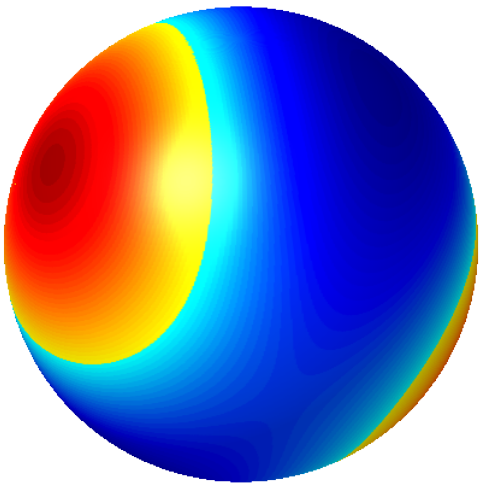

mtfit

A Bayesian Method for Source Inversion

Release 1.0.0

David Pugh

Nov 19, 2017

mtfit: A Bayesian Method for Source Inversion

Release 1.0.0

David Pugh

Nov 19, 2017

Abstract

`mtfit` is a Bayesian forward model inversion code for moment tensor and double-couple source inversion using different data types, based on the Bayesian approach presented in *Pugh et al, 2016a* and *Pugh, 2015*. The code has been developed as part of a PhD project (*Pugh, 2015*). The solutions are estimated using polarity and amplitude ratio data, although the code is extensible (see *Extending mtfit*) so it is possible to include other data-types in this framework. `mtfit` can incorporate uncertainty estimates both in the data (noise etc.) and the model (and location) in the resultant posterior probability density function. There are three sampling approaches that have been developed, with different advantages (Pugh et al, 2015t), and it is also possible to use the approach for relative amplitude inversion as well (*Pugh et al, 2015t*).

`mtfit` also works with the automated Bayesian polarity approach described in *Pugh et al, 2016b* as an alternative method of estimating polarity probabilities. This may be available on request as the `autopol` Python module.

Table of Contents

1 Introduction **1**

2 Installing mtfit **3**
2.1 Requirements . . . 3
2.2 Running the Test Suite . . . 5

3 Running mtfit **7**
3.1 Command Line . . . 7
3.2 Python Interpreter . . . 8
3.3 Input Data . . . 8
3.4 Output . . . 9
3.5 Running in parallel . . . 11

4 Tutorial: Using mtfit **13**
4.1 Creating a Data Dictionary . . . 13
4.2 P Polarity Inversion . . . 15
4.3 P/SH Amplitude Ratio Inversion . . . 17
4.4 Double-Couple Inversion . . . 18
4.5 Time Limited Inversion . . . 19
4.6 Parallel MPI Inversion . . . 20
4.7 Submitting to a Cluster . . . 20
4.8 Inversion from a CSV File . . . 21
4.9 Location Uncertainty . . . 22
4.10 Running from the Command Line . . . 25
4.11 Scatangle file binning . . . 25

5 Tutorial: Using mtfit with Real Data **27**
5.1 Synthetic Event . . . 27
5.2 Krafla Event . . . 33
5.3 Bayesian Evidence . . . 39
5.4 Joint Inversion . . . 41

6 Bayesian Approach **45**
6.1 Bayes Theory . . . 45
6.2 Uncertainties . . . 45
6.3 Posterior PDF . . . 46

7 Probability **47**
7.1 Polarity PDF . . . 47
7.2 Polarity Probability PDF . . . 48
7.3 Amplitude Ratio PDF . . . 48

8 mtfit.algorithms: Search Algorithms **49**
8.1 Algorithms . . . 49
8.2 Random Monte Carlo sampling . . . 49
8.3 Markov chain Monte Carlo sampling . . . 50
8.4 Relative Amplitude . . . 52
8.5 Running Time . . . 53
8.6 Summary . . . 54

9 convert **55**
9.1 moment_tensor_conversion.py . . . 55

10 mtfit command line options **63**
10.1 Positional Arguments: . . . 63
10.2 Optional Arguments: . . . 64
10.3 Scatangle: . . . 71
10.4 Cluster: . . . 71

11 Plotting the Moment Tensor **73**
11.1 Using MTplot from the command line . . . 73
11.2 Using MTplot from the Python interpreter . . . 74
11.3 Input Data . . . 74
11.4 MTplot Class . . . 75
11.5 MTData Class . . . 75
11.6 Beachball plot . . . 77
11.7 Fault Plane plot . . . 78
11.8 Riedesel-Jordan plot . . . 79
11.9 Radiation plot . . . 80
11.10 Hudson plot . . . 81
11.11 Lune plot . . . 81
11.12 Examples . . . 82

12 MTplot command line options **85**
12.1 Positional Arguments: . . . 85
12.2 Optional Arguments: . . . 85

13 mtfit.inversion: Inversion object **89**

14 Extending mtfit **97**
14.1 mtfit.cmd_opts . . . 98
14.2 mtfit.cmd_defaults . . . 99
14.3 mtfit.tests . . . 99
14.4 mtfit.pre_inversion . . . 100
14.5 mtfit.post_inversion . . . 100
14.6 mtfit.extensions . . . 101
14.7 mtfit.parsers . . . 101
14.8 mtfit.location_pdf_parsers . . . 102
14.9 mtfit.output_data_formats . . . 103
14.10 mtfit.output_formats . . . 103
14.11 mtfit.process_data_types . . . 104
14.12 mtfit.data_types . . . 104
14.13 mtfit.parallel_algorithms . . . 105
14.14 mtfit.directed_algorithms . . . 106
14.15 mtfit.sampling . . . 107
14.16 mtfit.sampling_prior . . . 107
14.17 mtfit.sample_distribution . . . 108
14.18 mtfit.plot . . . 108
14.19 mtfit.plot_read . . . 109
14.20 mtfit.documentation . . . 109
14.21 mtfit.source_code . . . 109

15 References 111

CHAPTER 1

Introduction

Source inversion is carried out at many seismological observatories and research groups around the world. `mtfit` builds on the approach introduced in *Pugh, 2015*, using a Bayesian approach to source inversion for polarity and amplitude ratios, as well as automated Bayesian polarity probability estimates. This approach differs from commonly used existing approaches, such as FPFIT (*Reasenberg and Oppenheimer 1985*), HASH (*Hardebeck and Shearer 2002*, *2003*) and FOCMEC (*Snoke 2003*), because it uses polarities and amplitude ratios in a Bayesian framework to estimate the full source PDF for the double-couple and full moment tensor model spaces. The approach can include location and velocity model uncertainty, as well as marginalising over measurement uncertainties in the data

Pugh et al. (2015) introduced a Bayesian approach to source inversion for polarity and amplitude ratios, as well as automated Bayesian polarity probability estimates (*Pugh et al., 2015*). These approaches have been developed into mtfit, a Python module for source inversion. Python is a common programming and scripting language with many scientific modules available, both for mathematical calculation such as NumPy[1] and SciPy[2], and for seismological methods such as ObsPy[3] (*Beyreuther et al. 2010*).

Python and many of its modules are open source, allowing easy development and removing licensing restrictions. Moreover, Python is platform independent, intuitive and accessible, with a good shell interface in the form of iPython[4]. It is used in many fields and is easy to install on almost any computer platform. Python can also interface easily with C and fortran libraries, as well as calling functions from compiled C modules, such as those generated with Cython[5], with no difference from normal Python functions. `mtfit` has been written in Python 2.7, and may not be compatible with Python 3.

mtfit has been used in several studies, including those reported by *Wilks et al. 2015*, *Greenfield and White 2015*, *Mildon et al. 2015*, and *Pugh et al. 2015*.

The polarity probability estimates can be calculated using the approach described in:ref:*Pugh, 2016a <Pugh-2016a>*.

This manual describes how to install and run `mtfit` in chapters 2 and 3. Chapters 4 and 5 provide an introduction to using `mtfit` with different algorithms and parameter choices. All the data and scripts to run these examples can be found in the `mtfit/examples` directory. The basic theory underlying `mtfit` is introduced in chapters 6 and 7, and *Pugh et al. (2016a)* provides a full derivation and explanation of the theory. The different search algorithms are described in chapter 8. These are investigated in more detail in *Pugh et al. (2015t)*. The moment tensor conversion submodule, `MTconvert` is introduced in chapter 9. Chapter 10 presents the different command line options

[1] http://www.numpy.org
[2] http://www.scipy.org
[3] http://www.obspy.org
[4] https://ipython.org/
[5] http://www.cython.org

for mtfit. Chapter 11 describes the different plotting options uisng the mtfit.plot submodule. Chapter 12 presents the different command line options for MTplot script. The *mtfit.inversion.Inversion* class, used for carrying out the forward model is documented in chapter 13. Lastly the different ways mtfit can be extended are shown in chapter 14.

CHAPTER 2

Installing mtfit

mtfit is available in several formats, including as a tar.gz file, a zip file, a Windows msi installer. Additionally the git repository can be cloned.

Apart from the Windows installer, all the other formats require installing from the source (after unpacking the compressed files e.g. tar.gz).

mtfit can be installed from the source by calling:

```
$ python setup.py install
```

To see additional command line options for the setup.py file use:

```
$ python setup.py --help
```

or:

```
$ python setup.py --help-commands
```

2.1 Requirements

mtfit requires three modules

- NumPy[6]
- SciPy[7]
- pyqsub[8] - a simple module to provide interfacing with qsub, and will be automatically installed.
- matplotlib[9]

2.1.1 Optional requirements

Additionally there are several optional requirements which allow additional features in mtfit.

[6] http://www.numpy.org
[7] http://www.scipy.org
[8] https://www.github.com/djpugh/pyqsub
[9] http://matplotlib.org/

HDF5 (Matlab -v7.3)

To use the HDF5 MATLAB format (format -v7.3) or disk-based storage of the in progress sampling (slower but saves on memory requirements) requires:

- h5py[10]
- hdf5storage[11]

If installing from source these modules require:

- HDF5[12] 1.8.4 or newer, shared library version with development headers (libhdf5-dev or similar)
- Python 2.6 - 3.3 with development headers (python-dev or similar)
- NumPy 1.6 or newer
- Optionally: Cython[13], if you want to access features introduced after HDF5 1.8.4, or Parallel HDF5.

Warning: HDF5 support is required if the output files are large (>2GB) and MATLAB output is used, beacause MATLAB cannot read older format files bigger than this.

MPI

To run on multiple nodes on a cluster requires `mpi4py` installed and a distribution of MPI[14] such as OpenMPI[15] to run in parallel on multiple nodes (single node multi-processor uses `multiprocessing`[16])

- mpi4py[17]

NonLinLoc Location Uncertainty

NonLinLoc[18] scatter files can be used for the location PDF. This requires:

- `Scat2Angle` from pyNLLoc[19].

Building the Documentation

To build this documentation from source requires:

- sphinx[20] 1.3.1 or newer

It can be built in the source directory:

```
$ python setup.py build-docs
```

and after installation:

```
>>mtfit.build_docs()
```

[10] http://www.h5py.org/
[11] http://pythonhosted.org/hdf5storage/
[12] http://www.hdfgroup.org/HDF5/
[13] http://cython.org/
[14] http://www.mcs.anl.gov/research/projects/mpi/
[15] http://www.open-mpi.org/
[16] https://docs.python.org/2/library/multiprocessing.html#module-multiprocessing
[17] http://mpi4py.scipy.org/
[18] http://alomax.free.fr/nlloc
[19] https://www.github.com/djpugh/pyNLLoc
[20] http://sphinx-doc.org

2.2 Running the Test Suite

`mtfit` comes with a complete test suite which can be run in the source directory:

```
$ python setup.py build
$ python setup.py test
```

and after installation from the python interpreter:

```
>>> import mtfit
>>> mtfit.run_tests()
```

CHAPTER 3

Running mtfit

There are several ways to run `mtfit`, and these are described here.

3.1 Command Line

`mtfit` can be run from the command line. A script should have been installed onto the path during installation and should be callable as:

```
$ mtfit
```

However it may be necessary to install the script manually. This is platform dependent.

3.1.1 Script Installation

Linux

Add this python script to a directory in the $PATH environmental variable:

```
#!/usr/bin/env python
import mtfit
mtfit.__run__()
```

And make sure it is executable.

Windows

Add the linux script (above) to the path or if using powershell edit the powershell profile (usually found in *Documents/WindowsPowerShell/* - if not present use `$PROFILE|Format-List -Force` to locate it, it may be necessary to create the profile) and add:

```
function mtfit{
    $script={
        python -c "import mtfit;mtfit.__run__()" $args
        }
```

```
    Invoke-Command -ScriptBlock $script -ArgumentList $args
    }
```

Windows Powershell does seem to have some errors with commandline arguments, if necessary these should be enclosed in quotation marks e.g. "-d=datafile.inv"

3.1.2 Command Line Options

When running `mtfit` from the command line, there are many options available, and these can be listed using:

```
$ mtfit -h
```

For a description of these options see Chapter 10.

The command line defaults can be set using a defaults file. This is recursively checked in 3 locations:

1. `MTFITDEFAULTSPATH` environmental variable (could be a system level setting)
2. `.mtfitdefaults` file in the users home directory
3. `.mtfitdefaults` file in the current working directory

The higher number file over-writes defaults in the lower files if they conflict.

The structure of the defaults file is simply:

```
key:attr
```

e.g.:

```
dc:True
algorithm:iterate
```

3.2 Python Interpreter

Running mtfit from the python interpreter is done as:

```
>>> import mtfit
>>> args=['-o','-d']
>>> mtfit.__run__(args)
```

Where args correspond to the command line arguments (see Chapter 10.

It is also possible to create the *Inversion* object:

```
>>> import mtfit
>>> inversion=mtfit.Inversion(*args,**kwargs)
>>> inversion.forward()
```

The descriptions of the *Inversion* initialisation arguments can be found in the `__init__` docstrings, and 13.

3.3 Input Data

There are several different input data types, and it is also possible to add additional parsers using the `mtfit.parsers` entry point.

The required data structure for running mtfit is very simple, the inversion expects a python dictionary of the data in the format:

```
>>> data={'PPolarity':{'Measured':numpy.matrix([[-1],[-1]...]),
                       'Error':numpy.matrix([[0.01],[0.02],...]),
                       'Stations':{'Name':['Station1','Station2',...],
                                   'Azimuth':numpy.matrix([[248.0],[122.3]...]),
                                   'TakeOffAngle':numpy.matrix([[24.5],[22.8]...]),
                                  }
                       },
           'PSHAmplitudeRatio':{...},
           ...
           'UID':'Event1'
           }
```

For more information on the data keywords and how to set them up, see *Inversion* docstrings.

The data dictionary can be passed directly to the *Inversion* object (simple if running within python), or from a binary pickled object, these can be made by simply using pickle (or cPickle):

```
>>> pickle.dump(data,open(filename,'wb'))
```

The coordinate system is that the Azimuth is angle from x towards y and TakeOffAngle is the angle from positive z.

For data in different formats it is necessary to write a parser to convert the data into this dictionary format.

There is a parser for csv files with format

3.3.1 CSV

There is a CSV format parser which reads CSV files. The CSV file format is to have events split by blank lines, a header line showing where the information is, UID and data-type information stored in the first column, e.g.:

```
UID=123,,,,
PPolarity,,,,
Name,Azimuth,TakeOffAngle,Measured,Error
S001,120,70,1,0.01
S002,160,60,-1,0.02
P/SHRMSAmplitudeRatio,,,,
Name,Azimuth,TakeOffAngle,Measured,Error
S003,110,10,1,0.05 0.04
,,,,
PPolarity ,,,,
Name,Azimuth,TakeOffAngle,Measured,Error
S003,110,10,1,0.05
```

This is a CSV file with 2 events, one event ID of 123, and PPolarity data at station S001 and station S002 and P/SHRMSAmplitude data at station S003, and a second event with no ID (will default to the event number, in this case 2) with PPolarity data at station S003.

3.3.2 hyp

There is a hyp format parser which reads hyp files as defined by NonLinLoc[21], this allows output files from NonLinLoc to be directly read.

3.4 Output

The default output is to output a MATLAB file containing 2 structures and a cell array, although there are two other possible formats, and others can be added (see mtfit.extensions). The `Events` structure has the following

[21] http://alomax.free.fr/nlloc/soft6.00/formats.html#_location_hypphs_

fieldnames: `MTspace` and `Probability`.

- `MTspace` - The moment tensor samples as a 6 by n vector of the form:

```
Mxx
Myy
Mzz
sqrt(2)*Mxy
sqrt(2)*Mxz
sqrt(2)*Myz
```

- `Probability` - The corresponding probability values

The `Other` structure contains information about the inversion

The `Stations` cell array contains the station information, including, if available, the polarity:

Name	Azimuth(angle from x)	TakeOffAngle(angle from z)	P Polarity (if available)

A log file for each event is also produced to help with debugging and understanding the results.

3.4.1 Pickle format

It is also possible to output the data structure as a pickled file using the pickle output options, storing the output dictionary as a pickled file.

3.4.2 hyp format

The results can be outputted in the NonLinLoc hyp format[22], with the range of solutions sampled outputted as a binary file with the following format:

```
binary file version (unsigned long integer)
total_number_samples(unsigned long integer)
number_of_saved_samples(unsigned long integer)
converted (bool flag)
Ln_bayesian_evidence (double)
Kullback-Liebeler Divergence from sampling prior (double)
```

Then for each moment tensor sample (up to `number_of_saved_samples`):

```
Probability (double)
Ln_probability(double)
Mnn (double)
Mee (double)
Mdd (double)
Mne (double)
Mnd (double)
Med (double)
```

if Converted is true then each sample also contains:

```
gamma (double)
delta (double)
kappa (double)
h (double)
sigma (double)
u (double)
v (double)
strike1 (double)
```

[22] http://alomax.free.fr/nlloc/soft6.00/formats.html#_location_hypphs_

```
dip1 (double)
rake1 (double)
strike2 (double)
dip2 (double)
rake2 (double)
```

If there are multiple events saved, then the next event starts immediately after the last with the same format. The output binary file can be re-read into python using `mtfit.inversion.read_binary_output()`.

3.5 Running in parallel

The code is written to run in parallel using multiprocessing, it will initialise as many threads as the system reports available. A single thread mode can be forced using:

- -l, –singlethread, –single, –single_thread flag on the command line
- parallel=False keyword in the mtfit.inversion.Inversion object initialisation

It is also possible to run this code on a cluster using qsub [requires pyqsub]. This can be called from the commandline using a flag:

- -q, –qsub, –pbs

This runs using a set of default parameters, however it is also possible to adjust these parameters using commandline flags (use -h flag for help and usage).

There is a bug when using mpi and very large result sizes, giving a size error (negative integer) in `mpi4py`. If this occurs, lower the sample size and it will be ok.

Warning: If running this on a server, be aware that not setting the number of workers option `--numberworkers`, when running in parallel, means that as many processes as processors will be spawned, slowing down the machine for any other users.

CHAPTER 4

Tutorial: Using mtfit

`mtfit` is a bayesian approach to moment tensor inversion, allowing rigorous inclusion of uncertainties. This section shows a simple series of examples for running `mtfit`.

The example data is included in `examples/example_data.py` and are purely data that are used as an example, rather than a necessarily good solution.

These examples show some of the common usage of `mtfit`. However, the reasons behind the choice of approach have not always been well explained. The next chapter (Chapter 5) includes real and synthetic data used in the *Pugh et al. 2016a* paper as an example of the results that can be obtained using `mtfit`, along with some explanation of the parameter choices made.

The tutorials described in this chapter are:

1. *Creating a Data Dictionary*
2. *P Polarity Inversion*
3. *P/SH Amplitude Ratio Inversion*
4. *Double-Couple Inversion*
5. *Time Limited Inversion*
6. *Parallel MPI Inversion*
7. *Submitting to a Cluster*
8. *Inversion from a CSV File*
9. *Location Uncertainty*
10. *Running from the Command Line*
11. *Scatangle file binning*

4.1 Creating a Data Dictionary

The input data dictionary (see *Input Data*) can either be pickled or not pickled. The structure is simple:

```
>>> import numpy as np
>>> data = {'PPolarity': {'Measured': np.matrix([[-1], [-1], [1], [1]]),
                          'Error': np.matrix([[0.01], [0.02], [0.4], [0.1]]),
```

```
                          'Stations':{'Name': ['Station1', 'Station2', 'Station3',
                                              'Station4'],
                                      'Azimuth': np.matrix([[248.0], [122.3],
                                                            [182.3], [35.2]]),
                                      'TakeOffAngle': np.matrix([[24.5], [22.8],
                                                                 [74.5], [54.3]])}}
↪,
          'UID': 'Event1'}
```

This has created a data dictionary for `Event1` with P Polarity observations at 4 stations:

```
>>> print data
{'PPolarity': {'Stations': {'TakeOffAngle': matrix([[ 24.5],
        [ 22.8],
        [ 74.5],
        [ 54.3]]),
    'Name': ['Station1', 'Station2', 'Station3', 'Station4'],
    'Azimuth': matrix([[ 248. ],
        [ 122.3],
        [ 182.3],
        [  35.2]])},
    'Measured': matrix([[-1],
        [-1],
        [ 1],
        [ 1]]),
    'Error': matrix([[ 0.01],
        [ 0.02],
        [ 0.4 ],
        [ 0.1 ]])},
'UID': 'Event1'}
```

If there were more observations such as P/SH Amplitude Ratios, the data dictionary above would need to be updated:

```
>>> data['P/SHAmplitudeRatio'] = {'Measured': np.matrix([[1242, 1113], [742, 2341],
                                                         [421, 112], [120, 87]]),
                      'Error': np.matrix([[102, 743], [66, 45], [342, 98], [14,␣
↪11]]),
                      'Stations': {'Name': ['Station5', 'Station6',
                                            'Station7', 'Station8'],
                                   'Azimuth': np.matrix([[163.0], [345.3],
                                                         [25.3], [99.2]]),
                                   'TakeOffAngle': np.matrix([[51.5], [76.8],
                                                              [22.5], [11.3]]),
                                  }
                      }
```

This has added P/SH Amplitude Ratio observations for 4 more stations to the data dictionary:

```
>>> print data
{'PPolarity': {'Stations': {'TakeOffAngle': matrix([[ 24.5],
        [ 22.8],
        [ 74.5],
        [ 54.3]]),
    'Name': ['Station1', 'Station2', 'Station3', 'Station4'],
    'Azimuth': matrix([[ 248. ],
        [ 122.3],
        [ 182.3],
        [  35.2]])},
    'Measured': matrix([[-1],
        [-1],
        [ 1],
```

```
        [ 1]]),
    'Error': matrix([[ 0.01],
        [ 0.02],
        [ 0.4 ],
        [ 0.1 ]])},
'P/SHAmplitudeRatio': {'Stations': {'TakeOffAngle': matrix([[ 51.5],
        [ 76.8],
        [ 22.5],
        [ 11.3]]),
    'Name': ['Station5', 'Station6', 'Station7', 'Station8'],
    'Azimuth': matrix([[ 163. ],
        [ 345.3],
        [  25.3],
        [  99.2]])},
    'Measured': matrix([[1242, 1113],
        [ 742, 2341],
        [ 421,  112],
        [ 120,   87]]),
    'Error': matrix([[102, 743],
        [ 66,  45],
        [342,  98],
        [ 14,  11]])},
'UID': 'Event1'}
```

The amplitude ratio `Measured` and `Error` numpy matrices have the observations of the ratio numerator and denominator at each station, i.e. in this case, `Station5` has P Amplitude is `1242` and SH Amplitude is `1113`, along with P error `102` and SH error `743`. The split into numerator and denominator is required because the appropriate PDF is the ratio PDF (see *Amplitude Ratio PDF*).

This dictionary can either be provided as a construction argument for the *`Inversion`* object:

```
>>> import mtfit
>>> inversion_object = mtfit.Inversion(data)
>>> inversion_object.forward()
```

Or read in from the command line:

```
>>> import cPickle
>>> cPickle.dump(data, open('Event1.inv', 'wb'))
```

This has created a pickled dictionary called `Event1.inv` in the current directory. To perform the inversion, open a shell in the same directory:

```
$ mtfit -d Event1.inv
```

This will create an output file `Event1MT.mat` which contains the MATLAB output data (see *Output*).

The creation of the dictionary can easily be automated from different data types by writing a simple parser for the format.

4.2 P Polarity Inversion

Using the above tutorial, it is simple to carry out a P polarity inversion, `examples/p_polarity.py` shows the example script and data and can be run in the examples directory.

The script can be run from the command line as:

```
$ python p_polarity.py
```

The parameters used are:

- algorithm = 'iterate' - uses an iterative random sampling approach (see *Random Monte Carlo sampling*).
- parallel = True - tries to run in parallel using `multiprocessing`[23].
- phy_mem = 0.5 - uses a soft limit of 500Mb of RAM for estimating the sample sizes (This is only a soft limit, so no errors are thrown if the memory usage increases above this).
- dc = False - runs the full moment tensor inversion.
- max_samples = 1000000 - runs the inversion for 1,000,000 samples.

The *Inversion* object is created and then the forward model run with the results automatically outputted:

```
# Create the inversion object with the set parameters..
inversion_object = Inversion(data, algorithm=algorithm, parallel=parallel,
                             phy_mem=phy_mem, dc=dc, max_samples=max_
↪samples,
                             convert=True)
# Run the forward model based inversion
inversion_object.forward()
```

The output file is `P_Polarity_Example_OutputMT.mat`.

The source PDF can be plotted (Fig. *4.1*)

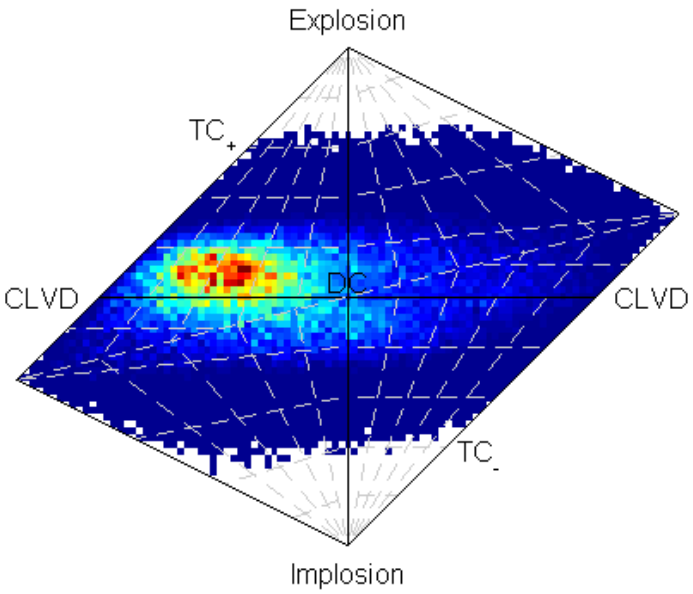

Fig. 4.1: *Hudson plot of the example results from* `examples/p_polarity.py` *(Plotted using MTplot MATLAB code)*

Increasing the number of samples can improve the fit at the expense of time taken to run the inversion. Re-running the inversion with more samples (10,000,000) takes longer, but produces a better density of sampling (output file is `P_Polarity_Example_Dense_OutputMT.mat`).

The source PDF can be plotted (Fig. *4.2*)

[23] https://docs.python.org/2/library/multiprocessing.html#module-multiprocessing

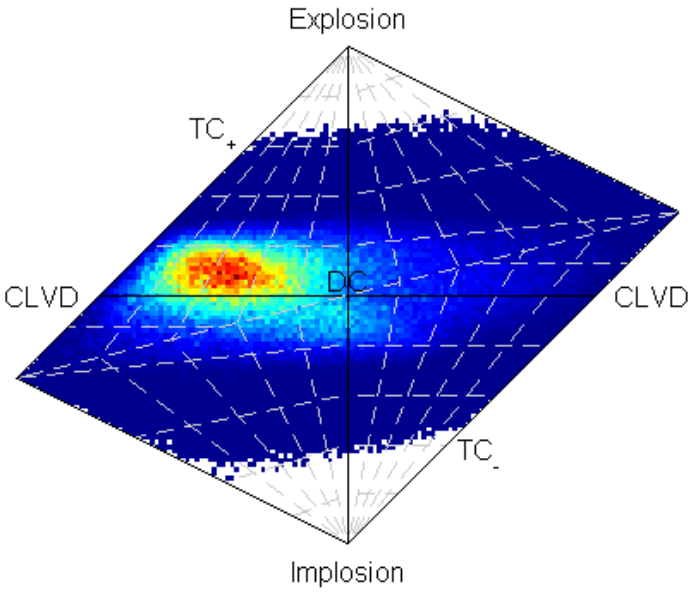

Fig. 4.2: *Hudson plot of the example results from* `examples/p_polarity.py` *(Plotted using MTplot MATLAB code)*

4.3 P/SH Amplitude Ratio Inversion

Example script for running P/SH amplitude ratio inversion is `examples/p_sh_amplitude_ratio.py` To run the script:

```
$ python p_sh_amplitude_ratio.py
```

The parameters used are:

- algorithm = 'iterate' - uses an iterative random sampling approach (see *Random Monte Carlo sampling*).
- parallel = True - tries to run in parallel using `multiprocessing`[24].
- phy_mem = 1 - uses a soft limit of 1Gb of RAM for estimating the sample sizes (This is only a soft limit, so no errors are thrown if the memory usage increases above this).
- dc = False - runs the full moment tensor inversion.
- max_samples = 1000000 - runs the inversion for 1,000,000 samples.

