FitBenchmarking Documentation

Release 0.1.dev1

STFC

May 18, 2021

Contents

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

Python Module Index **87**

Index **89**

CHAPTER 1

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/latest/users/install_instructions/index.html
- **User Documentation & Example Usage:** https://fitbenchmarking.readthedocs.io/en/latest/users/index.html
- **Community Guidelines:** https://fitbenchmarking.readthedocs.io/en/latest/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/latest/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 hightlighting 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:

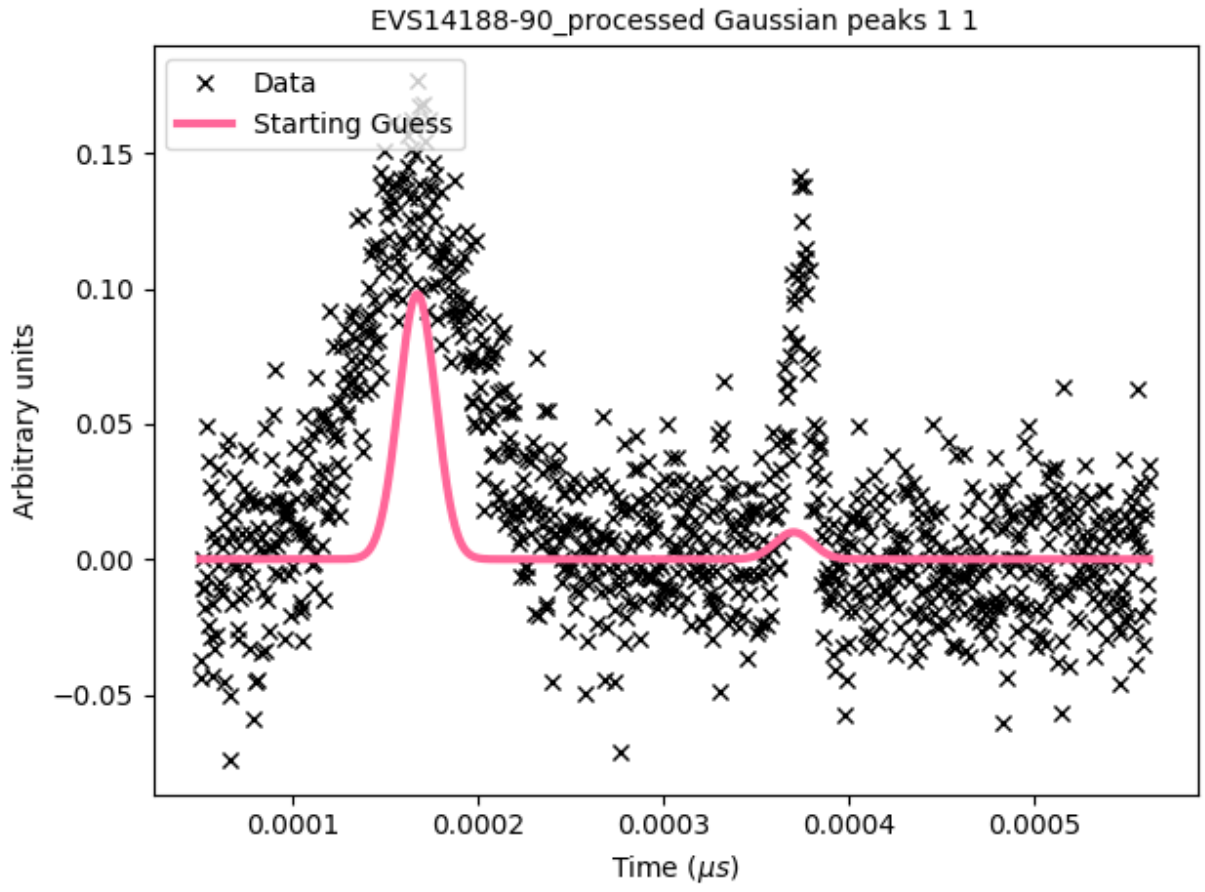


Fig. 1: VESUVIO experiment data

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 Nedler-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?*

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.

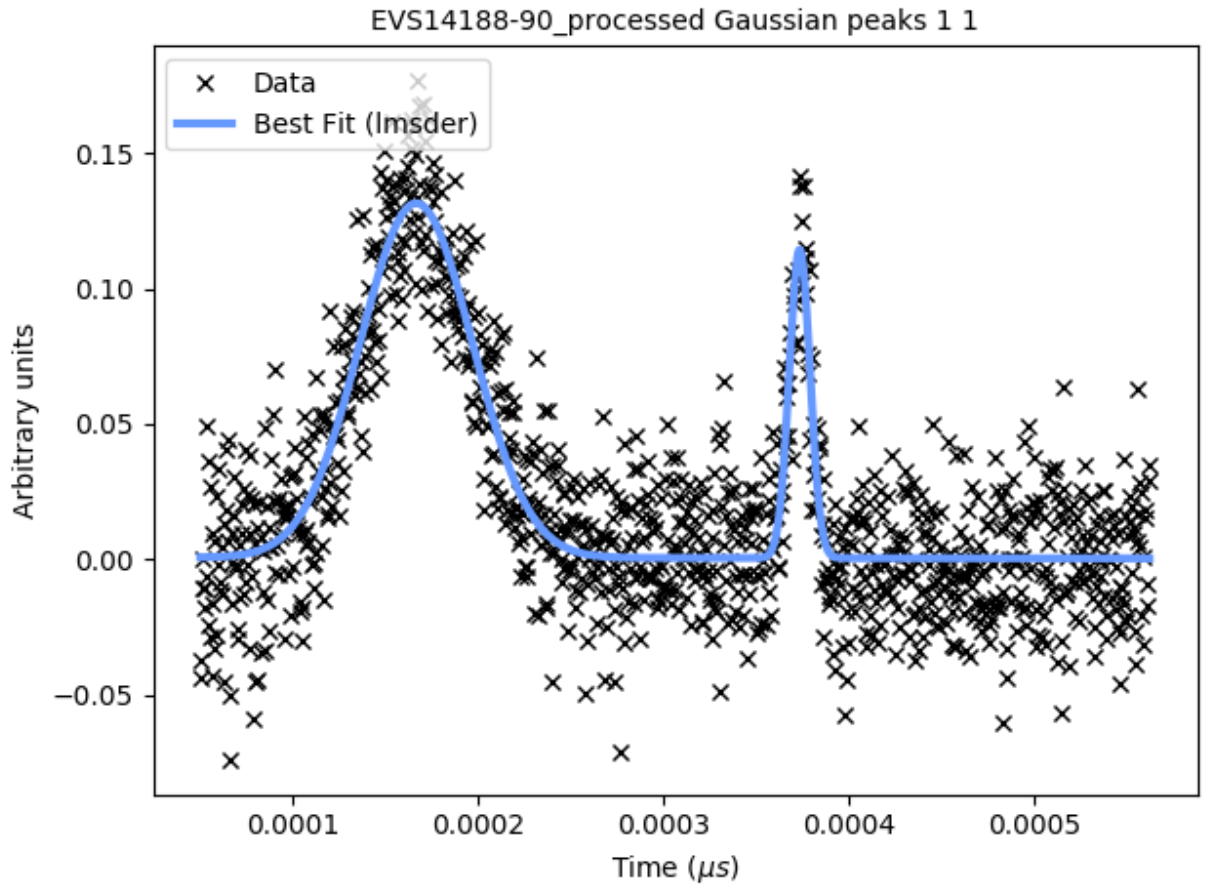


Fig. 2: GSL's `lmsder` (Levenberg-Marquardt) algorithm on the data

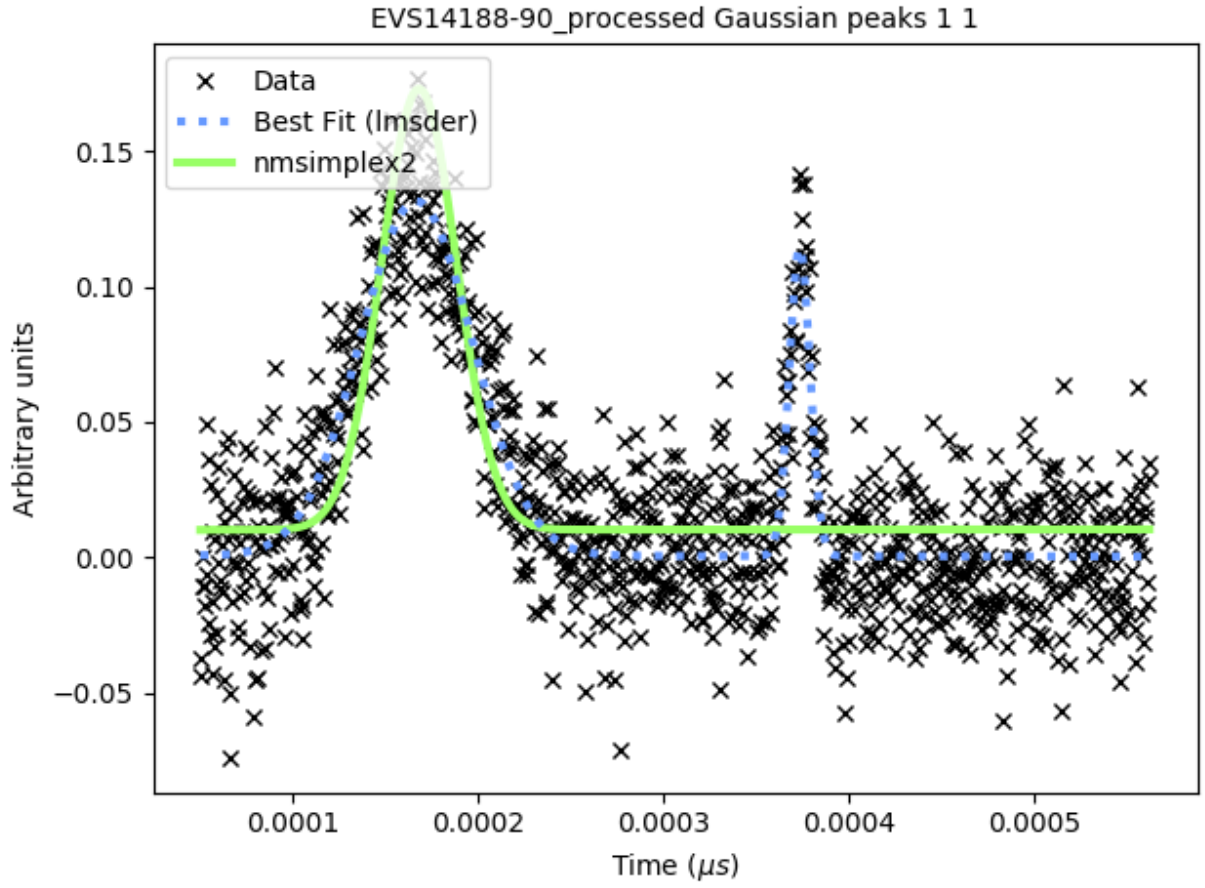
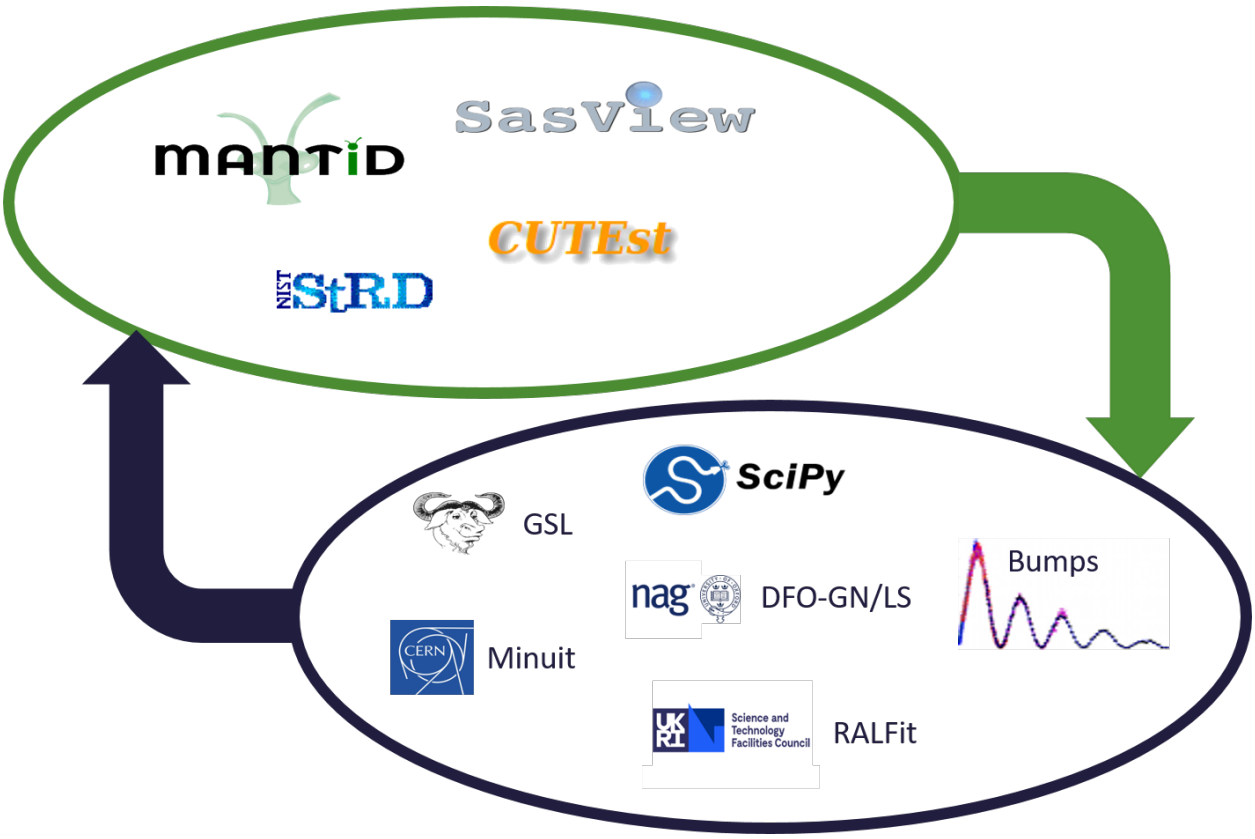


