steepest_descent'], 'levmar': ['levmar'], 'mantid': ['BFGS', 'Conjugate gradient
(Fletcher-Reeves imp.)', 'Conjugate gradient (Polak-Ribiere imp.)', 'Damped
GaussNewton', 'Levenberg-Marquardt', 'Levenberg-MarquardtMD', 'Simplex',
'SteepestDescent', 'Trust Region'], 'matlab': ['Nelder-Mead Simplex'],
'matlab_curve': ['Levenberg-Marquardt', 'Trust-Region'], 'matlab_opt':
['levenberg-marquardt', 'trust-region-reflective'], 'matlab_stats':
['Levenberg-Marquardt'], 'minuit': ['minuit'], 'ralfit': ['gn', 'gn_reg',
'hybrid', 'hybrid_reg'], 'scipy': ['Nelder-Mead', 'Powell', 'CG', 'BFGS',
'Newton-CG', 'L-BFGS-B', 'TNC', 'SLSQP'], 'scipy_go': ['differential_evolution',
'dual_annealing'], 'scipy_ls': ['lm-scipy', 'trf', 'dogbox']}, 'OUTPUT':
{'results_dir': 'fitbenchmarking_results'}, 'PLOTTING': {'cmap_range': [0.2, 0.8],
'colour_map': 'magma_r', 'colour_ulim': 100, 'comparison_mode': 'both',
'make_plots': True, 'table_type': ['acc', 'runtime', 'compare', 'local_min']}}
```

```
DEFAULT_FITTING = {'algorithm_type': ['all'], 'cost_func_type': ['weighted_nlls'],
'hes_method': ['default'], 'jac_method': ['scipy'], 'max_runtime': 600,
'num_runs': 5, 'software': ['scipy', 'scipy_ls']}
```

```
DEFAULT_HESSIAN = {'analytic': ['default'], 'default': ['default'],
'numdifftools': ['central'], 'scipy': ['2-point']}
```

```
DEFAULT_JACOBIAN = {'analytic': ['default'], 'default': ['default'],
'numdifftools': ['central'], 'scipy': ['2-point']}

DEFAULT_LOGGING = {'append': False, 'external_output': 'log_only', 'file_name':
'fitbenchmarking.log', 'level': 'INFO'}

DEFAULT_MINIMZERS = {'bumps': ['amoeba', 'lm-bumps', 'newton', 'mp'], 'dfo':
['dfogn', 'dfols'], 'gradient_free': ['HillClimbingOptimizer',
'RepulsingHillClimbingOptimizer', 'SimulatedAnnealingOptimizer',
'RandomSearchOptimizer', 'RandomRestartHillClimbingOptimizer',
'RandomAnnealingOptimizer', 'ParallelTemperingOptimizer', 'ParticleSwarmOptimizer',
'EvolutionStrategyOptimizer'], 'gsl': ['lmsder', 'lmder', 'nmsimplex',
'nmsimplex2', 'conjugate_pr', 'conjugate_fr', 'vector_bfgs', 'vector_bfgs2',
'steepest_descent'], 'levmar': ['levmar'], 'mantid': ['BFGS', 'Conjugate gradient
(Fletcher-Reeves imp.)', 'Conjugate gradient (Polak-Ribiere imp.)', 'Damped
GaussNewton', 'Levenberg-Marquardt', 'Levenberg-MarquardtMD', 'Simplex',
'SteepestDescent', 'Trust Region'], 'matlab': ['Nelder-Mead Simplex'],
'matlab_curve': ['Levenberg-Marquardt', 'Trust-Region'], 'matlab_opt':
['levenberg-marquardt', 'trust-region-reflective'], 'matlab_stats':
['Levenberg-Marquardt'], 'minuit': ['minuit'], 'ralfit': ['gn', 'gn_reg',
'hybrid', 'hybrid_reg'], 'scipy': ['Nelder-Mead', 'Powell', 'CG', 'BFGS',
'Newton-CG', 'L-BFGS-B', 'TNC', 'SLSQP'], 'scipy_go': ['differential_evolution',
'dual_annealing'], 'scipy_ls': ['lm-scipy', 'trf', 'dogbox']}

DEFAULT_OUTPUT = {'results_dir': 'fitbenchmarking_results'}

DEFAULT_PLOTTING = {'cmap_range': [0.2, 0.8], 'colour_map': 'magma_r',
'colour_ulim': 100, 'comparison_mode': 'both', 'make_plots': True, 'table_type':
['acc', 'runtime', 'compare', 'local_min']}
```

```
VALID = {'FITTING': {'algorithm_type': ['all', 'ls', 'deriv_free', 'general',
'simplex', 'trust_region', 'levenberg-marquardt', 'gauss_newton', 'bfgs',
'conjugate_gradient', 'steepest_descent', 'global_optimization'], 'cost_func_type':
['nlls', 'weighted_nlls', 'hellinger_nlls', 'poisson'], 'hes_method': ['scipy',
'analytic', 'default', 'numdifftools'], 'jac_method': ['scipy', 'analytic',
'default', 'numdifftools'], 'software': ['bumps', 'dfo', 'gradient_free', 'gsl',
'levmar', 'mantid', 'matlab', 'matlab_curve', 'matlab_opt', 'matlab_stats',
'minuit', 'ralfit', 'scipy', 'scipy_ls', 'scipy_go']}, 'HESSIAN': {'analytic':
['default'], 'default': ['default'], 'numdifftools': ['central', 'complex',
'multicomplex', 'forward', 'backward'], 'scipy': ['2-point', '3-point', 'cs']},
'JACOBIAN': {'analytic': ['default'], 'default': ['default'], 'numdifftools':
['central', 'complex', 'multicomplex', 'forward', 'backward'], 'scipy': ['2-point',
'3-point', 'cs']}, 'LOGGING': {'append': [True, False], 'external_output':
['debug', 'display', 'log_only'], 'level': ['NOTSET', 'DEBUG', 'INFO', 'WARNING',
'ERROR', 'CRITICAL']}, 'MINIMIZERS': {'bumps': ['amoeba', 'lm-bumps', 'newton',
'de', 'mp'], 'dfo': ['dfogn', 'dfols'], 'gradient_free': ['HillClimbingOptimizer',
'RepulsingHillClimbingOptimizer', 'SimulatedAnnealingOptimizer',
'RandomSearchOptimizer', 'RandomRestartHillClimbingOptimizer',
'RandomAnnealingOptimizer', 'ParallelTemperingOptimizer', 'ParticleSwarmOptimizer',
'EvolutionStrategyOptimizer', 'BayesianOptimizer', 'TreeStructuredParzenEstimators',
'DecisionTreeOptimizer'], 'gsl': ['lmsder', 'lmder', 'nmsimplex', 'nmsimplex2',
'conjugate_pr', 'conjugate_fr', 'vector_bfgs', 'vector_bfgs2', 'steepest_descent'],
'levmar': ['levmar'], 'mantid': ['BFGS', 'Conjugate gradient (Fletcher-Reeves
imp.)', 'Conjugate gradient (Polak-Ribiere imp.)', 'Damped GaussNewton',
'Levenberg-Marquardt', 'Levenberg-MarquardtMD', 'Simplex', 'SteepestDescent', 'Trust
Region', 'FABADA'], 'matlab': ['Nelder-Mead Simplex'], 'matlab_curve':
['Levenberg-Marquardt', 'Trust-Region'], 'matlab_opt': ['levenberg-marquardt',
'trust-region-reflective'], 'matlab_stats': ['Levenberg-Marquardt'], 'minuit':
['minuit'], 'ralfit': ['gn', 'gn_reg', 'hybrid', 'hybrid_reg'], 'scipy':
['Nelder-Mead', 'Powell', 'CG', 'BFGS', 'Newton-CG', 'L-BFGS-B', 'TNC', 'SLSQP'],
'scipy_go': ['differential_evolution', 'shgo', 'dual_annealing'], 'scipy_ls':
['lm-scipy', 'trf', 'dogbox']}, 'OUTPUT': {}, 'PLOTTING': {'colour_map': ['Accent',
'Accent_r', 'Blues', 'Blues_r', 'BrBG', 'BrBG_r', 'BuGn', 'BuGn_r', 'BuPu',
'BuPu_r', 'CMRmap', 'CMRmap_r', 'Dark2', 'Dark2_r', 'GnBu', 'GnBu_r', 'Greens',
'Greens_r', 'Greys', 'Greys_r', 'OrRd', 'OrRd_r', 'Oranges', 'Oranges_r', 'PRGn',
'PRGn_r', 'Paired', 'Paired_r', 'Pastel1', 'Pastel1_r', 'Pastel2', 'Pastel2_r',
'PiYG', 'PiYG_r', 'PuBu', 'PuBuGn', 'PuBuGn_r', 'PuBu_r', 'PuOr', 'PuOr_r', 'PuRd',
'PuRd_r', 'Purples', 'Purples_r', 'RdBu', 'RdBu_r', 'RdGy', 'RdGy_r', 'RdPu',
'RdPu_r', 'RdYlBu', 'RdYlBu_r', 'RdYlGn', 'RdYlGn_r', 'Reds', 'Reds_r', 'Set1',
'Set1_r', 'Set2', 'Set2_r', 'Set3', 'Set3_r', 'Spectral', 'Spectral_r', 'Wistia',