The *`Inversion`* object is created and then the forward model run with the results automatically outputted:

```
# Create the inversion object with the set parameters..
inversion_object = Inversion(data, algorithm=algorithm, parallel=parallel,
                             phy_mem=phy_mem, dc=dc, max_samples=max_
↪samples,
                             convert=True)

# Run the forward model based inversion
inversion_object.forward()
```

The output file is `P_SH_Amplitude_Ratio_Example_OutputMT.mat`.

[24] https://docs.python.org/2/library/multiprocessing.html#module-multiprocessing

It is also possible to run the inversion for as many samples as possible in a given time (output file is `P_Polarity_Example_Time_OutputMT.mat`) by setting the parameters:

- algorithm = 'time' - uses an iterative random sampling approach (see *Random Monte Carlo sampling*) until a specified time has elapsed.
- max_time = 300 - runs the inversion for 300 seconds.

The *Inversion* object is created and then the forward model run with the results automatically outputted:

```
ns the inversion for a given time period.
rithm = 'time'
est:
# Run inversion for test
max_time = 10
:
# Length of time to run for in seconds.
max_time = 300
eate the inversion object with the set parameters..
rsion_object = Inversion(data, algorithm=algorithm, parallel=parallel,
                         phy_mem=phy_mem, dc=dc, max_time=max_time,␣
↪convert=True)
n the forward model based inversion
rsion_object.forward()
```

4.4 Double-Couple Inversion

Sometimes it may be better to constrain the solution to only the double-couple space, this is easy to do from the command line using the `-c` flag (see *mtfit command line options*):

```
$ mtfit -c ...
```

An example script for running a mixed inversion constrained to double-couple space is `examples/double_couple.py`. To run the script:

```
$ python double_couple.py
```

The inversion is run from a data file, which is the pickled (`pickle`[25]/`cPickle`[26]) data dictionary:

```
import cPickle
cPickle.dump(data, open('Double_Couple_Example.inv', 'wb'))
```

The inversion parameters used are:

- algorithm = 'iterate' - uses an iterative random sampling approach (see *Random Monte Carlo sampling*)
- parallel = True - tries to run in parallel using `multiprocessing`[27]
- phy_mem = 1 - uses a soft limit of 1Gb of RAM for estimating the sample sizes (This is only a soft limit, so no errors are thrown if the memory usage increases above this)
- dc = True - runs the inversion in the double-couple space.
- max_samples = 100000 - runs the inversion for 100,000 samples.

Since the double-couple space has fewer dimensions than the moment tensor space, fewer samples are required for good coverage of the space, so only 100,000 samples are used.

The *Inversion* object is created and then the forward model run with the results automatically outputted:

[25] https://docs.python.org/2/library/pickle.html#module-pickle
[26] https://docs.python.org/2/library/pickle.html#module-cPickle
[27] https://docs.python.org/2/library/multiprocessing.html#module-multiprocessing

```
# Create the inversion object with the set parameters.
inversion_object = Inversion(data_file='Double_Couple_Example.inv',
                             algorithm=algorithm, parallel=parallel,
                             phy_mem=phy_mem, dc=dc,
                             max_samples=max_samples, convert=True)
# Run the forward model based inversion
inversion_object.forward()
```

4.5 Time Limited Inversion

A different algorithm for the inverson can be set using the algorithm option. In this case the time constrained algorithm is used (for other options see *mtfit.algorithms: Search Algorithms*). An example script for running a time constrained inversion is `examples/time_inversion.py`. To run the script:

```
$ python time_inversion.py
```

The time option for the inversion algorithm sets a maximum time (in seconds) to run the inversion for rather than a maximum number of samples. To select the algorithm from the command line use:

```
$mtfit --algorithm=time ...
```

For the other options see *Command Line Options*. The inversion parameters used in `examples/time_inversion.py` are:

- algorithm = 'time' - uses an time limited random sampling approach (see *Random Monte Carlo sampling*)
- parallel = False - runs in a single thread.
- phy_mem = 1 - uses a soft limit of 1Gb of RAM for estimating the sample sizes (This is only a soft limit, so no errors are thrown if the memory usage increases above this)
- dc = False - runs the inversion in the double-couple space.
- max_time = 120 - runs the inversion for 120 seconds.
- inversion_options = 'PPolarity,P/SHAmplitudeRatio' - Just uses PPolarity and P/SH Amplitude Ratios rather than all the data in the dictionary

In this case the `inversion_options` keyword argument is used to set the data types used in the inversion. If this is not set the inversion will use all of the available data types in the dictionary that match possible data types (see *Inversion* documentation), this is because the example data has other data types that are not desired or not independent:

```
>>> data.keys()=['PPolarity','P/SHRMSAmplitudeRatio','P/SVRMSAmplitudeRatio','P/
↪SHAmplitudeRatio','UID]
```

The `P/SHRMSAmplitudeRatio` and the `P/SHAmplitudeRatio` are not independent, and so cannot both be used in this inversion.

The *Inversion* object is created and then the forward model run with the results automatically outputted:

```
# Create the inversion object with the set parameters.
inversion_object = Inversion(data, algorithm=algorithm, parallel=parallel,
                             phy_mem=phy_mem, dc=dc, inversion_
↪options=inversion_options,
                             max_time=max_time, convert=True)

# Run the forward model based inversion
inversion_object.forward()
```

The output file is `Time_Inversion_Example_OutputMT.mat`.

It is also possible to run the inversion for the double-couple constrained inversion (output file is `Time_Inversion_Example_OutputDC.mat`):

```
dc = True
# Create the inversion object with the set parameters.
inversion_object = Inversion(data, algorithm=algorithm, parallel=parallel,
                             phy_mem=phy_mem, dc=dc, inversion_
↪options=inversion_options,
                             max_time=max_time, convert=True)
# Run the forward model based inversion
inversion_object.forward()
```

4.6 Parallel MPI Inversion

Running the inversion using MPI on a multi-node environment (such as a cluster) is done from the command line using:

```
$ mtfit -M ...
```

Warning: Do not use the `--mpi-call` flag as this is a flag set automatically by the code

The script `examples/mpi.py` is an example script for running using MPI (It will test if mpi4py[28] is installed)

The data file is pickled using `cPickle`[29]:

```
# pickle data using cPickle
try:
    import cPickle as pickle
except ImportError:
    import pickle
# data saved to MPI_Example.inv using cPickle
with open('MPI_Example.inv', 'wb') as f:
    pickle.dump(f, data)
```

And then `subprocess`[30] is used to call the inversion:

```
# Use subprocess to call mtfit
import subprocess
subprocess.call(['mtfit', '-M', '--data_file=MPI_Example.inv',
                 '--algorithm=iterate', '--max_samples=100000'])
```

This is equivalent to (see *command line options* for more information on the command line options):

```
$ mtfit -M --data_file=MPI_Example.inv --algorithm=iterate --max_samples=100000
```

The output file is `MPI_Inversion_Example_OutputMT.mat`.

The main advantage of running using MPI is to allow for more samples to be tried in a given time by using more processors.

4.7 Submitting to a Cluster

Submitting an `mtfit` job to a cluster using `qsub` uses a simple module called `pyqsub` (from https://www.github.com/djpugh/pyqsub) which provides command line options for running `qsub`.

[28] http://mpi4py.scipy.org/
[29] https://docs.python.org/2/library/pickle.html#module-cPickle
[30] https://docs.python.org/2/library/subprocess.html#module-subprocess

To submit to the cluster from command line, on a computer with qsub available use:

```
$ mtfit -q ...
```

There are other available options when submitting to the cluster:

```
$ mtfit -q  --walltime=48:00:00 --nodes=4 --ppn=4 --pmem=2 --emailoptions=ae
    --email=example@example.com --name=mtfitClusterTest --queue=auto ...
```

This submits an mtfit job to the cluster using `qsub` (`-q`) with a walltime of 48 hours (`--walltime`) using 4 nodes (`--nodes`) and 4 processors per node (`--ppn`) with a maximum amount of physical memory per process of 2Gb (`--pmem`). The job will send emails on abort and end (`--emailoptions`) to email `example@example.com` (`--email`). It has a job name of `mtfitClusterTest` (`--name`) and is submitted to the auto queue (`--queue`).

These options, combined with the other *command line options*, will be saved to a job script named `JobName.pJobID`. For the above case, if the JobID was `207642` a `PBS` script is saved called `mtfitClusterTest.p207642`

4.8 Inversion from a CSV File

`mtfit` can use a CSV file as input. An example CSV file can be made by running `examples/make_csv_file.py` in the examples folder:

```
$ python make_csv_file.py
```

This makes a CSV file (called csv_example_file.csv):

```
UID=Event1,,,,
PPolarity,,,,
Error,Name,TakeOffAngle,Measured,Azimuth
0.1,S0006,112.8,1,210.6
0.3,S0573,110.0,-1,306.7
0.1,S0563,131.4,-1,23.1
0.1,S0016,117.6,1,167.8
0.1,S0567,123.7,-1,41.3
0.1,S0654,110.0,-1,323.4
0.1,S0634,119.7,-1,342.5
0.1,S0533,138.3,-1,354.1
0.1,S0249,155.2,1,153.5
0.1,S0571,113.7,-1,54.5
0.1,S0065,125.6,1,184.2
0.1,S0095,127.4,1,159.2
0.1,S0537,134.9,-1,25.6
0.1,S0372,145.9,1,288.2
0.1,S0097,124.5,1,150.0
P/SHAmplitudeRatio,,,,
TakeOffAngle,Measured,Error,Name,Azimuth
112.8,1.91468406e-08   3.22758296e-08,9.58863666e-10   7.70965062e-09,S0006,210.6
110.0,4.88113677e-09   1.96675583e-08,2.45607268e-10   3.45469389e-09,S0573,306.7
131.4,1.45833761e-07   1.79089155e-09,7.28757867e-09   3.45820500e-09,S0563,23.1
117.6,9.31790661e-08   2.93385249e-08,4.65480572e-09   8.95408759e-09,S0016,167.8
123.7,1.20612039e-07   3.84818185e-08,6.02547046e-09   9.23059636e-09,S0567,41.3
110.0,2.07444768e-08   3.27506473e-08,1.03738569e-09   3.93335483e-09,S0654,323.4
119.7,7.83955802e-08   5.52997744e-08,3.91683797e-09   7.86172468e-10,S0634,342.5
138.3,1.38297893e-07   4.90243560e-08,6.91029070e-09   9.79988215e-10,S0533,354.1
155.2,1.74815653e-07   3.48061608e-08,8.75143170e-09   7.61184113e-10,S0249,153.5
113.7,8.41802958e-08   4.60234127e-08,4.20431936e-09   1.17189815e-08,S0571,54.5
125.6,1.09705743e-07   4.42081432e-08,5.48271153e-09   9.58851515e-10,S0065,184.2
127.4,1.35994091e-07   1.03528610e-08,6.79566727e-09   2.75097217e-09,S0095,159.2
134.9,1.54309735e-07   1.22170773e-08,7.71089395e-09   2.61801853e-09,S0537,25.6
```

```
145.9,6.88684554e-09   8.43199415e-08,3.43601244e-10   1.79928175e-09,S0372,288.2
124.5,1.24505851e-07   6.84587855e-09,6.22146156e-09   2.83710916e-09,S0097,150.0
,,,,
P/SVAmplitudeRatio,,,,
Name,Azimuth,Measured,Error,TakeOffAngle
S0006,210.6,3.22758296e-08   8.19892140e-08,7.70965062e-09   9.80424095e-09,112.8
S0573,306.7,1.96675583e-08   3.68506966e-08,3.45469389e-09   3.35913629e-09,110.0
S0563,23.1,1.79089155e-09   3.56992402e-08,3.45820500e-09   3.64333023e-09,131.4
S0016,167.8,2.93385249e-08   6.26397384e-08,8.95408759e-09   8.69575530e-09,117.6
S0567,41.3,3.84818185e-08   1.55744928e-08,9.23059636e-09   4.07140152e-09,123.7
S0654,323.4,3.27506473e-08   4.94388184e-08,3.93335483e-09   4.17167829e-09,110.0
S0634,342.5,5.52997744e-08   3.26269606e-08,7.86172468e-10   1.20208387e-09,119.7
S0533,354.1,4.90243560e-08   4.51596183e-08,9.79988215e-10   1.97681026e-09,138.3
S0249,153.5,3.48061608e-08   8.71989457e-08,7.61184113e-10   1.37314781e-09,155.2
S0571,54.5,4.60234127e-08   4.20042749e-09,1.17189815e-08   4.50190885e-09,113.7
S0065,184.2,4.42081432e-08   6.15020436e-08,9.58851515e-10   3.53524312e-09,125.6
S0095,159.2,1.03528610e-08   3.56854812e-08,2.75097217e-09   2.22496836e-09,127.4
S0537,25.6,1.22170773e-08   5.41945269e-08,2.61801853e-09   2.74678803e-09,134.9
S0372,288.2,8.43199415e-08   1.80916924e-08,1.79928175e-09   2.95196095e-10,145.9
S0097,150.0,6.84587855e-09   3.48806733e-08,2.83710916e-09   1.82493870e-09,124.5
,,,,
```

This is a CSV file with 2 events, one event ID of Event 1 with PPolarity and P/SHAmplitudeRatio and P/SVAmplitudeRatio data at 15 receivers, and a second event with no ID (will default to the event number, in this case 2) with PPolarity data at 15 receivers.

Running an inversion using a CSV file is the same as running a normal inversion. Calling from the command line is simply called by:

```
$ mtfit --datafile=thecsvfile.csv ...
```

The `--invext` flag sets the file ending that the inversion searches for when no datafile is specified, so to search for CSV files in the current directory:

```
$ mtfit --invext=csv
```

This will try to invert the data from all the CSV files in the current directory.

mtfit can be extended for other inversion file formats using *setuptools entry-points*

4.9 Location Uncertainty

`mtfit` can include location uncertainty in the resultant PDF. This requires samples from the location PDF. The location uncertainty is included in the inversion using a Monte Carlo method (see *Bayesian Approach*).

This file can be made from the NonLinLoc[31] `*.scat` file using `Scat2Angle` in the pyNLLoc[32] module.

The expected format for the location uncertainty file is:

```
Probability
StationName Azimuth TakeOffAngle
StationName Azimuth TakeOffAngle

Probability
.
.
.
```

[31] http://alomax.free.fr/nlloc
[32] https://github.com/djpugh/pyNLLoc

e.g.:

```
504.7
S0271   231.1   154.7
S0649   42.9    109.7
S0484   21.2    145.4
S0263   256.4   122.7
S0142   197.4   137.6
S0244   229.7   148.1
S0415   75.6    122.8
S0065   187.5   126.1
S0362   85.3    128.2
S0450   307.5   137.7
S0534   355.8   138.2
S0641   14.7    120.2
S0155   123.5   117
S0162   231.8   127.5
S0650   45.9    108.2
S0195   193.8   147.3
S0517   53.7    124.2
S0004   218.4   109.8
S0588   12.9    128.6
S0377   325.5   165.3
S0618   29.4    120.5
S0347   278.9   149.5
S0529   326.1   131.7
S0083   223.7   118.2
S0595   42.6    117.8
S0236   253.6   118.6

502.7
S0271   233.1   152.7
S0649   45.9    101.7
S0484   25.2    141.4
S0263   258.4   120.7
.
.
.
```

mtfit can be extended to use other location PDF file formats using *setuptools entry-points*

Running with the location uncertainty included will slow the inversion as this requires more memory to store each of the location samples in the inversion. The number of samples used can be changed by setting the `number_location_samples` parameter in the *`Inversion`* object:

```
>>> import mtfit
>>> mtfit.Inversion(...,number_location_samples=10000,...)
```

This limits the number of station samples to 10,000, reducing the memory requirements and improving the speed.

The script `examples/location_uncertainty.py` contains an example for the location uncertainty inversion. To run the script:

```
$ python location_uncertainty.py
```

The angle scatter file path option can be set from the command line using:

```
$ mtfit --anglescatterfilepath=./ --angleext=.scatangle ...
```

This will search in the current directory for *scatangle* files (default is to search for *scatangle* files if `--angleext` is not specified). The files are matched to the input data files if mtfit is called from the command line. A specific file or list of files can be set using:

```
$ mtfit --anglescatterfilepath=./thisanglefile.scatangle ...
```

Which uses the *thisanglefile.scatangle* file in the current directory.

The inversion parameters used in `examples/location_uncertainty.py` are:

- algorithm = 'time' - uses an time limited random sampling approach (see *Random Monte Carlo sampling*)
- parallel = True - runs in multiple threads using `multiprocessing`[33].
- phy_mem = 1 - uses a soft limit of 1Gb of RAM for estimating the sample sizes (This is only a soft limit, so no errors are thrown if the memory usage increases above this)
- dc = False - runs the inversion in the double-couple space.
- max_time = 60 - runs the inversion for 60 seconds.
- inversion_options = 'PPolarity' - Just uses PPolarity rather than all the data in the dictionary
- location_pdf_file_path = 'Location_Uncertainty.scatangle'

The *Inversion* object is created and then the forward model run with the results automatically outputted:

```
# Location uncertainty path
location_pdf_file_path = 'Location_Uncertainty.scatangle'
# Create the inversion object with the set parameters..
inversion_object = Inversion(data, location_pdf_file_path=location_pdf_
↪file_path,
                             algorithm=algorithm, parallel=parallel, phy_
↪mem=phy_mem, dc=dc,
                             inversion_options=inversion_options, max_
↪time=max_time,
                             convert=True)
# Run the forward model based inversion
inversion_object.forward()
```

The output file is `Location_Uncertainty_Example_OutputMT.mat`.

Including the location uncertainty in an inversion is slower, since fewer samples are used in a given time. Setting the number of station samples parameter to a smaller number can reduce this:

```
# Reduce the number of station samples to increase the
# number of moment tensor samples tried
number_station_samples = 10000
# Create the inversion object with the set parameters..
inversion_object = Inversion(data, location_pdf_file_path=location_pdf_
↪file_path,
                             algorithm=algorithm, parallel=parallel, phy_
↪mem=phy_mem, dc=dc,
                             inversion_options=inversion_options, max_
↪time=max_time,
                             number_station_samples=number_station_
↪samples, convert=True)
# Run the forward model based inversion
inversion_object.forward()
```

This tries more samples, however it has a worse sampling of the location PDF than before. Taking this to extremes, reducing the `number_location_samples` to `100` improves the number of samples tried but reduces the quality of the location uncertainty sampling.

The method of including location uncertainty can also be used to include **velocity model** uncertainty by drawing location samples from a range of models and combining (see `scripts/model_sampling.py`).

[33] https://docs.python.org/2/library/multiprocessing.html#module-multiprocessing

4.10 Running from the Command Line

`mtfit` is easy to run from the command line. The installation should install a script onto the path so that:

```
$ mtfit -h
```

Gives the command line options. If this does not work see *Running mtfit* to install the script.

There are many command line options available (see *mtfit command line options*) but the default settings are usually ok.

`examples/command_line.sh` (*nix) or `examples/command_line.bat` is an example script for running the inversion from the command line:

```
#!/usr/bin/sh
echo "Command line mtfit example script\n"
echo "Needs mtfit to have been installed to run.\n"
echo "Making data file - csv_example_file.csv\n"
python make_csv_file.py

echo "mtfit --version:\n"
#Output mtfit version
mtfit --version

echo "Running mtfit from command line:\n"
echo "mtfit --data_file=csv_example*.inv --algorithm=iterate \
        --max_samples=100000 -b  --inversionoptions=PPolarity"
#Run mtfit from the command line. Options are:
#   --data_file=csv_example*.inv - use the data files matching csv_
↪example*.inv
#   --algorithm=iterate - use the iterative algorithm
#   --max_samples=100000 - run for 100,000 samples
#   -b - carry out the inversion for both the double couple constrained
#           and full moment tensor spaces
#   --inversionoptions=PPolarity - carry out the inversion using
#           PPolarity data only
#   --convert - convert the solution using MTconvert.

mtfit --data_file=csv_example*.csv --algorithm=iterate --max_
↪samples=100000 \
     -b  --inversionoptions=PPolarity --convert
```

This uses the data from the CSV example file (see *Inversion from a CSV File*), prints the version of mtfit being used and then calls mtfit from the command line. The parameters used are:

- –data_file=csv_*.inv - use the data files matching csv_*.inv
- –algorithm=iterate - use the iterative algorithm
- –max_samples=100000 - run for 100,000 samples
- -b - carry out the inversion for both the double couple constrained and full moment tensor spaces
- –inversionoptions=PPolarity - carry out the inversion using PPolarity data only
- –convert - convert the solution using `mtfit.MTconvert`.

4.11 Scatangle file binning

Often the scatangle files are large with many samples at similar station angles. The size of these files can be reduced by binning these samples into similar bins. This can be done either before running mtfit or as a pre-inversion step using the command line parameters:

- –bin-scatangle=True - run the scatangle binning before the inversion
- –bin-size=1.0 - set a bin size of 1.0 degrees.

This can be run in parallel, which can speed up the process, using the same command line arguments as before.

The new files are outputted with _bin_1.0 appended if the bin-size is 1.0, and are automatically used in the inversio

CHAPTER 5

Tutorial: Using mtfit with Real Data

The previous chapter has introduced many of the different options available in `mtfit`. This chapter explains the reasoning behind the choice of these parameters using synthetic and real examples.

5.1 Synthetic Event

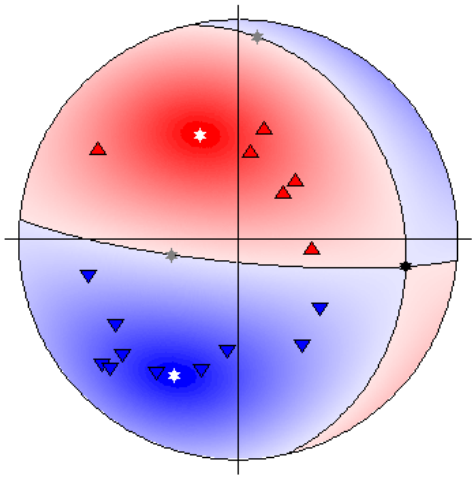

Fig. 5.1: *Beachball plot of the synthetic source and receiver positions for the location (Plotted using MTplot MATLAB code)*

The synthetic event shown here (Fig. *5.1*)has been generated from a double-couple source using a finite difference approach (*Bernth and Chapman, 2011*). Gaussian random noise was added to the data, which were manually picked for both P and S arrival times and P polarities, and then located using NonLinLoc[34] and a simplified version of the known velocity model. The arrival time picks were used to automatically window and measure P, SH, and SV amplitudes.

[34] http://alomax.free.fr/nlloc

5.1.1 Synthetic P Polarity Inversion

There are 16 P-polarity arrivals for the synthetic event, and the locations of these receivers are shown in the figure to the right. These receivers provide quite good coverage of the focal sphere, although there is not much constraint on the fault planes due to the large spacings between receivers of contrasting polarities.

```
>>> from example_data import synthetic_event
>>> data=synthetic_event()
>>> print data['PPolarity']
{'Stations': {'TakeOffAngle': matrix([[ 122.8],
        [ 120.8],
        [ 152.4],
        [ 138.7],
        [ 149.6],
        [ 120. ],
        [ 107.4],
        [ 117. ],
        [ 156.4],
        [ 115.3],
        [ 133.3],
        [ 109.1],
        [ 139.9],
        [ 147.2],
        [ 128.7],
                    [ 137.6]]), 'Name': ['S0517', 'S0415', 'S0347', 'S0534
↪',
            'S0244', 'S0618', 'S0650', 'S0595', 'S0271', 'S0155', 'S0529',
↪
            'S0649', 'S0450', 'S0195', 'S0588', 'S0142'], 'Azimuth':␣
↪matrix([[
            55.9],
        [  76.9],
        [ 277.9],
        [   5.4],
        [ 224.7],
        [  31.9],
        [  47.9],
        [  45.2],
        [ 224.6],
        [ 122.6],
        [ 328.4],
        [  45.2],
        [ 309.3],
        [ 187.7],
        [  16.1],
        [ 193.4]])}, 'Measured': matrix([[-1],
        [-1],
        [ 1],
        [-1],
        [ 1],
        [-1],
        [-1],
        [-1],
        [ 1],
        [ 1],
        [-1],
        [-1],
        [-1],
        [ 1],
        [-1],
        [ 1]]), 'Error': matrix([[ 0.05],
        [ 0.05],
        [ 0.05],
```

```
        [ 0.01],
        [ 0.01],
        [ 0.01],
        [ 0.01],
        [ 0.01],
        [ 0.01],
        [ 1.  ],
        [ 0.01],
        [ 1.  ],
        [ 1.  ],
        [ 0.01],
        [ 0.05],
        [ 0.05]])}
```

`examples/synthetic_event.py` contains a script for the double-couple and full moment tensor inversion of the source. It can be run as:

```
$ python synthetic_event.py case=ppolarity
```

Adding a `-l` flag will run the inversion in a single thread.

The important part of the script is:

```
# print output data
print(data['PPolarity'])
data['UID'] += '_ppolarity'
# Set inversion parameters
# Use an iteration random sampling algorithm
algorithm = 'iterate'
# Run in parallel if set on command line
parallel = parallel
# uses a soft memory limit of 1Gb of RAM for estimating the sample sizes
# (This is only a soft limit, so no errors are thrown if the memory usage
#       increases above this)
phy_mem = 1
# Run in double-couple space only
dc = True
# Run for one hundred thousand samples
max_samples = 100000
# Set to only use P Polarity data
inversion_options = 'PPolarity'
# Set the convert flag to convert the output to other source␣
↪parameterisations
convert = True
# Create the inversion object with the set parameters.
inversion_object = Inversion(data, algorithm=algorithm, parallel=parallel,
                             inversion_options=inversion_options, phy_
↪mem=phy_mem, dc=dc,
                             max_samples=max_samples, convert=convert)
# Run the forward model
inversion_object.forward()
# Run the full moment tensor inversion
# Increase the max samples due to the larger source space to 10 million␣
↪samples
max_samples = 10000000
# Create the inversion object with the set parameters.
inversion_object = Inversion(data, algorithm=algorithm, parallel=parallel,
                             inversion_options=inversion_options, phy_
↪mem=phy_mem, dc=not dc,
                             max_samples=max_samples, convert=convert)
# Run the forward model
inversion_object.forward()
# Equivalent to pickling the data:
```

```
#  >>> from example_data import synthetic_event
#  >>> data=synthetic_event()
#  >>> import cPickle
#  >>> cPickle.dump(data,open('synthetic_event_data.inv','wb'))
# And then calling from the command line
#  mtfit --algorithm=iterate --pmem=1 --double-couple --max-
↪samples=100000 \
#    --inversion-options=PPolarity --convert synthetic_event_data.inv
#  mtfit --algorithm=iterate --pmem=1 --max-samples=10000000  \
#    --inversion-options=PPolarity --convert synthetic_event_data.inv
```

The chosen algorithm is the `iterate` algorithm (see *Random Monte Carlo sampling*) for `100 000` samples for the double-couple case and `10 000 000` for the full moment tensor inversion. `100 000` samples in the double-couple space corresponds to approximately `50` samples in each parameter (strike, dip cosine, and rake). While this more dense samplings are possible, this produces a good sampling of the posterior PDF quickly, especially when run in parallel.

Warning: If running this in parallel on a server, be aware that because the number of workers option `number_workers` is not set, as many processes as processors will be spawned, slowing down the machine for any other users.

`10 000 000` samples for the full moment tensor inversion may seem like a coarse sampling, however, due to the higher noise level and the use of P polarity data only, there is much less constraint on the full moment tensor source PDF so this sampling still provides a good approximation of the results.

This script is equivalent to pickling the data:

```
>>> from example_data import synthetic_event
>>> data=synthetic_event()
>>> import cPickle
>>> cPickle.dump(data,open('synthetic_event_data.inv','wb'))
```

And then calling from the command line (Assuming parallel running: `-l` flag to run on a single processor):

```
$ mtfit --algorithm=iterate --pmem=1 --double-couple --max-samples=100000 \
    --inversion-options=PPolarity --convert synthetic_event_data.inv
$ mtfit --algorithm=iterate --pmem=1 --max-samples=10000000  \
    --inversion-options=PPolarity --convert synthetic_event_data.inv
```

These inversions should not take long to run (running on a single core of an i5 processor, the two inversions take 2 and 72 seconds respectively), although the conversions using `mtfit.MTconvert` can add to this time, but will reduce the time when plotting the results. The solutions are outputted as a MATLAB file for the DC and MT solutions respectively, with name `synthetic_example_event_ppolarityDC.mat` and `synthetic_example_event_ppolarityMT.mat` respectively.