Fig. 3: GSL's `nmsimplex2` (Nelder-Mead Simplex) algorithm on the data



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:

Performance Profile

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

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.

	bumps		dfo	gsl		mantid	minuit	ralfit		scipy		scipy-ls	
	mp	amoeba	dfols	lmsder: scipy 2-point	lmder: scipy 2-point	Levenberg-MarquardtMD: scipy 2-point	minuit	gn: scipy 2-point	gn_reg: scipy 2-point	CG: scipy 2-point	L-BFGS-B: scipy 2-point	lm-scipy-no-jac	trf: scipy 2-point
BENNETT5	1.912e-05 (1.191)	1.608e-05 (1.001)	2.039 (1.27e+08)[2]	1.639e-05 (1.021)[1]	1.612e-05 (1.004)[1]	0.01929 (1201)[1]	2.127e-05 (1.325)	1.606e-05 (1)	1.606e-05 (1)	1.652e-05 (1.029)	0.02056 (1281)	1.905e-05 (1.186)[1]	1.649e-05 (1.027)[1]
BoxBOD, Start 1	61.81 (7.724)	61.81 (7.724)	80.73 (10.09)[2]	inf (inf)[3]	8.003 (1)	61.81 (7.724)[1]	8.003 (1)	61.81 (7.724)	61.81 (7.724)	61.81 (7.724)	8.003 (1)	8.003 (1)	8.003 (1)
BoxBOD, Start 2	8.003 (1)	8.003 (1)	8.003 (1)	8.003 (1)	8.003 (1)	61.81 (7.723)[1]	8.003 (1)	8.003 (1)	8.003 (1)	24.11 (3.282)[1]	61.81 (7.724)	8.003 (1)	8.003 (1)
CERI651A	334.8 (1)	348.3 (1.04)	1889 (5.641)[2]	334.8 (1)	334.8 (1)	508 (1.517)	349.9 (1.045)	inf (inf)[3]	inf (inf)[3]	374.8 (1.119)[1]	383.2 (1.144)	334.8 (1)	334.8 (1)
ENGINX 193749 calibration, spectrum 651, peak 5	29.11 (1.996)	31.5 (2.16)	24.71 (1.695)[2]	14.58 (1)	14.58 (1)	14.69 (1.007)	15.12 (1.037)	inf (inf)[3]	inf (inf)[3]	20.07 (1.377)[1]	20.16 (1.383)	14.58 (1)	15.38 (1.054)
Gauss3, Start 1	76.64 (1)	76.64 (1)	407.4 (5.489)[2]	76.64 (1)	76.64 (1)	412.8 (5.386)[1]	76.64 (1)	76.64 (1)	76.64 (1)	76.67 (1)[1]	76.66 (1)	76.64 (1)	76.64 (1)
Gauss3, Start 2	76.64 (1)	76.64 (1)	76.64 (1)	76.64 (1)	76.64 (1)	196.6 (2.565)[1]	76.64 (1)	76.64 (1)	76.64 (1)	77.98 (1.017)[1]	81.8 (1.067)	76.64 (1)	76.64 (1)
HIFI 116891	1.508e+04 (16.36)	920.4 (1)	920.4 (1)	inf (inf)[3]	920.4 (1)	920.4 (1)	inf (inf)[3]	inf (inf)[3]	inf (inf)[3]	922 (1.002)[2]	922 (1.002)	920.4 (1)	920.4 (1)
Lanczos2, Start 1	2.189e-11 (1)	6.94e-08 (4541)	2.508e-10 (11.46)[2]	2.189e-11 (1)	2.189e-11 (1)	1.699e-05 (7.765e+05)	0.0004954 (2.263e+07)	2.189e-11 (1)	2.189e-11 (1)	1.268e-05 (5.793e+05)	1.378e-05 (6.297e+05)	1.416e-09 (64.72)[1]	2.189e-11 (1)
Lanczos2, Start 2	2.189e-11 (1)	7.474e-07 (3.13e+04)	2.232e-11 (1.02)[2]	2.189e-11 (1)	2.189e-11 (1)	2.251e-11 (1.029)	0.0003615 (1.652e+07)	2.189e-11 (1)	2.189e-11 (1)	1.499e-06 (6.848e+04)	1.372e-05 (6.267e+05)	2.189e-11 (1)	2.189e-11 (1)
Thurber, Start 1	7.591 (1)	7.591 (1)	7.591 (1)[2]	7.591 (1)	7.591 (1)	687.6 (90.57)[1]	7.591 (1)	7.591 (1)	7.591 (1)	66.88 (8.811)[2]	15 (1.976)	7.591 (1)	7.591 (1)
Thurber, Start 2	7.591 (1)	7.591 (1)	1285 (169.3)[2]	7.591 (1)	7.591 (1)	337.6 (44.47)[1]	7.591 (1)	7.591 (1)	7.591 (1)	3086 (935.2)[2]	8.697 (1.146)	7.591 (1)	7.591 (1)
WISH17701 panel 103 tube 3 calibration, peak 2	7.014e+06 (1)	7.014e+06 (1)	7.014e+06 (1)[1]	7.014e+06 (1)	7.014e+06 (1)	7.014e+06 (1)	7.014e+06 (1)	7.014e+06 (1)	7.014e+06 (1)	7.145e+06 (1.019)[2]	1.634e+07 (2.329)	7.014e+06 (1)	7.014e+06 (1)

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.