'Wistia_r', 'YlGn', 'YlGnBu', 'YlGnBu_r', 'YlGn_r', 'YlOrBr', 'YlOrBr_r', 'YlOrRd',
'YlOrRd_r', 'afmhot', 'afmhot_r', 'autumn', 'autumn_r', 'binary', 'binary_r',
'bone', 'bone_r', 'brg', 'brg_r', 'bwr', 'bwr_r', 'cividis', 'cividis_r', 'cool',
'cool_r', 'coolwarm', 'coolwarm_r', 'copper', 'copper_r', 'cubehelix',
'cubehelix_r', 'flag', 'flag_r', 'gist_earth', 'gist_earth_r', 'gist_gray',
'gist_gray_r', 'gist_heat', 'gist_heat_r', 'gist_ncar', 'gist_ncar_r',
'gist_rainbow', 'gist_rainbow_r', 'gist_stern', 'gist_stern_r', 'gist_yarg',
'gist_yarg_r', 'gnuplot', 'gnuplot2', 'gnuplot2_r', 'gnuplot_r', 'gray', 'gray_r',
'hot', 'hot_r', 'hsv', 'hsv_r', 'inferno', 'inferno_r', 'jet', 'jet_r', 'magma',
'magma_r', 'nipy_spectral', 'nipy_spectral_r', 'ocean', 'ocean_r', 'pink', 'pink_r',
'plasma', 'plasma_r', 'prism', 'prism_r', 'rainbow', 'rainbow_r', 'seismic',
'seismic_r', 'spring', 'spring_r', 'summer', 'summer_r', 'tab10', 'tab10_r',
'tab20', 'tab20_r', 'tab20b', 'tab20b_r', 'tab20c', 'tab20c_r', 'terrain',
'terrain_r', 'turbo', 'turbo_r', 'twilight', 'twilight_r', 'twilight_shifted',
'twilight_shifted_r', 'viridis', 'viridis_r', 'winter', 'winter_r'],
'comparison_mode': ['abs', 'rel', 'both'], 'make_plots': [True, False],
'table_type': ['acc', 'runtime', 'compare', 'local_min']}}
```

```
VALID_FITTING = {'algorithm_type': ['all', 'ls', 'deriv_free', 'general',
'simplex', 'trust_region', 'levenberg-marquardt', 'gauss_newton', 'bfgs',
'conjugate_gradient', 'steepest_descent', 'global_optimization'], 'cost_func_type':
['nlls', 'weighted_nlls', 'hellinger_nlls', 'poisson'], 'hes_method': ['scipy',
'analytic', 'default', 'numdifftools'], 'jac_method': ['scipy', 'analytic',
'default', 'numdifftools'], 'software': ['bumps', 'dfo', 'gradient_free', 'gsl',
'levmar', 'mantid', 'matlab', 'matlab_curve', 'matlab_opt', 'matlab_stats',
'minuit', 'ralfit', 'scipy', 'scipy_ls', 'scipy_go']}

VALID_HESSIAN = {'analytic': ['default'], 'default': ['default'], 'numdifftools':
['central', 'complex', 'multicomplex', 'forward', 'backward'], 'scipy': ['2-point',
'3-point', 'cs']}

VALID_JACOBIAN = {'analytic': ['default'], 'default': ['default'], 'numdifftools':
['central', 'complex', 'multicomplex', 'forward', 'backward'], 'scipy': ['2-point',
'3-point', 'cs']}

VALID_LOGGING = {'append': [True, False], 'external_output': ['debug', 'display',
'log_only'], 'level': ['NOTSET', 'DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL']}

VALID_MINIMIZERS = {'bumps': ['amoeba', 'lm-bumps', 'newton', 'de', 'mp'], 'dfo':
['dfogn', 'dfols'], 'gradient_free': ['HillClimbingOptimizer',
'RepulsingHillClimbingOptimizer', 'SimulatedAnnealingOptimizer',
'RandomSearchOptimizer', 'RandomRestartHillClimbingOptimizer',
'RandomAnnealingOptimizer', 'ParallelTemperingOptimizer', 'ParticleSwarmOptimizer',
'EvolutionStrategyOptimizer', 'BayesianOptimizer', 'TreeStructuredParzenEstimators',
'DecisionTreeOptimizer'], 'gsl': ['lmsder', 'lmder', 'nmsimplex', 'nmsimplex2',
'conjugate_pr', 'conjugate_fr', 'vector_bfgs', 'vector_bfgs2', 'steepest_descent'],
'levmar': ['levmar'], 'mantid': ['BFGS', 'Conjugate gradient (Fletcher-Reeves
imp.)', 'Conjugate gradient (Polak-Ribiere imp.)', 'Damped GaussNewton',
'Levenberg-Marquardt', 'Levenberg-MarquardtMD', 'Simplex', 'SteepestDescent', 'Trust
Region', 'FABADA'], 'matlab': ['Nelder-Mead Simplex'], 'matlab_curve':
['Levenberg-Marquardt', 'Trust-Region'], 'matlab_opt': ['levenberg-marquardt',
'trust-region-reflective'], 'matlab_stats': ['Levenberg-Marquardt'], 'minuit':
['minuit'], 'ralfit': ['gn', 'gn_reg', 'hybrid', 'hybrid_reg'], 'scipy':
['Nelder-Mead', 'Powell', 'CG', 'BFGS', 'Newton-CG', 'L-BFGS-B', 'TNC', 'SLSQP'],
'scipy_go': ['differential_evolution', 'shgo', 'dual_annealing'], 'scipy_ls':
['lm-scipy', 'trf', 'dogbox']}

VALID_OUTPUT = {}
```

VALID_PLOTTING = {'colour_map': ['Accent', 'Accent_r', 'Blues', 'Blues_r', 'BrBG', 'BrBG_r', 'BuGn', 'BuGn_r', 'BuPu', 'BuPu_r', 'CMRmap', 'CMRmap_r', 'Dark2', 'Dark2_r', 'GnBu', 'GnBu_r', 'Greens', 'Greens_r', 'Greys', 'Greys_r', 'OrRd', 'OrRd_r', 'Oranges', 'Oranges_r', 'PRGn', 'PRGn_r', 'Paired', 'Paired_r', 'Pastel1', 'Pastel1_r', 'Pastel2', 'Pastel2_r', 'PiYG', 'PiYG_r', 'PuBu', 'PuBuGn', 'PuBuGn_r', 'PuBu_r', 'PuOr', 'PuOr_r', 'PuRd', 'PuRd_r', 'Purples', 'Purples_r', 'RdBu', 'RdBu_r', 'RdGy', 'RdGy_r', 'RdPu', 'RdPu_r', 'RdYlBu', 'RdYlBu_r', 'RdYlGn', 'RdYlGn_r', 'Reds', 'Reds_r', 'Set1', 'Set1_r', 'Set2', 'Set2_r', 'Set3', 'Set3_r', 'Spectral', 'Spectral_r', 'Wistia', 'Wistia_r', 'YlGn', 'YlGnBu', 'YlGnBu_r', 'YlGn_r', 'YlOrBr', 'YlOrBr_r', 'YlOrRd', 'YlOrRd_r', 'afmhot', 'afmhot_r', 'autumn', 'autumn_r', 'binary', 'binary_r', 'bone', 'bone_r', 'brg', 'brg_r', 'bwr', 'bwr_r', 'cividis', 'cividis_r', 'cool', 'cool_r', 'coolwarm', 'coolwarm_r', 'copper', 'copper_r', 'cubehelix', 'cubehelix_r', 'flag', 'flag_r', 'gist_earth', 'gist_earth_r', 'gist_gray', 'gist_gray_r', 'gist_heat', 'gist_heat_r', 'gist_ncar', 'gist_ncar_r', 'gist_rainbow', 'gist_rainbow_r', 'gist_stern', 'gist_stern_r', 'gist_yarg', 'gist_yarg_r', 'gnuplot', 'gnuplot2', 'gnuplot2_r', 'gnuplot_r', 'gray', 'gray_r', 'hot', 'hot_r', 'hsv', 'hsv_r', 'inferno', 'inferno_r', 'jet', 'jet_r', 'magma', 'magma_r', 'nipy_spectral', 'nipy_spectral_r', 'ocean', 'ocean_r', 'pink', 'pink_r', 'plasma', 'plasma_r', 'prism', 'prism_r', 'rainbow', 'rainbow_r', 'seismic', 'seismic_r', 'spring', 'spring_r', 'summer', 'summer_r', 'tab10', 'tab10_r', 'tab20', 'tab20_r', 'tab20b', 'tab20b_r', 'tab20c', 'tab20c_r', 'terrain', 'terrain_r', 'turbo', 'turbo_r', 'twilight', 'twilight_r', 'twilight_shifted', 'twilight_shifted_r', 'viridis', 'viridis_r', 'winter', 'winter_r'], 'comparison_mode': ['abs', 'rel', 'both'], 'make_plots': [True, False], 'table_type': ['acc', 'runtime', 'compare', 'local_min']}

VALID_SECTIONS = ['MINIMIZERS', 'FITTING', 'JACOBIAN', 'HESSIAN', 'PLOTTING', 'OUTPUT', 'LOGGING']

property minimizers
Returns the minimizers in a software package

read_value(*func*, *option*)
Helper function which loads in the value

Parameters

- **func** (*callable*) – configparser function
- **option** (*str*) – option to be read for file

Returns value of the option

Return type list/str/int/bool

reset()
Resets options object when running multiple problem groups.