The results are shown in Fig. *5.2*.

5.1.2 Synthetic Polarity and Amplitude Ratio Inversion

Including amplitude ratios (`P/SH` and `P/SV`) in the inversion can improve the source constraint, but, as shown in *Pugh et al. 2016a*, the amplitude ratio can include a systematic error that is dependent on the noise level, leading to deviations from the "True" source.

Warning: Amplitudes and amplitude ratios can show systematic deviations from the "True" value due to the noise and the method of measuring the amplitudes. Consequently, care must be taken when using amplitude ratio data in the source inversion, including tests to see if the different amplitude ratios are consistent with each other and the polarity only inversion.

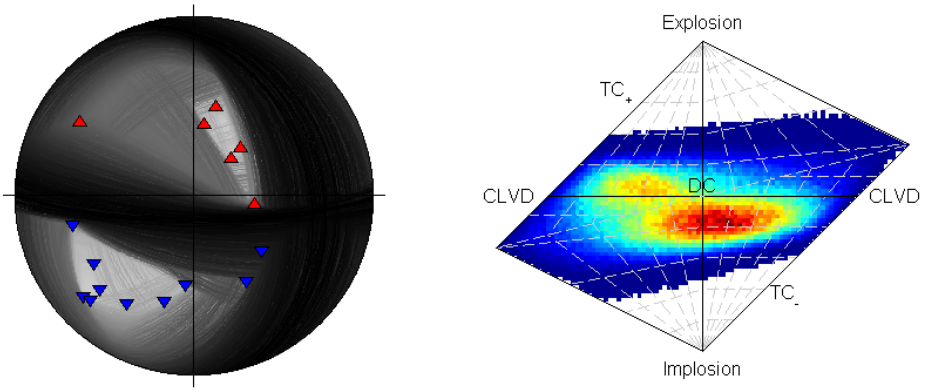

Fig. 5.2: *Beachball plot showing the fault plane orientations for the double-couple constrained inversion and the marginalised source-type PDF for the full moment tensor inversion of the synthetic data using polarities(Plotted using MTplot MATLAB code).*

These tests are ignored in this example, but can lead to very sparse solutions in the source inversion.

`examples/synthetic_event.py` contains a script for the double-couple and full moment tensor inversion of the source. It can be run as:

```
$ python synthetic_event.py case=ar
```

Adding a `-l` flag will run the inversion in a single thread.

The important part of the script is:

```
# print output data
print(data['PPolarity'])
print(data['P/SHRMSAmplitudeRatio'])
print(data['P/SVRMSAmplitudeRatio'])
data['UID'] += '_ar'
# Set inversion parameters
# Use an iteration random sampling algorithm
algorithm = 'iterate'
# Run in parallel if set on command line
parallel = parallel
# uses a soft memory limit of 1Gb of RAM for estimating the sample sizes
# (This is only a soft limit, so no errors are thrown if the memory usage
#     increases above this)
phy_mem = 1
# Run in double-couple space only
dc = True
# Run for one hundred thousand samples
max_samples = 100000
# Set to only use P Polarity data
inversion_options = ['PPolarity', 'P/SHRMSAmplitudeRatio',
                     'P/SVRMSAmplitudeRatio']
# Set the convert flag to convert the output to other source␣
↪parameterisations
convert = False
# Create the inversion object with the set parameters.
inversion_object = Inversion(data, algorithm=algorithm, parallel=parallel,
                             inversion_options=inversion_options, phy_
↪mem=phy_mem, dc=dc,
                             max_samples=max_samples, convert=convert)
# Run the forward model
```

```
inversion_object.forward()
# Run the full moment tensor inversion
# Increase the max samples due to the larger source space to 50 million␣
↪samples
max_samples = 50000000
# Create the inversion object with the set parameters.
inversion_object = Inversion(data, algorithm=algorithm, parallel=parallel,
                             inversion_options=inversion_options, phy_
↪mem=phy_mem, dc=not dc,
                             max_samples=max_samples, convert=convert)
# Run the forward model
inversion_object.forward()
# Equivalent to pickling the data:
#  >>> from example_data import synthetic_event
#  >>> data=synthetic_event()
#  >>> import cPickle
#  >>> cPickle.dump(data,open('synthetic_event_data.inv','wb'))
# And then calling from the command line
#  mtfit --algorithm=iterate --pmem=1 --double-couple --max-
↪samples=100000 \
#    --inversion-options=PPolarity,P/SHRMSAmplitudeRatio,P/
↪SVRMSAmplitudeRatio \
#    --convert synthetic_event_data.inv
#  mtfit --algorithm=iterate --pmem=1 --max-samples=50000000  \
#    --inversion-options=PPolarity,P/SHRMSAmplitudeRatio,P/
↪SVRMSAmplitudeRatio \
#    --convert synthetic_event_data.inv
```

The chosen algorithm is the `iterate` algorithm (see *Random Monte Carlo sampling*) for `100 000` samples for the double-couple case and `50 000 000` for the full moment tensor inversion. `100 000` samples in the double-couple space corresponds to approximately `50` samples in each parameter (strike, dip cosine, and rake). While this more dense samplings are possible, this produces a good sampling of the posterior PDF quickly, especially when run in parallel.

Warning: If running this in parallel on a server, be aware that because the number of workers option `number_workers` is not set, as many processes as processors will be spawned, slowing down the machine for any other users.

`50 000 000` samples are used instead of the `10 000 000` samples in the *Synthetic P Polarity Inversion* example because amplitude ratios provide a much stronger constraint on the source type, so more samples are required for an effective sampling of the source PDF.

This script is equivalent to pickling the data:

```
>>> from example_data import synthetic_event
>>> data=synthetic_event()
>>> import cPickle
>>> cPickle.dump(data,open('synthetic_event_data.inv','wb'))
```

And then calling from the command line (Assuming parallel running: `-l` flag to run on a single processor):

```
$ mtfit --algorithm=iterate --pmem=1 --double-couple --max-samples=100000 \
    --inversion-options=PPolarity,P/SHRMSAmplitudeRatio,P/SVRMSAmplitudeRatio \
    --convert synthetic_event_data.inv
$ mtfit --algorithm=iterate --pmem=1 --max-samples=10000000  \
    --inversion-options=PPolarity,P/SHRMSAmplitudeRatio,P/SVRMSAmplitudeRatio \
    --convert synthetic_event_data.inv
```

These inversions should not take long to run (running on a single core of an i5 processor, the two inversions take 20 and 436 seconds respectively), although the conversions using `mtfit.MTconvert` can add to this time,

but will reduce the time when plotting the results. These run times are longer than for only polarity data, both because of the additional data, and the increased complexity of the ratio-pdf-label. The solutions are outputted as a MATLAB file for the DC and MT solutions respectively, with name `synthetic_example_event_arDC.mat` and `synthetic_example_event_arMT.mat` respectively.

The results are shown in Fig. *5.3*.

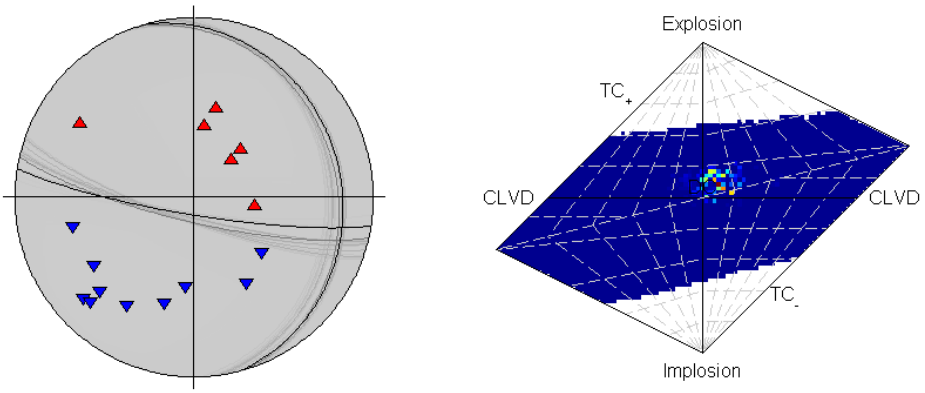

Fig. 5.3: *Beachball plot showing the fault plane orientations for the double-couple constrained inversion and the marginalised source-type PDF for the full moment tensor inversion of the synthetic data using polarities and amplitude ratios (Plotted using MTplot MATLAB code).*

5.2 Krafla Event

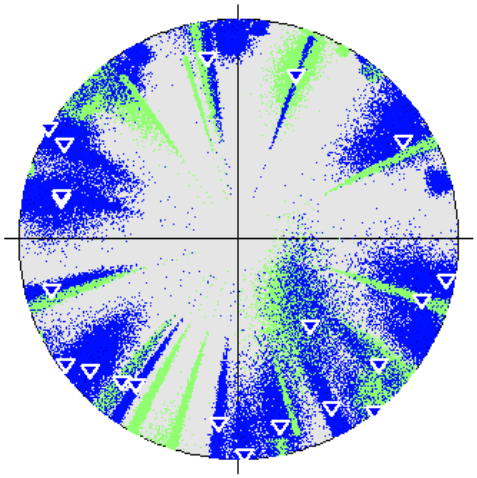

Fig. 5.4: *Beachball plot of the effect of the location uncertainty on the receivers (dots, darker are higher probability locations) (Plotted using MTplot MATLAB code)*

A strongly non-double-couple event, with manually picked P and S arrival times and P polarities, and then located using NonLinLoc[35] is used as an example for inversions with real data. In this case the S arrivals were hard to

[35] http://alomax.free.fr/nlloc

measure an amplitude for, so amplitude ratios are ignored. Instead, polarities and polarity probabilities (calculated using the approach described in *Pugh et al. 2016a* and implemented in :ref:rom `autopol`) are used separately to invert for the source, along with including the location data (Fig. *5.4*). This event is shown in *Pugh et al. 2016a* and investigated in more detail in *Watson et al. 2015*.

5.2.1 Krafla P Polarity Inversion

There are 21 P-polarity arrivals for the synthetic event, and the locations of these receivers are shown in the figure on the right. There is quite a large depth uncertainty, leading to a large variation in take-off angles (also shown in the figure on the right)

```
>>> from example_data import krafla_event
>>> data=krafla_event()
>>> print data['PPolarity']
{'Stations': {'TakeOffAngle': matrix([[  77. ],
        [ 108.8],
        [ 104.3],
        [  87.4],
        [  69.7],
        [  86.2],
        [  90.4],
        [  72.1],
        [  78.8],
        [  90. ],
        [ 103.2],
        [  73.2],
        [  74.7],
        [ 101.1],
        [ 103.6],
        [  76.3],
        [  71.5],
        [  68.1],
        [  86.3],
        [ 137.5],
        [  72.6]]), 'Name': ['KVO0', 'K080', 'K040', 'K200', 'REN0',
'K120', 'K100', 'K008', 'K020', 'K140', 'K060', 'K250', 'K170',
'K050', 'K090', 'K210', 'K010', 'GHA0', 'K110', 'K070', 'K220'],
'Azimuth': matrix([[
254.9],
        [ 102.5],
        [ 347.4],
        [ 178.3],
        [ 215. ],
        [ 234. ],
        [ 120.3],
        [ 350.4],
        [ 298.6],
        [ 141.8],
        [ 288.4],
        [ 186.2],
        [ 131.4],
        [  48.4],
        [ 239.6],
        [ 151. ],
        [ 284. ],
        [  19.6],
        [ 100.9],
        [ 319.8],
        [ 219.4]])}, 'Measured': matrix([[-1],
        [-1],
        [-1],
```

```
        [-1],
        [-1],
        [-1],
        [-1],
        [-1],
        [-1],
        [-1],
        [-1],
        [-1],
        [-1],
        [-1],
        [-1],
        [-1],
        [-1],
        [-1],
        [-1],
        [-1]]), 'Error': matrix([[ 0.01],
        [ 0.01],
        [ 0.01],
        [ 0.01],
        [ 0.01],
        [ 0.01],
        [ 0.01],
        [ 0.01],
        [ 0.01],
        [ 0.01],
        [ 0.01],
        [ 0.01],
        [ 0.01],
        [ 0.01],
        [ 0.01],
        [ 0.01],
        [ 0.01],
        [ 0.01],
        [ 0.01]])}
```

`examples/krafla_event.py` contains a script for the double-couple and full moment tensor inversion of the source. It can be run as:

```
$ python krafla_event.py case=ppolarity
```

Adding a `-l` flag will run the inversion in a single thread.

The important part of the script is:

```
# print output data
print(data['PPolarity'])
data['UID'] += '_ppolarity'
# Set inversion parameters
# Use an iteration random sampling algorithm
algorithm = 'iterate'
# Run in parallel if set on command line
parallel = parallel
# uses a soft memory limit of 1Gb of RAM for estimating the sample sizes
# (This is only a soft limit, so no errors are thrown if the memory usage
#         increases above this)
phy_mem = 1
# Run in double-couple space only
dc = True
# Run for one hundred thousand samples
```

```
max_samples = 100000
# Set to only use P Polarity data
inversion_options = 'PPolarity'
# Set the convert flag to convert the output to other source␣
↪parameterisations
convert = True
# Set location uncertainty file path
location_pdf_file_path = 'krafla_event.scatangle'
# Handle location uncertainty
# Set number of location samples to use (randomly sampled from PDF) as␣
↪this
#    reduces calculation time
# (each location sample is equivalent to running an additional event)
number_location_samples = 5000
# Bin Scatangle File
bin_scatangle = True
# Use mtfit.__core__.mtfit function
mtfit(data, location_pdf_file_path=location_pdf_file_path,␣
↪algorithm=algorithm,
      parallel=parallel, inversion_options=inversion_options, phy_mem=phy_
↪mem, dc=dc,
      max_samples=max_samples, convert=convert, bin_scatangle=bin_
↪scatangle,
      number_location_samples=number_location_samples)
# Change max_samples for MT inversion
max_samples = 1000000
# Create the inversion object with the set parameters.
mtfit(data, location_pdf_file_path=location_pdf_file_path,␣
↪algorithm=algorithm,
      parallel=parallel, inversion_options=inversion_options, phy_mem=phy_
↪mem,
      dc=not dc, max_samples=max_samples, convert=convert,
      bin_scatangle=bin_scatangle, number_location_samples=number_
↪location_samples)
# Equivalent to pickling the data and outputting the location uncertainty:
#  >>> from example_data import krafla_event,krafla_location
#  >>> data=krafla_event()
#  >>> open('krafla_event.scatangle','w').write(krafla_location())
#  >>> import cPickle
#  >>> cPickle.dump(data,open('krafla_event.inv','wb'))
# And then calling from the command line
#  mtfit --location_pdf_file_path=krafla_event.scatangle --
↪algorithm=iterate \
#    --pmem=1 --double-couple --max-samples=100000 \
#    --inversion-options=PPolarity --convert --bin-scatangle krafla_event.
↪inv
#  mtfit --location_pdf_file_path=krafla_event.scatangle --
↪algorithm=iterate \
#    --pmem=1 --max-samples=10000000  --inversion-options=PPolarity --
↪convert \
#    --bin-scatangle krafla_event.inv
```

In this example the `mtfit.__core__.mtfit()` function is used instead of creating the inversion object directly. Again, the chosen algorithm is the `iterate` algorithm (see *Random Monte Carlo sampling*) for `100 000` samples for the double-couple case and `1 000 000` for the full moment tensor inversion. The location uncertainty distribution is binned (`--bin_scatangle`), which runs before the main inversion is carried out. This uses the source-scatangle extension to both parse and bin the location PDF distribution.

Warning: If running this in parallel on a server, be aware that because the number of workers option `number_workers` is not set, as many processes as processors will be spawned, slowing down the machine

for any other users.

This script is equivalent to pickling the data:

```
>>> from example_data import krafla_event
>>> data=krafla_event()
>>> import cPickle
>>> cPickle.dump(data,open('krafla_event_data.inv','wb'))
```

And then calling from the command line (Assuming parallel running: `-l` flag to run on a single processor):

```
$ mtfit --location_pdf_file_path=krafla_event.scatangle --algorithm=iterate \
    --pmem=1 --double-couple --max-samples=100000 --inversion-options=PPolarity \
    --convert --bin-scatangle krafla_event_data.inv
$ mtfit --location_pdf_file_path=krafla_event.scatangle --algorithm=iterate \
    --pmem=1 --max-samples=10000000  --inversion-options=PPolarity --convert \
    --bin-scatangle krafla_event_data.inv
```

These inversions will take longer to run than the previous examples, due to the location uncertainty (running using 8 cores, the two inversions take 4 and 11 minutes respectively), although the conversions using `mtfit.MTconvert` can add to this time, but will reduce the time when plotting the results. The solutions are outputted as a MATLAB file for the DC and MT solutions respectively, with name `krafla_event_ppolarityDC.mat` and `krafla_event_ppolarityMT.mat` respectively.

Fig. *5.5* shows the results plotted using MATLAB.

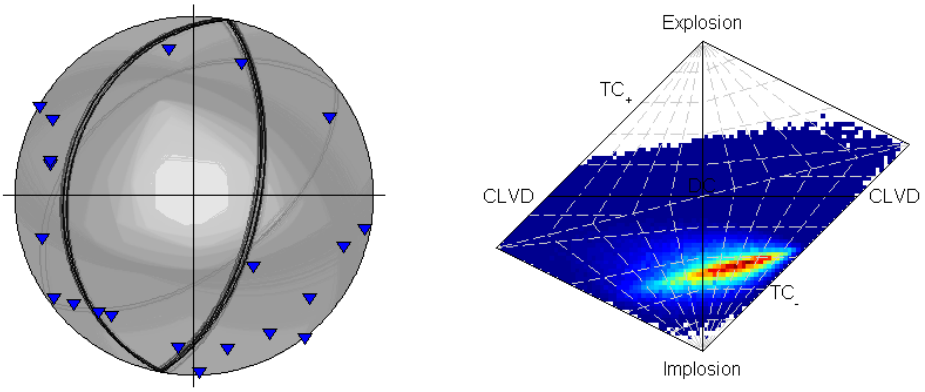

Fig. 5.5: *Beachball plot showing the fault plane orientations for the double-couple constrained inversion and the marginalised source-type PDF for the full moment tensor inversion of the krafla data using polarities(Plotted using MTplot MATLAB code).*

5.2.2 Krafla P Polarity Probability Inversion

There are 21 P-polarity arrivals for the synthetic event, but more observations, and a better understanding of the uncertainties on the polarities can be obtained using the automated Bayesian polarity probabilities generated using the approach described in *Pugh et al., 2016b*, Pugh, 2016a. Nevertheless, much of this example is the same as the *Krafla P Polarity Inversion* example.

`examples/krafla_event.py` contains a script for the double-couple and full moment tensor inversion of the source. It can be run as:

```
$ python krafla_event.py case=ppolarityprob
```

Adding a `-l` flag will run the inversion in a single thread.

The important part of the script is:

```
# print output data
print(data['PPolarityProb'])
data['UID'] += '_ppolarityprob'
# Set inversion parameters
# Use an mcmc sampling algorithm
algorithm = 'mcmc'
# Set parallel to false as running McMC
parallel = False
# uses a soft memory limit of 0.5Gb of RAM for estimating the sample sizes
# (This is only a soft limit, so no errors are thrown if the memory usage
#     increases above this)
phy_mem = 0.5
# Run both inversions
dc_mt = True
# Run for one hundred thousand samples
chain_length = 100000
# Set to only use P Polarity data
inversion_options = 'PPolarityProb'
# Set the convert flag to convert the output to other source␣
↪parameterisations
convert = True
# Set location uncertainty file path
location_pdf_file_path = 'krafla_event.scatangle'
# Handle location uncertainty
# Set number of location samples to use (randomly sampled from PDF) as␣
↪this
#    reduces calculation time
# (each location sample is equivalent to running an additional event)
number_location_samples = 5000
# Bin Scatangle File
bin_scatangle = True
# Use mtfit.__core__.mtfit function
mtfit(data, location_pdf_file_path=location_pdf_file_path,␣
↪algorithm=algorithm,
      parallel=parallel, inversion_options=inversion_options, phy_mem=phy_
↪mem,
      chain_length=chain_length, convert=convert, bin_scatangle=bin_
↪scatangle,
      dc_mt=dc_mt, number_location_samples=number_location_samples)
# Trans-Dimensional inversion
data['UID'] += '_transd'
# Use a transdmcmc sampling algorithm
algorithm = 'transdmcmc'
# Use mtfit.__core__.mtfit function
mtfit(data, location_pdf_file_path=location_pdf_file_path,␣
↪algorithm=algorithm,
      parallel=parallel, inversion_options=inversion_options, phy_mem=phy_
↪mem,
      chain_length=chain_length, convert=convert, bin_scatangle=bin_
↪scatangle,
      number_location_samples=number_location_samples)
# Equivalent to pickling the data and outputting the location uncertainty:
#  >>> from example_data import krafla_event,krafla_location
#  >>> data=krafla_event()
#  >>> open('krafla_event.scatangle','w').write(krafla_location())
#  >>> import cPickle
#  >>> cPickle.dump(data,open('krafla_event_data.inv','wb'))
```

```
# And then calling from the command line
# mtfit --location_pdf_file_path=krafla_event.scatangle --algorithm=mcmc -
↪b \
#     --pmem=1 --double-couple --max-samples=100000 \
#     --inversion-options=PPolarityProb --convert --bin-scatangle \
#     krafla_event.inv
# mtfit --location_pdf_file_path=krafla_event.scatangle --
↪algorithm=transdmcmc \
#     --pmem=1 --max-samples=100000  --inversion-options=PPolarityProb \
#     --convert --bin-scatangle krafla_event.inv
```

In this example the `mtfit.__core__.mtfit()` function is used instead of creating the inversion object directly. The chosen algorithm is the `mcmc` algorithm (see *Markov chain Monte Carlo sampling*) for a chain length of `100 000` samples for both the double-couple case and the full moment tensor inversion. Additionally a trans-dimensional McMC approach is run, allowing comparison between the two.

The location uncertainty distribution is binned (`--bin_scatangle`), which runs before the main inversion is carried out. This uses the `extensions/scatangle.py` extension to both parse and bin the location PDF distribution.

Warning: If running this in parallel on a server, be aware that because the number of workers option `number_workers` is not set, as many processes as processors will be spawned, slowing down the machine for any other users.

This script is equivalent to pickling the data:

```
>>> from example_data import krafla_event
>>> data=krafla_event()
>>> import cPickle
>>> cPickle.dump(data,open('krafla_event_data.inv','wb'))
```

And then calling from the command line:

```
$ mtfit --location_pdf_file_path=krafla_event.scatangle -l --algorithm=mcmc -b \
    --pmem=1 --double-couple --max-samples=100000 --inversion-
↪options=PPolarityProb \
    --convert --bin-scatangle krafla_event_data.inv
$ mtfit --location_pdf_file_path=krafla_event.scatangle -l --algorithm=transdmcmc \
    --pmem=1 --max-samples=100000  --inversion-options=PPolarityProb --convert \
    --bin-scatangle krafla_event_data.inv
```

These inversions will take longer to run than the previous examples, both due to the location uncertainty, the McMC algorithms, and the non-parallel running (running on a single process on an i5 processor, the inversions take 7 hours), although the conversions using `mtfit.MTconvert` can add to this time, but will reduce the time when plotting the results. The solutions are outputted as a MATLAB file for the DC and MT solutions respectively, with name `krafla_event_ppolarityDC.mat` and `krafla_event_ppolarityMT.mat` respectively.

Fig. *5.6* shows the results plotted using MATLAB.

The `transdmcmc` algorithm option produces similar results to the normal McMC example, and both results are consistent with the random sampling approach for polarity data (*Krafla P Polarity*). The chain-lengths may be on the long side, and it is possible that shorter chains would produce satisfactory sampling of the solution PDF.

Fig. *5.7* shows the results for the `transdmcmc` inversion plotted using MATLAB.

5.3 Bayesian Evidence

The Bayesian evidence estimates the evidence for the model given the posterior PDF and the parameter priors. It accounts for the model dimensions, and can penalise the higher dimensional model by the parameter priors. The

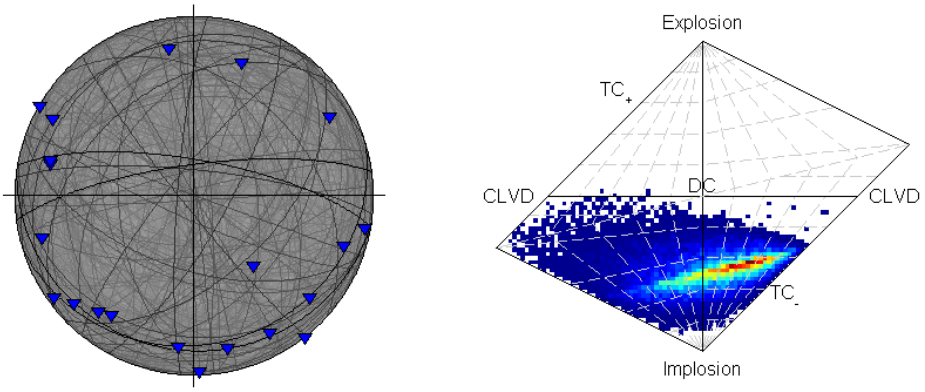

Fig. 5.6: *Beachball plot showing the fault plane orientations for the double-couple constrained inversion and the marginalised source-type PDF for the full moment tensor inversion of the krafla data using polarity probabilities (Plotted using MTplot MATLAB code).*

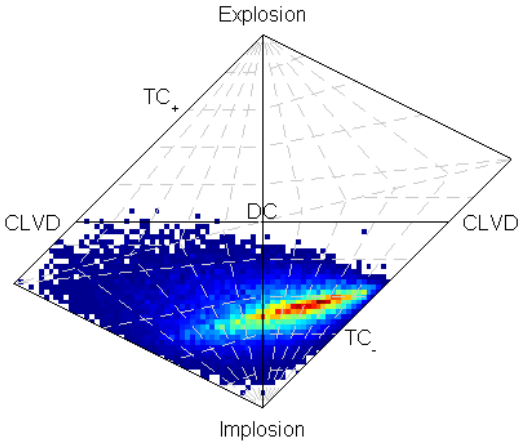

Fig. 5.7: *The marginalised source-type PDF for the full moment tensor inversion of the krafla data using polarity probabilities and the reversible jump (trans-dimensional) McMC approach (Plotted using MTplot MATLAB code).*

Bayesian evidence is:

$$\mathrm{p}\left(data|model\right) = \int_{\mathrm{x}} \mathrm{p}\left(data|x\right)\mathrm{p}\left(x|model\right)\mathrm{d}x.$$

Since `mtfit` produces random samples of the source PDF, so the Bayesian evidence is calculated as a sum over the samples:

$$\mathrm{p}\left(data|model\right) = \sum_{x} \mathrm{p}\left(data|x\right)\mathrm{p}\left(x|model\right)\delta x,$$

but care must be taken with the choice of the prior parameterisation. This must correspond to the same parameterisation in which the Monte Carlo samples were drawn, either directly or by correcting both the prior distribution and the δx values. A Monte Carlo approach can be affected by small sample sizes in the integration, which is sometimes the case when the source PDF is dominated by a few very large probability samples.

`mtfit` produces a Bayesian evidence estimate for a constrained inversion. These can be combined for the DC and MT models, with $\mathcal{B}$ corresponding to a Bayesian evidence estimate:

$$\ln\left(\mathcal{B}_{\max}\right) = \max\left(\ln\left(\mathcal{B}_{\mathrm{DC}}\right), \ln\left(\mathcal{B}_{\mathrm{MT}}\right)\right),$$
$$p_{\mathrm{DC}} = \frac{\mathrm{e}^{\ln(\mathcal{B}_{\mathrm{DC}})-\ln(\mathcal{B}_{\max})}}{\mathrm{e}^{\ln(\mathcal{B}_{\mathrm{DC}})-\ln(\mathcal{B}_{\max})} + \mathrm{e}^{\ln(\mathcal{B}_{\mathrm{MT}})-\ln(\mathcal{B}_{\max})}},$$
$$p_{\mathrm{MT}} = \frac{\mathrm{e}^{\ln(\mathcal{B}_{\mathrm{MT}})-\ln(\mathcal{B}_{\max})}}{\mathrm{e}^{\ln(\mathcal{B}_{\mathrm{DC}})-\ln(\mathcal{B}_{\max})} + \mathrm{e}^{\ln(\mathcal{B}_{\mathrm{MT}})-\ln(\mathcal{B}_{\max})}}.$$

For the examples shown above, the model probability estimates from the Bayesian evidence are:

Example	p_{DC}	p_{MT}
Synthetic Polarities	0.64	0.36
Synthetic Polarity and Amplitude Ratios	0.82	0.18
Krafla P Polarity	0.0008	0.9992

These can be calculated using the *`mtfit.probability.model_probabilities()`* function, which takes the logarithm of the Bayesian evidence for each model type as an argument:

`mtfit.probability.`**`model_probabilities`**`(`*`*args`*`)`

Calculate the model probabilities for a discrete set of models using the ln_bayesian_evidences, provided as args.

e.g. to compare between DC and MT:

pDC,pMT=model_probabilities(dc_ln_bayesian_evidence,mt_ln_bayesian_evidence)

Args floats - ln_bayesian_evidence for each model type

Returns

tuple: Tuple of the normalised model probabilities for the corresponding ln_bayesian_evidence inputs

The normal McMC approach cannot be used to estimate the model probabilities, however the trans-dimensional example is consistent with the *Krafla P Polarity* model probabilities as that produces an estimate $p_{\mathrm{DC}} = 0$.