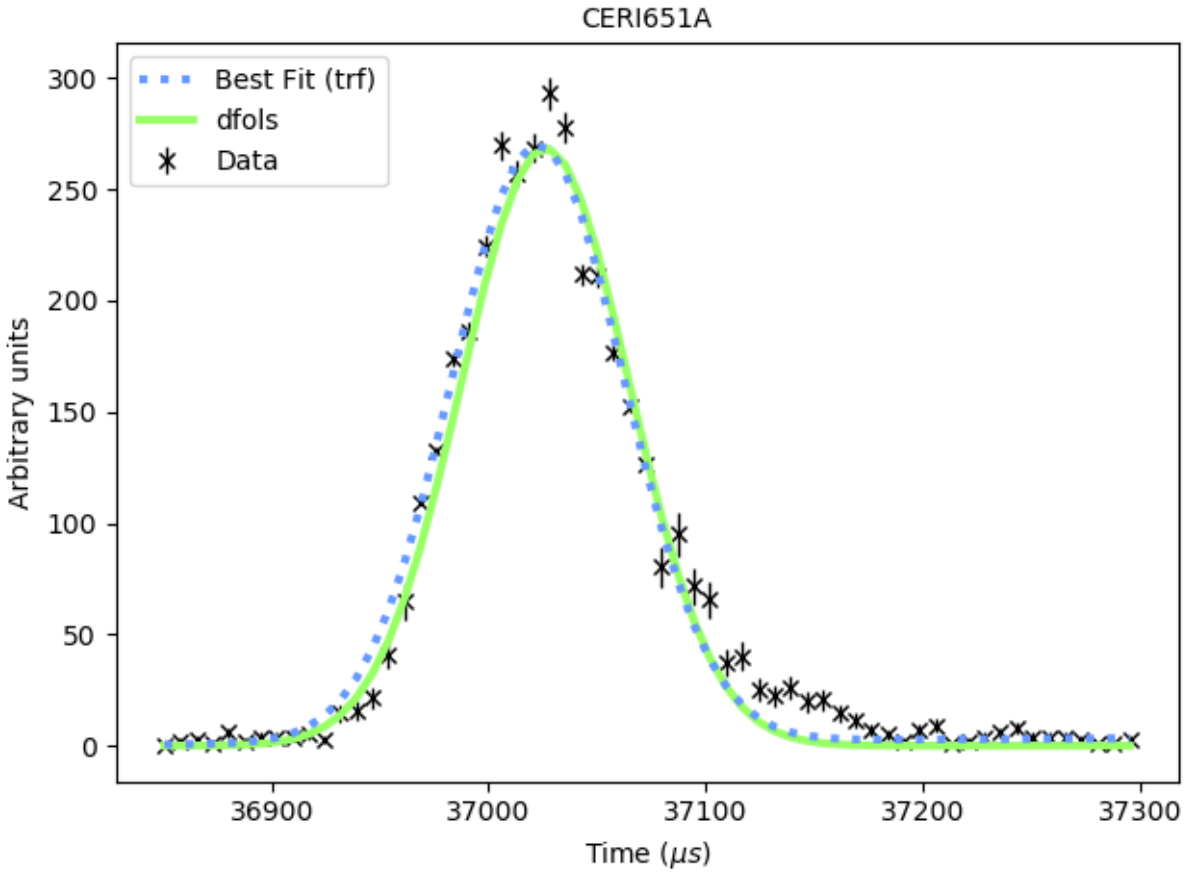
Installing via `pip`

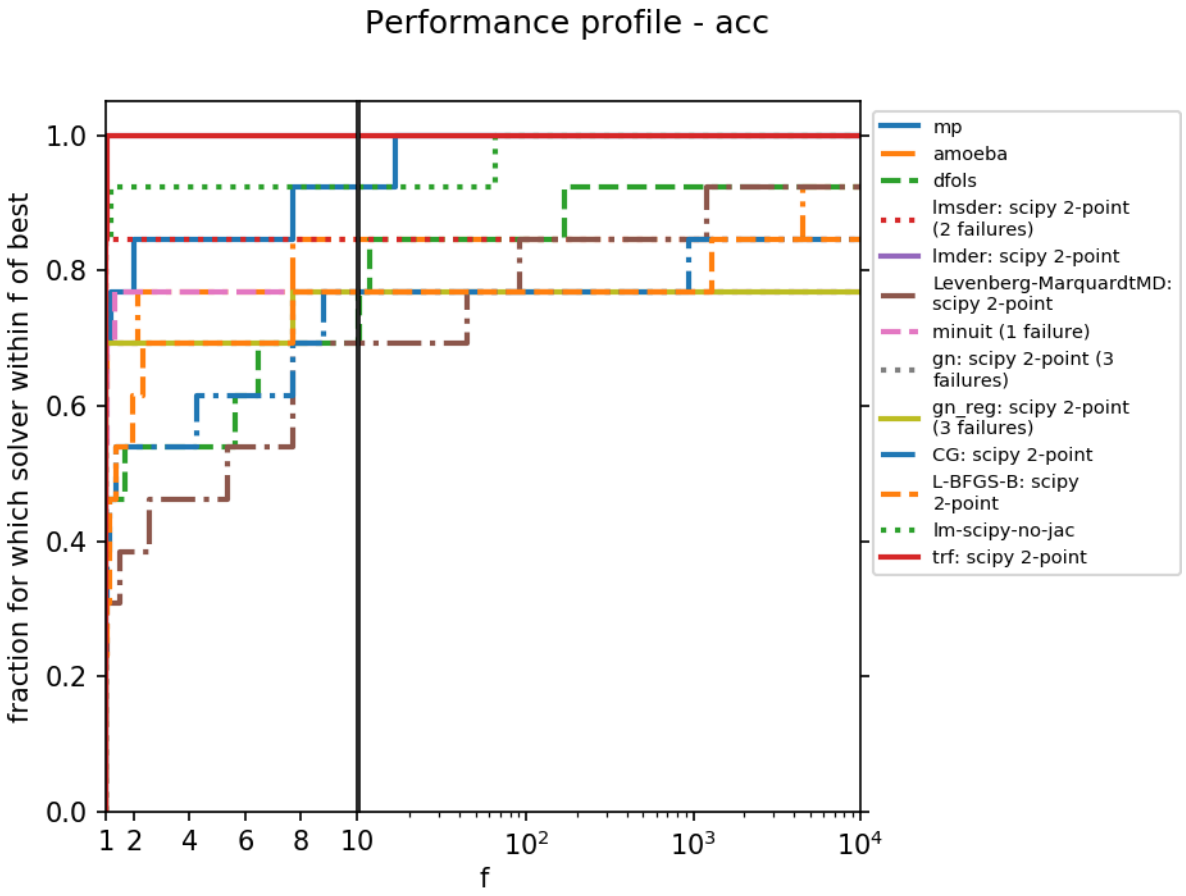
FitBenchmarking can be installed via the command line by entering:

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

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

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 - acc
fraction for which solver within f of best
f
1.0
0.8
0.6
0.4
0.2
0.0
1
2
4
6
8
10
10^2
10^3
10^4
mp
amoeba
dfols
lmsder: scipy 2-point (2 failures)
lmder: scipy 2-point
Levenberg-MarquardtMD: scipy 2-point
minuit (1 failure)
gn: scipy 2-point (3 failures)
gn_reg: scipy 2-point (3 failures)
CG: scipy 2-point
L-BFGS-B: scipy 2-point
lm-scipy-no-jac
trf: scipy 2-point

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,minuit,SAS]
```

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.
- `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.
- `minuit` – installs the Minuit fitting package.
- `SAS` – installs the Sasmodels fitting package.
- `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 range of systems are available at https://download.mantidproject.org/.

RALFit

RALFit is availble 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 test example 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 `--problem-sets` (or `-p`) option. 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` 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*.

Running `fitbenchmarking -h` will give more guidance about available commands, including examples of how to run multiple problem sets.

Cost functions

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

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**(*probl*
  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 as outputs. The links below give descriptions of these 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 using the timeit module in python. The number of runtimes can be 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.

Table formats

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.

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:

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 MultFit 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.

> **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.

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.

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.

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.

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).

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)
- `gsl` (external software – see *Installing External Software*)
- `levmar` (external software – see *Extra dependencies*)
- `mantid` (external software – see *Installing External Software*)
- `minuit` (default software)
- `ralfit` (external software – see *Installing External Software*)
- `scipy` (default software)
- `scipy_ls` (default software)

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

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

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. 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`)
- `general` - minimizers which solve a generic *min f(x)*

Default is `all`

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

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

Use errors (`use_errors`)

This will switch between weighted and unweighted least squares. If `use_errors=True`, and no errors are supplied, then `e[i]` will be set to `sqrt(abs(y[i]))`. Errors below `1.0e-8` will be clipped to that value.

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

```
[FITTING]
use_errors: yes
```

Jacobian method (`jac_method`)

This sets the Jacobian used. 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 approxmation implemented in the minimizer.

- `numdifftools` - uses the python package *numdifftools*.

Default is `default`

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

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

Cost function (`cost_func_type`)

This sets the cost function to be used for the given data. Current cost functions supported 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
```

Minimizer Options

This section is used to declare the minimizers to use for each fitting software.

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.

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*.

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`)

  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*.

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

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`)
- 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

The Mantid minimizers are set as follows:

```
[MINIMIZERS]
mantid: BFGS
        Conjugate gradient (Fletcher-Reeves imp.)
        Conjugate gradient (Polak-Ribiere imp.)
        Damped GaussNewton
        Levenberg-Marquardt
```

(continues on next page)

(continued from previous page)

```
        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

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.

Minuit (`minuit`)

CERN developed the Minuit 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

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

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 the 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

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 the 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

The SciPy least squares minimizers are set as follows:

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

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`
- `NIST`

Default is `cutest`

```
[JACOBIAN]
analytic: cutest
```

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`

```
[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 approxmation 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 approximaton) 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`.

```
[JACOBIAN]
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
```

Colour scale (`colour_scale`)

Lists thresholds for each colour in the html table. In the example below, this means that values less than 1.1 will have the top ranking (brightest) and values over 3 will show as the worst ranking (deep red).

If *parameter_ranges* have been set for a problem, then solutions flagged as not respecting the specified bounds will be given the worst ranking colour.

Default thresholds are `1.1, 1.33, 1.75, 3, and inf`

```
[PLOTTING]
colour_scale: 1.1, #fef0d9
              1.33, #fdcc8a
              1.75, #fc8d59
              3, #e34a33
              inf, #b30000
```

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 values 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
```

Results directory (`results_dir`)

This is used to select where the output should be saved

Default is `results`

```
[PLOTTING]
results_dir: 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.

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.

Native File Format

In FitBenchmarking, the native file format is used to read 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 = '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]}
```

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 'Mantid' or 'SasView' (case insensitive).

> This defines whether to use 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)*.

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 is currently unused within FitBenchmarking, but can be useful for explaining problems.

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
>
> x1 y1 [e1]
> x2 y2 [e2]
> ...
> ```
>
> 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.

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.

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, 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']
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]}]
```

(continues on next page)

(continued from previous page)

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 two categories:

- `default`: denotes tests involving `pip` installable *software packages*,
- `all`: in addition to `default`, also runs tests on *external packages*.

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 truely 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.

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 Mantids 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. 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 is 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 neccessary 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.

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

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 method:

   - `CostFunc.`**`eval_cost`**`()`
     Evaluate the cost function
     Parameters **`params`** (*list*) – The parameters to calculate residuals for
     Returns evaluated cost function
     Return type float

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`.

4. Update the analytic Jacobian for the new cost function, *Analytic*.

The FittingProblem and CostFunc

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

cache_model_x = None
dict Container cached function evaluation

data_e = None
numpy array The errors or weights

data_x = None
numpy array The x-data

data_y = None
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.cost_func.base_cost_func.**CostFunc**(*problem*)
Base class for the cost functions.

cache_cost_x = None
dict Container cached residual squared evaluation (cost function)

Adding new Jacobians

This section describes how to add further methods to approximate the Jacobian 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 methods:

   - Jacobian.**eval**()
     Evaluates Jacobian of the model
     Parameters **params** (*list*) – The parameter values to find the Jacobian at
     Returns Computed Jacobian
     Return type numpy array
   - Jacobian.**eval_cost**()
     Evaluates Jacobian of the cost function
     Parameters **params** (*list*) – The parameter values to find the Jacobian at
     Returns Computed derivative of the cost function
     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*.
   - updating any example files in the `examples` directory
4. Create tests for the Jacobian evaluation in `fitbenchmarking/jacobian/tests/test_jacobians.py`.

The `FittingProblem` and `CostFunc`

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

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.

cache_model_x = None

dict Container cached function evaluation

data_e = None

numpy array The errors or weights

data_x = None

numpy array The x-data

data_y = None

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.cost_func.base_cost_func.**CostFunc**(*problem*)

Base class for the cost functions.

cache_cost_x = None

dict Container cached residual squared evaluation (cost function)

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

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

Returns evaluated cost function

Return type float

Note: If using cached values, use the *cached_func_values()* method, which first checks if a cached function evaluation is available to use for the given parameters.

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*. This should implement five functions:

- `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). In this function, you must initialize the a dictionary, `self.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 derivative information. For example, the `Simplex` method in `Mantid` does not require Jacobians, and so is derivative free.
  - `general` - minimizers which solve a generic *min f(x)*.

  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.

  Parameters **cost_func** (subclass of *CostFunc*) – Cost function object selected from options.

- `Controller.jacobian_information()`
  Sets up Jacobian information for the controller.