property results_dir: str
Returns the directory to store the results in.

write(*file_name*)
Write the contents of the options object to a new options file.

Parameters **file_name** (*str*) – The path to the new options file

write_to_stream(*file_object*)
Write the contents of the options object to a file object.

fitbenchmarking.utils.options.**read_list**(*s*)
Utility function to allow lists to be read by the config parser

Parameters **s** (*string*) – string to convert to a list

Returns list of items

Return type list of str

fitbenchmarking.utils.options.**read_range**(*s*)
Utility function to allow ranges to be read by the config parser

Parameters **s** (*string*) – string to convert to a list

Returns two element list [lower_lim, upper lim]

Return type list

fitbenchmarking.utils.output_grabber module

This file will capture output that would be printed to the terminal

class fitbenchmarking.utils.output_grabber.**OutputGrabber**(*options*)
Bases: object

Class used to grab outputs for stdout and stderr

class fitbenchmarking.utils.output_grabber.**StreamGrabber**(*stream*)
Bases: object

Class used to grab an output stream.

escape_char = '\x08'

readOutput()
Read the stream data (one byte at a time) and save the text in *capturedtext*.

start()
Start capturing the stream data.

stop()
Stop capturing the stream data and save the text in *capturedtext*.

fitbenchmarking.utils.timer module

Implements the TimerWithMaxTime class used for checking the 'max_runtime' is not exceeded.

class fitbenchmarking.utils.timer.**TimerWithMaxTime**(*max_runtime: float*)
Bases: object

A timer class used for checking if the 'max_runtime' is exceeded when executing the fits.

check_elapsed_time() → None
Checks whether the max runtime has been exceeded. Raises a MaxRuntimeError exception if it has been exceeded. Otherwise, it carries on.

reset() → None
Resets the timer so it can be used for timing a different fit combination.

start() → None
Starts the timer by recording the current time.

stop() → None
Stops the timer if it is timing something. The elapsed time since starting the timer is added onto the total elapsed time.

fitbenchmarking.utils.write_files module

Utility decorator function for saving and writing files.

fitbenchmarking.utils.write_files.**write_file**(*function*)
A decorator function used for catching exceptions which can happen specifically when writing files. It will log a useful error message if the problem is identified.

Parameters **function** (*A callable function.*) – A callable function which writes files.

Returns A callable wrapped function which writes files.

Return type A callable function.

Module contents

Module contents

BIBLIOGRAPHY

[Nocedal] Jorge Nocedal, Stephen J. Wright (2006), Numerical Optimization

[Singer] Saša Singer, John Nelder (2009) Nelder-Mead algorithm. Scholarpedia, 4(7):2928.

[Nocedal] Jorge Nocedal, Stephen J. Wright (2006), Numerical Optimization

[Poczos] Barnabas Poczos, Ryan Tibshirani (2012), Lecture 10: Optimization, School of Computer Science, Carnegie Mellon University

[Floater] Michael S. Floater (2018), Lecture 13: Non-linear least squares and the Gauss-Newton method, University of Oslo

[Nocedal] Jorge Nocedal, Stephen J. Wright (2006), Numerical Optimization

[Ranganathan] Ananth Ranganathan (2004), The Levenberg-Marquardt Algorithm, University of California, Santa Barbara

PYTHON MODULE INDEX

f

fitbenchmarking, 141
fitbenchmarking.cli, 78
fitbenchmarking.cli.exception_handler, 77
fitbenchmarking.cli.main, 77
fitbenchmarking.controllers, 97
fitbenchmarking.controllers.base_controller, 78
fitbenchmarking.controllers.bumps_controller, 80
fitbenchmarking.controllers.controller_factory, 81
fitbenchmarking.controllers.dfo_controller, 82
fitbenchmarking.controllers.gradient_free_controller, 83
fitbenchmarking.controllers.gsl_controller, 84
fitbenchmarking.controllers.levmar_controller, 85
fitbenchmarking.controllers.mantid_controller, 86
fitbenchmarking.controllers.matlab_controller, 88
fitbenchmarking.controllers.matlab_curve_controller, 89
fitbenchmarking.controllers.matlab_mixin, 90
fitbenchmarking.controllers.matlab_opt_controller, 90
fitbenchmarking.controllers.matlab_stats_controller, 91
fitbenchmarking.controllers.minuit_controller, 92
fitbenchmarking.controllers.ralfit_controller, 93
fitbenchmarking.controllers.scipy_controller, 94
fitbenchmarking.controllers.scipy_go_controller, 95
fitbenchmarking.controllers.scipy_ls_controller, 96
fitbenchmarking.core, 102
fitbenchmarking.core.fitting_benchmarking, 97
fitbenchmarking.core.results_output, 101
fitbenchmarking.cost_func, 108
fitbenchmarking.cost_func.base_cost_func, 102
fitbenchmarking.cost_func.cost_func_factory, 103
fitbenchmarking.cost_func.hellinger_nlls_cost_func, 104
fitbenchmarking.cost_func.nlls_base_cost_func, 104
fitbenchmarking.cost_func.nlls_cost_func, 105
fitbenchmarking.cost_func.poisson_cost_func, 106
fitbenchmarking.cost_func.weighted_nlls_cost_func, 107
fitbenchmarking.hessian, 110
fitbenchmarking.hessian.analytic_hessian, 108
fitbenchmarking.hessian.base_hessian, 108
fitbenchmarking.hessian.hessian_factory, 109
fitbenchmarking.hessian.scipy_hessian, 109
fitbenchmarking.jacobian, 112
fitbenchmarking.jacobian.analytic_jacobian, 110
fitbenchmarking.jacobian.base_jacobian, 110
fitbenchmarking.jacobian.default_jacobian, 111
fitbenchmarking.jacobian.jacobian_factory, 111
fitbenchmarking.jacobian.scipy_jacobian, 111
fitbenchmarking.parsing, 117
fitbenchmarking.parsing.base_parser, 112
fitbenchmarking.parsing.cutest_parser, 112
fitbenchmarking.parsing.fitbenchmark_parser, 113
fitbenchmarking.parsing.fitting_problem, 113
fitbenchmarking.parsing.ivp_parser, 115
fitbenchmarking.parsing.mantid_parser, 115
fitbenchmarking.parsing.nist_data_functions, 115
fitbenchmarking.parsing.nist_parser, 116
fitbenchmarking.parsing.parser_factory, 117
fitbenchmarking.results_processing, 128

fitbenchmarking.results_processing.acc_table, 117
fitbenchmarking.results_processing.base_table, 118
fitbenchmarking.results_processing.compare_table, 121
fitbenchmarking.results_processing.fitting_report, 122
fitbenchmarking.results_processing.local_min_table, 122
fitbenchmarking.results_processing.performance_profiler, 124
fitbenchmarking.results_processing.plots, 125
fitbenchmarking.results_processing.problem_summary_page, 126
fitbenchmarking.results_processing.runtime_table, 126
fitbenchmarking.results_processing.tables, 127
fitbenchmarking.utils, 141
fitbenchmarking.utils.create_dirs, 128
fitbenchmarking.utils.debug, 129
fitbenchmarking.utils.exceptions, 129
fitbenchmarking.utils.fitbm_result, 133
fitbenchmarking.utils.log, 134
fitbenchmarking.utils.misc, 134
fitbenchmarking.utils.options, 135
fitbenchmarking.utils.output_grabber, 140
fitbenchmarking.utils.timer, 140
fitbenchmarking.utils.write_files, 141

INDEX

A

AccTable (*class in fitbenchmarking.results_processing.acc_table*), 117
additional_info (*fitbenchmarking.parsing.fitting_problem.FittingProblem attribute*), 113
algorithm_check (*fitbenchmarking.controllers.base_controller.Controller attribute*), 78
algorithm_check (*fitbenchmarking.controllers.bumps_controller.BumpsController attribute*), 80
algorithm_check (*fitbenchmarking.controllers.dfo_controller.DFOController attribute*), 82
algorithm_check (*fitbenchmarking.controllers.gradient_free_controller.GradientFreeController attribute*), 83
algorithm_check (*fitbenchmarking.controllers.gsl_controller.GSLController attribute*), 84
algorithm_check (*fitbenchmarking.controllers.levmar_controller.LevmarController attribute*), 85
algorithm_check (*fitbenchmarking.controllers.mantid_controller.MantidController attribute*), 86
algorithm_check (*fitbenchmarking.controllers.matlab_controller.MatlabController attribute*), 88
algorithm_check (*fitbenchmarking.controllers.matlab_curve_controller.MatlabCurveController attribute*), 89
algorithm_check (*fitbenchmarking.controllers.matlab_opt_controller.MatlabOptController attribute*), 90
algorithm_check (*fitbenchmarking.controllers.matlab_stats_controller.MatlabStatsController attribute*), 91
algorithm_check (*fitbenchmarking.controllers.minuit_controller.MinuitController attribute*), 92