`mtfit` also calculates the Kullback-Liebler divergence of the resultant PDF from the sampling prior, which is a measure of how much difference there is between the resultant PDF and the prior information. This is outputted during the inversion approach (when using Monte Carlo random sampling), and saved to the output file.

5.4 Joint Inversion

`mtfit` can also carry out joint invesions and include relative amplitude data (*Pugh et al., 2015t*). This examples uses data from two co-located synthetic events with overlapping receivers. `examples/relative_event.py` contains a script for the double-couple and full moment tensor inversion of the source. It can be run as:

```
$ python relative_event.py
```

Adding a `-l` flag will run the inversion in a single thread.

The important part of the script is:

```
# Set inversion parameters
# Use an iteration random sampling algorithm
algorithm = 'iterate'
# Run in parallel if set on command line
parallel = parallel
# uses a soft memory limit of 1Gb of RAM for estimating the sample sizes
# (This is only a soft limit, so no errors are thrown if the memory usage
#       increases above this)
phy_mem = 1
# Run in double-couple space only
dc = True
# Run for 10 million samples - quite coarse for relative inversion of two␣
↪events
max_samples = 10000000
# Set to only use P Polarity data
inversion_options = ['PPolarity', 'PAmplitude']
# Set the convert flag to convert the output to other source␣
↪parameterisations
convert = True
# Create the inversion object with the set parameters.
inversion_object = Inversion(data, algorithm=algorithm, parallel=parallel,
                             inversion_options=inversion_options, phy_
↪mem=phy_mem, dc=dc,
                             max_samples=max_samples, convert=convert,␣
↪multiple_events=True,
                             relative_amplitude=True)
# Run the forward model
inversion_object.forward()
# Run the full moment tensor inversion
# Use mcmc algorithm for full mt space as random sampling can require a
# prohibitive number of samples
algorithm = 'mcmc'
# Set McMC parameters
burn_length = 30000
chain_length = 100000
min_acceptance_rate = 0.1
max_acceptance_rate = 0.3
# Create the inversion object with the set parameters.
inversion_object = Inversion(data, algorithm=algorithm, parallel=False,
                             inversion_options=inversion_options, phy_
↪mem=phy_mem, dc=not dc,
                             chain_length=chain_length, max_acceptance_
↪rate=max_acceptance_rate,
                             min_acceptance_rate=min_acceptance_rate,␣
↪burn_length=burn_length,
                             convert=convert, multiple_events=True,␣
↪relative_amplitude=True)
# Run the forward model
inversion_object.forward()
# Equivalent to pickling the data:
#  >>> from example_data import relative_event
#  >>> data=relative_event()
#  >>> import cPickle
#  >>> cPickle.dump(data,open('relative_event_data.inv','wb'))
# And then calling from the command line
#  mtfit --algorithm=iterate --pmem=1 --double-couple --max-
↪samples=10000000 \
```

```
#     --inversion-options=PPolarity,PAmplitude --convert --relative \
#     --multiple-events relative_event_data.inv
#  mtfit --algorithm=mcmc --pmem=1 --chain-length=100000  \
#     --burn_in=30000 --min_acceptance_rate=0.1 \
#     --max_acceptance_rate=0.3 --inversion-options=PPolarity,PAmplitude \
#     --convert --relative --multiple-events relative_event_data.inv
```

In this example the *mtfit.inversion.Inversion* class is created directly. The chosen algorithm for the double-couple inversion is the `iterate` algorithm (see *Random Monte Carlo sampling*) for `10 000 000` samples for the double-couple case. A large sample size is required when running the joint inversion because if the probabilities of obtaining a non-zero probability sample for both events is less than or equal to the product of the probabilities of obtaining a non-zero probability sample for the events individually, i.e if the fraction of non-zero probability samples for event 1 is f_1 and the fraction for event 2 is f_2, then the fraction for the joint samping $f_j \leq f_1.f_2$. Consequently it soon becomes infeasible to run the monte-carlo sampling algorithm for the full moment tensor case.

The full moment tensor inversion uses the `mcmc` algorithm to reduce the dependence of the result on the sample size. The burn in length is set to `30 000` and the chain length to `100 000`, while the targeted acceptance rate window is reduced from the normal McMC selection (0.3 - 0.5), instead targetting between 0.1 and 0.3.

Warning: If running this in parallel on a server, be aware that because the number of workers option `number_workers` is not set, as many processes as processors will be spawned, slowing down the machine for any other users.

This script is equivalent to pickling the data:

```
>>> from example_data import relative_event
>>> data=relative_event()
>>> import cPickle
>>> cPickle.dump(data,open('relative_event_data.inv','wb'))
```

And then calling from the command line:

```
$ mtfit --algorithm=iterate --pmem=1 --double-couple --max-samples=10000000 \
    --inversion-options=PPolarity,PAmplitude --convert --relative \
    --multiple-events relative_event_data.inv

$ mtfit --algorithm=mcmc --pmem=1 --chain-length=100000  \
    --burn_in=30000 --min_acceptance_rate=0.1 \
    --max_acceptance_rate=0.3 --inversion-options=PPolarity,PAmplitude \
    --convert --relative --multiple-events relative_event_data.inv
```

These inversions will take longer to run than the previous examples, due to the increased sample size required for the joint PDF. The solutions are outputted as a MATLAB file for the DC and MT solutions respectively, with name `mtfitOutput_joint_inversionDC.mat` and `mtfitOutput_joint_inversionMT.mat` respectively.

Fig. *5.8* shows the results plotted using the plot module of `mtfit`.

These have improved the constraint compared to the joint inversion without relative amplitudes (Fig. *5.9*).

Althought the constraint has improved, the relative amplitudes have changed the full moment tensor solution for the second event, possibly due to errors in the P-amplitude estimation (*Pugh et al., 2016a*).

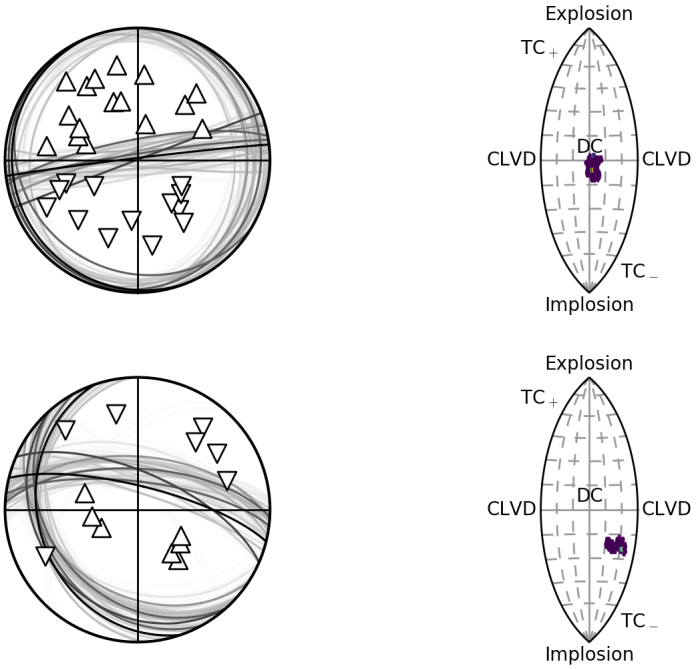

Fig. 5.8: *Beachball plot showing the fault plane orientations for the double-couple constrained inversion and the marginalised source-type PDF for the full moment tensor inversion for two events, inverted using P polarities and relative P amplitude ratios.*

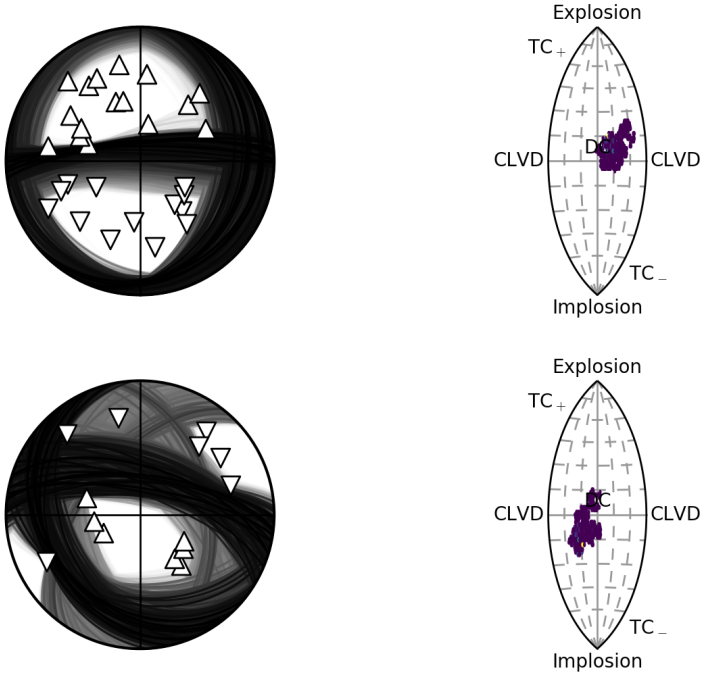

Fig. 5.9: *Beachball plot showing the fault plane orientations for the double-couple constrained inversion and the marginalised source-type PDF for the full moment tensor inversion for two events, inverted using P polarities.*

CHAPTER 6

Bayesian Approach

The Bayesian approach used by `mtfit` is based on the approach described in *Pugh et al, 2016a*

6.1 Bayes Theory

Inversion approaches fit model parameters to observed data, to find the best fitting parameters. In `mtfit`, the probability of the data being correct is evaluated for the possible sources. The resulting estimates of the PDF can be combined for all the data to approximate the true PDF for the source. This PDF can be considered to describe the probability of the observed data for a given source, the likelihood, $p\left(\mathrm{data}\,|\,\mathrm{model}\right)$. However the value of interest in such an inversion is the probability of the model given the observed data, the posterior PDF. This can be evaluated from the likelihood using Bayes' Theory (*Bayes and Price, 1763* , *Laplace, 1812* , *Sivia, 2000*) is given simply by:

$$\mathrm{p}\left(\mathrm{model}\,|\,\mathrm{data}\right)=\frac{\mathrm{p}\left(\mathrm{data}\,|\,\mathrm{model}\right)\,\mathrm{p}\left(\mathrm{model}\right)}{\mathrm{p}\left(\mathrm{data}\right)}$$

This relates the data likelihood $\mathrm{p}\left(\mathrm{data}\,|\,\mathrm{model}\right)$ to the posterior probability $\mathrm{p}\left(\mathrm{model}\,|\,\mathrm{data}\right)$ via the prior probabilities $\mathrm{p}\left(\mathrm{model}\right)$ and $\mathrm{p}\left(\mathrm{data}\right)$. The prior distributions incorporate the known information about the two parameters, and although the choice of prior is not trivial, it does not have a large affect on the posterior PDF given enough data.

6.2 Uncertainties

Source inversion is particularly sensitive to uncertainties, and since it is usually carried out after several other required steps, the effect of the uncertainties may be difficult to understand. The uncertainties can be broken down into several different types: instrument errors, model errors and background noise.

Although the interdependencies between the uncertainties can be explored, a quantitative relationship is not known. For such a treatment to be truly rigorous, the variations in these errors must be included throughout the inversion. The Bayesian formulation allows rigorous inclusion of uncertainties in the problem using marginalisation. Marginalisation removes the dependence of the joint PDF, $P(A,B)$, on one of the variables, by integrating over the PDF for that variable:

$$\mathrm{p}\left(A\right)=\int\mathrm{p}\left(A\,,\,B\right)dB=\int\mathrm{p}\left(A\,|\,B\right)\mathrm{p}\left(B\right)dB$$

Marginalisation can be used to incorporate the uncertainties in the inversion into the final PDF.

It is assumed that the noise has a mean and variance that are measurable, and therefore the most ambiguous (maximum entropy) distribution for these measurements is the Gaussian distribution (*Pugh et al, 2015*). For de-meaned data, the noise can be assumed also to have zero mean, and a standard deviation σ_{mes}, so that the PDF $\mathrm{p}\left(\Delta_{mes}\right)$, is:

$$\mathrm{p}\left(\Delta_{mes}\right) = \frac{1}{\sqrt{2\pi\sigma_{mes}^2}}\mathrm{e}^{-\frac{\Delta_{mes}^2}{2\sigma_{mes}^2}}$$

There are several different types of uncertainties however most can be simplified to independent uncertainties on each trace, and uncertainties in the underlying model.

6.3 Posterior PDF

The likelihoods for the two different data types are described in chapter 7.

The posterior PDFs for the data used in `mtfit` are, for a known Earth model:

$$\mathrm{p}\left(\mathbf{d}'\mid\mathbf{M},\mathbf{t},\epsilon,\mathbf{k}\right) = \iint\sum_{j=1}^{M}\prod_{i=1}^{N}\left[\mathrm{p}\left(Y_i\mid\mathbf{A_{ij}}=\mathbf{a}_j\cdot\tilde{\mathbf{M}},\sigma_\mathbf{i},\varpi_i\right)\mathrm{p}\left(\mathbf{R_i}\mid\mathbf{A_i}=\mathbf{a}_j\cdot\tilde{\mathbf{M}},\sigma_\mathbf{i},\varpi_i\right)\right]\mathrm{p}\left(\sigma\right)\mathrm{p}\left(\varpi\right)d\sigma d\varpi$$

An unknown earth model, samples from the model distribution, $\mathbf{G}_k$, are included using a Monte Carlo method based marginalisation:

$$\mathrm{p}\left(\mathbf{d}'\mid\mathbf{M},\mathbf{t},\epsilon,\mathbf{k}\right) = \iint\sum_{k=1}^{Q}\sum_{j=1}^{M}\prod_{i=1}^{N}\left[\mathrm{p}\left(Y_i\mid\mathbf{A_{ijk}}=\mathbf{a}_{jk}\cdot\tilde{\mathbf{M}},\sigma_\mathbf{i},\varpi_i\right)\mathrm{p}\left(\mathbf{R_i}\mid\mathbf{A_{ijk}}=\mathbf{a}_{jk}\cdot\tilde{\mathbf{M}},\sigma_\mathbf{i},\varpi_i\right)\right]\mathrm{p}\left(\sigma\right)\mathrm{p}\left(\varpi\right)d\sigma d\varpi,$$

where $\mathbf{a}_{jk} = \mathbf{a}\left(\mathbf{x}_j,\mathbf{G}_k\right)$ refers to the station propagation coefficients associated with the location at $\mathbf{x}_j$ and earth model $\mathbf{G}_k$.

A common location method is NonLinLoc by Anthony Lomax[36] which can produces samples from the location PDF to be used in the Monte Carlo method described above (see the location uncertainty tutorial in chapter 4).

The symbols used in these PDFs are:

Symbol	*Meaning*
Y	Polarity
A	Amplitude
σ_y	Uncertainty
ϖ	Mispick Probability
$\tilde{\mathbf{M}}$	Moment Tensor 6-vector
$\mathbf{M}$	Moment Tensor
$\mathbf{a}$	Receiver Ray Path Coefficients
σ	Measurment Uncertainty
R	Amplitude Ratio
A	Amplitude
ϵ	Nuisance Parameters
$\mathbf{k}$	Known Parameters
$\mathbf{t}$	Arrival times

[36] http://alomax.free.fr/nlloc

CHAPTER 7

Probability

`mtfit` has PDF s for two data types, with two different approaches to measuring the polarity.

However, it is possible to add PDF s for other data-types (see *Extending mtfit*)

7.1 Polarity PDF

Pugh et al, 2016a has a derivation of the polarity PDF used in `mtfit`. It is given by:

$$\mathrm{p}\left(Y=y\,|\,A,\sigma_y,\varpi\right)=\frac{1}{2}\left(1+\mathrm{erf}\left(\frac{yA}{\sqrt{2}\sigma_y}\right)\right)\left(1-\varpi\right)+\frac{1}{2}\left(1+\mathrm{erf}\left(\frac{-yA}{\sqrt{2}\sigma_y}\right)\right)\varpi$$

The different symbols in this equation are:

Symbol	*Meaning*
Y	Polarity
A	Amplitude
σ_y	Uncertainty
ϖ	Mispick Probability

and $\mathrm{erf}\left(x\right)$ is the error function, given by $\mathrm{erf}\left(x\right)=\frac{2}{\sqrt{\pi}}\int_0^x \mathrm{e}^{-t^2}\mathrm{d}t$.

This approach requires an estimate of the uncertainty σ_y. This is not the noise at the arrival, since it does not scale correctly in comparison to the modelled amplitude due to the propagation effects. It could be estimated from the fractional amplitude uncertainty, but this will be greater than or equal to the true value, because the amplitude at a receiver is only ever less than or equal to the maximum theoretical amplitude (accounting for propagation effects). Consequently, this would most likely overestimate the uncertainty. It is clear that the uncertainty value should be station-specific as noise environments at different stations often vary, so the maximum estimate of the event signal-to-noise ratio (SNR) fails to account for the variation across the receivers.

The difficulty in estimating σ_y is increased further when polarity picking is done manually, so the uncertainty on the trace is perhaps not even known. Due to the difficulty in quantifying the uncertainty, it is best left as a user-defined parameter that reflects the confidence in the arrival polarity pick, which can be mapped to the pick quality. However, *Pugh et al, 2016b* proposes an alternate method for calculating polarity uncertainties that can be included in this framework (see *Polarity Probability PDF*)

7.2 Polarity Probability PDF

Pugh et al, 2016b introduces an alternate method for estimating the polarity, using an automated Bayesian probability estimate. This approach results in estimates of the postive and negative polarity probabilities. `autopol` provides a Python module for calculating these values (*Pugh, 2016a*), and may be available on request. These observations can be included in `mtfit`, although the data independence must be preserved. The PDF is:

$$\mathrm{p}\left(\psi|A,\sigma,\tau,\sigma_\tau,\varpi\right)=1-\varpi+\left(2\varpi-1\right)\left[\mathrm{H}\left(A\right)+\psi-2\mathrm{H}\left(A\right)\varpi\right]$$

The different symbols in this equation are:

Symbol	*Meaning*
ψ	Polarity Probability
A	Amplitude
σ	Trace Noise
τ	Pick Time
σ_τ	Pick Time Noise
ϖ	Mispick Probability

and $\mathrm{H}\left(x\right)$ is the Heaviside step function, given by $\mathrm{H}\left(x\right)=\int_{-\infty}^{x}\delta\left(s\right)\mathrm{d}s$.

7.3 Amplitude Ratio PDF

The amplitude ratio PDF used in `mtfit` is based on the ratio PDF for two gaussian distributed variables (*Hinkley, 1969*):

$$P\left(r\right)=\frac{b\left(r\right)d\left(r\right)}{\sigma_x\sigma_y a^3\left(r\right)\sqrt{2\pi}}\left[\Phi\left(\frac{b\left(r\right)}{a\left(r\right)\sqrt{1-\rho^2}}\right)-\Phi\left(\frac{-b\left(r\right)}{a\left(r\right)\sqrt{1-\rho^2}}\right)\right]+\frac{\sqrt{1-\rho^2}}{\pi\sigma_x\sigma_y a^2\left(r\right)}e^{\left(-\frac{c}{2\left(1-\rho^2\right)}\right)}$$

With coefficients $a\left(r\right)$, $b\left(r\right)$, c and $d\left(r\right)$ given by :

$$\begin{aligned}a\left(r\right)&=\sqrt{\frac{r^2}{\sigma_x^2}-2\rho\frac{r}{\sigma_x\sigma_y}+\frac{1}{\sigma_y^2}}\\ b\left(r\right)&=\frac{\mu_x r}{\sigma_x^2}-\rho\frac{\mu_x+\mu_y r}{\sigma_x\sigma_y}+\frac{\mu_y}{\sigma_y^2}\\ c&=\frac{\mu_x^2}{\sigma_x^2}-2\rho\frac{\mu_x\mu_y}{\sigma_x\sigma_y}+\frac{\mu_y^2}{\sigma_y^2}\\ d\left(r\right)&=e^{\left(\frac{b^2\left(r\right)-ca^2\left(r\right)}{2\left(1-\rho^2\right)a^2\left(r\right)}\right)}\end{aligned}$$

The resultant PDF is (unsigned amplitude ratios):

$$P\left(R=r\,|\,A_x,A_y,\sigma_x,\sigma_y\right)=\mathcal{R}_{\mathcal{N}}\left(r,A_x,A_y,\sigma_x,\sigma_y\right)+\mathcal{R}_{\mathcal{N}}\left(-r,A_x,A_y,\sigma_x,\sigma_y\right)$$

With $\mathcal{R}_{\mathcal{N}}\left(r,\mu_x,\mu_y,\sigma_x,\sigma_y\right)$ referring to the ratio PDF above, since ρ, the correlation between the variables, is zero.

The different symbols in this equation are:

Symbol	*Meaning*
R	Amplitude Ratio
A	Amplitude
σ	Amplitude Noise

CHAPTER 8

mtfit.algorithms: Search Algorithms

These algorithms and their effects are explored in *Pugh (2015)*, which expands in further detail on the topics covered here.

8.1 Algorithms

This module contains the sampling approaches used. Two main approaches are used:

- Random Monte Carlo sampling
- Markov chain Monte Carlo sampling

However, there are also two variants of the Markov chain Monte Carlo (McMC) method:

- Metropolis-Hastings
- Trans-Dimensional Metropolis-Hastings (Reversible Jump)

8.2 Random Monte Carlo sampling

The simplest approach is that of random sampling over the moment tensor or double-couple space. Stochastic Monte Carlo sampling introduces no biases and provides an estimate for the true PDF, but requires a sufficient density of sampling to reduce the uncertainties in the estimate. The sampled PDF approaches the true distribution in the limit of infnite samples. However, this approach is limited both by time and memory. Some benefits can be gained by only keeping the samples with non-zero probability.

The assumed prior distribution is a uniform distribution on the unit 6-sphere in moment tensor space. This is equivalent to unit normalisation of the moment tensor six vector:

$$\tilde{\mathbf{M}} = \begin{pmatrix} M_{11} \\ M_{22} \\ M_{33} \\ \sqrt{2}M_{12} \\ \sqrt{2}M_{13} \\ \sqrt{2}M_{23} \end{pmatrix}.$$

This sampling is explored further in *Pugh et al, 2015t*.

8.3 Markov chain Monte Carlo sampling

An alternative approach is to use Markov chain Monte Carlo (McMC) sampling. This constructs a Markov chain (*Norris, 1998*) of which the equilibrium distribution is a good sample of the target probability distribution.

A Markov chain is a memoryless stochastic process of transitioning between states. The probability of the next value depends only on the current value, rather than all the previous values, which is known as the Markov property (*Markov, 1954*):

$$\mathrm{p}\left(d_n|d_{n-1}, d_{n-2}, d_{n-3}, \dots d_0\right) = \mathrm{p}\left(d_n|d_{n-1}\right).$$

A suitable McMC method should converge on the target distribution rapidly. As an approach it is more complex than the Monte Carlo random sampling approach described above, and by taking samples close to other non-zero samples, there is moreintelligence to the sampling than in the random Monte Carlo sampling approach.

A Metropolis-Hastings approach is used here (*Metropolis, 1953* and *Hastings, 1970*). The Metropolis-Hastings approach is a common method for McMC sampling and satisfies the detailed balance condition (Robert and Casella, 2004, eq. 6.22), which means that the probability density of the chain is stationary. New samples are drawn from a probability density $\mathrm{q}\left(\mathbf{x}'|\mathbf{x}_\mathrm{t}\right)$ to evaluate the target probability density $\mathrm{p}\left(\mathbf{x}|\mathbf{d}\right)$.

The Metropolis-Hastings algorithm begins with a random starting point and then iterates until this initial state is forgotten (Algorithm). Each iteration evaluates whether a new proposed state is accepted or not. If $\mathrm{q}\left(\mathbf{x}'|\mathbf{x}_\mathrm{t}\right)$ is symmetric, then the ratio $\frac{\mathrm{q}\left(\mathbf{x}_\mathrm{t}|\mathbf{x}'\right)}{\mathrm{q}\left(\mathbf{x}'|\mathbf{x}_\mathrm{t}\right)} = 1$. The acceptance, alpha, is given by

$$\alpha = \min\left(1, \frac{\mathrm{p}\left(\mathbf{x}'|\mathbf{d}\right)}{\mathrm{p}\left(\mathbf{x}_\mathrm{t}|\mathbf{d}\right)}.\frac{\mathrm{q}\left(\mathbf{x}_\mathrm{t}|\mathbf{x}'\right)}{\mathrm{q}\left(\mathbf{x}'|\mathbf{x}_\mathrm{t}\right)}\right),$$

which can be expanded using *Bayes' Theorem* to give the acceptance in terms of the likelihood $\mathrm{p}\left(\mathbf{d}|\mathbf{x}\right)$ and prior $\mathrm{p}\left(\mathbf{x}\right)$,

$$\alpha = \min\left(1, \frac{\mathrm{p}\left(\mathbf{d}|\mathbf{x}'\right)\mathrm{p}\left(\mathbf{x}'\right)}{\mathrm{p}\left(\mathbf{d}|\mathbf{x}_\mathrm{t}\right)\mathrm{p}\left(\mathbf{x}_\mathrm{t}\right)}.\frac{\mathrm{q}\left(\mathbf{x}_t|\mathbf{x}'\right)}{\mathrm{q}\left(\mathbf{x}'|\mathbf{x}_\mathrm{t}\right)}\right).$$

The acceptance is the probability that the new sample in the chain, $\mathbf{x}_{\mathrm{t}+1}$ is the new sample, $\mathbf{x}'$, otherwise the original value, $\mathbf{x}_\mathrm{t}$, is added to the chain again,

$$\mathbf{x_{t+1}} = \begin{cases} \mathbf{x}' & probability = \alpha \\ \mathbf{x}_t & probability = 1 - \alpha \end{cases}.$$

The algorithm used in `mtfit` is:

Metropolis-Hastings Markov chain Monte Carlo Sampling Algorithm	
1	Determine initial value for $\mathbf{x}_0$ with non-zero likelihood
2	Draw new sample $\mathbf{x}'$ from transition PDF $\mathrm{q}\left(\mathbf{x}'|\mathbf{x}_\mathrm{t}\right)$
3	Evaluate likelihood for sample $\mathbf{x}'$
4	Calculate acceptance, α, for $\mathbf{x}'$.
5	Determine sample $\mathbf{x}_{\mathrm{t}+1}$.
6(a)	If in learning period:
	i If sufficient samples (> 100) have been obtained, update transition PDF parameters to target ideal acceptance rate.
	ii Return to 2 until end of learning period and discard learning samples.
6(b)	Otherwise return to 2 until sufficient samples are drawn.

The source parameterisation is from *Tape and Tape (2012)*, and the algorithm uses an iterative parameterisation for the learning parameters during a learning period, then generates a Markov chain from the pdf.

8.3.1 Reversible Jump Markov chain Monte Carlo Sampling

The Metropolis-Hastings approach does not account for variable dimension models. *Green (1995)* introduced a new type of move, a *jump*, extending the approach to variable dimension problems. The jump introduces a dimension-balancing vector, so it can be evaluated like the normal Metropolis-Hastings shift.

Green (1995) showed that the acceptance for a pair of models M_t and M' is given by:

$$\alpha = \min\left(1, \frac{\mathrm{p}\left(\mathbf{d}|\mathbf{x}', M'\right)\mathrm{p}\left(\mathbf{x}'|M'\right)\mathrm{p}\left(M'\right)}{\mathrm{p}\left(\mathbf{d}|\mathbf{x}_\mathrm{t}, M_\mathrm{t}\right)\mathrm{p}\left(\mathbf{x}_\mathrm{t}|M_\mathrm{t}\right)\mathrm{p}\left(M_\mathrm{t}\right)}.\frac{\mathrm{q}\left(\mathbf{x}_\mathrm{t}|\mathbf{x}'\right)}{\mathrm{q}\left(\mathbf{x}'|\mathbf{x}_\mathrm{t}\right)}\right),$$

where $\mathrm{q}\left(\mathbf{x}'|\mathbf{x}_\mathrm{t}\right)$ is the probability of making the transition from parameters $\mathbf{x}_\mathrm{t}$ from model M_t to parameters $\mathbf{x}'$ from model M', and $\mathrm{p}\left(M_\mathrm{t}\right)$ is the prior for the model M_t.

If the models M_t and M' are the same, the reversible jump acceptance is the same as the Metropolis-Hastings acceptance, because the model priors are the same. The importance of the reversible jump approach is that it allows a transformation between different models, and even different dimensions.