  This should return the following arguments:
  - `has_jacobian`: a True or False value whether the controller requires Jacobian information.
  - `jacobian_free_solvers`: a list of minimizers in a specific software that do not allow Jacobian information to be passed into the fitting algorithm. For example in the `Scipy` controller this would return `Nelder-Mead` and `Powell`.

Returns (has_jacobian, jacobian_free_solvers)

Return type (*string*, *list*)

- 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),...]

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

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

- 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:

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*). 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.
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 = None
numpy array The errors or weights

data_x = None
numpy array The x-data

data_y = None
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** –

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.

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**(*cost_func*)
Base class for Jacobian.

eval(*params*, ***kwargs*)
Evaluates Jacobian of the model

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

Returns Computed Jacobian

Return type numpy array

eval_cost(*params*, ***kwargs*)
Evaluates Jacobian of the cost function

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

Returns Computed derivative of the cost function

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:

- `Table.`**`get_values`**(*results_dict*)
  Gets the main values to be reported in the tables
  Parameters **`results_dict`** (*dictionary*) – dictionary containing results where the keys are the problem sets and the values are lists of results objects
  Returns tuple of dictionaries which contain the main values in the tables
  Return type tuple

- `Table.`**`display_str`**(*results*)
  Function which converts the results from *get_values()* into a string respresentation to be used in the tables. Base class implementation takes the absolute and relative values and uses `self.output_string_type` as a template for the string format. This can be overwritten to adequately display the results.
  Parameters **`results`** (*tuple*) – tuple containing absolute and relative values
  Returns dictionary containing the string representation of the values in the table.
  Return type dict

Additional functions to be changed are:

- `Table.`**`get_colour`**(*results*)
  Converts the result from *get_values()* into the HTML colours used in the tables. The base class implementation, for example, uses the relative results and `colour_scale` within *Options*. :param results: tuple containing absolute and relative values :type results: tuple :return: dictionary containing HTML colours for the table :rtype: dict

- `Table.`**`colour_highlight`**(*value*, *colour*)
  Takes the HTML colour values from *get_colour()* and maps it over the HTML table using the Pandas style mapper.
  Parameters
  - **`value`** (*pandas.core.series.Series*) – Row data from the pandas array
  - **`colour`** (*dict*) – dictionary containing error codes from the minimizers

  Returns list of HTML colours
  Return type list

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 substutions 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 approprate 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.

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 neccessary.

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 descision. 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.

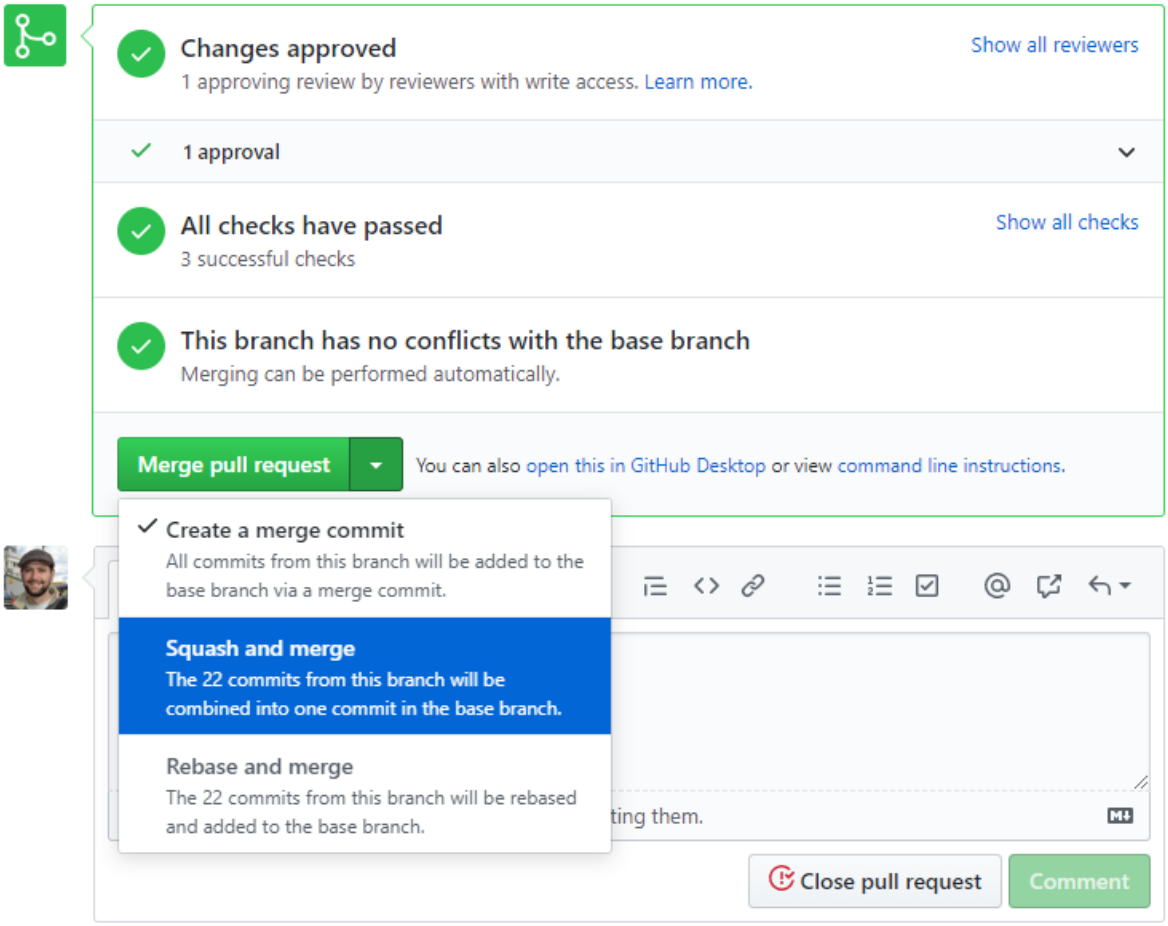
Release branches should have the format *release-major.minor.x*, starting from *release-0.1.x*. When the code is released, we will tag that commit with a version number *v0.1.0*. Any hotfixes will increment *x* by one, and a new tag will be created accordingly. If at some point we don't want to provide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


Changes approved
1 approving review by reviewers with write access. Learn more.
Show all reviewers
1 approval
All checks have passed
3 successful checks
Show all checks
This branch has no conflicts with the base branch
Merging can be performed automatically.
Merge pull request
You can also open this in GitHub Desktop or view command line instructions.
Create a merge commit
All commits from this branch will be added to the base branch via a merge commit.
Squash and merge
The 22 commits from this branch will be combined into one commit in the base branch.
Rebase and merge
The 22 commits from this branch will be rebased and added to the base branch.
ting them.
Close pull request
Comment

```
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>
```

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 lastest 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 lastest 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.

Jacobian (`jacobian`)

This directory holds the `Jacobian` class, and subclasses, which are used by the controllers to approximate derivatives. Currenlty 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*, *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
- **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]

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

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:

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

fit()
Run the fitting.

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

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*

jacobian_information()
Sets up Jacobian information for the controller.

This should return the following arguments:

- `has_jacobian`: a True or False value whether the controller requires Jacobian information.
- `jacobian_free_solvers`: a list of minimizers in a specific software that do not allow Jacobian information to be passed into the fitting algorithm. For example in the `Scipy` controller this would return `Nelder-Mead` and `Powell`.

Returns (`has_jacobian`, `jacobian_free_solvers`)

Return type (*string*, *list*)

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

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),...]

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** (*str*) – 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.

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

fit()
Run problem with Bumps.

jacobian_information()
Bumps cannot use Jacobian information

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.

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

fit()
Run problem with DFO.

jacobian_information()
DFO cannot use Jacobian information

setup()
Setup for DFO

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

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

fit()
Run problem with GSL

jacobian_information()
GSL can use Jacobian information

setup()
Setup for GSL

fitbenchmarking.controllers.levmar_controller module

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', 'weigh
A map from fitbenchmarking cost functions to mantid ones.

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_information()
Mantid cannot use Jacobian information

setup()
Setup problem ready to run with Mantid.

Adds a custom function to Mantid for calling in fit().

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

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

fit()
Run problem with Minuit

jacobian_information()
Minuit cannot use Jacobian information

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.

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

fit()
Run problem with RALFit.

jacobian_information()
RALFit can use Jacobian information

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.

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

fit()
Run problem with Scipy.

jacobian_information()
Scipy can use Jacobian information

setup()
Setup problem ready to be run with SciPy

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.

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

fit()
Run problem with Scipy LS.