algorithm_check (*fitbenchmarking.controllers.ralfit_controller.RALFitController attribute*), 93
algorithm_check (*fitbenchmarking.controllers.scipy_controller.ScipyController attribute*), 94
algorithm_check (*fitbenchmarking.controllers.scipy_go_controller.ScipyGOController attribute*), 95
algorithm_check (*fitbenchmarking.controllers.scipy_ls_controller.ScipyLSController attribute*), 96
Analytic (*class in fitbenchmarking.hessian.analytic_hessian*), 108
Analytic (*class in fitbenchmarking.jacobian.analytic_jacobian*), 110

B

BaseNLLSCostFunc (*class in fitbenchmarking.cost_func.nlls_base_cost_func*), 104
benchmark() (*in module fitbenchmarking.core.fitting_benchmarking*), 97
best_fit_plot_options (*fitbenchmarking.results_processing.plots.Plot attribute*), 125
BumpsController (*class in fitbenchmarking.controllers.bumps_controller*), 80

C

check_attributes() (*fitbenchmarking.controllers.base_controller.Controller method*), 78
check_bounds_respected() (*fitbenchmarking.controllers.base_controller.Controller method*), 78
check_elapsed_time() (*fitbenchmarking.utils.timer.TimerWithMaxTime method*), 140
check_minimizer_bounds() (*fitbenchmarking.controllers.base_controller.Controller method*), 78

class_message (*fitbenchmarking.utils.exceptions.ControllerAttributeError attribute*), 129
class_message (*fitbenchmarking.utils.exceptions.CostFuncError attribute*), 129
class_message (*fitbenchmarking.utils.exceptions.FilepathTooLongError attribute*), 129
class_message (*fitbenchmarking.utils.exceptions.FitBenchmarkException attribute*), 130
class_message (*fitbenchmarking.utils.exceptions.FittingProblemError attribute*), 130
class_message (*fitbenchmarking.utils.exceptions.IncompatibleHessianError attribute*), 130
class_message (*fitbenchmarking.utils.exceptions.IncompatibleJacobianError attribute*), 130
class_message (*fitbenchmarking.utils.exceptions.IncompatibleMinimizerError attribute*), 130
class_message (*fitbenchmarking.utils.exceptions.IncompatibleProblemError attribute*), 130
class_message (*fitbenchmarking.utils.exceptions.IncompatibleTableError attribute*), 130
class_message (*fitbenchmarking.utils.exceptions.IncorrectBoundsError attribute*), 131
class_message (*fitbenchmarking.utils.exceptions.MaxRuntimeError attribute*), 131
class_message (*fitbenchmarking.utils.exceptions.MissingBoundsError attribute*), 131
class_message (*fitbenchmarking.utils.exceptions.MissingSoftwareError attribute*), 131
class_message (*fitbenchmarking.utils.exceptions.NoAnalyticJacobian attribute*), 131
class_message (*fitbenchmarking.utils.exceptions.NoControllerError attribute*), 131
class_message (*fitbenchmarking.utils.exceptions.NoDataError attribute*), 131
class_message (*fitbenchmarking.utils.exceptions.NoHessianError attribute*), 132
class_message (*fitbenchmarking.utils.exceptions.NoJacobianError attribute*), 132
class_message (*fitbenchmarking.utils.exceptions.NoParserError attribute*), 132
class_message (*fitbenchmarking.utils.exceptions.NoResultsError attribute*), 132
class_message (*fitbenchmarking.utils.exceptions.OptionsError attribute*), 132
class_message (*fitbenchmarking.utils.exceptions.ParsingError attribute*), 132
class_message (*fitbenchmarking.utils.exceptions.PlottingError attribute*), 132
class_message (*fitbenchmarking.utils.exceptions.UnknownMinimizerError attribute*), 132
class_message (*fitbenchmarking.utils.exceptions.UnknownTableError attribute*), 133
class_message (*fitbenchmarking.utils.exceptions.UnsupportedMinimizerError attribute*), 133
class_message (*fitbenchmarking.utils.exceptions.ValidationException attribute*), 133
cleanup() (*fitbenchmarking.controllers.base_controller.Controller method*), 79
cleanup() (*fitbenchmarking.controllers.bumps_controller.BumpsController method*), 81
cleanup() (*fitbenchmarking.controllers.dfo_controller.DFOController method*), 82
cleanup() (*fitbenchmarking.controllers.gradient_free_controller.GradientFreeController method*), 84
cleanup() (*fitbenchmarking.controllers.gsl_controller.GSLController method*), 84
cleanup() (*fitbenchmarking.controllers.levmar_controller.LevmarController method*), 85
cleanup() (*fitbenchmarking.controllers.mantid_controller.MantidController method*), 87
cleanup() (*fitbenchmarking.controllers.matlab_controller.MatlabController method*), 88

cleanup() (*fitbenchmarking.controllers.matlab_curve_controller.MatlabCurveController method*), 89
cleanup() (*fitbenchmarking.controllers.matlab_opt_controller.MatlabOptController method*), 90
cleanup() (*fitbenchmarking.controllers.matlab_stats_controller.MatlabStatsController method*), 92
cleanup() (*fitbenchmarking.controllers.minuit_controller.MinuitController method*), 92
cleanup() (*fitbenchmarking.controllers.ralfit_controller.RALFitController method*), 93
cleanup() (*fitbenchmarking.controllers.scipy_controller.ScipyController method*), 95
cleanup() (*fitbenchmarking.controllers.scipy_go_controller.ScipyGOController method*), 96
cleanup() (*fitbenchmarking.controllers.scipy_ls_controller.ScipyLSController method*), 97
CompareTable (*class in fitbenchmarking.results_processing.compare_table*), 121
Controller (*class in fitbenchmarking.controllers.base_controller*), 78
controller_name (*fitbenchmarking.controllers.base_controller.Controller attribute*), 79
controller_name (*fitbenchmarking.controllers.gradient_free_controller.GradientFreeController attribute*), 84
controller_name (*fitbenchmarking.controllers.matlab_curve_controller.MatlabCurveController attribute*), 89
controller_name (*fitbenchmarking.controllers.matlab_opt_controller.MatlabOptController attribute*), 90
controller_name (*fitbenchmarking.controllers.matlab_stats_controller.MatlabStatsController attribute*), 92
controller_name (*fitbenchmarking.controllers.scipy_go_controller.ScipyGOController attribute*), 96
controller_name (*fitbenchmarking.controllers.scipy_ls_controller.ScipyLSController attribute*), 97
ControllerAttributeError, 129
ControllerFactory (*class in fitbenchmarking.controllers.controller_factory*), 81
correct_data() (*fitbenchmarking.parsing.fitting_problem.FittingProblem method*), 113
correct_data() (*in module fitbenchmarking.parsing.fitting_problem*), 115
COST_FUNCTION_MAP (*fitbenchmarking.controllers.mantid_controller.MantidController attribute*), 86
CostFunc (*class in fitbenchmarking.cost_func.base_cost_func*), 102
CostFuncError, 129
create() (*in module fitbenchmarking.results_processing.fitting_report*), 122
create() (*in module fitbenchmarking.results_processing.problem_summary_page*), 126
create_controller() (*fitbenchmarking.controllers.controller_factory.ControllerFactory static method*), 81
create_cost_func() (*in module fitbenchmarking.cost_func.cost_func_factory*), 103
create_directories() (*in module fitbenchmarking.core.results_output*), 101
create_hessian() (*in module fitbenchmarking.hessian.hessian_factory*), 109
create_jacobian() (*in module fitbenchmarking.jacobian.jacobian_factory*), 111
create_pandas_data_frame() (*fitbenchmarking.results_processing.base_table.Table method*), 118
create_parser() (*fitbenchmarking.parsing.parser_factory.ParserFactory static method*), 117
create_plot() (*in module fitbenchmarking.results_processing.performance_profiler*), 124
create_plots() (*in module fitbenchmarking.core.results_output*), 101
create_prob_group() (*in module fitbenchmarking.results_processing.fitting_report*), 122
create_problem_level_index() (*in module fitbenchmarking.core.results_output*), 101
create_results_dict() (*fitbenchmarking.results_processing.base_table.Table method*), 118
create_results_tables() (*in module fitbenchmarking.results_processing.tables*), 127
css() (*in module fitbenchmarking.utils.create_dirs*), 128
CutestParser (*class in fitbenchmarking.parsing.cutest_parser*), 112

D

data_e (*fitbenchmarking.parsing.fitting_problem.FittingProblem attribute*), 113
data_plot_options (*fitbenchmarking.results_processing.plots.Plot attribute*),

125
data_x (*fitbenchmarking.parsing.fitting_problem.FittingProblem attribute*), 113
data_y (*fitbenchmarking.parsing.fitting_problem.FittingProblem attribute*), 113
Default (*class in fitbenchmarking.jacobian.default_jacobian*), 111
DEFAULT_FITTING (*fitbenchmarking.utils.options.Options attribute*), 135