The dimension balancing vector requires a bijection between the parameters of the two models, so that the transformation is not singular and a reverse jump can occur. In the case where $\dim\left(M'\right) > \dim\left(M_\mathrm{t}\right)$, a vector $\mathbf{u}$ of length $\dim\left(M'\right) - \dim\left(M_\mathrm{t}\right)$ needs to be introduced to balance the number of parameters in $\mathbf{x}'$. The values of $\mathbf{u}$ have probability density $\mathrm{q}\left(\mathbf{u}\right)$ and some bijection that maps $\mathbf{x}_\mathrm{t}, \mathbf{u} \rightarrow \mathbf{x}', \mathbf{x}' = \mathrm{h}\left(\mathbf{x}_\mathrm{t}, \mathbf{u}\right)$.

If the jump has a probability $\mathrm{j}\left(\mathbf{x}\right)$ of occurring for a sample $\mathbf{x}$, the transition probability depends on the jump parameters, $\mathbf{u}$ and the transition ratio is given by:

$$\frac{\mathrm{q}\left(\mathbf{x}_\mathrm{t}, \mathbf{u}|\mathbf{x}'\right)}{\mathrm{q}\left(\mathbf{x}'|\mathbf{x}_\mathrm{t}, \mathbf{u}\right)} = \frac{\mathrm{j}\left(\mathbf{x}'\right)}{\mathrm{j}\left(\mathbf{x}_\mathrm{t}\right)\mathrm{q}\left(\mathbf{u}\right)}\left|\mathbf{J}\right|,$$

with the Jacobian matrix, $\mathbf{J} = \frac{\partial \mathrm{h}\left(\mathbf{x}_\mathrm{t}, \mathbf{u}\right)}{\partial\left(\mathbf{x}_\mathrm{t}, \mathbf{u}\right)}$.

The general form of the jump acceptance involves a prior on the models, along with a prior on the parameters. The acceptance for this case is:

$$\alpha = \min\left(1, \frac{\mathrm{p}\left(\mathbf{d}|\mathbf{x}', M'\right)\mathrm{p}\left(\mathbf{x}'|M'\right)\mathrm{p}\left(M'\right)}{\mathrm{p}\left(\mathbf{d}|\mathbf{x}_\mathrm{t}, M_\mathrm{t}\right)\mathrm{p}\left(\mathbf{x}_\mathrm{t}|M_\mathrm{t}\right)\mathrm{p}\left(M_\mathrm{t}\right)}.\frac{\mathrm{j}\left(\mathbf{x}'\right)}{\mathrm{j}\left(\mathbf{x}_\mathrm{t}\right)\mathrm{q}\left(\mathbf{u}\right)}\left|\mathbf{J}\right|\right).$$

If the jump is from higher dimensions to lower, the bijection describing the transformation has an inverse that describes the transformation $\mathbf{x}' \rightarrow \mathbf{x}_\mathrm{t}, \mathbf{u}$, and the acceptance is given by:

$$\alpha = \min\left(1, \frac{\mathrm{p}\left(\mathbf{d}|\mathbf{x}_\mathrm{t}, M_\mathrm{t}\right)\mathrm{p}\left(\mathbf{x}_\mathrm{t}|M_\mathrm{t}\right)\mathrm{p}\left(M_\mathrm{t}\right)}{\mathrm{p}\left(\mathbf{d}|\mathbf{x}', M'\right)\mathrm{p}\left(\mathbf{x}'|M'\right)\mathrm{p}\left(M'\right)}.\frac{\mathrm{j}\left(\mathbf{x}_\mathrm{t}\right)\mathrm{q}\left(\mathbf{u}\right)}{\mathrm{j}\left(\mathbf{x}'\right)}\left|\mathbf{J}^{-1}\right|\right).$$

A simple example used in the literature (see Green, 1995; Brooks et al., 2003) is a mapping from a one dimensional model with parameter θ to a two dimensional model with parameters θ_1, θ_2. A possible bijection is given by:

$$h\left(\theta, u\right) = \begin{cases}\theta_1 & = \theta - u \\ \theta_2 & = \theta + u\end{cases}$$

with the reverse bijection given by:

$$h\left(\theta_1, \theta_2\right) = \begin{cases}\theta & = \frac{1}{2}\left(\theta_1 + \theta_2\right) \\ u & = \frac{1}{2}\left(\theta_1 - \theta_2\right)\end{cases}$$

Reversible Jump McMC in Source Inversion

The reversible jump McMC approach allows switching between different source models, such as the double-couple model and the higher dimensional model of the full moment tensor, and can be extended to other source models.

The full moment tensor model is nested around the double-couple point, leading to a simple description of the jumps by keeping the common parameters constant. The Tape parameterisation (*Tape and Tape, 2012*) allows for easy movement both in the source space and between models. The moment tensor model has five parameters:

- strike (κ)

- dip cosine (h)
- slip (σ)
- eigenvalue co-latitude (δ)
- eigenvalue longitude (γ)

while the double-couple model has only the three orientation parameters: κ, h, and σ. Consequently, the orientation parameters are left unchanged between the two models, and the dimension balancing vector for the jump has two elements, which can be mapped to γ and δ:

$$\mathrm{h}\left(\kappa_{\mathrm{DC}}, h_{\mathrm{DC}}, \sigma_{\mathrm{DC}}, \mathbf{u}\right) \begin{cases} \kappa_{\mathrm{MT}} & = \kappa_{\mathrm{DC}} \\ h_{\mathrm{MT}} & = h_{\mathrm{DC}} \\ \sigma_{\mathrm{MT}} & = \sigma_{\mathrm{DC}} \\ \gamma_{\mathrm{MT}} & = u_1 \\ \delta_{\mathrm{MT}} & = u_2 \end{cases}.$$

The algorithm used is:

Reversible Jump Markov chain Monte Carlo sampling Algorithm	
1	Determine initial value for $\mathbf{x}_0$ with non-zero likelihood
2(a)	Carry out jump with probability p_{jump}: Draw new sample $\mathbf{x}' = (\mathbf{x}, \mathbf{u})$ with dimension balancing vector $\mathbf{u}$ drawn from $\mathrm{q}\left(u1, u2\right)$.
2(b)	Carry out shift with probability $1 - p_{\mathrm{jump}}$: Draw new sample $\mathbf{x}'$ from transition PDF $\mathrm{q}\left(\mathbf{x}'|\mathbf{x}_{\mathrm{t}}\right)$.
3	Evaluate likelihood for sample $\mathbf{x}'$
4	Calculate acceptance, α, for $\mathbf{x}'$.
5	Determine sample $\mathbf{x}_{\mathrm{t}+1}$.
6(a)	If in learning period:
	i If sufficient samples (> 100) have been obtained, update transition PDF parameters to target ideal acceptance rate.
	ii Return to 2 until end of learning period and discard learning samples.
6(b)	Otherwise return to 2 until sufficient samples are drawn.

8.4 Relative Amplitude

Inverting for multiple events increases the dimensions of the source space for each event. This leads to a much reduced probability of obtaining a non-zero likelihood sample, because sampling from the n-event distribution leads to multiplying the probabilities of drawing a non-zero samples, resulting in sparser sampling of the joint source PDF.

The elapsed time for the random sampling is longer per sample than the individual sampling, and longer than the combined sampling for both events due to evaluating the relative amplitude PDF. Moreover, increasing the number of samples 10-fold raises the required time by a factor of 10, requiring some method of reducing the running time for the inversion, since, given current processor speeds, 10^{15} samples would take many years to calculate on a single core. As a result, more intelligent search algorithms are required for the full moment tensor case.

Markov chain approaches are less dependent on the model dimensionality. To account for the fact that the uncertainties in each parameter can differ between the events, the Markov chain shape parameters can be scaled based on the relative non-zero percentages of the events when they are initialised. The initialisation approaches also need to be adjusted to account for the reduced non-zero sample probability, such as by initialising the Markov chain independently for each event. The trans-dimensional McMC algorithm allows model jumping independently for each event.

Tuning the Markov chain acceptance rate is difficult, as it is extremely sensitive to small changes in the proposal distribution widths, and with the higher dimensionality it may be necessary to lower the targeted acceptance rate

to improve sampling. Consequently, care needs to be taken when tuning the parameters to effectively implement the approaches for relative amplitude data.

8.5 Running Time

Comparing different sample sizes shows that the McMC approaches require far fewer samples than random sampling. However, the random sampling algorithm is quick to calculate the likelihood for a large number of samples, unlike the McMC approach, because of the extra computations in calculating the acceptance and obtaining new samples. Some optimisations have been included in the McMC algorithms, including calculating the probability for multiple new samples at once, with sufficient samples that there is a high probability of containing an accepted sample. This is more efficient than repeatedly updating the algorithm.

Despite these optimisations, the McMC approach is still much slower to reach comparable sample sizes, and is slower than would be expected just given the targeted acceptance rate, because of the additional computational overheads (Fig. :ref:‘8.1<algorithm-run-time>).

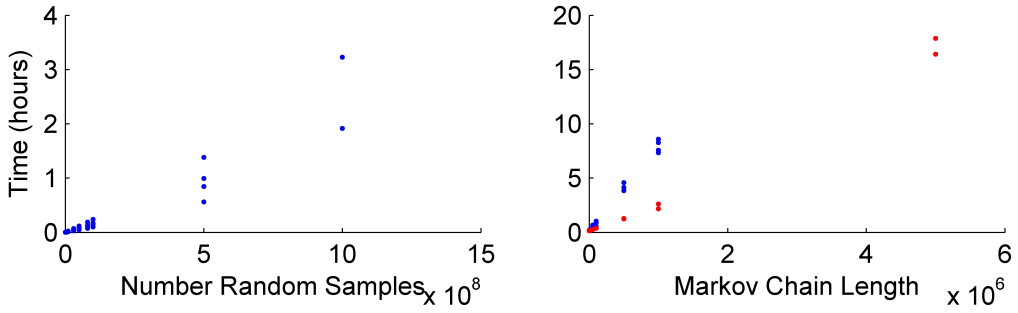

Fig. 8.1: *Elapsed Time for different sample sizes of the random sampling algorithm and for McMC algorithms with different number of unique samples.*

Including location uncertainty and model uncertainty in the forward model causes a rapid reduction of the available samples for a given amount of RAM and increases the number of times the forward model must be evaluated, lengthening the time for sufficient sampling (Fig. *8.2*).

The location uncertainty has less of an effect on the McMC algorithms, since the number of samples being tested at any given iteration are small. Consequently, as the random sampling approach becomes slower, the fewer samples required to construct the Markov chain starts to produce good samples of the source PDF at comparable times. However, there is an initial offset in the elapsed time for the Markov chain Monte Carlo approaches due to the burn in and initialisation of the algorithm.

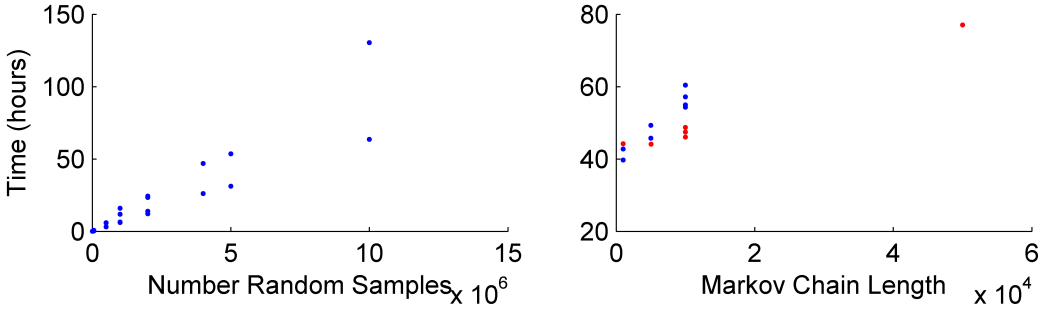

Fig. 8.2: *Elapsed Time for different sample sizes of the random sampling algorithm and for McMC algorithms with different number of unique samples. The velocity model and location uncertainty in the source was included with a one degree binning reducing the number of location samples from 50,000 to 5, 463.*

The relative amplitude algorithms require an exponential increase in the number of samples as the number of events being sampled increases (Fig. *8.3*). However, the McMC approaches are more intelligent and do not require the same increase in sample size, but these algorithms can prove difficult to estimate the appropriate sampling parameters for the proposal distribution.

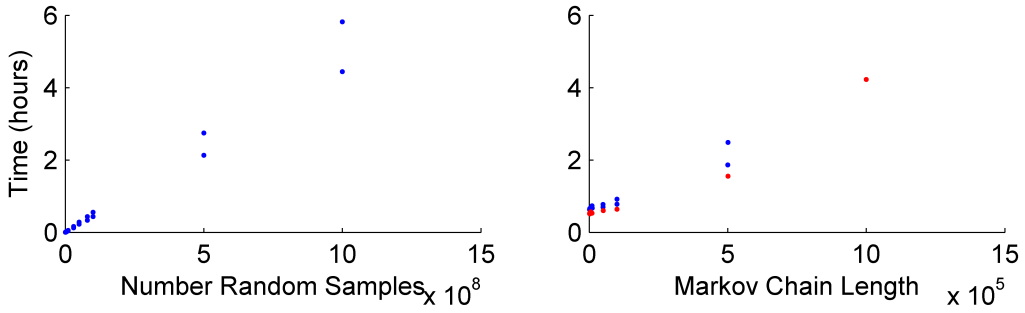

Fig. 8.3: *Elapsed Time for different sample sizes of the random sampling algorithm and for McMC algorithms with different number of unique samples for a two event joint PDF with relative P-amplitudes.*

8.6 Summary

There are several different approaches to sampling for the forward model. The Monte Carlo random sampling is quick, although it requires a large number of samples to produce good samplings of the PDF. The McMC approaches produce good sampling of the source for far fewer samples, but the evaluation time can be long due to the overhead of calculating the acceptance and generating new samples. Furthermore, these approaches rely on achieving the desired acceptance rate to sufficiently sample the PDF given the chain length. Including location uncertainty increases the random sampling time drastically, and has less of an effect on the McMC approaches.

The solutions for the different algorithms and different source types are consistent, but the McMC approach can be poor at sampling multi-modal distributions if the acceptance rate is too high.

The two different approaches for estimating the probability of the double-couple source type model being correct are consistent, apart from cases where the PDF is dominated by a few very high probability samples. For these solutions, the number of random samples needs to be increased sufficiently to estimate the probability well. The McMC approach cannot easily be used to estimate the double-couple model probabilities, unlike both the random sampling and trans-dimensional approaches.

Extending these models to relative source inversion leads to a drastically increased number of samples required for the Monte Carlo random sampling approach that it becomes infeasible with current computation approaches. However, the McMC approaches can surmount this problem due to the improved sampling approach, but care must be taken when initialising the algorithm to make sure that sufficient samples are discarded that any initial bias has been removed, as well as targeting an acceptance rate that does not require extremely large chain lengths to successfully explore the full space.

The required number of samples depends on each event, and the constraint on the source that is given by the data. Usually including more data-types can sharpen the PDF, requiring more samples to get a satisfactory sampling of the event.

CHAPTER 9

convert

`mtfit` contains a submodule `mtfit.convert` that can handle moment tensor conversions. Mostly it is used with the `--convert` flag for converting the moment tensor results to the different source parameterisations. However it can be used separately, and the functions are described here.

9.1 moment_tensor_conversion.py

Module containing moment tensor conversion functions. Acts on the parameters of the moment tensor 3x3 form or the modified 6-vector form, dependent on the name

The function naming is OriginalVariables_NewVariables

The coordinate system is North (X), East (Y), Down (Z)

`mtfit.convert.moment_tensor_conversion.`**`E_GD`**(*E*)
Convert the eigenvalues to the Tape parameterisation gamma and delta.

Args E: array of eigenvalues

Returns (numpy.array, numpy.array): tuple of gamma, delta

`mtfit.convert.moment_tensor_conversion.`**`E_tk`**(*E*)
Convert the moment tensor eigenvalues to the Hudson tau, k parameters

Args E: indexable list/array (e.g numpy.array) of moment tensor eigenvalues

Returns (float, float): tau, k tuple

`mtfit.convert.moment_tensor_conversion.`**`E_uv`**(*E*)
Convert the eigenvalues to the Hudson i, v parameters

Args E: indexable list/array (e.g numpy.array) of moment tensor eigenvalues

Returns (float, float): u, v tuple

`mtfit.convert.moment_tensor_conversion.`**`FP_SDR`**(*normal*, *slip*)
Convert fault normal and slip to strike, dip and rake

Coordinate system is North East Down.

Args normal: numpy matrix - Normal vector slip: numpy matrix - Slip vector

Returns (float, float, float): tuple of strike, dip and rake angles in radians

mtfit.convert.moment_tensor_conversion.**FP_SDSD**(*N1*, *N2*)
Convert the the fault normal and slip vectors to the strike and dip pairs (all angles in radians).

Args Normal: numpy matrix - Normal vector Slip: numpy matrix - Slip vector

Returns

(float, float, float, float): tuple of strike1, dip1, strike2, dip2 angles in radians

mtfit.convert.moment_tensor_conversion.**FP_TNP**(*normal*, *slip*)
Convert fault normal and slip to TNP axes

Coordinate system is North East Down.

Args normal: numpy matrix - normal vector slip: numpy matrix - slip vector

Returns (numpy.matrix, numpy.matrix, numpy.matrix): tuple of T, N, P vectors

mtfit.convert.moment_tensor_conversion.**GD_E**(*gamma*, *delta*)
Convert the Tape parameterisation gamma and delta to the eigenvalues.

Args gamma: numpy array of gamma values delta: numpy array of delta values

Returns numpy.array: array of eigenvalues

mtfit.convert.moment_tensor_conversion.**GD_basic_cdc**(*gamma*, *delta*)
Convert gamma, delta to basic crack+double-couple parameters

Gamma and delta are the source type parameters from the Tape parameterisation.

Args gamma: numpy array of gamma values delta: numpy array of delta values

Returns tuple of alpha, poisson

Return type (numpy.array, numpy.array)

mtfit.convert.moment_tensor_conversion.**MT33_GD**(*MT33*)
Convert the 3x3 Moment Tensor to theTape parameterisation gamma and delta.

Args MT33: 3x3 numpy matrix

Returns (numpy.array, numpy.array): tuple of gamma, delta

mtfit.convert.moment_tensor_conversion.**MT33_MT6**(*MT33*)
Convert a 3x3 matrix to six vector maintaining normalisation. 6-vector has the form:

```
Mxx
Myy
Mzz
sqrt(2)*Mxy
sqrt(2)*Mxz
sqrt(2)*Myz
```

Args M33: 3x3 numpy matrix

Returns numpy.matrix: MT 6-vector

mtfit.convert.moment_tensor_conversion.**MT33_SDR**(*MT33*)
Convert the 3x3 Moment Tensor to the strike, dip and rake.

Args MT33: 3x3 numpy matrix

Returns (float, float, float): tuple of strike, dip, rake angles in radians

mtfit.convert.moment_tensor_conversion.**MT33_TNPE**(*MT33*)
Convert the 3x3 Moment Tensor to the T,N,P vectors and the eigenvalues.

Args MT33: 3x3 numpy matrix

Returns

(numpy.matrix, numpy.matrix, numpy.matrix, numpy.array): tuple of T, N, P vectors and Eigenvalue array

mtfit.convert.moment_tensor_conversion.**MT6_MT33**(*MT6*)

Convert a six vector to a 3x3 MT maintaining normalisation. 6-vector has the form:

```
Mxx
Myy
Mzz
sqrt(2)*Mxy
sqrt(2)*Mxz
sqrt(2)*Myz
```

Args MT6: numpy matrix Moment tensor 6-vector

Returns numpy.matrix: 3x3 Moment Tensor

mtfit.convert.moment_tensor_conversion.**MT6_TNPE**(*MT6*)

Convert the 6xn Moment Tensor to the T,N,P vectors and the eigenvalues.

Args MT6: 6xn numpy matrix

Returns

(numpy.matrix, numpy.matrix, numpy.matrix, numpy.array): tuple of T, N, P vectors and Eigenvalue array

mtfit.convert.moment_tensor_conversion.**MT6_Tape**(*MT6*)

Convert the moment tensor 6-vector to the Tape parameters.

6-vector has the form:

```
Mxx
Myy
Mzz
sqrt(2)*Mxy
sqrt(2)*Mxz
sqrt(2)*Myz
```

Args MT6: numpy matrix six-vector

Returns

(numpy.array, numpy.array, numpy.array, numpy.array, numpy.array): tuple of gamma, delta, strike, cos(dip) and slip (angles in radians)

mtfit.convert.moment_tensor_conversion.**MT6_biaxes**(*MT6, c=[3.0, 1.0, 1.0, 0, 0, 0, 3.0, 1.0, 0, 0, 0, 3.0, 0, 0, 0, 1.0, 0, 0, 1.0, 0, 1.0]*)

Convert six vector to bi-axes

Convert the moment tensor 6-vector to the bi-axes decomposition from Chapman, C and Leaney,S, 2011. A new moment-tensor decomposition for seismic events in anisotropic media, GJI, 188(1), 343-370. The 6-vector has the form:

```
Mxx
Myy
Mzz
sqrt(2)*Mxy
sqrt(2)*Mxz
sqrt(2)*Myz
```

The stiffness tensor can be provided as an input, as a list of the 21 elements of the upper triangular part of the Voigt stiffness matrix:

```
C(21) = ( C_11, C_12, C_13, C_14, C_15, C_16,
                C_22, C_23, C_24, C_25, C_26,
                      C_33, C_34, C_35, C_36,
                            C_44, C_45, C_46,
                                  C_55, C_56,
                                        C_66 )

         (upper triangular part of Voigt stiffness matrix)
```

Alternatively, the default isotropic parameters can be used or a possible isotropic stiffness tensor can be genereated using (isotropic_c)

Args MT6: numpy matrix Moment tensor 6-vector c: list or numpy array of the 21 element input stiffness tensor

Returns

(numpy.array, numpy.array, numpy.array): tuple of phi (bi-axes) vectors, explosion value and area_displacement value.

mtfit.convert.moment_tensor_conversion.**MT6c_D6**(*MT6, c=[3.0, 1.0, 1.0, 0, 0, 0, 3.0, 1.0, 0, 0, 0, 3.0, 0, 0, 0, 1.0, 0, 0, 1.0, 0, 1.0]*)

Convert the moment tensor 6-vector to the potency tensor. The 6-vector has the form:

```
Mxx
Myy
Mzz
sqrt(2)*Mxy
sqrt(2)*Mxz
sqrt(2)*Myz
```

The stiffness tensor can be provided as an input, as a list of the 21 elements of the upper triangular part of the Voigt stiffness matrix:

```
C(21) = ( C_11, C_12, C_13, C_14, C_15, C_16,
                C_22, C_23, C_24, C_25, C_26,
                      C_33, C_34, C_35, C_36,
                            C_44, C_45, C_46,
                                  C_55, C_56,
                                        C_66 )

         (upper triangular part of Voigt stiffness matrix)
```

Alternatively, the default isotropic parameters can be used or a possible isotropic stiffness tensor can be genereated using (isotropic_c).

Args MT6: numpy matrix Moment tensor 6-vector c: list or numpy array of the 21 element input stiffness tensor

Returns

numpy.array: numpy array of the potency 6 vector (in the same ordering as the moment tensor six vector)

mtfit.convert.moment_tensor_conversion.**SDR_FP**(*strike*, *dip*, *rake*)

Convert the strike, dip and rake in radians to the fault normal and slip.

Args strike: float strike angle of fault plane in radians dip: float dip angle of fault plane in radians rake: float rake angle of fault plane in radians

Returns (numpy.matrix, numpy.matrix): tuple of Normal and slip vectors

mtfit.convert.moment_tensor_conversion.**SDR_SDR**(*strike*, *dip*, *rake*)
Convert strike, dip rake to strike, dip rake for other fault plane

Coordinate system is North East Down.

Args strike: float radians dip: float radians rake: float radians

Returns

(float, float, float): tuple of strike, dip and rake angles of alternate fault plane in radians

mtfit.convert.moment_tensor_conversion.**SDR_SDSD**(*strike*, *dip*, *rake*)
Convert the strike, dip and rake to the strike and dip pairs (all angles in radians).

Args strike: float strike angle of fault plane in radians dip: float dip angle of fault plane in radians rake: float rake angle of fault plane in radians

Returns (float, float, float, float): tuple of strike1, dip1, strike2, dip2 angles in radians

mtfit.convert.moment_tensor_conversion.**SDR_TNP**(*strike*, *dip*, *rake*)
Convert strike, dip rake to TNP vectors

Coordinate system is North East Down.

Args strike: float radians dip: float radians rake: float radians

Returns (numpy.matrix, numpy.matrix, numpy.matrix): tuple of T,N,P vectors.

mtfit.convert.moment_tensor_conversion.**SDSD_FP**(*strike1*, *dip1*, *strike2*, *dip2*)
Convert strike and dip pairs to fault normal and slip

Converts the strike and dip pairs in radians to the fault normal and slip.

Args strike1: float strike angle of fault plane 1 in radians dip1: float dip angle of fault plane 1 in radians strike2: float strike angle of fault plane 2 in radians dip2: float dip of fault plane 2 in radians

Returns (numpy.matrix, numpy.matrix): tuple of Normal and slip vectors

mtfit.convert.moment_tensor_conversion.**TNP_SDR**(*T*, *N*, *P*)
Convert the T,N,P vectors to the strike, dip and rake in radians

Args T: numpy matrix of T vectors. N: numpy matrix of N vectors. P: numpy matrix of P vectors.

Returns (float, float, float): tuple of strike, dip and rake angles of fault plane in radians

mtfit.convert.moment_tensor_conversion.**TP_FP**(*T*, *P*)
Convert the 3x3 Moment Tensor to the fault normal and slip vectors.

Args T: numpy matrix of T vectors. P: numpy matrix of P vectors.

Returns (numpy.matrix, numpy.matrix): tuple of Normal and slip vectors

mtfit.convert.moment_tensor_conversion.**Tape_MT33**(*gamma*, *delta*, *kappa*, *h*, *sigma*, ***kwargs*)
Convert Tape parameters to a 3x3 moment tensor

Args

gamma: float, gamma parameter (longitude on funamental lune takes values between -pi/6 and pi/6).

delta: float, delta parameter (latitude on funamental lune takes values between -pi/2 and pi/2)

kappa: float, strike (takes values between 0 and 2*pi) h: float, cos(dip) (takes values between 0 and 1) sigma: float, slip angle (takes values between -pi/2 and pi/2)

Returns numpy.matrix: 3x3 moment tensor

mtfit.convert.moment_tensor_conversion.**Tape_MT6**(*gamma*, *delta*, *kappa*, *h*, *sigma*)
Convert the Tape parameterisation to the moment tensor six-vectors.

Args

gamma: Gamma parameter (longitude on funamental lune takes values between -pi/6 and pi/6).

delta: Delta parameter (latitude on funamental lune takes values between -pi/2 and pi/2)

kappa: Strike (takes values between 0 and 2*pi) h: Cos(dip) (takes values between 0 and 1) sigma: Slip angle (takes values between -pi/2 and pi/2)

Returns np.array: Array of MT 6-vectors

mtfit.convert.moment_tensor_conversion.**Tape_TNPE**(*gamma*, *delta*, *kappa*, *h*, *sigma*)
Convert the Tape parameterisation to the T,N,P vectors and the eigenvalues.

Args

gamma: Gamma parameter (longitude on funamental lune takes values between -pi/6 and pi/6).

delta: Delta parameter (latitude on funamental lune takes values between -pi/2 and pi/2)

kappa: Strike (takes values between 0 and 2*pi) h: Cos(dip) (takes values between 0 and 1) sigma: Slip angle (takes values between -pi/2 and pi/2)

Returns

(numpy.matrix, numpy.matrix, numpy.matrix, numpy.array): T,N,P vectors and Eigenvalues tuple

mtfit.convert.moment_tensor_conversion.**basic_cdc_GD**(*alpha*, *poisson=0.25*)
Convert alpha and poisson ratio to gamma and delta

alpha is opening angle, poisson : ratio lambda/(2(lambda+mu)) Defaults to 0.25. Uses basic crack+double-couple model of Minson et al (Seismically and geodetically determined nondouble-couple source mechanisms from the 2000 Miyakejima volcanic earthquake swarm, Minson et al, 2007, JGR 112) and Tape and Tape (The classical model for moment tensors, Tape and Tape, 2013, GJI)

Args alpha: Opening angle in radians (between 0 and pi/2) poisson:[0.25] Poisson ratio on the fault surface.