jacobian_information()
Scipy LS can use Jacobian information

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()
```

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 prob_results array of fitting results for the problem group, list of failed problems and dictionary of unselected minimizers, rst description of the cost function from the docstring

Return type tuple(list, list, dict, 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 prob_results array of fitting results for the problem group, list of failed problems and dictionary of unselected minimizers, rst description of the cost function from the docstring

Return type tuple(list, list, dict, 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 list of all results, failed problem names, dictionary of unselected minimizers based on algorithm_type and dictionary of minimizers together with the Jacobian used

Return type tuple(list of fibenchmarking.utils.fitbm_result.FittingResult, list of failed problem names, dictionary of minimizers. dictionary of minimizers and Jacobians)

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 list of all results, dictionary of unselected minimizers based on algorithm_type and dictionary of minimizers together with the Jacobian used

Return type tuple(list of fibenchmarking.utils.fitbm_result.FittingResult, list of failed minimizers, list of minimizers and Jacobians)

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 list of all results, dictionary of unselected minimizers based on algorithm_type and dictionary of minimizers together with the Jacobian used

Return type tuple(list of fibenchmarking.utils.fitbm_result.FittingResult, list of failed minimizers, list of minimizers and Jacobians)

fitbenchmarking.core.fitting_benchmarking.**loop_over_starting_values**(*cost_func*, *options*, *grabbed_output*)

Loops over starting values from the fitting cost_func

Parameters

- **cost_func** (CostFunc) – a cost_func object containing information 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 prob_results array of fitting results for the problem group, list of failed problems, dictionary of unselected minimizers and dictionary of minimizers together with the Jacobian used

Return type tuple(list, list, dict, dict)

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 results, group results, support pages, and figures directories

Return type (str, 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 of fitbenchmarking.utils.fitbm_result.FittingResult*) – best result for each problem
- **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*, *cost_func_description*)
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
- **cost_func_description** (*str*) – cost function description

fitbenchmarking.core.results_output.**preproccess_data**(*results_per_test*)
Preprocess data into the right format for printing and find the best result for each problem

Parameters **results_per_test** (*list of list of fitbenchmarking.utils.fitbm_result.FittingResult*) – results nested array of objects

Returns The best result for each problem

Return type list of fitbenchmarking.utils.fitbm_result.FittingResult

fitbenchmarking.core.results_output.**save_results**(*options*, *results*, *group_name*, *failed_problems*, *unselected_minimzers*, *cost_func_description*)
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 of list of fitbenchmarking.utils.fitbm_result.FittingResult*) – results nested array of objects
- **group_name** (*str*) – name of the problem group
- **failed_problems** (*list*) – list of failed problems to be reported in the html output
- **cost_func_description** (*str*) – cost function description

Params unselected_minimzers 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.

cache_cost_x = None
dict Container cached residual squared evaluation (cost function)

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

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

Returns evaluated cost function

Return type float

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.

cache_rx = None
dict Container cached residual evaluation

eval_r(*params*, ***kwargs*)
Calculate residuals

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

Returns The residuals for the datapoints at the given parameters

Return type numpy array

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*

cache_rx = None
dict Container cached residual evaluation

eval_cost(*params*, ***kwargs*)
Evaluate the square of the L2 norm of the residuals

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

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

Return type numpy array

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

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.

cache_rx = None
dict Container cached residual evaluation

eval_r(*params*, ***kwargs*)
Calculate residuals

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

Returns The residuals for the datapoints at the given parameters

Return type numpy array

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 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

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.

cache_rx = None
dict Container cached residual evaluation

eval_r (*params*, ***kwargs*)
Calculate residuals

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

Returns The residuals for the data points at the given parameters

Return type 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**(*cost_func*)
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

eval_cost(*params*, ***kwargs*)
Evaluates derivative of the cost function

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

Returns Computed derivative of the cost function

Return type numpy array

fitbenchmarking.jacobian.base_jacobian module

Implements the base class for the Jacobian.

class fitbenchmarking.jacobian.base_jacobian.**Jacobian**(*cost_func*)
Bases: object

Base class for Jacobian.

cached_func_values(*cached_dict*, *eval_model*, *params*, ***kwargs*)
Computes function values using cached or function evaluation

Parameters

- **cached_dict** (*dict*) – Cached function values
- **eval_modelunc** (*Callable*) – Function to find the Jacobian for
- **params** (*list*) – The parameter values to find the Jacobian at

Returns Function evaluation

Return type numpy array

eval(*params*, ***kwargs*)
Evaluates Jacobian of the model

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

Returns Computed Jacobian

Return type numpy array

eval_cost (*params*, ***kwargs*)
Evaluates Jacobian of the cost function

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

Returns Computed derivative of the cost function

Return type numpy array

method
Utility function to get the numerical method

Returns the names of the parameters

Return type list of str

fitbenchmarking.jacobian.default_jacobian module

Module which uses the minimizer's default jacobian

class fitbenchmarking.jacobian.default_jacobian.**default** (*cost_func*)
Bases: *fitbenchmarking.jacobian.base_jacobian.Jacobian*

Use the minimizer's jacobian/derivative approximation

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** (*cost_func*)
Bases: *fitbenchmarking.jacobian.base_jacobian.Jacobian*

Implements SciPy finite difference approximations to the derivative

eval (*params*, ***kwargs*)
Evaluates Jacobian

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

Returns Approximation of the Jacobian

Return type numpy array

eval_cost (*params*, ***kwargs*)
Evaluates derivative of the cost function

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

Returns Computed derivative of the cost function

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.

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()

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 = None

dict Container for software specific information. This should be avoided if possible.

cache_model_x = None

dict Container cached function evaluation

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 = None

numpy array The errors or weights

data_x = None

numpy array The x-data

data_y = None

numpy array The y-data

end_x = None

float The end of the range to fit model data over (if different from entire range) (/float/)

equation = None

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 = None
string Name of the problem definition type (e.g., 'cutest')

function = None
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

jacobian = None
Callable function for the Jacobian

multifit = None
bool Used to check if a problem is using multifit.

name = None
string Name (title) of the fitting problem

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** –

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

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

sorted_index = None
numpy array The index for sorting the data (used in plotting)

start_x = None
float The start of the range to fit model data over (if different from entire range)

starting_values = None
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 = None
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.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_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*

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_values(*results_dict*)

Gets the main values to be reported in the tables

Parameters **results_dict** (*dictionary*) – dictionary containing results where the keys are the problem sets and the values are lists of results objects

Returns two dictionaries containing the absolute chi_sq and the normalised chi_sq with respect to the smallest chi_sq value.

Return type tuple(dict, dict)

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.

colour_highlight(*value*, *colour*)

Takes the HTML colour values from *get_colour()* and maps it over the HTML table using the Pandas style mapper.

Parameters

- **value** (*pandas.core.series.Series*) – Row data from the pandas array
- **colour** (*dict*) – dictionary containing error codes from the minimizers

Returns list of HTML colours

Return type list

create_pandas_data_frame(*str_results*)

Converts dictionary of results into a pandas data frame

Parameters **str_results** (*dict*) – dictionary containing the string representation of the values in the table.

Returns pandas data frame with from results

Return type Pandas DataFrame

create_results_dict()
Generates a dictionary used to create HTML and txt tables.

Returns dictionary containing results where the keys are the problem sets and the values are lists of results objects

Return type dictionary

display_str(*results*)
Function which converts the results from *get_values()* into a string respresentation to be used in the tables. Base class implementation takes the absolute and relative values and uses `self.output_string_type` as a template for the string format. This can be overwritten to adequately display the results.

Parameters **results** (*tuple*) – tuple containing absolute and relative values

Returns dictionary containing the string representation of the values in the table.

Return type dict

enable_error(*value*, *error*, *template*)
Enable error codes in table

Parameters

- **value** (*pandas.core.series.Series*) – Row data from the pandas array
- **error** (*dict*) – dictionary containing error codes from the minimizers

Returns Row data from the pandas array with error codes enabled

Return type pandas.core.series.Series

enable_link(*value*, *links*)
Enable HTML links in values

Parameters

- **value** (*pandas.core.series.Series*) – Row data from the pandas array
- **links** (*dict*) – dictionary containing links to the support pages

Returns Row data from the pandas array with links enabled

Return type pandas.core.series.Series

file_path
Getter function for the path to the table

Returns path to table

Return type str

get_colour(*results*)
Converts the result from *get_values()* into the HTML colours used in the tables. The base class implementation, for example, uses the relative results and `colour_scale` within *Options*. :param results: tuple containing absolute and relative values :type results: tuple :return: dictionary containing HTML colours for the table :rtype: dict

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

get_error(*results_dict*)
Pulls out error code from the results object

Parameters **results** (*tuple*) – tuple containing absolute and relative values

Returns dictionary containing error codes from the minimizers

Return type dict