DEFAULT_HESSIAN (*fitbenchmarking.utils.options.Options attribute*), 135
DEFAULT_JACOBIAN (*fitbenchmarking.utils.options.Options attribute*), 135
DEFAULT_LOGGING (*fitbenchmarking.utils.options.Options attribute*), 136
DEFAULT_MINIMZERS (*fitbenchmarking.utils.options.Options attribute*), 136
DEFAULT_OUTPUT (*fitbenchmarking.utils.options.Options attribute*), 136
DEFAULT_PLOTTING (*fitbenchmarking.utils.options.Options attribute*), 136
DEFAULTS (*fitbenchmarking.utils.options.Options attribute*), 135
del_contents_of_dir() (*in module fitbenchmarking.utils.create_dirs*), 128
description (*fitbenchmarking.parsing.fitting_problem.FittingProblem attribute*), 113
DFOController (*class in fitbenchmarking.controllers.dfo_controller*), 82
display_str() (*fitbenchmarking.results_processing.base_table.Table method*), 118
display_str() (*fitbenchmarking.results_processing.compare_table.CompareTable method*), 121
display_str() (*fitbenchmarking.results_processing.local_min_table.LocalMinTable method*), 122

E

end_x (*fitbenchmarking.parsing.fitting_problem.FittingProblem attribute*), 113
equation (*fitbenchmarking.parsing.fitting_problem.FittingProblem attribute*), 113
error_code (*fitbenchmarking.utils.exceptions.ControllerAttributeError attribute*), 129
error_code (*fitbenchmarking.utils.exceptions.CostFuncError attribute*), 129
error_code (*fitbenchmarking.utils.exceptions.FilepathTooLongError attribute*), 130
error_code (*fitbenchmarking.utils.exceptions.FitBenchmarkException attribute*), 130
error_code (*fitbenchmarking.utils.exceptions.FittingProblemError attribute*), 130
error_code (*fitbenchmarking.utils.exceptions.IncompatibleHessianError attribute*), 130
error_code (*fitbenchmarking.utils.exceptions.IncompatibleJacobianError attribute*), 130
error_code (*fitbenchmarking.utils.exceptions.IncompatibleMinimizerError attribute*), 130
error_code (*fitbenchmarking.utils.exceptions.IncompatibleProblemError attribute*), 130
error_code (*fitbenchmarking.utils.exceptions.IncompatibleTableError attribute*), 131
error_code (*fitbenchmarking.utils.exceptions.IncorrectBoundsError attribute*), 131
error_code (*fitbenchmarking.utils.exceptions.MaxRuntimeError attribute*), 131
error_code (*fitbenchmarking.utils.exceptions.MissingBoundsError attribute*), 131
error_code (*fitbenchmarking.utils.exceptions.MissingSoftwareError attribute*), 131
error_code (*fitbenchmarking.utils.exceptions.NoAnalyticJacobian attribute*), 131
error_code (*fitbenchmarking.utils.exceptions.NoControllerError attribute*), 131
error_code (*fitbenchmarking.utils.exceptions.NoDataError attribute*), 131
error_code (*fitbenchmarking.utils.exceptions.NoHessianError attribute*), 132
error_code (*fitbenchmarking.utils.exceptions.NoJacobianError attribute*), 132
error_code (*fitbenchmarking.utils.exceptions.NoParserError attribute*), 132
error_code (*fitbenchmarking.utils.exceptions.NoResultsError attribute*),

132
error_code (*fitbenchmarking.utils.exceptions.OptionsError attribute*), 132
error_code (*fitbenchmarking.utils.exceptions.ParsingError attribute*), 132
error_code (*fitbenchmarking.utils.exceptions.PlottingError attribute*), 132
error_code (*fitbenchmarking.utils.exceptions.UnknownMinimizerError attribute*), 133
error_code (*fitbenchmarking.utils.exceptions.UnknownTableError attribute*), 133
error_code (*fitbenchmarking.utils.exceptions.UnsupportedMinimizerError attribute*), 133
escape_char (*fitbenchmarking.utils.output_grabber.StreamGrabber attribute*), 140
eval() (*fitbenchmarking.hessian.analytic_hessian.Analytic method*), 108
eval() (*fitbenchmarking.hessian.base_hessian.Hessian method*), 108
eval() (*fitbenchmarking.hessian.scipy_hessian.Scipy method*), 109
eval() (*fitbenchmarking.jacobian.analytic_jacobian.Analytic method*), 110
eval() (*fitbenchmarking.jacobian.base_jacobian.Jacobian method*), 110
eval() (*fitbenchmarking.jacobian.default_jacobian.Default method*), 111
eval() (*fitbenchmarking.jacobian.scipy_jacobian.Scipy method*), 111
eval_chisq() (*fitbenchmarking.controllers.base_controller.Controller method*), 79
eval_chisq() (*fitbenchmarking.controllers.mantid_controller.MantidController method*), 87
eval_cost() (*fitbenchmarking.cost_func.base_cost_func.CostFunc method*), 102
eval_cost() (*fitbenchmarking.cost_func.nlls_base_cost_func.BaseNLLSCostFunc method*), 104
eval_cost() (*fitbenchmarking.cost_func.poisson_cost_func.PoissonCostFunc method*), 106
eval_model() (*fitbenchmarking.parsing.fitting_problem.FittingProblem method*), 113
eval_r() (*fitbenchmarking.cost_func.hellinger_nlls_cost_func.HellingerNLLSCostFunc method*), 104
eval_r() (*fitbenchmarking.cost_func.nlls_base_cost_func.BaseNLLSCostFunc method*), 105
eval_r() (*fitbenchmarking.cost_func.nlls_cost_func.NLLSCostFunc method*), 105
eval_r() (*fitbenchmarking.cost_func.weighted_nlls_cost_func.WeightedNLLSCostFunc method*), 107
exception_handler() (*in module fitbenchmarking.cli.exception_handler*), 77
execute() (*fitbenchmarking.controllers.base_controller.Controller method*), 79

F

figures() (*in module fitbenchmarking.utils.create_dirs*), 128
file_path (*fitbenchmarking.results_processing.base_table.Table property*), 118
FilepathTooLongError, 129
fit() (*fitbenchmarking.controllers.base_controller.Controller method*), 79
fit() (*fitbenchmarking.controllers.bumps_controller.BumpsController method*), 81
fit() (*fitbenchmarking.controllers.dfo_controller.DFOController method*), 82
fit() (*fitbenchmarking.controllers.gradient_free_controller.GradientFree method*), 84
fit() (*fitbenchmarking.controllers.gsl_controller.GSLController method*), 85
fit() (*fitbenchmarking.controllers.levmar_controller.LevmarController method*), 86
fit() (*fitbenchmarking.controllers.mantid_controller.MantidController method*), 87
fit() (*fitbenchmarking.controllers.matlab_controller.MatlabController method*), 88
fit() (*fitbenchmarking.controllers.matlab_curve_controller.MatlabCurve method*), 89
fit() (*fitbenchmarking.controllers.matlab_opt_controller.MatlabOptCont method*), 91
fit() (*fitbenchmarking.controllers.matlab_stats_controller.MatlabStatsCo method*), 92
fit() (*fitbenchmarking.controllers.minuit_controller.MinuitController method*), 93
fit() (*fitbenchmarking.controllers.ralfit_controller.RALFitController method*), 93
fit() (*fitbenchmarking.controllers.scipy_controller.ScipyController method*), 95

fit() (*fitbenchmarking.controllers.scipy_go_controller.ScipyGOController method*), 96
fit() (*fitbenchmarking.controllers.scipy_ls_controller.ScipyLSController method*), 97
fit_plot_options (*fitbenchmarking.results_processing.plots.Plot attribute*), 125
FitBenchmarkException, 130
fitbenchmarking
    module, 141
fitbenchmarking.cli
    module, 78
fitbenchmarking.cli.exception_handler
    module, 77
fitbenchmarking.cli.main
    module, 77
fitbenchmarking.controllers
    module, 97
fitbenchmarking.controllers.base_controller
    module, 78
fitbenchmarking.controllers.bumps_controller
    module, 80
fitbenchmarking.controllers.controller_factory
    module, 81
fitbenchmarking.controllers.dfo_controller
    module, 82
fitbenchmarking.controllers.gradient_free_controller
    module, 83
fitbenchmarking.controllers.gsl_controller
    module, 84
fitbenchmarking.controllers.levmar_controller
    module, 85
fitbenchmarking.controllers.mantid_controller
    module, 86
fitbenchmarking.controllers.matlab_controller
    module, 88
fitbenchmarking.controllers.matlab_curve_controller
    module, 89
fitbenchmarking.controllers.matlab_mixin
    module, 90
fitbenchmarking.controllers.matlab_opt_controller
    module, 90
fitbenchmarking.controllers.matlab_stats_controller
    module, 91
fitbenchmarking.controllers.minuit_controller
    module, 92
fitbenchmarking.controllers.ralfit_controller
    module, 93
fitbenchmarking.controllers.scipy_controller
    module, 94
fitbenchmarking.controllers.scipy_go_controller
    module, 95
fitbenchmarking.controllers.scipy_ls_controller
    module, 96
fitbenchmarking.core
    module, 102
fitbenchmarking.core.fitting_benchmarking
    module, 97
fitbenchmarking.core.results_output
    module, 101
fitbenchmarking.cost_func
    module, 108
fitbenchmarking.cost_func.base_cost_func
    module, 102
fitbenchmarking.cost_func.cost_func_factory