Returns tuple of gamma, delta

Return type (numpy.array, numpy.array)

mtfit.convert.moment_tensor_conversion.**c21_cvoigt**(*c*)
Convert an input stiffness tensor to voigt form

The stiffness tensor needs to be provided as a list of the 21 elements of the upper triangular part of the Voigt stiffness matrix:

```
C(21) = ( C_11, C_12, C_13, C_14, C_15, C_16,
                C_22, C_23, C_24, C_25, C_26,
                      C_33, C_34, C_35, C_36,
                            C_44, C_45, C_46,
                                  C_55, C_56,
                                        C_66 )

         (upper triangular part of Voigt stiffness matrix)
```

Args c: input list of stiffness parameters (21 required)

Returns numpy.array: voigt form of the stiffness tensor

mtfit.convert.moment_tensor_conversion.**c_norm**(*c*)
Calculate norm of the stiffness tensor.

The stiffness tensor needs to be provided as a list of the 21 elements of the upper triangular part of the Voigt stiffness matrix:

```
C(21) = ( C_11, C_12, C_13, C_14, C_15, C_16,
                C_22, C_23, C_24, C_25, C_26,
                      C_33, C_34, C_35, C_36,
                            C_44, C_45, C_46,
                                  C_55, C_56,
                                        C_66 )

         (upper triangular part of Voigt stiffness matrix)
```

Args c: input list of stiffness parameters (21 required)

Returns float: Euclidean norm of the tensor

mtfit.convert.moment_tensor_conversion.**is_isotropic_c**(*c*)

Evaluate if an input stiffness tensor is isotropic

The stiffness tensor needs to be provided as a list of the 21 elements of the upper triangular part of the Voigt stiffness matrix:

```
C(21) = ( C_11, C_12, C_13, C_14, C_15, C_16,
                C_22, C_23, C_24, C_25, C_26,
                      C_33, C_34, C_35, C_36,
                            C_44, C_45, C_46,
                                  C_55, C_56,
                                        C_66 )

         (upper triangular part of Voigt stiffness matrix)
```

Args c: input list of stiffness parameters (21 required)

Returns bool: result of is_isotropic check

mtfit.convert.moment_tensor_conversion.**isotropic_c**(*lambda_=1*, *mu=1*, *c=False*)

Calculate the isotropic stiffness tensor

Calculate isotropic stiffness parameters. The input parameters are either the two lame parameters lambda and mu, or is a full 21 element stiffness tensor:

```
C(21) = ( C_11, C_12, C_13, C_14, C_15, C_16,
                C_22, C_23, C_24, C_25, C_26,
                      C_33, C_34, C_35, C_36,
                            C_44, C_45, C_46,
                                  C_55, C_56,
                                        C_66 )

         (upper triangular part of Voigt stiffness matrix)
```

If the full stiffness tensor is used, the "average" isotropic approximation is calculated using Eqns 81a and 81b from Chapman, C and Leaney,S, 2011. A new moment-tensor decomposition for seismic events in anisotropic media, GJI, 188(1), 343-370.

Args **lambda_**: lambda value mu: mu value c: list or numpy array of the 21 element input stiffness tensor (overrides

lambda and mu arguments)

Returns list: list of 21 elements of the stiffness tensor

mtfit.convert.moment_tensor_conversion.**normal_SD**(*normal*)

Convert a plane normal to strike and dip

Coordinate system is North East Down.

Args normal: numpy matrix - Normal vector

Returns (float, float): tuple of strike and dip angles in radians

mtfit.convert.moment_tensor_conversion.**output_convert**(*mts*)
Convert the moment tensors into several different parameterisations

The moment tensor six-vectors are converted into the Tape gamma,delta,kappa,h,sigma parameterisation; the Hudson u,v parameterisation; and the strike, dip and rakes of the two fault planes are calculated.

Args mts: numpy array of moment tensor six-vectors

Returns dict: dictionary of numpy arrays for each parameter

mtfit.convert.moment_tensor_conversion.**tk_uv**(*tau*, *k*)
Convert the Hudson tau, k parameters to the Hudson u, v parameters

Args tau: float, Hudson tau parameter k: float, Hudson k parameter

Returns (float, float): u, v tuple

mtfit.convert.moment_tensor_conversion.**toa_vec**(*azimuth*, *plunge*, *radians=False*)
Convert the azimuth and plunge of a vector to a cartesian description of the vector

Args azimuth: float, vector azimuth plunge: float, vector plunge

Keyword Arguments radians: boolean, flag to use radians [default = False]

Returns np.matrix: vector

CHAPTER 10

mtfit command line options

Command line usage:

```
mtfit [-h] [-d DATAFILE] [-s LOCATION_PDF_FILE_PATH]
[-a {iterate,time,mcmc,transdmcmc}] [-l] [-n N] [-m MEM] [-c]
[-b] [--nstations NUMBER_STATIONS]
[--nanglesamples NUMBER_LOCATION_SAMPLES] [-f] [--not_file_safe]
[-i INVERSION_OPTIONS] [-o FID] [-x MAX_SAMPLES] [-t MAX_TIME]
[-e] [-r] [--marginalise_relative] [-R] [--invext DATA_EXTENSION]
[--angleext ANGLE_EXTENSION] [-S MINIMUM_NUMBER_INTERSECTIONS]
[-M] [-B] [-X MIN_NUMBER_INITIALISATION_SAMPLES] [-T]
[-Q [QUALITY_CHECK]] [-D] [-V VERBOSITY] [-g]
[-j DIMENSION_JUMP_PROB] [-y {grid}] [-u MIN_ACCEPTANCE_RATE]
[-v MAX_ACCEPTANCE_RATE] [-w ACCEPTANCE_RATE_WINDOW]
[-W WARNINGS] [-z LEARNING_LENGTH] [--version] [--mpi_call]
[--output-format {hyp,pickle,matlab}]
[--results-format {hyp,full_pdf}] [--no-dist]
[--dc-prior DC_PRIOR] [--sampling SAMPLING]
[--sample-models SAMPLE_DISTRIBUTION]
[--sampling-prior SAMPLING_PRIOR] [--no-normalise] [--convert]
[--discard DISCARD] [--mpioutput] [--combine_mpi_output]
[--c_generate] [--relative_loop] [--bin-scatangle]
[--bin-size BIN_SCATANGLE_SIZE] [-q] [--nodes QSUB_NODES]
[--ppn QSUB_PPN] [--pmem QSUB_PMEM] [--email QSUB_M]
[--emailoptions QSUB_M] [--name QSUB_N]
[--walltime QSUB_WALLTIME] [--queue QSUB_Q]
[--bladeproperties QSUB_BLADE_PROPERTIES]
[--feature QSUB_BLADE_FEATURE]
[data_file]
```

10.1 Positional Arguments:

```
data_file
```

Data file to use for the inversion, optional but must be specified either as a positional argument or as an optional argument (see -d below) If not specified defaults to all *.inv files in current directory, and searches for all anglescatterfiles in the directory too. Inversion file extension can be set using –invext option. Angle scatter file extension can be set using –angleext option

10.2 Optional Arguments:

```
-h, --help
```

show this help message and exit

```
-d DATAFILE, --datafile DATAFILE, --data_file DATAFILE
```

Data file to use for the inversion. Can be provided as a positional argument. There are several different data file types:

- pickled dictionary
- csv file
- NLLOC hyp file

The data file is a pickled python dictionary of the form:

```
{'DataType':{'Stations':{'Name':['STA1','STA2',...],
'Azimuth':np.matrix([[190],[40],...]),
'TakeOffAngle':np.matrix([[70],[40],...])},
'Measured':np.matrix([[1],[-1],...]),
'Error':np.matrix([[0.01],[0.02],...])}}
```

e.g.:

```
{'P/SHRMSAmplitudeRatio':{'Stations':{'Name':['S
0649',"S0162"],
'Azimuth':np.array([90.0,270.0]),
'TakeOffAngle':np.array([30.0,60.0])},
'Measured':np.matrix([[1],[-1]]),
'Error':np.matrix([[ 0.001,0.02],[
0.001,0.001]])}}
```

Or a CSV file with events split by blank lines, a header line showing which row corresponds to which information (default is as shown here), UID and data-type information stored in the first column, e.g.:

```
UID=123,,,,
PPolarity,,,,
Name,Azimuth,TakeOffAngle,Measured,Error
S001,120,70,1,0.01
S002,160,60,-1,0.02
P/SHRMSAmplitudeRatio,,,,
Name,Azimuth,TakeOffAngle,Measured,Error
S003,110,10,1,0.05 0.04
,,,,
PPolarity ,,,,
Name,Azimuth,TakeOffAngle,Measured,Error
S003,110,10,1,0.05
```

Is a CSV file with 2 events, one event UID of 123, and PPolarity data at S001 and S002 and P/SHRMSAmplitude data at S003, and a second event with no UID (will default to the event number, in this case 2) with PPolarity data at S003.

This data format can be constructed manually or automatically.

```
-s LOCATION_PDF_FILE_PATH, --anglescatterfilepath LOCATION_PDF_FILE_PATH,
--location_pdf_file_path LOCATION_PDF_FILE_PATH,
--location_file_path LOCATION_PDF_FILE_PATH,
--scatterfilepath LOCATION_PDF_FILE_PATH,
--scatter_file_path LOCATION_PDF_FILE_PATH
```

Path to location scatter angle files - wild cards behave as normal. To include the model and location uncertainty, a ray path angle pdf file must be provided. This is of the form:

```
probability1
Station1    Azimuth1    TakeOffAngle1
Station2    Azimuth2    TakeOffAngle2
.
.
.
StationN    AzimuthN    TakeOffAngleN
```

probability2 Station1 Azimuth1 TakeOffAngle1 Station2 Azimuth2 TakeOffAngle2 . . . StationN AzimuthN TakeOffAngleN

e.g.:

```
504.7
S0529   326.1   131.7
S0083   223.7   118.2
S0595   42.6    117.8
S0236   253.6   118.6
&&
504.7
S0529   326.1   131.8
S0083   223.7   118.2
S0595   42.7    117.9
S0236   253.5   118.7
```

```
-a {iterate,time,mcmc,transdmcmc}, --algorithm {iterate,time,mcmc,transdmcmc}
```

Selects the algorithm used for the search. [default=time] Possible algorithms are: iterate (random sampling of the source space for a set number of samples) time (random sampling of the source space for a set time) mcmc (Markov chain Monte Carlo sampling)

```
-l, --singlethread, --single, --single_thread
```

Flag to disable parallel computation

```
-n N, --numberworkers N, --number_workers N
```

Set the number of workers used in the parallel computation. [default=all available cpus]

```
-m MEM, --mem MEM, --memory MEM, --physical_memory MEM, --physicalmemory MEM
```

Set the maximum memory used in Gb if psutil not available [default=8Gb]

```
-c, --doublecouple, --double-couple, --double_couple, --dc, --DC
```

Flag to constrain the inversion to double-couple sources only

---

```
-b, --compareconstrained, --compare_constrained
```

Flag to run two inversions, one constrained to double-couple and one unconstrained

---

```
--nstations NUMBER_STATIONS
```

Set the maximum number of stations without having to load an angle pdf file - used for calculating sample sizes and memory sizes, and can speed up the calculation a bit, but has no effect on result.

```
--nanglesamples NUMBER_LOCATION_SAMPLES,
--nlocationsamples NUMBER_LOCATION_SAMPLES,
--number_location_samples NUMBER_LOCATION_SAMPLES,
--number-location-samples NUMBER_LOCATION_SAMPLES
```

Set the maximum number of angle pdf samples to use. If this is less than the total number of samples, a subset are randomly selected [default=0].

```
-f, --file_sample, --file-sample, --filesample, --disk_sample,
--disk-sample, --disksample
```

Save sampling to disk (allows for easier recovery and reduces memory requirements, but can be slower)

```
--not_file_safe, --not-file-safe, --notfilesafe, --no_file_safe,
--no-file-safe, --nofilesafe
```

Disable file safe saving (i.e. copy and write to .mat~ then copy back

---

```
-i INVERSION_OPTIONS, --inversionoptions INVERSION_OPTIONS,
--inversion_options INVERSION_OPTIONS
```

Set the inversion data types to use: comma delimited. If not set, the inversion uses all the data types in the data file. e.g. PPolarity,P/SHRMSAmplitudeRatio

Needs to correspond to the data types in the data file.

If not specified can lead to independence errors: e.g. P/SH Amplitude Ratio and P/SV Amplitude Ratio can give SH/SV Amplitude Ratio. Therefore using SH/SV Amplitude Ratio in the inversion is reusing data and will artificially sharpen the PDF. This applies to all forms of dependent measurements.

```
-o FID, --out FID, --fid FID, --outputfile FID, --outfile FID
```

Set output file basename [default=mtfitOutput]

```
-x MAX_SAMPLES, --samples MAX_SAMPLES, --maxsamples MAX_SAMPLES,
--max_samples MAX_SAMPLES, --chain_length MAX_SAMPLES,
--max-samples MAX_SAMPLES, --chain-length MAX_SAMPLES, --chainlength MAX_SAMPLES
```

Iteration algorithm: Set maximum number of samples to use [default=6000000]. McMC algorithms: Set chain length [default=10000], trans-d McMC [default=100000]

```
-t MAX_TIME, --time MAX_TIME, --maxtime MAX_TIME, --max_time MAX_TIME
```

Time algorithm: Set maximum time to use [default=600]

```
-e, --multiple_events, --multiple-events
```

Run using events using joint PDF approach

```
-r, --relative_amplitude, --relative-amplitude
```

Run using events using joint PDF approach

```
--marginalise_relative, --marginalise, --marginalise-relative
```

Flag to marginalise location uncertainty in relative amplitude case [default=False]

```
-R, --recover
```

Recover crashed run (ie restart from last event not written out)]

```
--invext DATA_EXTENSION, --dataextension DATA_EXTENSION,
--dataext DATA_EXTENSION, --data-extension DATA_EXTENSION,
--data_extension DATA_EXTENSION
```

Set data file extension to search for when inverting on a folder

---

```
--angleext ANGLE_EXTENSION, --locationextension ANGLE_EXTENSION,
--locationext ANGLE_EXTENSION, --location-extension ANGLE_EXTENSION,
--location_extension ANGLE_EXTENSION
```

Set location sample file extension to search for when inverting on a folder

---

```
-S MINIMUM_NUMBER_INTERSECTIONS,
--minimum_number_intersections MINIMUM_NUMBER_INTERSECTIONS,
--min_number_intersections MINIMUM_NUMBER_INTERSECTIONS,
--minimum-number-intersections MINIMUM_NUMBER_INTERSECTIONS,
--min-number-intersections MINIMUM_NUMBER_INTERSECTIONS
```

For relative amplitude inversion, the minimum number of intersecting stations required (must be greater than 1) [default=2]

```
-M, --mpi, --MPI
```

Run using mpi - will reinitialise using mpirun (mpi etc needs to be added to path)

```
-B, --benchmark, --benchmarking
```

Run benchmark tests for the event

```
-X MIN_NUMBER_INITIALISATION_SAMPLES,
--min_number_check_samples MIN_NUMBER_INITIALISATION_SAMPLES,
--min_number_initialisation_samples MIN_NUMBER_INITIALISATION_SAMPLES
```

Minimum number of samples for McMC initialiser, or the minimum number of samples required when using quality check (-Q)

```
-T, --test, --test
```

Run mtfit Test suite (if combined with -q runs test suite on cluster

---

```
-Q [QUALITY_CHECK], --quality [QUALITY_CHECK]
```

Run mtfit with quality checks enabled [default=False]. Checks if an event has a percentage of non-zero samples lower than the flag - values from 0-100.

```
-D, --debug
```

Run mtfit with debugging enabled.

```
-V VERBOSITY, --verbosity VERBOSITY
```

Set verbosity level for non-fatal errors [default=0].

```
-g, --diagnostics
```

Run mtfit with diagnostic output. Outputs the full chain and sampling - wil make a large file.

```
-j DIMENSION_JUMP_PROB, --jumpProbability DIMENSION_JUMP_PROB,
--jumpProb DIMENSION_JUMP_PROB, --jumpprob DIMENSION_JUMP_PROB,
--jumpProb DIMENSION_JUMP_PROB, --dimensionJumpProb DIMENSION_JUMP_PROB,
--dimensionjumpprob DIMENSION_JUMP_PROB
```

Sets the probability of making a dimension jump in the Trans-Dimensional McMC algorithm [default=0.01]

```
-y {grid}, --initialSampling {grid}
```

Sets the initialisation sampling method for McMC algorithms choices: grid - use grid based sampling to find non-zero initial sample [default=grid]

```
-u MIN_ACCEPTANCE_RATE, --minAcceptanceRate MIN_ACCEPTANCE_RATE,
--minacceptancerate MIN_ACCEPTANCE_RATE,
--min_acceptance_rate MIN_ACCEPTANCE_RATE
```

Set the minimum acceptance rate for the McMC algorithm [mcmc default=0.3, transdmcmc default=0.05]

```
-v MAX_ACCEPTANCE_RATE, --maxAcceptanceRate MAX_ACCEPTANCE_RATE,
--maxacceptancerate MAX_ACCEPTANCE_RATE,
--max_acceptance_rate MAX_ACCEPTANCE_RATE
```

Set the maximum acceptance rate for the McMC algorithm [mcmc default=0.5, transdmcmc default=0.2]

```
-w ACCEPTANCE_RATE_WINDOW,
--acceptanceLearningWindow ACCEPTANCE_RATE_WINDOW,
--acceptancelearningwindow ACCEPTANCE_RATE_WINDOW
```

Sets the window for calculating and updating the acceptance rate for McMC algorithms [default=500]

```
-W WARNINGS, --warnings WARNINGS, --Warnings WARNINGS
```

Sets the warning visibility.

options are:

- "e","error" - turn matching warnings intoexceptions
- "i","ignore" - never print matching warnings
- "a","always" - always print matchingwarnings
- "d","default" - print the first occurrenceof matching warnings for each location where thewarning is issued
- "m","module" - print the first occurrence ofmatching warnings for each module where the warning isissued
- "o","once" - print only the first occurrenceof matching warnings, regardless of location

---

```
-z LEARNING_LENGTH, --learningLength LEARNING_LENGTH,
--learninglength LEARNING_LENGTH, --learning_length LEARNING_LENGTH
```

Sets the number of samples to discard as the learning period [default=5000]

```
--version
```

show program's version number and exit

```
--mpi_call
```

Warning: Do not use - automatically set when spawning mpi subprocess

---

```
--output-format {hyp,pickle,matlab}, --output_format {hyp,pickle,matlab},
--outputformat {hyp,pickle,matlab}, --format {hyp,pickle,matlab}
```

Output file format [default=matlab]

```
--results-format {hyp,full_pdf}, --results_format {hyp,full_pdf},
--resultsformat {hyp,full_pdf}
```

Output results data format (extensible) [default=full_pdf]

```
--no-dist, --no_dist, --nodist
```

Do not output station distribution if running location samples

---

```
--dc-prior DC_PRIOR, --dc_prior DC_PRIOR, --dcprior DC_PRIOR
```

Prior probability for the double-couple model when using the Trans-Dimensional McMC algorithm

---

```
--sampling SAMPLING, --sampling SAMPLING, --sampling SAMPLING
```

Random moment tensor sampling distribution

```
--sample-models SAMPLE_DISTRIBUTION,
--sample_distribution SAMPLE_DISTRIBUTION, --samplemodels SAMPLE_DISTRIBUTION
```

Alternate models for random sampling (Monte Carlo algorithms only)

```
--sampling-prior SAMPLING_PRIOR, --sampling_prior SAMPLING_PRIOR,
--samplingprior SAMPLING_PRIOR
```

Prior probability for the model distribution when using the McMC algorithm, alternatively the prior distribution for the source type parameters gamma and delta for use by the Bayesian evidence calculation for the MC algorithms

---

```
--no-normalise, --no-norm, --no_normalise, --no_norm
```

Do not normalise the output pdf

```
--convert
```

Convert the output MTs to Tape parameters, hudson parameters and strike dip rakes.

```
--discard DISCARD
```

Fraction of maxProbability * total samples to discard as negligeable.

```
--mpioutput, --mpi_output, --mpi-output
```

When the mpi flag -M is used outputs each processor individually rather than combining

---

```
--combine_mpi_output, --combine-mpi-output, --combinempioutput
```

Combine the mpi output from the mpioutput flag. The data path corresponds to the root path for the mpi output

---

```
--c_generate, --c-generate, --generate
```

Generate moment tensor samples in the probability evaluation

```
--relative_loop, --relative-loop, --relativeloop, --loop
```

Loop over independent non-zero samples randomly to construct joint rather than joint samples

10.3 Scatangle:

```
--bin-scatangle, --binscatangle, --bin_scatangle
```

Bin the scatangle file to reduce the number of samples [default=False]. –bin-size Sets the bin size parameter .

```
--bin-size BIN_SCATANGLE_SIZE, --binsize BIN_SCATANGLE_SIZE,
--bin_size BIN_SCATANGLE_SIZE
```

Sets the scatangle bin size parameter [default=1.0]

10.4 Cluster:

```
Commands for using mtfit on a cluster environment using qsub/PBS
```

```
-q, --qsub, --pbs
```

Flag to set mtfit to submit to cluster

```
--nodes QSUB_NODES
```

Set number of nodes to use for job submission. [default=1]

```
--ppn QSUB_PPN
```

Set ppn to use for job submission. [default=8]

```
--pmem QSUB_PMEM
```

Set pmem (Gb) to use for job submission. [default=2Gb]

```
--email QSUB_M
```

Set user email address.

```
--emailoptions QSUB_M
```

Set PBS -m mail options. Requires email address using -M. [default=bae]

```
--name QSUB_N
```

Set PBS -N job name options. [default=mtfit]

```
--walltime QSUB_WALLTIME
```

Set PBS maximum wall time. Needs to be of the form HH:MM:SS. [default=24:00:00]

```
--queue QSUB_Q
```

Set PBS -q Queue options. [default=batch]

```
--bladeproperties QSUB_BLADE_PROPERTIES
```

Set desired PBS blade properties. [default=False]

```
--feature QSUB_BLADE_FEATURE
```

Set desired Torque feature arguments. [default=False]

CHAPTER 11

Plotting the Moment Tensor

`mtfit` has a plotting submodule that can be used to represent the source. There are several different plot types, shown below, and MTplot can be used both from the command line, and from within the python interpreter.

This section describes how to use the plotting tools and shows the different plot types:

- *Beachball*
- *Fault plane*
- *Riedesel-Jordan*
- *Radiation*
- *Lune*
- *Hudson*

These are plotted using matplotlib[37], using a class based system. The main plotting class is the `MTplot` class, which stores the figure and handles the plotting, and each axes plotted is shown using a plot class from `mtfit.plot.plot_classes`. The plotting methods are designed to enable easy plotting without much user input, but also allow more complex plots to be made. The *examples section* shows two examples of using the plotting submodule.

The source code is shown in source-plot_classes.

Warning: `matplotlib` does not plot 3d plots very well, as each object is converted to a 2d object and plotted, and given a constant zorder for the whole plot. Consequently, the bi-axes plot (*Chapman and Leaney, 2011*) is not included as an option, and other 3D plots may not always work correctly.

11.1 Using MTplot from the command line

`MTplot` can be run from the command line. A script should have been installed onto the path during installation and should be callable as:

```
$ MTplot
```

However it may be necessary to install the script manually. This is platform dependent.

[37] http://matplotlib.org/

11.1.1 Script Installation

Linux

Add this python script to a directory on the $PATH environmental variable:

```
#!/usr/bin/env python
import mtfit
mtfit.plot.__run__()
```

and make sure it is executable.

Windows

Add the linux script (above) to the path or if using powershell edit the powershell profile (usually found in *Documents/WindowsPowerShell/* - if not present use `$PROFILE|Format-List -Force` to locate it, it may be necessary to create the profile) and add:

```
function MTplot{
    $script={
        python -c "import mtfit;mtfit.plot.__run__()" $args
        }
    Invoke-Command -ScriptBlock $script -ArgumentList $args
    }
```

Windows Powershell does seem to have some errors with commandline arguments, if necessary these should be enclosed in quotation marks e.g. "-d=datafile.inv"

11.1.2 Command Line Options

There are several command line options available, these can be found by calling:

```
$ MTplot -h
```

The command line defaults can be set using the same defaults file as for mtfit (see *Running mtfit*).

11.2 Using MTplot from the Python interpreter

Although MTplot can be run from the command line, it is much more powerful to run it from within the python interpreter. To run MTplot from the python interpreter, create the `MTplot` object:

```
>>> from mtfit.plot import MTplot
>>> MTplot(MTs,plot_type='beachball',stations={},plot=True,*args,**kwargs)
```

See *Making the MTplot class* for more information on the `MTplot` object.

11.3 Input Data

`mtfit.plot.__core__.read()` can read the three default output formats (MATLAB, hyp and pickle) for mtfit results.

Additional parsers can be installed using the *mtfit.plot_read* entry point described in *Extending mtfit*.

11.4 MTplot Class

The MTplot class is used to handle plotting the moment tensors. The moment tensors are stored as an *MTData* class.

class mtfit.plot.plot_classes.**MTplot**(*MTs*, *plot_type='beachball'*, *stations={}*, *plot=True*, *label=False*, *save_file=''*, *save_dpi=200*, **args*, ***kwargs*)

MTplot class - handles plotting of the moment tensor using different plot_classes

This object acts as a handle round a matplotlib figure, handling the moment tensors and creating the plot classes for axes.

Initialises the MTplot object.

Multiple plots (subplots) are handled using the matplotlib GridSpec - to create multiple plots, use nested lists for the MTs, with the inner lists corresponding to each column and the outer each row, e.g.:

```
[[x,y=1,1;x,y=2,1],[x,y=1,2;x,y=2,2]]
```

The total number of columns is given by the max number of points in the nested list. Lists with fewer points will have the last plot stretched to cover the remaining columns. Setting None in the list will stretch the plot to it's left to cover the columns.

The args and kwargs, which are passed through to the plot_class object can also be nested in this way. Additionally, if the parameters are equal to the number of columns, or number of rows, the same values are used for each plot in the corresponding column or row. If the dimension of the multiplot is square, the parameters are assumed to correspond to the rows.

Args MTs: Moment tensor samples for creating MTData object, such as a numpy array of MT 6 vectors (see MTData initialisation for different types, and MTplot docstring for handling multiple plots

Keyword Args plot_type: str - plot type selection (Default is 'beachball') stations: dict - Dictionary of stations, containing keys: 'names','azimuth','takeoff_angle' and 'polarity' or an empty dict (default) plot: bool - flag to actually plot and show the figure label: bool - flag to show axis labels if multiple plots are being shown save_file: string - filename to save plot to (if set) save_dpi: int - output dpi for file save args: passed through to plot_class initialisation kwargs: passed through to plot_class initialisation

ax_labels(*show=True*)

Set or update axis label to axis corners.

plot(**args*, ***kwargs*)

Plots the figure and shows it.

Calls the plot functions for each plot_class in the multiplot.

11.5 MTData Class

The MTData class is used for storing and converting the moment tensors for plotting.

class mtfit.plot.plot_classes.**MTData**(*MTs*, *probability=[]*, *c=False*, ***kwargs*)

MTData object manages the moment tensors.

Additionally it enables conversion between different parameterisations and calculation of several statistics.

This is used by all of the plot classes for handling the moment tensor, and is transparent to converting and accessing the moment tensors, as it behaves like a numpy array (e.g. indexing and properties like shape are given for the moment tensor).

The MTData object also stores the probability, and is used for calculating the orientation mean.

MTData initialisation

Parameters corresponding to the moment tensor samples can be set as kwargs. These parameters include:

T, N, P, E, u, v, tau, k, gamma, delta, kappa, h, sigma, strike1, dip1, rake1, strike2, dip2, rake2, N1, N2

Args

MTs: numpy array of moment tensor six vectors with shape (6,n). Alternatively, the input can be a dictionary from the mtfit output.

Keyword Args

probability: list or numpy array of probabilities - should be the same length as the number of moment tensors, or empty (Default is [])

****kwargs: Attributes that correspond to converted parameters** of the moment tensor, as described above

cluster_normals()
Cluster the normals.

This is a very simplistic algorithm that clusters the normals by dividing them into two groups with the shortest distance between them.

N.B. This depends on the initial values given, so if the normals are not well clustered, or the initial normals chosen are outliers, the results will not be a good clustering.

Consequently, only use this approach for well clustered fault plane normals (in at least one normal direction). The more tightly clustered normals are set to clustered_N1.

Returns numpy array, numpy array, numpy array, numpy array: tuple of numpy arrays, clustered_N1, clustered_N2, clustered_rake1,clustered_rake2

get_max_probability(*single=False*)
Returns an MTData object containing the maximum probability solutions

Keyword Args single: bool - flag to return only one moment tensor (the first maximum probability moment tensor)

Returns New MTData object with max probability MT samples.

Return type MTData

get_mean()
Calculates the mean moment tensor and variance of the moment tensor samples

Returns tuple of numpy arrays for the mean moment tensor six vector and the moment tensor six vector covariance.

Return type numpy array,numpy array

get_mean_orientation()
Get the mean orientation from the clustered normals and rake parameters using the more tightly clustered parameters.