get_links(*results_dict*)
Pulls out links to the individual support pages from the results object

Parameters **results** (*tuple*) – tuple containing absolute and relative values

Returns dictionary containing links to the support pages

Return type dict

get_values(*results_dict*)
Gets the main values to be reported in the tables

Parameters **results_dict** (*dictionary*) – dictionary containing results where the keys are the problem sets and the values are lists of results objects

Returns tuple of dictionaries which contain the main values in the tables

Return type tuple

table_title
Getter function for table name if self._table_title is None

Returns name of table

Return type str

to_html(*table*, *colour*, *links*, *error*)
Takes Pandas data frame and converts it into the HTML table output

Parameters

- **table** (*Pandas DataFrame*) – pandas data frame with from results
- **colour** (*dict*) – dictionary containing error codes from the minimizers
- **links** (*dict*) – dictionary containing links to the support pages
- **error** (*dict*) – dictionary containing error codes from the minimizers

Returns HTLM table output

Return type str

to_txt(*table*, *error*)
Takes Pandas data frame and converts it into the plain text table output

Parameters

- **table** (*Pandas DataFrame*) – pandas data frame with from results
- **error** (*dict*) – dictionary containing error codes from the minimizers

Returns plain text table output

Return type 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.

colour_highlight(*value*, *colour*)

Enable HTML colours in the HTML output.

Parameters

- **value** (*pandas.core.series.Series*) – Row data from the pandas array
- **colour** (*dict*) – dictionary containing error codes from the minimizers

Returns list of HTML colours

Return type list

display_str(*results*)

Function that combines the accuracy and runtime results into the string representation.

Parameters **results** (*tuple*) – tuple containing absolute and relative values

Returns dictionary containing the string representation of the values in the table.

Return type dict

get_values(*results_dict*)

Gets the main values to be reported in the tables

Parameters **results_dict** (*dictionary*) – dictionary containing results where the keys are the problem sets and the values are lists of results objects

Returns First dictionary containing the absolute chi_sq and runtime results and the second contains the normalised chi_sq and runtime results with respect to the smallest values.

Return type tuple(dict, dict)

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(*results*)

Function that combines the True and False value from variable local_min with the normalised residual

Parameters **results** (*tuple*) – a dictionary containing true or false values whether the return parameters is a local minimizer and a dictionary containing $\frac{||J^T r||}{||r||}$ values

Returns dictionary containing the string representation of the values in the table.

Return type dict

get_colour(*results*)

Uses the local minimizer dictionary values to set the HTML colour :param results: a dictionary containing true or false values whether

the return parameters is a local minimizer and a dictionary containing $\frac{||J^T r||}{||r||}$ values

Returns dictionary containing error codes from the minimizers

Return type dict

get_values(*results_dict*)

Gets the main values to be reported in the tables

Parameters **results_dict** (*dictionary*) – dictionary containing results where the keys are the problem sets and the values are lists of results objects

Returns a dictionary containing true or false values whether the return parameters is a local minimizer and a dictionary containing $\frac{||J^T r||}{||r||}$ values

Return type tuple(dict, dict)

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*, *solvers*)

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 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 number of occurrences that the minimizer produces a normalised result which is less than the bounds in PROFILER_BOUNDS

Parameters **results** (*list of list of fitbenchmarking.utils.fitbm_result.FittingResult*) – results nested array of objects

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** (*list of list of fitbenchmarking.utils.fitbm_result.FittingResult*) – results nested array of objects
- **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.

format_plot()
Performs post plot processing to annotate the plot correctly

plot_best(*minimizer*, *params*)
Plots the fit along with the data using the "best_fit" style and saves to a file

Parameters

- **minimizer** (*str*) – name of the best fit minimizer
- **params** (*list*) – fit parameters returned from the best fit minimizer

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(*minimizer*, *params*)
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

- **minimizer** (*str*) – name of the fit minimizer
- **params** (*list*) – fit parameters returned from the best fit minimizer

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

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 using the timeit module in python. The number of runtimes can be set in *FitBenchmarking Options*.

get_values(*results_dict*)
Gets the main values to be reported in the tables

Parameters **results_dict** (*dictionary*) – dictionary containing results where the keys are the problem sets and the values are lists of results objects

Returns two dictionaries containing the absolute runtime and the normalised runtime with respect to the quickest time.

Return type tuple(dict, dict)

fitbenchmarking.results_processing.support_page module

Set up and build the support pages for various types of problems.

fitbenchmarking.results_processing.support_page.**create**(*results_per_test*, *group_name*, *support_pages_dir*, *options*)
Iterate through problem results and create a support html page for each.

Parameters

- **results_per_test** (*list[list[list]]*) – results object
- **group_name** (*str*) – name of the problem group
- **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.support_page.**create_prob_group**(*prob_results*, *group_name*, *support_pages_dir*, *options*)
Creates a support page containing figures and other details about the fit for a problem. A link to the support page is stored in the results object.

Parameters

- **prob_results** (*list[*fitbenchmarking.utils.fitbm_result.FittingResult*]*) – problem results objects containing results for each minimizer and a certain fitting function
- **group_name** (*str*) – name of the problem group
- **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.support_page.**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.tables module

Set up and build the results tables.

fitbenchmarking.results_processing.tables.**create_results_tables**(*options*, *results*, *best_results*, *group_name*, *group_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** (*list of list of fitbenchmarking.utils.fitbm_result.FittingResult*) – results nested array of objects
- **best_results** (*list of fitbenchmarking.utils.fitbm_result.FittingResult*) – best result for each problem
- **group_name** (*str*) – name of the problem group
- **group_dir** (*str*) – path to the directory where group results should be stored
- **table_descriptions** (*dict*) – dictionary containing descriptions of the tables and the comparison mode
- **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*, *pp_locations*, *table_name*, *suffix*)

Generate html/txt tables.

Parameters

- **results** (*list of list of fitbenchmarking.utils.fitbm_result.FittingResult*) – results nested array of objects
- **best_results** (*list of fitbenchmarking.utils.fitbm_result.FittingResult*) – best result for each problem

- **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
- **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, HTML string of table and text string of table.

Return type tuple(Table object, 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.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

exception fitbenchmarking.utils.exceptions.**CostFuncError**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates a problem with the cost function class.

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

exception fitbenchmarking.utils.exceptions.**FittingProblemError**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates a problem with the fitting problem.

exception fitbenchmarking.utils.exceptions.**IncompatibleMinimizerError**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates that the selected minimizer is not compatible with selected options/problem set

exception fitbenchmarking.utils.exceptions.**IncompatibleTableError**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates that selected cost function and table are not compatible

exception fitbenchmarking.utils.exceptions.**IncorrectBoundsError**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates that *parameter_ranges* have been set incorrectly

exception fitbenchmarking.utils.exceptions.**MissingSoftwareError**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates that the requirements for a software package are not available.

exception fitbenchmarking.utils.exceptions.**NoAnalyticJacobian**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates when no Jacobian data files can be found

exception fitbenchmarking.utils.exceptions.**NoControllerError**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates a controller could not be found

exception fitbenchmarking.utils.exceptions.**NoDataError**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates that no data could be found.

exception fitbenchmarking.utils.exceptions.**NoJacobianError**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates a problem with the Jacobian import.

exception fitbenchmarking.utils.exceptions.**NoParserError**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates a parser could not be found.

exception fitbenchmarking.utils.exceptions.**NoResultsError**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates a problem with the fitting problem.

exception fitbenchmarking.utils.exceptions.**OptionsError**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates an error during processing options.

exception fitbenchmarking.utils.exceptions.**ParsingError**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates an error during parsing.

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.

exception fitbenchmarking.utils.exceptions.**UnknownTableError**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates a problem with the fitting problem.

exception fitbenchmarking.utils.exceptions.**UnsupportedMinimizerError**(*message=''*)
Bases: *fitbenchmarking.utils.exceptions.FitBenchmarkException*

Indicates that the controller does not support a given minimizer.

fitbenchmarking.utils.fitbm_result module

FitBenchmarking results object

class fitbenchmarking.utils.fitbm_result.**FittingResult**(*options*, *cost_func*, *jac*, *initial_params*, *params*, *name=None*, *chi_sq=None*, *runtime=None*, *minimizer=None*, *error_flag=None*, *dataset_id=None*)

Bases: object

Minimal definition of a class to hold results from a fitting problem test.

norm_acc

Getting function for norm_acc attribute

Returns normalised accuracy value

Return type float

norm_runtime

Getting function for norm_runtime attribute

Returns normalised runtime value

Return type float

sanitised_min_name

Sanitise the minimizer name into one which can be used as a filename.

Returns sanitised name

Return type str

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*)
Bases: object

An options class to store and handle all options for fitbenchmarking

DEFAULTS = {'FITTING': {'algorithm_type': 'all', 'cost_func_type': 'weighted_nlls',

DEFAULT_FITTING = {'algorithm_type': 'all', 'cost_func_type': 'weighted_nlls', 'jac_

DEFAULT_JACOBIAN = {'analytic': ['cutest'], 'default': ['default'], 'numdifftools':

DEFAULT_LOGGING = {'append': False, 'external_output': 'log_only', 'file_name': 'fi

DEFAULT_MINIMZERS = {'bumps': ['amoeba', 'lm-bumps', 'newton', 'mp'], 'dfo': ['dfogn

DEFAULT_PLOTTING = {'colour_scale': [(1.1, '#fef0d9'), (1.33, '#fdcc8a'), (1.75, '#fc

VALID = {'FITTING': {'algorithm_type': ['all', 'ls', 'deriv_free', 'general'], 'cost_

VALID_FITTING = {'algorithm_type': ['all', 'ls', 'deriv_free', 'general'], 'cost_func_

VALID_JACOBIAN = {'analytic': ['cutest'], 'default': ['default'], 'numdifftools': [