    module, 103
fitbenchmarking.cost_func.hellinger_nlls_cost_func
    module, 104
fitbenchmarking.cost_func.nlls_base_cost_func
    module, 104
fitbenchmarking.cost_func.nlls_cost_func
    module, 105
fitbenchmarking.cost_func.poisson_cost_func
    module, 106
fitbenchmarking.cost_func.weighted_nlls_cost_func
    module, 107
fitbenchmarking.hessian
    module, 110
fitbenchmarking.hessian.analytic_hessian
    module, 108
fitbenchmarking.hessian.base_hessian
    module, 108
fitbenchmarking.hessian.hessian_factory
    module, 109
fitbenchmarking.hessian.scipy_hessian
    module, 109
fitbenchmarking.jacobian
    module, 112
fitbenchmarking.jacobian.analytic_jacobian
    module, 110
fitbenchmarking.jacobian.base_jacobian
    module, 110
fitbenchmarking.jacobian.default_jacobian
    module, 111
fitbenchmarking.jacobian.jacobian_factory
    module, 111
fitbenchmarking.jacobian.scipy_jacobian
    module, 111
fitbenchmarking.parsing
    module, 117
fitbenchmarking.parsing.base_parser
    module, 112
fitbenchmarking.parsing.cutest_parser
    module, 112
fitbenchmarking.parsing.fitbenchmark_parser
    module, 113
fitbenchmarking.parsing.fitting_problem
    module, 113

fitbenchmarking.parsing.ivp_parser module, 115
fitbenchmarking.parsing.mantid_parser module, 115
fitbenchmarking.parsing.nist_data_functions module, 115
fitbenchmarking.parsing.nist_parser module, 116
fitbenchmarking.parsing.parser_factory module, 117
fitbenchmarking.results_processing module, 128
fitbenchmarking.results_processing.acc_table module, 117
fitbenchmarking.results_processing.base_table module, 118
fitbenchmarking.results_processing.compare_tables module, 121
fitbenchmarking.results_processing.fitting_report module, 122
fitbenchmarking.results_processing.local_min_table module, 122
fitbenchmarking.results_processing.performance_profiler module, 124
fitbenchmarking.results_processing.plots module, 125
fitbenchmarking.results_processing.problem_summary_page module, 126
fitbenchmarking.results_processing.runtime_table module, 126
fitbenchmarking.results_processing.tables module, 127
fitbenchmarking.utils module, 141
fitbenchmarking.utils.create_dirs module, 128
fitbenchmarking.utils.debug module, 129
fitbenchmarking.utils.exceptions module, 129
fitbenchmarking.utils.fitbm_result module, 133
fitbenchmarking.utils.log module, 134
fitbenchmarking.utils.misc module, 134
fitbenchmarking.utils.options module, 135
fitbenchmarking.utils.output_grabber module, 140
fitbenchmarking.utils.timer module, 140
fitbenchmarking.utils.write_files module, 141
FitbenchmarkParser (*class in fitbenchmarking.parsing.fitbenchmark_parser*), 113
FittingProblem (*class in fitbenchmarking.parsing.fitting_problem*), 113
FittingProblemError, 130
FittingResult (*class in fitbenchmarking.utils.fitbm_result*), 133
flag (*fitbenchmarking.controllers.base_controller.Controller property*), 79
format (*fitbenchmarking.parsing.fitting_problem.FittingProblem attribute*), 113
format_function_scipy() (*in module fitbenchmarking.parsing.nist_data_functions*), 115
format_plot() (*fitbenchmarking.results_processing.plots.Plot method*), 125
function (*fitbenchmarking.parsing.fitting_problem.FittingProblem attribute*), 114

G

generate_table() (*in module fitbenchmarking.results_processing.tables*), 127
get_colour_df() (*fitbenchmarking.results_processing.base_table.Table method*), 118
get_colours_for_row() (*fitbenchmarking.results_processing.base_table.Table method*), 119
get_css() (*in module fitbenchmarking.utils.misc*), 134
get_description() (*fitbenchmarking.results_processing.base_table.Table method*), 119
get_error_str() (*fitbenchmarking.results_processing.base_table.Table static method*), 119
get_figure_paths() (*in module fitbenchmarking.results_processing.fitting_report*), 122
get_function_params() (*fitbenchmarking.parsing.fitting_problem.FittingProblem method*), 114
get_js() (*in module fitbenchmarking.utils.misc*), 134
get_link_str() (*fitbenchmarking.results_processing.base_table.Table method*), 119
get_logger() (*in module fitbenchmarking.utils.log*), 134
get_parser() (*in module fitbenchmarking.cli.main*), 77
get_printable_table() (*in module fitbenchmarking.utils.debug*), 129
get_problem_files() (*in module fitbenchmarking.utils.misc*), 135
get_str_dict() (*fitbenchmarking.results_processing.base_table.Table*

method), 119
get_str_result() (*fitbenchmarking.results_processing.base_table.Table method*), 119
get_value() (*fitbenchmarking.results_processing.acc_table.AccTable method*), 117
get_value() (*fitbenchmarking.results_processing.base_table.Table method*), 120
get_value() (*fitbenchmarking.results_processing.compare_table.CompareTable method*), 121
get_value() (*fitbenchmarking.results_processing.local_min_table.LocalMinTable method*), 123
get_value() (*fitbenchmarking.results_processing.runtime_table.RuntimeTable method*), 126
GradientFreeController (*class in fitbenchmarking.controllers.gradient_free_controller*), 83
group_results() (*in module fitbenchmarking.utils.create_dirs*), 128
GSLController (*class in fitbenchmarking.controllers.gsl_controller*), 84

H

HellingerNLLSCostFunc (*class in fitbenchmarking.cost_func.hellinger_nlls_cost_func*), 104
hes_cost() (*fitbenchmarking.cost_func.base_cost_func.CostFunc method*), 102
hes_cost() (*fitbenchmarking.cost_func.nlls_base_cost_func.BaseNLLSCostFunc method*), 105
hes_cost() (*fitbenchmarking.cost_func.poisson_cost_func.PoissonCostFunc method*), 106
hes_eval() (*fitbenchmarking.controllers.ralfit_controller.RALFitController method*), 93
hes_res() (*fitbenchmarking.cost_func.base_cost_func.CostFunc method*), 102
hes_res() (*fitbenchmarking.cost_func.hellinger_nlls_cost_func.HellingerNLLSCostFunc method*), 104
hes_res() (*fitbenchmarking.cost_func.nlls_cost_func.NLLSCostFunc method*), 105
hes_res() (*fitbenchmarking.cost_func.poisson_cost_func.PoissonCostFunc method*), 106
hes_res() (*fitbenchmarking.cost_func.weighted_nlls_cost_func.WeightedNLLSCostFunc method*), 107
Hessian (*class in fitbenchmarking.hessian.base_hessian*), 108
hessian (*fitbenchmarking.parsing.fitting_problem.FittingProblem attribute*), 114
hessian_enabled_solvers (*fitbenchmarking.controllers.base_controller.Controller attribute*), 79
hessian_enabled_solvers (*fitbenchmarking.controllers.ralfit_controller.RALFitController attribute*), 94
hessian_enabled_solvers (*fitbenchmarking.controllers.scipy_controller.ScipyController attribute*), 95

I

incompatible_problems (*fitbenchmarking.controllers.base_controller.Controller attribute*), 79
incompatible_problems (*fitbenchmarking.controllers.matlab_controller.MatlabController attribute*), 88
incompatible_problems (*fitbenchmarking.controllers.matlab_curve_controller.MatlabCurveController attribute*), 89
incompatible_problems (*fitbenchmarking.controllers.matlab_opt_controller.MatlabOptController attribute*), 91
incompatible_problems (*fitbenchmarking.controllers.matlab_stats_controller.MatlabStatsController attribute*), 92
INCOMPATIBLE_PROBLEMS (*fitbenchmarking.hessian.base_hessian.Hessian attribute*), 108
INCOMPATIBLE_PROBLEMS (*fitbenchmarking.hessian.scipy_hessian.Scipy attribute*), 109
INCOMPATIBLE_PROBLEMS (*fitbenchmarking.jacobian.base_jacobian.Jacobian attribute*), 110
INCOMPATIBLE_PROBLEMS (*fitbenchmarking.jacobian.scipy_jacobian.Scipy attribute*), 111
IncompatibleHessianError, 130
IncompatibleJacobianError, 130
IncompatibleMinimizerError, 130
IncompatibleProblemError, 130
IncompatibleTableError, 130
IncorrectBoundsError, 131
ini_guess_plot_options (*fitbenchmarking.results_processing.plots.Plot attribute*),

125
is_int() (*in module fitbenchmarking.parsing.nist_data_functions*), 115
is_safe() (*in module fitbenchmarking.parsing.nist_data_functions*), 115
IVPParser (*class in fitbenchmarking.parsing.ivp_parser*), 115

J

jac_cost() (*fitbenchmarking.cost_func.base_cost_func.CostFunc method*), 103
jac_cost() (*fitbenchmarking.cost_func.nlls_base_cost_func.BaseNLLSCostFunc method*), 105
jac_cost() (*fitbenchmarking.cost_func.poisson_cost_func.PoissonCostFunc method*), 107
jac_res() (*fitbenchmarking.cost_func.base_cost_func.CostFunc method*), 103
jac_res() (*fitbenchmarking.cost_func.hellinger_nlls_cost_func.HellingerNLLSCostFunc method*), 104