This also calculates the variance of the rake distributions and covariance of the clustered normals (using the probability if it is set)

Returns numpy array, numpy array, numpy array: tuple of the mean strike dip and rake of the fault planes.

get_unique_McMC()
Gets the unique McMC samples, with the probability scaled by the number of samples

Returns MTData: New MTData object with unique MT samples and probabilities corresponding to the moment tensor frequencies.

is_dc()
Returns array of DC values

11.6 Beachball plot

The simplest plot is a beachball plot using the `mtfit.plot.plot_classes._AmplitudePlot` class.

Using the MTplot function, it can be made with the following commands:

```
>>> import mtfit
>>> import numpy as np
>>> mtfit.plot.MTplot(np.array([[1],[0],[-1],[0],[0],[0]]),'beachball',
        fault_plane=True)
```

This plots the equal area projection of the source (a double-couple).

Stations can be included as a dictionary, with the azimuths and takeoff angles in degrees, such as:

```
>>> stations={'names':['S01','S02','S03','S04'],
              'azimuth':np.array([120.,5.,250.,75.]),
              'takeoff_angle':np.array([30.,60.,45.,10.]),
              'polarity':np.array([1,0,1,-1])}
>>> mtfit.plot.MTplot(np.array([[1],[0],[-1],[0],[0],[0]]),'beachball',
            stations=stations,fault_plane=True)
```

If the polarity probabilities have been used in the inversion, the probabilities can be plotted on the receivers, by setting the stations `polarity` array as an array of the larger polarity probabilities, with negative polarity probabilities corresponding to polarities in the negative direction, e.g.:

```
>>> stations={'names':['S01','S02','S03','S04'],
              'azimuth':np.array([120.,5.,250.,75.]),
              'takeoff_angle':np.array([30.,60.,45.,10.]),
              'polarity':np.array([0.8,0.5,0.7,-0.9])}
>>> mtfit.plot.MTplot(np.array([[1],[0],[-1],[0],[0],[0]]),'beachball',
            stations=stations,fault_plane=True)
```

To tweak the plot further, the plot class can be used directly:

```
>>> import mtfit
>>> import numpy as np
>>> X=mtfit.plot.plot_classes._AmplitudePlot(False,False,
        np.array([[1],[0],[-1],[0],[0],[0]]),'beachball',
        stations=stations,fault_plane=True)
>>> X.plot()
```

The first two arguments correspond to the subplot_spec and matplotlib figure to be used - if these are False, then a new figure is created.

It uses the `mtfit.plot.plot_classes._AmplitudePlot` class:

class mtfit.plot.plot_classes.**_AmplitudePlot**(*subplot_spec*, *fig*, *MTs*, *stations={}*, *phase='P'*, **args*, ***kwargs*)

Amplitude plotting (beachball)

parameter single is set to True, as can only plot one source per axis

Args subplot_spec: matplotlib subplot spec fig: matplotlib figure MTs: moment tensor samples (see MTData initialisation docstring for formats)

Keyword Args phase: str - phase to plot (default='p') lower: bool - project lower hemisphere (ie. downward hemisphere) full_sphere: bool - plot the full sphere fault_plane: bool - plot the fault planes nodal_line: bool - plot the nodal lines stations: dict - station dict containing keys:

'names','azimuth','takeoff_angle' and 'polarity' station_distribution: list - list of station dictionaries corresponding to a location PDF distribution show_zero_polarity: bool - flag to show zero polarity receivers show_stations: bool - flag to show stations when station_distribution present station_markersize: float - station marker size (squared to get area) station_colors: tuple - tuple of colors for negative, no, and positive polarity TNP: bool - show the TNP axes on the plot colormap: str - matplotlib colormap selection (using matplotlib.cm.get_cmap()) fontsize: int - fontsize for text linewidth: float - base linewidth (sometimes thinner or thicker values are used, but relative to this parameter) text: bool - flag to show or hide text on the plot axis_lines: bool - flag to show or hide axis lines on the plot resolution: int - resolution for spherical sampling etc

plot (*MTs=False*, **args*, ***kwargs*)

Plots the result

Keyword Args MTs: Moment tensors to plot (see MTData initialisation docstring for formats) args: args passed to the _ax_plot function (e.g. set local parameters to be different from initialisation values) kwargs: kwargs passed to the _ax_plot function (e.g. set local parameters to be different from initialisation values)

11.7 Fault Plane plot

A similar plot to the amplitude beachball plot is the fault plane plot, made using the `mtfit.plot.plot_classes._FaultPlanePlot` class.

Using the MTplot function, it can be made with the following commands:

```
>>> import mtfit
>>> import numpy as np
>>> mtfit.plot.MTplot(np.array([[1],[0],[-1],[0],[0],[0]]),'faultplane',
        fault_plane=True)
```

This plots the equal area projection of the source (a double-couple).

Stations can be included as a dictionary, like with the beachball plot.

The fault plane plot also can plot the solutions for multiple moment tensors, so the input array can be longer:

```
>>> import mtfit
>>> import numpy as np
>>> mtfit.plot.MTplot(np.array([[ 1,0.9, 1.1,0.4],
                                [ 0,0.1,-0.1,0.6],
                                [-1, -1,  -1, -1],
                                [ 0,  0,   0,  0],
                                [ 0,  0,   0,  0],
                                [ 0,  0,   0,  0]]),
                       'faultplane',fault_plane=True)
```

There are additional initialisation arguments, such as `show_max_likelihood` and `show_mean` boolean flags, which shows the maximum likelihood fault planes in the color given by the default color argument, and the mean orientation in green.

Additionally, if the probability argument is set, the fault planes are coloured by the probability, with more likely planes darker.

It uses the `mtfit.plot.plot_classes._FaultPlanePlot` class:

class `mtfit.plot.plot_classes.`**_FaultPlanePlot** (*subplot_spec*, *fig*, *MTs*, *stations={}*, *probability=[]*, *phase='p'*, **args*, ***kwargs*)

Plots the fault plane distribution

parameter single is set to False as multiple sources can be plotted per axes

Args subplot_spec: matplotlib subplot spec fig: matplotlib figure MTs: moment tensor samples (see MTData initialisation docstring for formats)

Keyword Args probability: numpy array - moment tensor probabilities phase: str - phase to plot (default='p') lower: bool - project lower hemisphere (ie. downward hemisphere) full_sphere: bool - plot the full sphere fault_plane: bool - plot the fault planes nodal_line: bool - plot the nodal lines stations: dict - station dict containing keys: 'names','azimuth','takeoff_angle' and 'polarity' station_distribution: list - list of station dictionaries corresponding to a location PDF distribution show_zero_polarity: bool - flag to show zero polarity receivers show_stations: bool - flag to show stations when station_distribution present station_markersize: float - station marker size (squared to get area) station_colors: tuple - tuple of colors for negative, no, and positive polarity TNP: bool - show the TNP axes on the plot colormap: str - matplotlib colormap selection (using matplotlib.cm.get_cmap()) fontsize: int - fontsize for text linewidth: float - base linewidth (sometimes thinner or thicker values are used, but relative to this parameter) text: bool - flag to show or hide text on the plot axis_lines: bool - flag to show or hide axis lines on the plot resolution: int - resolution for spherical sampling etc show_max_likelihood: bool - show the maximum likelihood solution in the default color show_mean: bool - show the mean orientation source in green (thicker plane is better constrained) color: set color for maximum likelihood fault plane (show_max_likelihood)

plot(*MTs=False*, **args*, ***kwargs*)
Plots the result

Keyword Args MTs: Moment tensors to plot (see MTData initialisation docstring for formats) args: args passed to the _ax_plot function (e.g. set local parameters to be different from initialisation values) kwargs: kwargs passed to the _ax_plot function (e.g. set local parameters to be different from initialisation values)

11.8 Riedesel-Jordan plot

The Riedesel-Jordan plot is more complicated, and is described in *Riedesel and Jordan (1989)*. It plots the source type on the focal sphere, in a region described by the source eigenvectors.

Using the MTplot function, it can be made with the following commands:

```
>>> import mtfit
>>> import numpy as np
>>> mtfit.plot.MTplot(np.array([[1],[0],[-1],[0],[0],[0]]),'riedeseljordan')
```

This plots the equal area projection of the source (a double-couple).

Stations cannot be shown on this plot.

The Riedesel-Jordan plot cannot plot the solutions for multiple moment tensors, so the input array can only be one moment tensor.

It uses the `mtfit.plot.plot_classes._RiedeselJordanPlot` class:

class `mtfit.plot.plot_classes.`**_RiedeselJordanPlot**(*subplot_spec*, *fig*, *MTs*, **args*, ***kwargs*)
Plots the source as a Riedesel-Jordan type plot

Args subplot_spec: matplotlib subplot spec fig: matplotlib figure MTs: moment tensor samples (see MTData initialisation docstring for formats)

Keyword Args phase: str - phase to plot (default='p') lower: bool - project lower hemisphere (ie. downward hemisphere) full_sphere: bool - plot the full sphere fault_plane: bool - plot the fault planes nodal_line: bool - plot the nodal lines show_stations: bool - flag to show stations when station_distribution present station_markersize: float - sets the source type marker sizes (squared to get area) station_colors: tuple - tuple of colors for negative, no, and positive polarity TNP: bool - show the TNP axes on the plot colormap: str - matplotlib colormap selection (using matplotlib.cm.get_cmap()) color: str - matplotlib color for source region fontsize: int - fontsize for text linewidth: float - base linewidth (sometimes thinner or thicker values are used, but relative to this parameter) text: bool - flag

to show or hide text on the plot axis_lines: bool - flag to show or hide axis lines on the plot resolution: int - resolution for spherical sampling etc

plot(*MTs=False*, **args*, ***kwargs*)
Plots the result

Keyword Args MTs: Moment tensors to plot (see MTData initialisation docstring for formats) args: args passed to the _ax_plot function (e.g. set local parameters to be different from initialisation values) kwargs: kwargs passed to the _ax_plot function (e.g. set local parameters to be different from initialisation values)

11.9 Radiation plot

The radiation plot shows the same pattern as the beachball plot, except the shape is scaled by the amplitude on the focal sphere.

Using the MTplot function, it can be made with the following commands:

```
>>> import mtfit
>>> import numpy as np
>>> mtfit.plot.MTplot(np.array([[1],[0],[-1],[0],[0],[0]]),'radiation')
```

This plots the equal area projection of the source (a double-couple).

Stations cannot be shown on this plot.

The radiation plot cannot plot the solutions for multiple moment tensors, so the input array can only be one moment tensor.

It uses the `mtfit.plot.plot_classes._RadiationPlot` class:

class mtfit.plot.plot_classes.**_RadiationPlot**(*subplot_spec*, *fig*, *MTs*, *stations={}*, *phase='P'*, **args*, ***kwargs*)
Plot the radiation pattern

Args subplot_spec: matplotlib subplot spec fig: matplotlib figure MTs: moment tensor samples (see MTData initialisation docstring for formats)

Keyword Args phase: str - phase to plot (default='p') lower: bool - project lower hemisphere (ie. downward hemisphere) full_sphere: bool - plot the full sphere fault_plane: bool - plot the fault planes nodal_line: bool - plot the nodal lines stations: dict - station dict containing keys: 'names','azimuth','takeoff_angle' and 'polarity' station_distribution: list - list of station dictionaries corresponding to a location PDF distribution show_zero_polarity: bool - flag to show zero polarity receivers show_stations: bool - flag to show stations when station_distribution present station_markersize: float - station marker size (squared to get area) station_colors: tuple - tuple of colors for negative, no, and positive polarity TNP: bool - show the TNP axes on the plot colormap: str - matplotlib colormap selection (using matplotlib.cm.get_cmap()) fontsize: int - fontsize for text linewidth: float - base linewidth (sometimes thinner or thicker values are used, but relative to this parameter) text: bool - flag to show or hide text on the plot axis_lines: bool - flag to show or hide axis lines on the plot resolution: int - resolution for spherical sampling etc

plot(*MTs=False*, **args*, ***kwargs*)
Plots the result

Keyword Args MTs: Moment tensors to plot (see MTData initialisation docstring for formats) args: args passed to the _ax_plot function (e.g. set local parameters to be different from initialisation values) kwargs: kwargs passed to the _ax_plot function (e.g. set local parameters to be different from initialisation values)

11.10 Hudson plot

The Hudson plot is a source type plot, described in *Hudson et al. (1989)*. It plots the source type in a quadrilateral, depending on the chosen projection. There are two projections, the tau-k plot and the u-v plot, with the latter being more common (and the default).

Using the MTplot function, it can be made with the following commands:

```
>>> import mtfit
>>> import numpy as np
>>> mtfit.plot.MTplot(np.array([[1],[0],[-1],[0],[0],[0]]),'hudson')
```

This plots the u-v plot of the source (a double-couple).

Stations cannot be shown on this plot.

The Hudson plot can plot the solutions for multiple moment tensors, so the input array can be longer. Additionally, it can also plot a histogram of the PDF, if the probability argument is set.

It uses the `mtfit.plot.plot_classes._HudsonPlot` class:

class `mtfit.plot.plot_classes.`**_HudsonPlot**(*subplot_spec*, *fig*, *MTs*, *projection='uv'*, *probability=[]*, ***kwargs*)

Hudson plot class

Plots the source on the u-v or tau-k Hudson plot

Args subplot_spec: matplotlib subplot spec fig: matplotlib figure MTs: moment tensor samples (see MTData initialisation docstring for formats)

Keyword Args projection: str - select the projection type from uv or tauk [default is uv] probability: numpy array - moment tensor probabilities colormap: str - matplotlib colormap selection (using matplotlib.cm.get_cmap()) fontsize: int - fontsize for text linewidth: float - base linewidth (sometimes thinner or thicker values are used, but relative to this parameter) text: bool - flag to show or hide text on the plot axis_lines: bool - flag to show or hide axis lines on the plot resolution: int - resolution for spherical sampling etc grid_lines: bool - show the interior grid lines marginalised: bool - marginalise the PDF (defailt is True) color: set marker color type_label: bool - show the label of the different types hex_bin: bool - use the hex-bin histogram type (slightly smoother) bins: int/array/list of arrays - bins for numpy histogram call

plot(*MTs=False*, **args*, ***kwargs*)

Plots the result

Keyword Args MTs: Moment tensors to plot (see MTData initialisation docstring for formats) args: args passed to the _ax_plot function (e.g. set local parameters to be different from initialisation values) kwargs: kwargs passed to the _ax_plot function (e.g. set local parameters to be different from initialisation values)

11.11 Lune plot

The Lune plot is a source type plot, described in *Tape and Tape (2012)*. It plots the source type in the fundamental eigenvalue lune, which can be projected into 2 dimensions.

Using the MTplot function, it can be made with the following commands:

```
>>> import mtfit
>>> import numpy as np
>>> mtfit.plot.MTplot(np.array([[1],[0],[-1],[0],[0],[0]]),'lune')
```

Stations cannot be shown on this plot.

The Lune plot can plot the solutions for multiple moment tensors, so the input array can be longer. Additionally, it can also plot a histogram of the PDF, if the probability argument is set.

It uses the `mtfit.plot.plot_classes._LunePlot` class:

class `mtfit.plot.plot_classes.`**_LunePlot**(*subplot_spec*, *fig*, *MTs*, *stations={}*, *phase='P'*, **args*, ***kwargs*)

Plots the source on the fundamental eigenvalue lune

Can be either a histogram or scatter plot.

parameter single is set to False as multiple sources can be plotted per axes

Args subplot_spec: matplotlib subplot spec fig: matplotlib figure MTs: moment tensor samples (see MTData initialisation docstring for formats)

Keyword Args phase: str - phase to plot (default='p') lower: bool - project lower hemisphere (ie. downward hemisphere) full_sphere: bool - plot the full sphere fault_plane: bool - plot the fault planes nodal_line: bool - plot the nodal lines stations: dict - station dict containing keys: 'names','azimuth','takeoff_angle' and 'polarity' station_distribution: list - list of station dictionaries corresponding to a location PDF distribution show_zero_polarity: bool - flag to show zero polarity receivers show_stations: bool - flag to show stations when station_distribution present station_markersize: float - station marker size (squared to get area) station_colors: tuple - tuple of colors for negative, no, and positive polarity TNP: bool - show the TNP axes on the plot colormap: str - matplotlib colormap selection (using matplotlib.cm.get_cmap()) fontsize: int - fontsize for text linewidth: float - base linewidth (sometimes thinner or thicker values are used, but relative to this parameter) text: bool - flag to show or hide text on the plot axis_lines: bool - flag to show or hide axis lines on the plot resolution: int - resolution for spherical sampling etc

plot(*MTs=False*, **args*, ***kwargs*)

Plots the result

Keyword Args MTs: Moment tensors to plot (see MTData initialisation docstring for formats) args: args passed to the _ax_plot function (e.g. set local parameters to be different from initialisation values) kwargs: kwargs passed to the _ax_plot function (e.g. set local parameters to be different from initialisation values)

11.12 Examples

This section shows a pair of simple examples and their results.

The first example is to plot the data from *Krafla P Polarity example*:

```
import mtfit
import numpy as np
#Load Data
st_dist=mtfit.plot.read('krafla_event_ppolarityDCStationDistribution.mat',
    station_distribution=True)
DCs,DCstations=mtfit.plot.read('krafla_event_ppolarityDC.mat')
MTs,MTstations=mtfit.plot.read('krafla_event_ppolarityMT.mat')
#Plot
plot=mtfit.plot.MTplot([np.array([1,0,-1,0,0,0]),DCs,MTs],
    stations=[DCstations,DCstations,MTstations],
    station_distribution=[st_dist,False,False],
    plot_type=['faultplane','faultplane','hudson'],fault_plane=[False,True,False],
    show_mean=False,show_max=True,grid_lines=True,TNP=False,text=[False,False,
↪True])
```

This produces a `matplotlib` figure, shown in Fig. *11.1*.

The second example shows the different plot types:

```
import mtfit
import numpy as np
import scipy.stats as sp
#Generate Data
```

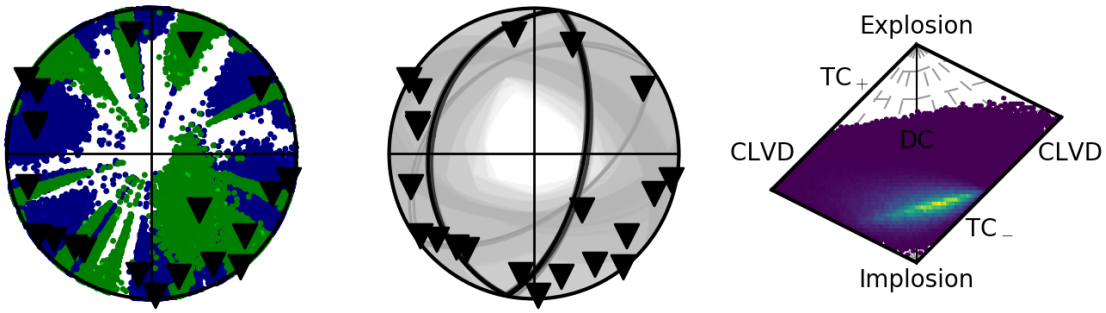

Fig. 11.1: *Beachball plots showing the station location uncertainty, and the fault plane orientations for the double-couple constrained inversion and the marginalised source-type PDF for the faultplane moment tensor inversion of the krafla data using polarity probabilities.*

```
n=100
DCs=mtfit.MTconvert.Tape_MT6(np.zeros(n),np.zeros(n),np.pi+0.1*np.random.randn(n),
        0.5+0.01*np.random.randn(n),0.1*np.random.randn(n))
probDCs=np.random.rand(n)
n=10000
g=-np.pi/12+0.01*np.random.randn(n)
d=np.pi/3+0.1*np.random.randn(n)
MTs=mtfit.MTconvert.Tape_MT6(g,d,np.pi+0.1*np.random.randn(n),
        0.5+0.01*np.random.randn(n),0.1*np.random.randn(n))
probMTs=sp.norm.pdf(g,-np.pi/12,0.01)*sp.norm.pdf(d,np.pi/3,0.1)
plot_sources=[np.array([1,0,1,-1,0,0]),DCs,MTs,MTs,np.array([1,0,1,-1,0,0])]
#Plot
plot=mtfit.plot.MTplot(plot_sources,
    plot_type=['beachball','faultplane','hudson','lune','riedeseljordan'],
    probability=[False,probDCs,probMTs,probMTs,False],
    colormap=['bwr','bwr','viridis','viridis','bwr'],
    stations=[{'names':['S01','S02','S03','S04'],
               'azimuth':np.array([120.,45.,238.,341.]),
               'takeoff_angle':np.array([12.,56.,37.,78.]),
               'polarity':[1,0,-1,-1]},{},{},{},{}],
    show_mean=True,show_max=True,grid_lines=True,TNP=False,fontsize=6,
    station_markersize=2,markersize=2)
```

This produces a `matplotlib` figure, shown in Fig. *11.2*.

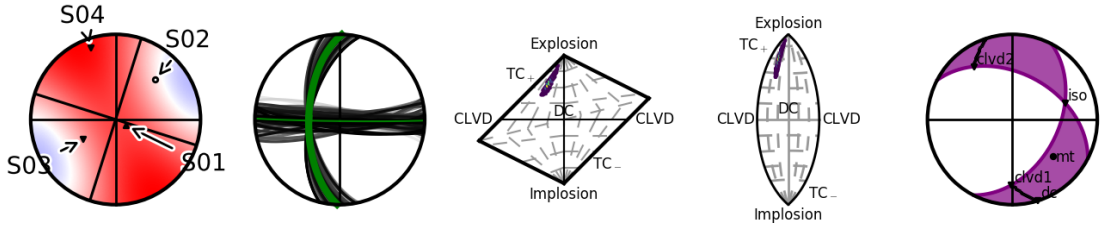

Fig. 11.2: *MTplot examples showing an equal area projection of a beachball for an example moment tensor source, fault plane distribution showing the mean orientation in green, Hudson and lune type plots of a full moment tensor PDF, and a Riedesel-Jordan type plot of an example moment tensor source.*

CHAPTER 12

MTplot command line options

Command line usage:

```
MTplot [-h] [-d DATAFILE] [-p PLOT_TYPE] [-c COLORMAP] [-f FONTSIZE]
[-l LINEWIDTH] [-t] [-r RESOLUTION] [-b BINS] [--fault_plane]
[-n] [--tnp] [--marker_size MARKERSIZE]
[--station_marker_size STATION_MARKERSIZE] [--showmaxlikelihood]
[--showmean] [--grid_lines] [--color COLOR] [--type_label]
[--hex_bin] [--projection PROJECTION] [--save SAVE_FILE]
[--save-dpi SAVE_DPI] [--version]
[data_file]
```

12.1 Positional Arguments:

```
data_file
```

Data file to use for plotting, optional but must be specified either as a positional argument or as an optional argument (see -d below)

12.2 Optional Arguments:

```
-h, --help
```

show this help message and exit

```
-d DATAFILE, --datafile DATAFILE, --data_file DATAFILE
```

MTplot can read the output data from mtfit

```
-p PLOT_TYPE, --plot_type PLOT_TYPE, --plottype PLOT_TYPE,
--plot-type PLOT_TYPE, --type PLOT_TYPE
```

Type of plot to make

```
-c COLORMAP, --colormap COLORMAP, --color_map COLORMAP, --color-map COLORMAP
```

Matplotlib colormap selection

```
-f FONTSIZE, --font_size FONTSIZE, --fontsize FONTSIZE, --font-size FONTSIZE
```

Fontsize

```
-l LINEWIDTH, --line_width LINEWIDTH, --linewidth LINEWIDTH,
--line-width LINEWIDTH
```

Linewidth

```
-t, --text, --show-text, --show_text, --showtext
```

Flag to show text or not

```
-r RESOLUTION, --resolution RESOLUTION
```

Resolution for the focal sphere plot types

```
-b BINS, --bins BINS  Number of bins for the histogram plot types
```

```
--fault_plane, --faultplane, --fault-plane
```

Show the fault planes on a focal sphere type plot

```
-n, --nodal_line, --nodal-line, --nodalline
```

Show the nodal lines on a focal sphere type plot

```
--tnp, --tp, --pt
```

Show TNP axes on focal sphere plots

```
--marker_size MARKERSIZE, --markersize MARKERSIZE, --marker-size MARKERSIZE
```

Set marker size

```
--station_marker_size STATION_MARKERSIZE,
--stationmarkersize STATION_MARKERSIZE,
--station-marker-size STATION_MARKERSIZE,
--station_markersize STATION_MARKERSIZE, --station-markersize STATION_MARKERSIZE
```

Set station marker size

```
--showmaxlikelihood, --show_max_likelihood, --show-max-likelihood
```

Show the maximum likelihood solution on a fault plane plot (shown in color set by –color).

```
--showmean, --show-mean, --show_mean
```

Show the mean orientaion on a fault plane plot (shown in green).

```
--grid_lines, --gridlines, --grid-lines
```

Show interior lines on Hudson and lune plots

```
--color COLOR
```

Set default color

```
--type_label, --typelabel, --type-label
```

Show source type labels on Hudson and lune plots.

```
--hex_bin, --hexbin, --hex-bin
```

Use hex bin for histogram plottings

```
--projection PROJECTION
```

Projection choice for focal sphere plots

```
--save SAVE_FILE, --save_file SAVE_FILE, --savefile SAVE_FILE,
--save-file SAVE_FILE
```

Set the filename to save to (if set the plot is saved to the file)

```
--save-dpi SAVE_DPI, --savedpi SAVE_DPI, --save_dpi SAVE_DPI
```

Output file dpi

```
--version
```

show program's version number and exit

CHAPTER 13

mtfit.inversion: Inversion object

The main `mtfit` inversion object is used to handle the forward model evaluation and process the results. The object initialisation and forward functions are documented here, although there are additional (private and special) functions that are documented in the source code.

class mtfit.inversion.**Inversion**(*data={}*, *data_file=False*, *location_pdf_file_path=False*, *algorithm='iterate'*, *parallel=True*, *n=0*, *phy_mem=8*, *dc=False*, ***kwargs*)

Main Inversion object

Runs the MT inversion, follwing a parameterisation set on initialisation. Parameters can be set for the algorithm to use in the kwargs.

Algorithm options are:

- Time - Monte Carlo random sampling - runs until time limit reached.
- Iterate - Monte Carlo random sampling - runs until sample limit reached.
- McMC - Markov chain Monte Carlo sampling.
- TransDMcMC - Markov chain Monte Carlo sampling.

These are discussed in more detail in the mtfit.algorithms documentation.

The inversion is run by calling the forward function (mtfit.inversion.Inversion.forward)

Data Format

The inversion expects a python dictionary of the data in the format:

```
>>data={'PPolarity':{'Measured':numpy.matrix([[-1],[-1]...]),
                     'Error':numpy.matrix([[0.01],[0.02],...]),
                     'Stations':{'Name':[Station1,Station2,...]
                                 'Azimuth':numpy.matrix([[248.0],[122.3]...]),
                                 'TakeOffAngle':numpy.matrix([[24.5],[2.8]...])
                                }
                   },
        'PSHAmplitudeRatio':{...},
        ...
        'UID':'Event1'
        }
```

The initial key arguments correspond to the data types that can be used in the inversion. The inversion uses polarity observations and amplitude ratios, and can also use relative amplitudes. This means that the useable data types are:

Polarity

- PPolarity
- SHPolarity
- SVPolarity

Polarity observations made manually or automatically. The corresponding data dictionary for polarities needs the following keys:

- *Measured*: numpy matrix of polarity observations
- *Error*: numpy matrix of fractional uncertainty in the polarity observations.
- *Stations*: dictionary of station information with keys:
  - *Name*: list of station names
  - *Azimuth*: numpy matrix of azimuth values in degrees
  - *TakeOffAngle*: numpy matrix of take off angle values in degrees - 0 down (NED coordinate system).

As with all of these dictionaries, the indexes of the observations and errors must correspond to the stations, i.e. *data['Measured'][0,0]* -> from *data['Stations']['Name'][0]* with *error data['Error'][0,:]* etc.

If polarity probabilities are being used, the keys are:

- PPolarityProbability
- SHPolarityProbability
- SVPolarityProbability

With a similar structure to the Polarity data, except the measured matrix has an additional dimension, i.e. *data['Measured'][0,0]* is the positive polarity probability and *data['Measured'][0,1]* is the negative polarity probability.

Amplitude ratios

- P/SHRMSAmplitudeRatio
- P/SVRMSAmplitudeRatio
- SH/SVRMSAmplitudeRatio
- P/SHQRMSAmplitudeRatio
- P/SVQRMSAmplitudeRatio
- SH/SVQRMSAmplitudeRatio
- P/SHAmplitudeRatio
- P/SVAmplitudeRatio
- SH/SVAmplitudeRatio
- P/SHQAmplitudeRatio
- P/SVQAmplitudeRatio
- SH/SVQAmplitudeRatio

Amplitude ratio observations made manually or automatically. The Q is not necessary but is useful to label the amplitudes with a Q correction. The corresponding data dictionary for amplitude ratios needs the following keys:

- *Measured*: numpy matrix of corrected numerator and denominator amplitude ratio observations, needs to have two columns, one for the numerator and one for the denominator.
- *Error*: numpy matrix of uncertainty (standard deviation) in the amplitude ratio observations, needs to have two columns, one for the numerator and one for the denominator.
- *Stations*: dictionary of station information with keys:
  - *Name*: list of station names
  - *Azimuth*: numpy matrix of azimuth values in degrees
  - *TakeOffAngle*: numpy matrix of take off angle values in degrees - 0 down (NED coordinate system).