VALID_LOGGING = {'append': [True, False], 'external_output': ['debug', 'display', 'l

VALID_MINIMIZERS = {'bumps': ['amoeba', 'lm-bumps', 'newton', 'de', 'mp'], 'dfo': ['

VALID_PLOTTING = {'comparison_mode': ['abs', 'rel', 'both'], 'make_plots': [True, Fa

VALID_SECTIONS = ['MINIMIZERS', 'FITTING', 'JACOBIAN', 'PLOTTING', 'LOGGING']

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.

results_dir
Returns the path to the results directory

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.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*.

Module contents

Module contents

Python Module Index

f

fitbenchmarking, 86
fitbenchmarking.cli, 51
fitbenchmarking.cli.exception_handler, 50
fitbenchmarking.cli.main, 50
fitbenchmarking.controllers, 56
fitbenchmarking.controllers.base_controller, 51
fitbenchmarking.controllers.bumps_controller, 53
fitbenchmarking.controllers.controller_factory, 53
fitbenchmarking.controllers.dfo_controller, 53
fitbenchmarking.controllers.gsl_controller, 54
fitbenchmarking.controllers.mantid_controller, 54
fitbenchmarking.controllers.minuit_controller, 55
fitbenchmarking.controllers.ralfit_controller, 55
fitbenchmarking.controllers.scipy_controller, 56
fitbenchmarking.controllers.scipy_ls_controller, 56
fitbenchmarking.core, 60
fitbenchmarking.core.fitting_benchmarking, 56
fitbenchmarking.core.results_output, 59
fitbenchmarking.cost_func, 64
fitbenchmarking.cost_func.base_cost_func, 60
fitbenchmarking.cost_func.cost_func_factory, 61
fitbenchmarking.cost_func.hellinger_nlls_cost_func, 61
fitbenchmarking.cost_func.nlls_base_cost_func, 61
fitbenchmarking.cost_func.nlls_cost_func, 62
fitbenchmarking.cost_func.poisson_cost_func, 62
fitbenchmarking.cost_func.weighted_nlls_cost_func, 63
fitbenchmarking.jacobian, 66
fitbenchmarking.jacobian.analytic_jacobian, 64
fitbenchmarking.jacobian.base_jacobian, 64
fitbenchmarking.jacobian.default_jacobian, 65
fitbenchmarking.jacobian.jacobian_factory, 65
fitbenchmarking.jacobian.scipy_jacobian, 65
fitbenchmarking.parsing, 71
fitbenchmarking.parsing.base_parser, 66
fitbenchmarking.parsing.cutest_parser, 66
fitbenchmarking.parsing.fitbenchmark_parser, 67
fitbenchmarking.parsing.fitting_problem, 67
fitbenchmarking.parsing.nist_data_functions, 69
fitbenchmarking.parsing.nist_parser, 70
fitbenchmarking.parsing.parser_factory, 70
fitbenchmarking.results_processing, 80
fitbenchmarking.results_processing.acc_table, 71
fitbenchmarking.results_processing.base_table, 71
fitbenchmarking.results_processing.compare_table, 74
fitbenchmarking.results_processing.local_min_table, 74

fitbenchmarking.results_processing.performance_profiler,
75
fitbenchmarking.results_processing.plots,
76
fitbenchmarking.results_processing.runtime_table,
77
fitbenchmarking.results_processing.support_page,
78
fitbenchmarking.results_processing.tables,
79
fitbenchmarking.utils, 86
fitbenchmarking.utils.create_dirs, 80
fitbenchmarking.utils.exceptions, 81
fitbenchmarking.utils.fitbm_result, 82
fitbenchmarking.utils.log, 83
fitbenchmarking.utils.misc, 84
fitbenchmarking.utils.options, 84
fitbenchmarking.utils.output_grabber,
85

Index

A

AccTable (*class in fitbenchmarking.results_processing.acc_table*), 71
additional_info (*fitbenchmarking.parsing.fitting_problem.FittingProblem attribute*), 67
Analytic (*class in fitbenchmarking.jacobian.analytic_jacobian*), 64

B

BaseNLLSCostFunc (*class in fitbenchmarking.cost_func.nlls_base_cost_func*), 61
benchmark() (*in module fitbenchmarking.core.fitting_benchmarking*), 56
BumpsController (*class in fitbenchmarking.controllers.bumps_controller*), 53

C

cache_cost_x (*fitbenchmarking.cost_func.base_cost_func.CostFunc attribute*), 60
cache_model_x (*fitbenchmarking.parsing.fitting_problem.FittingProblem attribute*), 67
cache_rx (*fitbenchmarking.cost_func.hellinger_nlls_cost_func.HellingerNLLSCostFunc attribute*), 61
cache_rx (*fitbenchmarking.cost_func.nlls_base_cost_func.BaseNLLSCostFunc attribute*), 62
cache_rx (*fitbenchmarking.cost_func.nlls_cost_func.NLLSCostFunc attribute*), 62
cache_rx (*fitbenchmarking.cost_func.weighted_nlls_cost_func.WeightedNLLSCostFunc attribute*), 63
cached_func_values() (*fitbenchmarking.jacobian.base_jacobian.Jacobian method*), 64
check_attributes() (*fitbenchmarking.controllers.base_controller.Controller method*), 51
check_bounds_respected() (*fitbenchmarking.controllers.base_controller.Controller method*), 51
check_minimizer_bounds() (*fitbenchmarking.controllers.base_controller.Controller method*), 51
cleanup() (*fitbenchmarking.controllers.base_controller.Controller method*), 51
cleanup() (*fitbenchmarking.controllers.bumps_controller.BumpsController method*), 53
cleanup() (*fitbenchmarking.controllers.dfo_controller.DFOController method*), 53
cleanup() (*fitbenchmarking.controllers.gsl_controller.GSLController method*), 54
cleanup() (*fitbenchmarking.controllers.mantid_controller.MantidController method*), 54
cleanup() (*fitbenchmarking.controllers.minuit_controller.MinuitController method*), 55
cleanup() (*fitbenchmarking.controllers.ralfit_controller.RALFitController method*), 55
cleanup() (*fitbenchmarking.controllers.scipy_controller.ScipyController method*), 56
cleanup() (*fitbenchmarking.controllers.scipy_ls_controller.ScipyLSController method*), 56
colour_highlight() (*fitbenchmarking.results_processing.base_table.Table method*), 71
colour_highlight() (*fitbenchmark-*

ing.results_processing.compare_table.CompareTable method), 74
CompareTable (class in fitbenchmarking.results_processing.compare_table), 74
Controller (class in fitbenchmarking.controllers.base_controller), 51
ControllerAttributeError, 81
ControllerFactory (class in fitbenchmarking.controllers.controller_factory), 53
correct_data() (fitbenchmarking.parsing.fitting_problem.FittingProblem method), 67
correct_data() (in module fitbenchmarking.parsing.fitting_problem), 69
COST_FUNCTION_MAP (fitbenchmarking.controllers.mantid_controller.MantidController attribute), 54
CostFunc (class in fitbenchmarking.cost_func.base_cost_func), 60
CostFuncError, 81
create() (in module fitbenchmarking.results_processing.support_page), 78
create_controller() (fitbenchmarking.controllers.controller_factory.ControllerFactory static method), 53
create_cost_func() (in module fitbenchmarking.cost_func.cost_func_factory), 61
create_directories() (in module fitbenchmarking.core.results_output), 59
create_jacobian() (in module fitbenchmarking.jacobian.jacobian_factory), 65
create_pandas_data_frame() (fitbenchmarking.results_processing.base_table.Table method), 71
create_parser() (fitbenchmarking.parsing.parser_factory.ParserFactory static method), 70
create_plot() (in module fitbenchmarking.results_processing.performance_profiler), 75
create_plots() (in module fitbenchmarking.core.results_output), 59
create_prob_group() (in module fitbenchmarking.results_processing.support_page), 78
create_problem_level_index() (in module fitbenchmarking.core.results_output), 59
create_results_dict() (fitbenchmarking.results_processing.base_table.Table method), 72
create_results_tables() (in module fitbenchmarking.results_processing.tables), 79
css() (in module fitbenchmarking.utils.create_dirs), 80
CutestParser (class in fitbenchmarking.parsing.cutest_parser), 66

D

data_e (fitbenchmarking.parsing.fitting_problem.FittingProblem attribute), 67
data_x (fitbenchmarking.parsing.fitting_problem.FittingProblem attribute), 67
data_y (fitbenchmarking.parsing.fitting_problem.FittingProblem attribute), 67
default (class in fitbenchmarking.jacobian.default_jacobian), 65
DEFAULT_FITTING (fitbenchmarking.utils.options.Options attribute), 84
DEFAULT_JACOBIAN (fitbenchmarking.utils.options.Options attribute), 84
DEFAULT_LOGGING (fitbenchmarking.utils.options.Options attribute), 84
DEFAULT_MINIMZERS (fitbenchmarking.utils.options.Options attribute), 84
DEFAULT_PLOTTING (fitbenchmarking.utils.options.Options attribute), 84
DEFAULTS (fitbenchmarking.utils.options.Options attribute), 84
del_contents_of_dir() (in module fitbenchmarking.utils.create_dirs), 80
DFOController (class in fitbenchmarking.controllers.dfo_controller), 53
display_str() (fitbenchmarking.results_processing.base_table.Table method), 72
display_str() (fitbenchmarking.results_processing.compare_table.CompareTable method), 74
display_str() (fitbenchmarking.results_processing.local_min_table.LocalMinTable method), 75

E

enable_error() (fitbenchmarking.results_processing.base_table.Table method), 72
enable_link() (fitbenchmarking.results_processing.base_table.Table method), 72
end_x (fitbenchmarking.parsing.fitting_problem.FittingProblem attribute), 67
equation (fitbenchmarking.parsing.fitting_problem.FittingProblem attribute), 67
escape_char (fitbenchmarking.utils.output_grabber.StreamGrabber attribute), 85

eval() (*fitbenchmarking.jacobian.analytic_jacobian.Analytic method*), 64
eval() (*fitbenchmarking.jacobian.base_jacobian.Jacobian method*), 64
eval() (*fitbenchmarking.jacobian.scipy_jacobian.Scipy method*), 65
eval_chisq() (*fitbenchmarking.controllers.base_controller.Controller method*), 51
eval_chisq() (*fitbenchmarking.controllers.mantid_controller.MantidController method*), 54
eval_cost() (*fitbenchmarking.cost_func.base_cost_func.CostFunc method*), 60
eval_cost() (*fitbenchmarking.cost_func.nlls_base_cost_func.BaseNLLSCostFunc method*), 62
eval_cost() (*fitbenchmarking.cost_func.poisson_cost_func.PoissonCostFunc method*), 63
eval_cost() (*fitbenchmarking.jacobian.analytic_jacobian.Analytic method*), 64
eval_cost() (*fitbenchmarking.jacobian.base_jacobian.Jacobian method*), 65
eval_cost() (*fitbenchmarking.jacobian.scipy_jacobian.Scipy method*), 65
eval_model() (*fitbenchmarking.parsing.fitting_problem.FittingProblem method*), 67
eval_r() (*fitbenchmarking.cost_func.hellinger_nlls_cost_func.HellingerNLLSCostFunc method*), 61
eval_r() (*fitbenchmarking.cost_func.nlls_base_cost_func.BaseNLLSCostFunc method*), 62
eval_r() (*fitbenchmarking.cost_func.nlls_cost_func.NLLSCostFunc method*), 62
eval_r() (*fitbenchmarking.cost_func.weighted_nlls_cost_func.WeightedNLLSCostFunc method*), 63
exception_handler() (*in module fitbenchmarking.cli.exception_handler*), 50

F

figures() (*in module fitbenchmarking.utils.create_dirs*), 80
file_path (*fitbenchmarking.results_processing.base_table.Table attribute*), 72
fit() (*fitbenchmarking.controllers.base_controller.Controller method*), 52
fit() (*fitbenchmarking.controllers.bumps_controller.BumpsController method*), 53
fit() (*fitbenchmarking.controllers.dfo_controller.DFOController method*), 53
fit() (*fitbenchmarking.controllers.gsl_controller.GSLController method*), 54
fit() (*fitbenchmarking.controllers.mantid_controller.MantidController method*), 55
fit() (*fitbenchmarking.controllers.minuit_controller.MinuitController method*), 55
fit() (*fitbenchmarking.controllers.ralfit_controller.RALFitController method*), 55
fit() (*fitbenchmarking.controllers.scipy_controller.ScipyController method*), 56
fit() (*fitbenchmarking.controllers.scipy_ls_controller.ScipyLSController method*), 56
FitBenchmarkException, 81
fitbenchmarking (*module*), 86
fitbenchmarking.cli (*module*), 51
fitbenchmarking.cli.exception_handler (*module*), 50
fitbenchmarking.cli.main (*module*), 50
fitbenchmarking.controllers (*module*), 56
fitbenchmarking.controllers.base_controller (*module*), 51
fitbenchmarking.controllers.bumps_controller (*module*), 53
fitbenchmarking.controllers.controller_factory (*module*), 53
fitbenchmarking.controllers.dfo_controller (*module*), 53
fitbenchmarking.controllers.gsl_controller (*module*), 54
fitbenchmarking.controllers.mantid_controller (*module*), 54
fitbenchmarking.controllers.minuit_controller (*module*), 55
fitbenchmarking.controllers.ralfit_controller (*module*), 55
fitbenchmarking.controllers.scipy_controller (*module*), 56
fitbenchmarking.controllers.scipy_ls_controller (*module*), 56
fitbenchmarking.core (*module*), 60
fitbenchmarking.core.fitting_benchmarking (*module*), 56
fitbenchmarking.core.results_output (*module*), 59
fitbenchmarking.cost_func (*module*), 64

fitbenchmarking.cost_func.base_cost_func (*module*), 60
fitbenchmarking.cost_func.cost_func_factory (*module*), 61
fitbenchmarking.cost_func.hellinger_nlls_cost_func (*module*), 61
fitbenchmarking.cost_func.nlls_base_cost_func (*module*), 61
fitbenchmarking.cost_func.nlls_cost_func (*module*), 62
fitbenchmarking.cost_func.poisson_cost_func (*module*), 62
fitbenchmarking.cost_func.weighted_nlls_cost_func (*module*), 63
fitbenchmarking.jacobian (*module*), 66
fitbenchmarking.jacobian.analytic_jacobian (*module*), 64