jac_res() (*fitbenchmarking.cost_func.nlls_cost_func.NLLSCostFunc method*), 106
jac_res() (*fitbenchmarking.cost_func.poisson_cost_func.PoissonCostFunc method*), 107
jac_res() (*fitbenchmarking.cost_func.weighted_nlls_cost_func.WeightedNLLSCostFunc method*), 107
Jacobian (*class in fitbenchmarking.jacobian.base_jacobian*), 110
jacobian (*fitbenchmarking.parsing.fitting_problem.FittingProblem attribute*), 114
jacobian_enabled_solvers (*fitbenchmarking.controllers.base_controller.Controller attribute*), 79
jacobian_enabled_solvers (*fitbenchmarking.controllers.gsl_controller.GSLController attribute*), 85
jacobian_enabled_solvers (*fitbenchmarking.controllers.levmar_controller.LevmarController attribute*), 86
jacobian_enabled_solvers (*fitbenchmarking.controllers.mantid_controller.MantidController attribute*), 87
jacobian_enabled_solvers (*fitbenchmarking.controllers.matlab_opt_controller.MatlabOptController attribute*), 91
jacobian_enabled_solvers (*fitbenchmarking.controllers.ralfit_controller.RALFitController attribute*), 94
jacobian_enabled_solvers (*fitbenchmarking.controllers.scipy_controller.ScipyController attribute*), 95
jacobian_enabled_solvers (*fitbenchmarking.controllers.scipy_go_controller.ScipyGOController attribute*), 96
jacobian_enabled_solvers (*fitbenchmarking.controllers.scipy_ls_controller.ScipyLSController attribute*), 97

L

LevmarController (*class in fitbenchmarking.controllers.levmar_controller*), 85
load_table() (*in module fitbenchmarking.results_processing.tables*), 128
LocalMinTable (*class in fitbenchmarking.results_processing.local_min_table*), 122
loop_over_benchmark_problems() (*in module fitbenchmarking.core.fitting_benchmarking*), 98
loop_over_cost_function() (*in module fitbenchmarking.core.fitting_benchmarking*), 98
loop_over_fitting_software() (*in module fitbenchmarking.core.fitting_benchmarking*), 98
loop_over_hessians() (*in module fitbenchmarking.core.fitting_benchmarking*), 99
loop_over_jacobians() (*in module fitbenchmarking.core.fitting_benchmarking*), 99
loop_over_minimizers() (*in module fitbenchmarking.core.fitting_benchmarking*), 99
loop_over_starting_values() (*in module fitbenchmarking.core.fitting_benchmarking*), 100

M

main() (*in module fitbenchmarking.cli.main*), 77
MantidController (*class in fitbenchmarking.controllers.mantid_controller*), 86
MantidParser (*class in fitbenchmarking.parsing.mantid_parser*), 115
MatlabController (*class in fitbenchmarking.controllers.matlab_controller*), 88
MatlabCurveController (*class in fitbenchmarking.controllers.matlab_curve_controller*), 89
MatlabMixin (*class in fitbenchmarking.controllers.matlab_mixin*), 90
MatlabOptController (*class in fitbenchmarking.controllers.matlab_opt_controller*), 90
MatlabStatsController (*class in fitbenchmarking.controllers.matlab_stats_controller*), 91
MaxRuntimeError, 131

method (*fitbenchmarking.hessian.base_hessian.Hessian property*), 108
method (*fitbenchmarking.jacobian.base_jacobian.Jacobian property*), 110
minimizer_dropdown_html() (*fitbenchmarking.results_processing.base_table.Table method*), 120
minimizers (*fitbenchmarking.utils.options.Options property*), 139
MinuitController (*class in fitbenchmarking.controllers.minuit_controller*), 92
MissingBoundsError, 131
MissingSoftwareError, 131
modified_minimizer_name() (*fitbenchmarking.utils.fitbm_result.FittingResult method*), 133
module
fitbenchmarking, 141
fitbenchmarking.cli, 78
fitbenchmarking.cli.exception_handler, 77
fitbenchmarking.cli.main, 77
fitbenchmarking.controllers, 97
fitbenchmarking.controllers.base_controller, 78
fitbenchmarking.controllers.bumps_controller, 80
fitbenchmarking.controllers.controller_factory, 81
fitbenchmarking.controllers.dfo_controller, 82
fitbenchmarking.controllers.gradient_free_controller, 83
fitbenchmarking.controllers.gsl_controller, 84
fitbenchmarking.controllers.levmar_controller, 85
fitbenchmarking.controllers.mantid_controller, 86
fitbenchmarking.controllers.matlab_controller, 88
fitbenchmarking.controllers.matlab_curve_controller, 89
fitbenchmarking.controllers.matlab_mixin, 90
fitbenchmarking.controllers.matlab_opt_controller, 90
fitbenchmarking.controllers.matlab_stats_controller, 91
fitbenchmarking.controllers.minuit_controller, 92
fitbenchmarking.controllers.ralfit_controller, 93
fitbenchmarking.controllers.scipy_controller, 94
fitbenchmarking.controllers.scipy_go_controller, 95
fitbenchmarking.controllers.scipy_ls_controller, 96
fitbenchmarking.core, 102
fitbenchmarking.core.fitting_benchmarking, 97
fitbenchmarking.core.results_output, 101
fitbenchmarking.cost_func, 108
fitbenchmarking.cost_func.base_cost_func, 102
fitbenchmarking.cost_func.cost_func_factory, 103
fitbenchmarking.cost_func.hellinger_nlls_cost_func, 104
fitbenchmarking.cost_func.nlls_base_cost_func, 104
fitbenchmarking.cost_func.nlls_cost_func, 105
fitbenchmarking.cost_func.poisson_cost_func, 106
fitbenchmarking.cost_func.weighted_nlls_cost_func, 107
fitbenchmarking.hessian, 110
fitbenchmarking.hessian.analytic_hessian, 108
fitbenchmarking.hessian.base_hessian, 108
fitbenchmarking.hessian.hessian_factory, 109
fitbenchmarking.hessian.scipy_hessian, 109
fitbenchmarking.jacobian, 112
fitbenchmarking.jacobian.analytic_jacobian, 110
fitbenchmarking.jacobian.base_jacobian, 110
fitbenchmarking.jacobian.default_jacobian, 111
fitbenchmarking.jacobian.jacobian_factory, 111
fitbenchmarking.jacobian.scipy_jacobian, 111
fitbenchmarking.parsing, 117
fitbenchmarking.parsing.base_parser, 112
fitbenchmarking.parsing.cutest_parser, 112
fitbenchmarking.parsing.fitbenchmark_parser, 113
fitbenchmarking.parsing.fitting_problem, 113
fitbenchmarking.parsing.ivp_parser, 115
fitbenchmarking.parsing.mantid_parser, 115
fitbenchmarking.parsing.nist_data_functions,

115
fitbenchmarking.parsing.nist_parser, 116
fitbenchmarking.parsing.parser_factory, 117
fitbenchmarking.results_processing, 128
fitbenchmarking.results_processing.acc_table, 117
fitbenchmarking.results_processing.base_table, 118
fitbenchmarking.results_processing.compare_table, 121
fitbenchmarking.results_processing.fitting_report, 122
fitbenchmarking.results_processing.local_min_table, 122
fitbenchmarking.results_processing.performance_profiles, 124
fitbenchmarking.results_processing.plots, 125
fitbenchmarking.results_processing.problem_summary_page, 126
fitbenchmarking.results_processing.runtime_table, 126
fitbenchmarking.results_processing.tables, 127
fitbenchmarking.utils, 141
fitbenchmarking.utils.create_dirs, 128
fitbenchmarking.utils.debug, 129
fitbenchmarking.utils.exceptions, 129
fitbenchmarking.utils.fitbm_result, 133
fitbenchmarking.utils.log, 134
fitbenchmarking.utils.misc, 134
fitbenchmarking.utils.options, 135
fitbenchmarking.utils.output_grabber, 140
fitbenchmarking.utils.timer, 140
fitbenchmarking.utils.write_files, 141
multifit (*fitbenchmarking.parsing.fitting_problem.FittingProblem attribute*), 114
multivariate (*fitbenchmarking.parsing.fitting_problem.FittingProblem attribute*), 114

N

name (*fitbenchmarking.parsing.fitting_problem.FittingProblem attribute*), 114
name() (*fitbenchmarking.hessian.analytic_hessian.Analytic method*), 108
name() (*fitbenchmarking.hessian.base_hessian.Hessian method*), 109
name() (*fitbenchmarking.jacobian.analytic_jacobian.Analytic method*), 110
name() (*fitbenchmarking.jacobian.base_jacobian.Jacobian method*), 110
name() (*fitbenchmarking.jacobian.default_jacobian.Default method*), 111
nist_func_definition() (*in module fitbenchmarking.parsing.nist_data_functions*), 116
nist_hessian_definition() (*in module fitbenchmarking.parsing.nist_data_functions*), 116
nist_jacobian_definition() (*in module fitbenchmarking.parsing.nist_data_functions*), 116
NISTParser (*class in fitbenchmarking.parsing.nist_parser*), 116
NLLSCostFunc (*class in fitbenchmarking.cost_func.nlls_cost_func*), 105
NoAnalyticJacobian, 131
NoCompatibleMinimizerError, 131
NoDataError, 131
NoHessianError, 131
NoJacobianError, 132
NoParserError, 132
NoResultsError, 132
norm_acc (*fitbenchmarking.utils.fitbm_result.FittingResult property*), 133
norm_runtime (*fitbenchmarking.utils.fitbm_result.FittingResult property*), 133

O

Options (*class in fitbenchmarking.utils.options*), 135
OptionsError, 132
OutputGrabber (*class in fitbenchmarking.utils.output_grabber*), 140

P

param_names (*fitbenchmarking.parsing.fitting_problem.FittingProblem property*), 114
parse() (*fitbenchmarking.parsing.base_parser.Parser method*), 112
parse() (*fitbenchmarking.parsing.cutest_parser.CutestParser method*), 112
parse() (*fitbenchmarking.parsing.fitbenchmark_parser.FitbenchmarkParser method*), 113
parse() (*fitbenchmarking.parsing.nist_parser.NISTParser method*), 116
parse_problem_file() (*in module fitbenchmarking.parsing.parser_factory*), 117
Parser (*class in fitbenchmarking.parsing.base_parser*), 112
ParserFactory (*class in fitbenchmarking.parsing.parser_factory*), 117
ParsingError, 132

perform_fit() (*in module fitbenchmarking.core.fitting_benchmarking*), 100
Plot (*class in fitbenchmarking.results_processing.plots*), 125
plot() (*in module fitbenchmarking.results_processing.performance_profiler*), 124
plot_best() (*fitbenchmarking.results_processing.plots.Plot method*), 125
plot_data() (*fitbenchmarking.results_processing.plots.Plot method*), 125
plot_fit() (*fitbenchmarking.results_processing.plots.Plot method*), 125
plot_initial_guess() (*fitbenchmarking.results_processing.plots.Plot method*), 125
plot_summary() (*fitbenchmarking.results_processing.plots.Plot class method*), 125
PlottingError, 132
PoissonCostFunc (*class in fitbenchmarking.cost_func.poisson_cost_func*), 106
prepare() (*fitbenchmarking.controllers.base_controller.Controller method*), 80
prepare_profile_data() (*in module fitbenchmarking.results_processing.performance_profiler*), 124
preprocess_data() (*in module fitbenchmarking.core.results_output*), 101
problem_dropdown_html() (*fitbenchmarking.results_processing.base_table.Table method*), 120
profile() (*in module fitbenchmarking.results_processing.performance_profiler*), 124
py_to_mat() (*fitbenchmarking.controllers.matlab_mixin.MatlabMixin static method*), 90

R

RALFitController (*class in fitbenchmarking.controllers.ralfit_controller*), 93
read_list() (*in module fitbenchmarking.utils.options*), 139
read_range() (*in module fitbenchmarking.utils.options*), 140
read_value() (*fitbenchmarking.utils.options.Options method*), 139
readOutput() (*fitbenchmarking.utils.output_grabber.StreamGrabber method*), 140
record_alg_type() (*fitbenchmarking.controllers.base_controller.Controller method*), 80
reset() (*fitbenchmarking.utils.options.Options method*), 139
reset() (*fitbenchmarking.utils.timer.TimerWithMaxTime method*), 140
results() (*in module fitbenchmarking.utils.create_dirs*), 129
results_dir (*fitbenchmarking.utils.options.Options property*), 139
run() (*in module fitbenchmarking.cli.main*), 77
RuntimeTable (*class in fitbenchmarking.results_processing.runtime_table*), 126

S

sanitised_min_name() (*fitbenchmarking.utils.fitbm_result.FittingResult method*), 134
sanitised_name (*fitbenchmarking.utils.fitbm_result.FittingResult property*), 134
save_colourbar() (*fitbenchmarking.results_processing.base_table.Table method*), 120
save_colourbar() (*fitbenchmarking.results_processing.local_min_table.LocalMinTable method*), 123
save_results() (*in module fitbenchmarking.core.results_output*), 102
Scipy (*class in fitbenchmarking.hessian.scipy_hessian*), 109
Scipy (*class in fitbenchmarking.jacobian.scipy_jacobian*), 111
ScipyController (*class in fitbenchmarking.controllers.scipy_controller*), 94
ScipyGOController (*class in fitbenchmarking.controllers.scipy_go_controller*), 95
ScipyLSController (*class in fitbenchmarking.controllers.scipy_ls_controller*), 96
set_value_ranges() (*fitbenchmarking.parsing.fitting_problem.FittingProblem method*), 114
setup() (*fitbenchmarking.controllers.base_controller.Controller method*), 80
setup() (*fitbenchmarking.controllers.bumps_controller.BumpsController method*), 81
setup() (*fitbenchmarking.controllers.dfo_controller.DFOController method*), 82

setup() (*fitbenchmarking.controllers.gradient_free_controller.GradientFreeController method*), 84
setup() (*fitbenchmarking.controllers.gsl_controller.GSLController method*), 85
setup() (*fitbenchmarking.controllers.levmar_controller.LevmarController method*), 86
setup() (*fitbenchmarking.controllers.mantid_controller.MantidController method*), 87
setup() (*fitbenchmarking.controllers.matlab_controller.MatlabController method*), 88
setup() (*fitbenchmarking.controllers.matlab_curve_controller.MatlabCurveController method*), 89
setup() (*fitbenchmarking.controllers.matlab_opt_controller.MatlabOptController method*), 91
setup() (*fitbenchmarking.controllers.matlab_stats_controller.MatlabStatsController method*), 92
setup() (*fitbenchmarking.controllers.minuit_controller.MinuitController method*), 93
setup() (*fitbenchmarking.controllers.ralfit_controller.RALFitController method*), 94
setup() (*fitbenchmarking.controllers.scipy_controller.ScipyController method*), 95
setup() (*fitbenchmarking.controllers.scipy_go_controller.ScipyGOController method*), 96
setup() (*fitbenchmarking.controllers.scipy_ls_controller.ScipyLSController method*), 97
setup_logger() (*in module fitbenchmarking.utils.log*), 134
software (*fitbenchmarking.controllers.base_controller.Controller property*), 80
sorted_index (*fitbenchmarking.parsing.fitting_problem.FittingProblem attribute*), 114
start() (*fitbenchmarking.utils.output_grabber.StreamGrabber method*), 140
start() (*fitbenchmarking.utils.timer.TimerWithMaxTime method*), 140
start_x (*fitbenchmarking.parsing.fitting_problem.FittingProblem attribute*), 114
starting_values (*fitbenchmarking.parsing.fitting_problem.FittingProblem attribute*), 114
stop() (*fitbenchmarking.utils.output_grabber.StreamGrabber method*), 140
stop() (*fitbenchmarking.utils.timer.TimerWithMaxTime method*), 140
StreamGrabber (*class in fitbenchmarking.utils.output_grabber*), 140
summary_best_plot_options (*fitbenchmarking.results_processing.plots.Plot attribute*), 126
summary_plot_options (*fitbenchmarking.results_processing.plots.Plot attribute*), 126
support_pages() (*in module fitbenchmarking.utils.create_dirs*), 129

T

Table (*class in fitbenchmarking.results_processing.base_table*), 118
table_title (*fitbenchmarking.results_processing.base_table.Table property*), 120
TimerWithMaxTime (*class in fitbenchmarking.utils.timer*), 140
to_html() (*fitbenchmarking.results_processing.base_table.Table method*), 120
to_txt() (*fitbenchmarking.results_processing.base_table.Table method*), 120

U

UnknownMinimizerError, 132
UnknownTableError, 133
UnsupportedMinimizerError, 133

V

VALID (*fitbenchmarking.utils.options.Options attribute*), 136
VALID_FITTING (*fitbenchmarking.utils.options.Options attribute*), 138
VALID_FLAGS (*fitbenchmarking.controllers.base_controller.Controller attribute*), 78
VALID_HESSIAN (*fitbenchmarking.utils.options.Options attribute*), 138
VALID_JACOBIAN (*fitbenchmarking.utils.options.Options attribute*), 138
VALID_LOGGING (*fitbenchmarking.utils.options.Options attribute*), 138

VALID_MINIMIZERS (*fitbenchmarking.utils.options.Options attribute*), 138
VALID_OUTPUT (*fitbenchmarking.utils.options.Options attribute*), 138
VALID_PLOTTING (*fitbenchmarking.utils.options.Options attribute*), 138
VALID_SECTIONS (*fitbenchmarking.utils.options.Options attribute*), 139
validate() (*fitbenchmarking.controllers.base_controller.Controller method*), 80
validate_algorithm_type() (*fitbenchmarking.cost_func.base_cost_func.CostFunc method*), 103
validate_minimizer() (*fitbenchmarking.controllers.base_controller.Controller method*), 80
ValidationException, 133
vals_to_colour() (*fitbenchmarking.results_processing.base_table.Table static method*), 121
vals_to_colour() (*fitbenchmarking.results_processing.compare_table.CompareTable method*), 121
vals_to_colour() (*fitbenchmarking.results_processing.local_min_table.LocalMinTable static method*), 123
value_ranges (*fitbenchmarking.parsing.fitting_problem.FittingProblem attribute*), 114
verify() (*fitbenchmarking.parsing.fitting_problem.FittingProblem method*), 114

W

WeightedNLLSCostFunc (*class in fitbenchmarking.cost_func.weighted_nlls_cost_func*), 107
write() (*fitbenchmarking.utils.options.Options method*), 139
write_file() (*in module fitbenchmarking.utils.write_files*), 141
write_to_stream() (*fitbenchmarking.utils.options.Options method*), 139