fitbenchmarking.jacobian.base_jacobian (*module*), 64
fitbenchmarking.jacobian.default_jacobian (*module*), 65
fitbenchmarking.jacobian.jacobian_factory (*module*), 65
fitbenchmarking.jacobian.scipy_jacobian (*module*), 65
fitbenchmarking.parsing (*module*), 71
fitbenchmarking.parsing.base_parser (*module*), 66
fitbenchmarking.parsing.cutest_parser (*module*), 66
fitbenchmarking.parsing.fitbenchmark_parser (*module*), 67
fitbenchmarking.parsing.fitting_problem (*module*), 67
fitbenchmarking.parsing.nist_data_functions (*module*), 69
fitbenchmarking.parsing.nist_parser (*module*), 70
fitbenchmarking.parsing.parser_factory (*module*), 70
fitbenchmarking.results_processing (*module*), 80
fitbenchmarking.results_processing.acc_table (*module*), 71
fitbenchmarking.results_processing.base_table (*module*), 71
fitbenchmarking.results_processing.compare_table (*module*), 74
fitbenchmarking.results_processing.local_min_table (*module*), 74
fitbenchmarking.results_processing.performance_profiler (*module*), 75
fitbenchmarking.results_processing.plots (*module*), 76
fitbenchmarking.results_processing.runtime_table (*module*), 77
fitbenchmarking.results_processing.support_page (*module*), 78
fitbenchmarking.results_processing.tables (*module*), 79
fitbenchmarking.utils (*module*), 86
fitbenchmarking.utils.create_dirs (*module*), 80
fitbenchmarking.utils.exceptions (*module*), 81
fitbenchmarking.utils.fitbm_result (*module*), 82
fitbenchmarking.utils.log (*module*), 83
fitbenchmarking.utils.misc (*module*), 84
fitbenchmarking.utils.options (*module*), 84
fitbenchmarking.utils.output_grabber (*module*), 85
FitbenchmarkParser (*class in fitbenchmarking.parsing.fitbenchmark_parser*), 67
FittingProblem (*class in fitbenchmarking.parsing.fitting_problem*), 67
FittingProblemError, 81
FittingResult (*class in fitbenchmarking.utils.fitbm_result*), 82
flag (*fitbenchmarking.controllers.base_controller.Controller attribute*), 52
format (*fitbenchmarking.parsing.fitting_problem.FittingProblem attribute*), 68
format_function_scipy() (*in module fitbenchmarking.parsing.nist_data_functions*), 69
format_plot() (*fitbenchmarking.results_processing.plots.Plot method*), 76
function (*fitbenchmarking.parsing.fitting_problem.FittingProblem attribute*), 68

G

generate_table() (*in module fitbenchmarking.results_processing.tables*), 79
get_colour() (*fitbenchmarking.results_processing.base_table.Table method*), 72
get_colour() (*fitbenchmarking.results_processing.local_min_table.LocalMinTable method*), 75
get_css() (*in module fitbenchmarking.utils.misc*), 84
get_description() (*fitbenchmarking.results_processing.base_table.Table method*), 72
get_error() (*fitbenchmarking.results_processing.base_table.Table*

method), 73
get_figure_paths() (*in module fitbenchmarking.results_processing.support_page*), 78
get_function_params() (*fitbenchmarking.parsing.fitting_problem.FittingProblem method*), 68
get_js() (*in module fitbenchmarking.utils.misc*), 84
get_links() (*fitbenchmarking.results_processing.base_table.Table method*), 73
get_logger() (*in module fitbenchmarking.utils.log*), 83
get_parser() (*in module fitbenchmarking.cli.main*), 50
get_problem_files() (*in module fitbenchmarking.utils.misc*), 84
get_values() (*fitbenchmarking.results_processing.acc_table.AccTable method*), 71
get_values() (*fitbenchmarking.results_processing.base_table.Table method*), 73
get_values() (*fitbenchmarking.results_processing.compare_table.CompareTable method*), 74
get_values() (*fitbenchmarking.results_processing.local_min_table.LocalMinTable method*), 75
get_values() (*fitbenchmarking.results_processing.runtime_table.RuntimeTable method*), 78
group_results() (*in module fitbenchmarking.utils.create_dirs*), 80
GSLController (*class in fitbenchmarking.controllers.gsl_controller*), 54

H

HellingerNLLSCostFunc (*class in fitbenchmarking.cost_func.hellinger_nlls_cost_func*), 61

I

IncompatibleMinimizerError, 81
IncompatibleTableError, 81
IncorrectBoundsError, 81
is_int() (*in module fitbenchmarking.parsing.nist_data_functions*), 69
is_safe() (*in module fitbenchmarking.parsing.nist_data_functions*), 69

J

Jacobian (*class in fitbenchmarking.jacobian.base_jacobian*), 64
jacobian (*fitbenchmarking.parsing.fitting_problem.FittingProblem attribute*), 68
jacobian_information() (*fitbenchmarking.controllers.base_controller.Controller method*), 52
jacobian_information() (*fitbenchmarking.controllers.bumps_controller.BumpsController method*), 53
jacobian_information() (*fitbenchmarking.controllers.dfo_controller.DFOController method*), 53
jacobian_information() (*fitbenchmarking.controllers.gsl_controller.GSLController method*), 54
jacobian_information() (*fitbenchmarking.controllers.mantid_controller.MantidController method*), 55
jacobian_information() (*fitbenchmarking.controllers.minuit_controller.MinuitController method*), 55
jacobian_information() (*fitbenchmarking.controllers.ralfit_controller.RALFitController method*), 55
jacobian_information() (*fitbenchmarking.controllers.scipy_controller.ScipyController method*), 56
jacobian_information() (*fitbenchmarking.controllers.scipy_ls_controller.ScipyLSController method*), 56

L

load_table() (*in module fitbenchmarking.results_processing.tables*), 80
LocalMinTable (*class in fitbenchmarking.results_processing.local_min_table*), 74
loop_over_benchmark_problems() (*in module fitbenchmarking.core.fitting_benchmarking*), 57
loop_over_fitting_software() (*in module fitbenchmarking.core.fitting_benchmarking*), 57
loop_over_jacobians() (*in module fitbenchmarking.core.fitting_benchmarking*), 58
loop_over_minimizers() (*in module fitbenchmarking.core.fitting_benchmarking*), 58
loop_over_starting_values() (*in module fitbenchmarking.core.fitting_benchmarking*), 58

M

main() (*in module fitbenchmarking.cli.main*), 51
MantidController (*class in fitbenchmarking.controllers.mantid_controller*), 54

method (*fitbenchmarking.jacobian.base_jacobian.Jacobian attribute*), 65
minimizers (*fitbenchmarking.utils.options.Options attribute*), 85
MinuitController (*class in fitbenchmarking.controllers.minuit_controller*), 55
MissingSoftwareError, 81
multifit (*fitbenchmarking.parsing.fitting_problem.FittingProblem attribute*), 68

N

name (*fitbenchmarking.parsing.fitting_problem.FittingProblem attribute*), 68
nist_func_definition() (*in module fitbenchmarking.parsing.nist_data_functions*), 69
nist_jacobian_definition() (*in module fitbenchmarking.parsing.nist_data_functions*), 70
NISTParser (*class in fitbenchmarking.parsing.nist_parser*), 70
NLLSCostFunc (*class in fitbenchmarking.cost_func.nlls_cost_func*), 62
NoAnalyticJacobian, 82
NoControllerError, 82
NoDataError, 82
NoJacobianError, 82
NoParserError, 82
NoResultsError, 82
norm_acc (*fitbenchmarking.utils.fitbm_result.FittingResult attribute*), 83
norm_runtime (*fitbenchmarking.utils.fitbm_result.FittingResult attribute*), 83

O

Options (*class in fitbenchmarking.utils.options*), 84
OptionsError, 82
OutputGrabber (*class in fitbenchmarking.utils.output_grabber*), 85

P

param_names (*fitbenchmarking.parsing.fitting_problem.FittingProblem attribute*), 68
parse() (*fitbenchmarking.parsing.base_parser.Parser method*), 66
parse() (*fitbenchmarking.parsing.cutest_parser.CutestParser method*), 66
parse() (*fitbenchmarking.parsing.fitbenchmark_parser.FitbenchmarkParser method*), 67
parse() (*fitbenchmarking.parsing.nist_parser.NISTParser method*), 70
parse_problem_file() (*in module fitbenchmarking.parsing.parser_factory*), 70
Parser (*class in fitbenchmarking.parsing.base_parser*), 66
ParserFactory (*class in fitbenchmarking.parsing.parser_factory*), 70
ParsingError, 82
Plot (*class in fitbenchmarking.results_processing.plots*), 76
plot() (*in module fitbenchmarking.results_processing.performance_profiler*), 76
plot_best() (*fitbenchmarking.results_processing.plots.Plot method*), 77
plot_data() (*fitbenchmarking.results_processing.plots.Plot method*), 77
plot_fit() (*fitbenchmarking.results_processing.plots.Plot method*), 77
plot_initial_guess() (*fitbenchmarking.results_processing.plots.Plot method*), 77
PoissonCostFunc (*class in fitbenchmarking.cost_func.poisson_cost_func*), 62
prepare() (*fitbenchmarking.controllers.base_controller.Controller method*), 52
prepare_profile_data() (*in module fitbenchmarking.results_processing.performance_profiler*), 76
preproccess_data() (*in module fitbenchmarking.core.results_output*), 59
profile() (*in module fitbenchmarking.results_processing.performance_profiler*), 76

R

RALFitController (*class in fitbenchmarking.controllers.ralfit_controller*), 55
read_list() (*in module fitbenchmarking.utils.options*), 85
read_value() (*fitbenchmarking.utils.options.Options method*), 85
readOutput() (*fitbenchmarking.utils.output_grabber.StreamGrabber method*), 85
reset() (*fitbenchmarking.utils.options.Options method*), 85

results() (*in module fitbenchmarking.utils.create_dirs*), 81
results_dir (*fitbenchmarking.utils.options.Options attribute*), 85
run() (*in module fitbenchmarking.cli.main*), 51
RuntimeTable (*class in fitbenchmarking.results_processing.runtime_table*), 77

S

sanitised_min_name (*fitbenchmarking.utils.fitbm_result.FittingResult attribute*), 83
sanitised_name (*fitbenchmarking.utils.fitbm_result.FittingResult attribute*), 83
save_results() (*in module fitbenchmarking.core.results_output*), 60
Scipy (*class in fitbenchmarking.jacobian.scipy_jacobian*), 65
ScipyController (*class in fitbenchmarking.controllers.scipy_controller*), 56
ScipyLSController (*class in fitbenchmarking.controllers.scipy_ls_controller*), 56
set_value_ranges() (*fitbenchmarking.parsing.fitting_problem.FittingProblem method*), 68
setup() (*fitbenchmarking.controllers.base_controller.Controller method*), 52
setup() (*fitbenchmarking.controllers.bumps_controller.BumpsController method*), 53
setup() (*fitbenchmarking.controllers.dfo_controller.DFOController method*), 54
setup() (*fitbenchmarking.controllers.gsl_controller.GSLController method*), 54
setup() (*fitbenchmarking.controllers.mantid_controller.MantidController method*), 55
setup() (*fitbenchmarking.controllers.minuit_controller.MinuitController method*), 55
setup() (*fitbenchmarking.controllers.ralfit_controller.RALFitController method*), 55
setup() (*fitbenchmarking.controllers.scipy_controller.ScipyController method*), 56
setup() (*fitbenchmarking.controllers.scipy_ls_controller.ScipyLSController method*), 56
setup_logger() (*in module fitbenchmarking.utils.log*), 83
sorted_index (*fitbenchmarking.parsing.fitting_problem.FittingProblem attribute*), 68
start() (*fitbenchmarking.utils.output_grabber.StreamGrabber method*), 86
start_x (*fitbenchmarking.parsing.fitting_problem.FittingProblem attribute*), 68
starting_values (*fitbenchmarking.parsing.fitting_problem.FittingProblem attribute*), 68
stop() (*fitbenchmarking.utils.output_grabber.StreamGrabber method*), 86
StreamGrabber (*class in fitbenchmarking.utils.output_grabber*), 85
support_pages() (*in module fitbenchmarking.utils.create_dirs*), 81

T

Table (*class in fitbenchmarking.results_processing.base_table*), 71
table_title (*fitbenchmarking.results_processing.base_table.Table attribute*), 73
to_html() (*fitbenchmarking.results_processing.base_table.Table method*), 73
to_txt() (*fitbenchmarking.results_processing.base_table.Table method*), 73

U

UnknownMinimizerError, 82
UnknownTableError, 82
UnsupportedMinimizerError, 82

V

VALID (*fitbenchmarking.utils.options.Options attribute*), 84
VALID_FITTING (*fitbenchmarking.utils.options.Options attribute*), 84
VALID_FLAGS (*fitbenchmarking.controllers.base_controller.Controller attribute*), 51
VALID_JACOBIAN (*fitbenchmarking.utils.options.Options attribute*), 84
VALID_LOGGING (*fitbenchmarking.utils.options.Options attribute*), 84
VALID_MINIMIZERS (*fitbenchmarking.utils.options.Options attribute*), 85

VALID_PLOTTING (*fitbenchmarking.utils.options.Options attribute*), 85
VALID_SECTIONS (*fitbenchmarking.utils.options.Options attribute*), 85
validate_algorithm_type() (*fitbenchmarking.cost_func.base_cost_func.CostFunc method*), 61
validate_minimizer() (*fitbenchmarking.controllers.base_controller.Controller method*), 52
value_ranges (*fitbenchmarking.parsing.fitting_problem.FittingProblem attribute*), 68
verify() (*fitbenchmarking.parsing.fitting_problem.FittingProblem method*), 68

W

WeightedNLLSCostFunc (*class in fitbenchmarking.cost_func.weighted_nlls_cost_func*), 63
write() (*fitbenchmarking.utils.options.Options method*), 85
write_to_stream() (*fitbenchmarking.utils.options.Options method*), 85