As with all of these dictionaries, the indexes of the observations and errors must correspond to the stations, i.e. *data['Measured'][0,0]* -> from *data['Stations']['Name'][0]* with *error data['Error'][0,:]* etc.

Relative Amplitude Ratios

- PAmplitude
- SHAmplitude
- SVAmplitude
- PQAmplitude
- SHQAmplitude
- SVQAmplitude
- PRMSAmplitude
- SHRMSAmplitude
- SVRMSAmplitude
- PQRMSAmplitude
- SHQRMSAmplitude
- SVQRMSAmplitude

Relative Amplitude ratios use amplitude observations for different events made manually or automatically. The Q is not necessary but is useful to label the amplitudes with a Q correction. The corresponding data dictionary for amplitude ratios needs the following keys:

- *Measured*: numpy matrix of amplitude observations for the event.
- *Error*: numpy matrix of uncertainty (standard deviation) in the amplitude observations.
- *Stations*: dictionary of station information with keys:
  - *Name*: list of station names
  - *Azimuth*: numpy matrix of azimuth values in degrees
  - *TakeOffAngle*: numpy matrix of take off angle values in degrees - 0 down (NED coordinate system).

As with all of these dictionaries, the indexes of the observations and errors must correspond to the stations, i.e. *data['Measured'][0,0]* -> from *data['Stations']['Name'][0]* with *error data['Error'][0,:]* etc.

Angle Scatter Format The angle scatter files can be generated using a utility Scat2Angle based on the NonLinLoc angle code. The angle scatter file is a text file with samples separated by blank lines. The expected format is for samples from the location PDF that have been converted into take off and azimuth angles (in degrees) for the stations, along with a probability value. It is important that the samples are drawn from the location PDF as a Monte Carlo based integration approach is used for marginalising over the uncertainty.

It is possible to use XYZ2Angle to samples drawn from a location PDF using the NonLinLoc angle approach.

Expected format is:

```
Probability
StationName Azimuth TakeOffAngle
StationName Azimuth TakeOffAngle

Probability
.
.
.
```

With TakeOffAngle as 0 down (NED coordinate system). e.g.:

```
504.7
S0271   231.1   154.7
S0649   42.9    109.7
S0484   21.2    145.4
S0263   256.4   122.7
S0142   197.4   137.6
S0244   229.7   148.1
S0415   75.6    122.8
S0065   187.5   126.1
S0362   85.3    128.2
S0450   307.5   137.7
S0534   355.8   138.2
S0641   14.7    120.2
S0155   123.5   117
S0162   231.8   127.5
S0650   45.9    108.2
S0195   193.8   147.3
S0517   53.7    124.2
S0004   218.4   109.8
S0588   12.9    128.6
S0377   325.5   165.3
S0618   29.4    120.5
S0347   278.9   149.5
S0529   326.1   131.7
S0083   223.7   118.2
S0595   42.6    117.8
S0236   253.6   118.6

502.7
S0271   233.1   152.7
S0649   45.9    101.7
S0484   25.2    141.4
S0263   258.4   120.7
.
.
.
```

Initialisation of inversion object

Parameters

- **data** (*dict/list*) – Dictionary or list of dictionaries containing data for inversion. Can be ignored if a data_file is passed as an argument (for data format, see below).
- **data_file** (*str*[38]) – Path or list of file paths containing (binary) pickled data dictionaries.
- **location_pdf_file_path** (*str*[39]) – Path or list of file paths to angle scatter files (for file format, see above - other format extensions can be added using setuptools entry points).
- **algorithm** (*str*[40]) – ['iterate'] algorithm selector
- **parallel** (*bool*[41]) – [True] Run the inversion in parallel using multiprocessing
- **n** (*int*[42]) – [0] Number of workers, default is to use as many as returned by multiprocessing.cpu_count
- **phy_mem** (*int*[43]) – [8] Estimated physical memory to use (used for determining array sizes, it is likely that more memory will be used, and if so no errors are forced). *On python versions <2.7.4 there is a bug (http://bugs.python.org/issue13555) with pickle that limits the total number of samples when running in parallel, so large (>20GB) phy_mem allocations per process are ignored.*
- **dc** (*bool*[44]) – [False] Boolean flag as to run inversion constrained to double-couple or allowed to explore the full moment tensor space.

Keyword Arguments

- **number_stations** (*int*[45]) – [0] Used for estimating sample sizes in the Monte Carlo random sampling algorithms (Time,Iterate) if set.
- **number_location_samples** (*int*[46]) – [0] Used for estimating sample sizes in the Monte Carlo random sampling algorithms (Time,Iterate) if set.
- **path** (*str*[47]) – File path for output. Default is current working dir (interactive) or PBS workdir.
- **fid** (*str*[48]) – File name root for output, default is to use mtfitOutput or the event UID if set in the data dictionary.
- **inversion_options** (*list*) – List of data types to be used in the iversion, if not set, the inversion uses all the data types in the data dictionary, irrespective of independence.
- **diagnostic_output** (*bool*[49]) – [False] Boolean flag to output diagnostic information in MATLAB save file.
- **marginalise_relative** (*bool*[50]) – [False] Boolean flag to marginalise over location/model uncertainty during relative amplitude inversion.
- **mpi** (*bool*[51]) – [False] Boolean flag to run using MPI from mpi4py (useful on cluster).
- **multiple_events** (*bool*[52]) – [False] Boolean flag to run all the events in the inversion in one single joint inversion.

[38] https://docs.python.org/2/library/functions.html#str
[39] https://docs.python.org/2/library/functions.html#str
[40] https://docs.python.org/2/library/functions.html#str
[41] https://docs.python.org/2/library/functions.html#bool
[42] https://docs.python.org/2/library/functions.html#int
[43] https://docs.python.org/2/library/functions.html#int
[44] https://docs.python.org/2/library/functions.html#bool
[45] https://docs.python.org/2/library/functions.html#int
[46] https://docs.python.org/2/library/functions.html#int
[47] https://docs.python.org/2/library/functions.html#str
[48] https://docs.python.org/2/library/functions.html#str
[49] https://docs.python.org/2/library/functions.html#bool
[50] https://docs.python.org/2/library/functions.html#bool
[51] https://docs.python.org/2/library/functions.html#bool
[52] https://docs.python.org/2/library/functions.html#bool

- **relative_amplitude** (*bool*[53]) – [False] Boolean flag to run multiple_events including relative amplitude inversion.
- **output_format** (*str*[54]) – ['matlab'] str format style.
- **minimum_number_intersections** (*int*[55]) – [2] Integer minimum number of station intersections between events in relative amplitude inversion.
- **quality_check** (*bool*[56]) – [False] Boolean flag/Float value for maximum non-zero percentage check (stops inversion if event quality is poor)
- **recover** (*bool*[57]) – [False] Tries to recover pre-existing inversions, and does not re-run if an output file exists and is readable.
- **file_sample** (*bool*[58]) – [False] Saves samples to a mat file (allowing for easier recovery and reduced memory requirements but can slow the inversion down as requires disk writing every non-zero iteration. Only works well for recovery with random sampling methods)
- **results_format** (*str*[59]) – ['full_pdf'] Data format for the output
- **marginalise_relative** – [False] Boolean flag to marginalise the location uncertainty between absolute and relative data.
- **file_sample** – [False] save samples to file rather than keeping in memory (not necessary in normal use).
- **normalise** (*bool*[60]) – [True] normalise output Probability (doesn't affect ln_pdf output).
- **convert** (*bool*[61]) – Convert output moment tensors to Tape parameters, Hudson u&v coordinates and strike-dip-rake triples.
- **discard** (*bool*[62]) – [False] Probability cut-off for discarding samples Discarding samples - samples less than 1/(discard*n_samples) of the maximum likelihood value are discarded as negligeable. False means no samples are discarded.
- **c_generate** (*bool*[63]) – [False] Generate samples in the probability calculation when using Cython.
- **generate_cutoff** (*int*[64]) – Set number of samples to cut-off at when using c_generate (Default is the value of max_samples)
- **relative_loop** (*bool*[65]) – [False] Loop over non-zero samples when using relative amplitudes.
- **bin_angle_coefficient_samples** (*int*[66]) – [0] Bin size in degrees when binning angle coefficients (All station angle differences must be within this range for samples to fall in the same bin)
- **no_station_distribution** (*bool*[67]) – [True] Boolean flag to output station distribution or not.

[53] https://docs.python.org/2/library/functions.html#bool
[54] https://docs.python.org/2/library/functions.html#str
[55] https://docs.python.org/2/library/functions.html#int
[56] https://docs.python.org/2/library/functions.html#bool
[57] https://docs.python.org/2/library/functions.html#bool
[58] https://docs.python.org/2/library/functions.html#bool
[59] https://docs.python.org/2/library/functions.html#str
[60] https://docs.python.org/2/library/functions.html#bool
[61] https://docs.python.org/2/library/functions.html#bool
[62] https://docs.python.org/2/library/functions.html#bool
[63] https://docs.python.org/2/library/functions.html#bool
[64] https://docs.python.org/2/library/functions.html#int
[65] https://docs.python.org/2/library/functions.html#bool
[66] https://docs.python.org/2/library/functions.html#int
[67] https://docs.python.org/2/library/functions.html#bool

- **max_samples** (*int*[68]) – [6000000] Max number of samples when using the iterate algorithm.
- **max_time** (*int*[69]) – [600] Max time when using the time algorithm.
- **verbosity** (*int*[70]) – [0] Set verbosity level (0-4) high numbers mean more logging output and verbosity==4 means the debugger will be called on errors.
- **debug** (*bool*[71]) – [False] Sets debug on or off (True means verbosity set to 4).

Other kwargs are passed to the algorithm - see mtfit.algorithms documentation for help on those.

forward()

Runs event forward model using the arguments when the Inversion object was initialised.

Depending on the algorithm selection uses either random sampling or Markov chain Monte Carlo sampling - for more information on the different algorithms see mtfit.algorithms documentation.

[68] https://docs.python.org/2/library/functions.html#int
[69] https://docs.python.org/2/library/functions.html#int
[70] https://docs.python.org/2/library/functions.html#int
[71] https://docs.python.org/2/library/functions.html#bool

CHAPTER 14

Extending mtfit

mtfit has been written with the view that it is desirable to be able to easily extend the code. This is done using entry points[72] from the setuptools[73] module.

The entry points are:

Entry Point	Descriptions
mtfit.cmd_opts	Command line options
mtfit.cmd_defaults	Default parameters for the command line options
mtfit.tests	Test functions for the extensions
mtfit.pre_inversion	Function to be called with all kwargs before the inversion object is initialised
mtfit.post_inversion	Function to be called with all available kwargs after the inversion has occurred
mtfit.extensions	Functions that replaces the call to the inversion using all the kwargs
mtfit.parsers	Functions that return the data dictionary from an input filename
mtfit.location_pdf_parsers	Functions that return the location PDF samples from an input filename
mtfit.output_data_formats	Functions that format the output data into a given type, often linked to the output format
mtfit.output_formats	Functions that output the results from the output_data_formats
mtfit.process_data_types	Functions to convert input data into correct format for new data types in forward model
mtfit.data_types	Functions to evaluate the forward model for new data types
mtfit.parallel_algorithms	Search algorithms that can be run (in parallel) like monte carlo random sampling
mtfit.directed_algorithms	Search algorithms that are dependent on the previous value (e.g. McMC)
mtfit.sampling	Function that generates new moment tensor samples in the Monte Carlo random sampling algorithm
mtfit.sampling_prior	Function that calculates the prior either in the McMC algorithm or the MC bayesian evidence estimate
mtfit.sample_distribution	Function that generates random samples according to some source model
mtfit.plot	Callable class for source plotting using matplotlib
mtfit.plot_read	Function that reads the data from a file for the MTplot class
mtfit.documentation	Installs the documentation for the extension
mtfit.source_code	Installs the source code documentation for the extension

These entry points can be accessed by adding some arguments to the `setuptools` module `setup.py` script:

[72] https://pythonhosted.org/setuptools/pkg_resources.html#entry-points
[73] https://pythonhosted.org/setuptools

```
kwargs['entry_points']={'entry_point_name': ['key = function']}
```

Where `kwargs` is the keyword dictionary passed to the `setuptools setup()` function, and the `entry_point_name` is the desired entry point in the other package. The `key` is the description of the `function()`, used for selecting it in the code (this should be described by the package), and the `function()` is the desired function to be called when this key is selected.

The different usages for these entry points are described below.

`extensions/scatangle.py` is an example extension structure, although it would be necessary to make a `setup.py` file to install it.

14.1 mtfit.cmd_opts

This entry point handles command line options for extensions that have been added. It is called when parsing the command line options, and should not conflict with the options described in *mtfit command line options*.

The function is called as:

```
parser_group, parser_check = cmd_opts(parser_group, argparse=[True/False],␣
↪defaults)
```

Where the `parser_group` is the `argparse`[74] or `optparse`[75] parser group depending on if `argparse`[76] is installed (Python version 2.7 or later), defaults are the command line defaults (with corresponding entry points *mtfit.cmd_defaults*), and `parser_check` is the function called to check/process the parsers results.

An example cmd_opts function is:

```
def parser_check(parser,options,defaults):
    flags=[]
    if options['bin_scatangle']:
        if not options['location_pdf_file_path']:
          options['location_pdf_file_path']=glob.glob(options['data_file']+\
                os.path.sep+'*'+options['angle_extension'])
        if not type(options['location_pdf_file_path'])==list:
          options['location_pdf_file_path']=[options['location_pdf_file_path']]
        flags=['no_location_update']
    return options,flags

def cmd_opts(group,argparse=ARGPARSE,defaults=PARSER_DEFAULTS):
    if argparse:
        group.add_argument("--bin-scatangle","--binscatangle","--bin_scatangle", \
            action="store_true",default=defaults['bin_scatangle'], \
            help="Bin the scatangle file to reduce the number of samples \
                [default=False]. --bin-size Sets the bin size parameter .", \
            dest="bin_scatangle")
        group.add_argument("--bin-size","--binsize","--bin_size",type=float, \
            default=defaults['bin_size'],help="Sets the scatangle bin size␣
↪parameter \
                [default="+str(defaults['bin_size'])+"].",dest="bin_scatangle_size
↪")
    else:
        group.add_option("--bin-scatangle","--binscatangle","--bin_scatangle", \
            action="store_true",default=defaults['bin_scatangle'],help="Bin the \
                scatangle file to reduce the number of samples [default=False]. \
                --bin-size Sets the bin size parameter .",dest="bin_scatangle")
        group.add_option("--bin-size","--binsize","--bin_size",type=float, \
            default=defaults['bin_size'],help="Sets the scatangle bin size \
```

[74] https://docs.python.org/2/library/argparse.html#module-argparse
[75] https://docs.python.org/2/library/optparse.html#module-optparse
[76] https://docs.python.org/2/library/argparse.html#module-argparse

```
                parameter [default="+str(defaults['bin_size'])+"].", \
            dest="bin_scatangle_size")
    return group,parser_check
```

This is taken from `extensions/scatangle.py`.

These command line options will be added to the options mtfit is called with so can then be parsed by other functions in the extension.

The command line options for an extension can be installed using `setuptools` by adding the `mtfit.cmd_opts` entry point to the extension `setup.py` script:

```
setup(...
      entry_points = {'mtfit.cmd_opts': ['extension = mymodule:cmd_opts']}
      ...)
```

14.2 mtfit.cmd_defaults

This entry point handles the default values and types for the command line options described in *mtfit.cmd_opts*. It is called when parsing the command line options.

The function is called as:

```
plugin_defaults, plugin_default_types = cmd_defaults()
```

Where both are dicts, and should contain defaults for the *mtfit.cmd_opts*, although they can also update the normal *mtfit command line options* defaults and default types. Both dictionaries are used for updating the defaults from the default file (see *Installing mtfit*).

An example cmd_defaults function is:

```
PARSER_DEFAULTS={
          'bin_scatangle': False,
          'bin_size': 1.0,
          }
PARSER_DEFAULT_TYPES = {'bin_scatangle': [bool], 'bin_size': [float]}

def cmd_defaults():
    return(PARSER_DEFAULTS, PARSER_DEFAULT_TYPES)
```

This is taken from `extensions/scatangle.py`.

The default command line options for an extension can be installed using `setuptools` by adding the `mtfit.cmd_defaults` entry point to the extension `setup.py` script:

```
setup(...
      entry_points = {'mtfit.cmd_defaults': ['extension = mymodule:cmd_defaults']}
      ...)
```

14.3 mtfit.tests

This entry point is used for any extensions to add tests to the test suite, which can be run using `mtfit --test` on the command line, or as `mtfit.run_tests()` from within python.

The function is called as:

```
test_suite = tests()
```

Where `test_suite` is the `unittest.TestSuite`[77] containing the TestSuite, created as:

```
tests=[]
tests.append(unittest.TestLoader().loadTestsFromTestCase(__ExtensionTestCase))
test_suite=unittest.TestSuite(tests)
```

from each `unittest.TestCase`[78].

A test suite for an extension can be installed using `setuptools` by adding the `mtfit.tests` entry point to the extension `setup.py` script:

```
setup(...
      entry_points = {'mtfit.tests': ['extension = mymodule:tests']}
      ...)
```

(N.B. the different test suites can be empty).

14.4 mtfit.pre_inversion

This entry point provides an opportunity to call a function before the *`mtfit.inversion.Inversion`* object is created (e.g. for some additional data processing).

The plugin is called as:

```
kwargs = pre_inversion(**kwargs)
```

And can change the kwargs passed to the *`Inversion`* object to create it.

The function should just return the initial kwargs if the command line option to select it is not `True`, otherwise it will always be called.

An pre_inversion function can be installed using `setuptools` by adding the `mtfit.pre_inversion` entry point to the extension `setup.py` script:

```
setup(...
      entry_points = {
            'mtfit.pre_inversion': ['my_fancy_function = mymodule:main_function'],
            'mtfit.cmd_opts': ['extension = mymodule:cmd_opts'],
            'mtfit.cmd_defaults': ['extension = mymodule:cmd_defaults']}
      ...)
```

Where the *mtfit.cmd_opts* and *mtfit.cmd_defaults* entry points have been included.

14.5 mtfit.post_inversion

This entry point provides an opportunity to call a function after the *`mtfit.inversion.Inversion`* object is created (e.g. for some additional data processing).

The plugin is called as:

```
post_inversion(**kwargs)
```

The function should just return nothing if the command line option to select it is not `True`, otherwise it will always be called.

An post_inversion function can be installed using `setuptools` by adding the `mtfit.post_inversion` entry point to the extension `setup.py` script:

[77] https://docs.python.org/2/library/unittest.html#unittest.TestSuite
[78] https://docs.python.org/2/library/unittest.html#unittest.TestCase

```
setup(...
      entry_points = {
            'mtfit.post_inversion': ['my_fancy_function = mymodule:main_function'],
            'mtfit.cmd_opts': ['extension = mymodule:cmd_opts'],
            'mtfit.cmd_defaults': ['extension = mymodule:cmd_defaults']}
      ...)
```

Where the *mtfit.cmd_opts* and *mtfit.cmd_defaults* entry points have been included.

14.6 mtfit.extensions

This entry point allows functions that can replace the main call to the *mtfit.inversion.Inversion* object and to the *mtfit.inversion.Inversion.forward()* function.

The plugin is called as:

```
result = ext(**kwargs)
if result != 1
    return result
```

Where kwargs are all the command line options that have been set.

If the result of the extension is `1` the program will not exit (this should be the case if the kwargs option to call the extension is not True), otherwise it exits.

N.B it is necessary for an extension to also have installed functions for the entry points:

- *mtfit.cmd_opts*,
- *mtfit.cmd_defaults*,

and the function should check if the appropriate option has been selected on the command line (if it doesn't it will always run).

An extension function can be installed using `setuptools` by adding the `mtfit.extensions` entry point to the extension `setup.py` script:

```
setup(...
      entry_points = {
            'mtfit.extensions': ['my_fancy_function = mymodule:main_function'],
            'mtfit.cmd_opts': ['extension = mymodule:cmd_opts'],
            'mtfit.cmd_defaults': ['extension = mymodule:cmd_defaults']}
      ...)
```

Where the *mtfit.cmd_opts* and *mtfit.cmd_defaults* entry points have been included.

14.7 mtfit.parsers

The *mtfit.parsers* entry point allows additional input file parsers to be added. The CSV parser is added using this in the `setup.py` script:

```
kwargs['entry_points'] = {'mtfit.parsers': ['.csv = mtfit.inversion:parse_csv']}
```

`mtfit` expects to call the plugin (if the data-file extension matches) as:

```
data = plugin(filename)
```

A parser for a new file format can be installed using `setuptools` by adding the `mtfit.parsers` entry point to the extension `setup.py` script:

```
setup(...
      entry_points = {
            'mtfit.parsers':
                ['.my_format = mymodule.all_parsers:my_format_parser_function']
            }
      ...
      )
```

The parser is called using:

```
data = my_new_format_parser_function(filename)
```

Where the `filename` is the data filename and `data` is the data dictionary (see *Creating a Data Dictionary*).

When a new parser is installed, the format (.my_new_format) will be called if it corresponds to the data-file extension. However if the extension doesn't match any of the parsers it will try all of them.

14.8 mtfit.location_pdf_parsers

This entry point allows additional location PDF file parsers to be added

`mtfit` expects to call the plugin (if the extension matches) as:

```
location_samples,location_probability=plugin(filename,number_station_samples)
```

Where number_station_samples is the number of samples to use (i.e subsampling if there are more samples in the location PDF).

A parser for a new format can be installed using `setuptools` by adding the `mtfit.location_pdf_parsers` entry point to the extension `setup.py` script:

```
setup(...
      entry_points = {
            'mtfit.location_pdf_parsers':
                ['.my_format = mymodule.all_parsers:my_format_parser_function']
        }
      ...)
```

The parser is called using:

```
location_samples,location_probability=my_format_parser_function(filename,
        number_location_samples)
```

Where the `filename` is the location PDF filename and `number_location_samples` is the number of samples to use (i.e subsampling if there are more samples in the location PDF).

The expected format for the location_samples and location_probability return values are:

```
location_samples = [
    {'Name': ['S01', 'S02', ...], 'Azimuth': np.matrix([[121.], [37.], ...]),
        'TakeOffAngle': np.matrix([[88.], [12.], ...])},
     {'Name': ['S01', 'S02', ...],'Azimuth': np.matrix([[120.], [36.], ...]),
        'TakeOffAngle': np.matrix([[87.], [11.], ...])}
    ]
location_probability=[0.8,1.2, ...]
```

These are then used in a Monte Carlo method approach to include the location uncertainty in the inversion (see *Bayesian Approach*).

When a new parser is installed, the format (.my_new_format) will be called if it corresponds to the data-file extension. However if the extension doesn't match any of the parsers it will try all of them.

14.9 mtfit.output_data_formats

A parser for a new output data format can be installed using `setuptools` by adding the `mtfit.output_data_formats` entry point to the extension `setup.py` script:

```
setup(...
      entry_points = {
            'mtfit.output_data_formats':
                ['my_format = mymodule.all_parsers:my_output_data_function']
            }
      ...)
```

The parser is called using:

```
output_data = my_output_data_function(event_data, self.inversion_options,
    output_data, location_samples, location_sample_multipliers,
    self.multiple_events, self._diagnostic_output, *args, **kwargs)
```

Where the `event_data` is the dictionary of event data, `self.inversion_options` are the inversion options set using t
samples described above, and the `multiple_events` and `_diagnostic_output` are corresponding boolean flags.

The format is set using the `--resultsformat` command line argument (see *mtfit command line options*) or the `results_format` function argument when initialising the *`Inversion`* object.

The resulting `output_data` is normally expected to be either a dictionary to be passed to the output_format function to write to disk, or a pair of dictionaries (`list`). However it is passed straight through to the output file format function so it is possible to have a custom `output_data` object that is then dealt with in the output file formats function (see *mtfit.output_formats*). When a new parser is installed, the format (`my_format`) will be added to the possible result formats on the command line (`--resultsformat` option in *mtfit command line options*).

14.10 mtfit.output_formats

mtfit has an entry point for the function that outputs the results to a specific file format.

The function outputs the results from the *output_data_formats function* and returns a string to be printed to the terminal and the output filename (it should change the extension as required) e.g.:

```
out_string,filename=output_formatter(out_data,filename,JobPool,*args,**kwargs)
```

`JobPool` is a `mtfit.inversion.JobPool`, which handles job tasking if the inversion is being run in parallel. It can be passed a task (callable object) to write to disk in parallel.

The format is set using the `--format` command line argument (see *mtfit command line options*) or the `format` function argument when initialising the *`Inversion`* object.

A new format can be installed using `setuptools` by adding the `mtfit.output_formats` entry point to the extension `setup.py` script:

```
setup(...
      entry_points = {
            'mtfit.output_formats':
                ['my_format = mymodule.all_parsers:my_output_format_function']
            }
      ...)
```

The parser is called using:

```
output_string, fname = my_output_format_function(output_data,
        fname, pool, *args, **kwargs)
```

Where the `fname` is the output filename and `output_data` is the output data from the output data parser (see :ref:entry_point-8‘). `pool` is the `mtfit.inversion.JobPool`.

When a new parser is installed, the format (`my_format`) will be added to the possible output formats on the command line (`--format` option in *mtfit command line options*).

14.11 mtfit.process_data_types

A function to process the data from the input data to the correct format for an *mtfit.data_types* extension. This can be installed can be installed using `setuptools` by adding the `mtfit.process_data_types` entry point to the extension `setup.py` script:

```
setup(...
      entry_points = {
            'mtfit.process_data_types':
                ['my_data_type = mymodule.all_parsers:my_data_type_preparation']
            }
      ...)
```

The function is called using:

```
extension_data_dict = extension_function(event)
```

where event is the data dictionary (keys correspond to different data types and the settings of the inversion_options parameter). The function returns a dict, with the station coefficients having keys `a_***` or `aX_***` where `X` is a single identifying digit. These station coefficients are a 3rd rank numpy array, with the middle index corresponding to the location samples.

14.12 mtfit.data_types

A function to evaluate the forward model likelihood for a new data-type. This can be installed can be installed using `setuptools` by adding the `mtfit.data_types` entry point to the extension `setup.py` script:

```
setup(...
      entry_points = {
            'mtfit.data_types':
                ['my_data_type = mymodule.all_parsers:my_data_type_likelihood']
            }
      ...)
```

The inputs are prepared using an *mtfit.process_data_types* extension.

The function is called using:

```
ln_pdf = extension_function(self.mt, **self.ext_data[key])
```

where `self.ext_data[key]` is the data prepared by the *mtfit.process_data_types* function for this extension. The `mt` variable is a numpy array of moment tensor six vectors in the form:

```
self.mt = np.array([[m11, ...],
                    [m22, ...],
                    [m33, ...],
                    [sqrt(2)*m12, ...],
                    [sqrt(2)*m13, ...],
                    [sqrt(2)*m23, ...]])
```

The station coefficients for the extension should be named as `a_***` or `aX_***` where `X` is a single identifying digit, and be a 3rd rank numpy array, with the middle index corresponding to the location samples. The function

returns a `mtfit.probability.LnPDF` for the moment tensors provided. If the function does not exist, an error is raised, and the result ignored.

The function should handle any c/cython calling internally.

> **Warning:** It is assumed that the data used is independent, but this must be checked by the user.

Relative inversions can also be handled, but the extension name requires `relative` in it.

Relative functions are called using:

```
ln_pdf, scale, scale_uncertainty = extension_function(self.mt, ext_data_1, ext_
↪data_2)
```

Where `ext_data_*` is the extension data for each event as a dictionary. This dictionary, generated using the *mtfit.process_data_types* function for this extension, should also contain a list of the receivers with observations, ordered in the same order as the numpy array of the data, as this is used for station indexing.

The `scale` and `scale_uncertainty` return variables correspond to estimates of the relative seismic moment between the two events, if it is generated by the extension function (if this is not estimated, `1.` and `0.` should be returned)

14.13 mtfit.parallel_algorithms

This extension provides an entry point for customising the search algorithm. This can be installed using `setuptools` by adding the `mtfit.parallel_algorithms` entry point to the extension `setup.py` script:

```
setup(...
      entry_points = {
            'mtfit.parallel_algorithms':
                ['my_new_algorithm = mymodule:my_new_algorithm_class']
            }
      ...)
```

The algorithm should inherit from `mtfit.algorithms.monte_carlo_random._MonteCarloRandomSample`, or have the functions `initialise()`, `iterate()`, `__output__()` and attributes `iteration`, `start_time`, and `pdf_sample` as a `mtfit.sampling.Sample` or `mtfit.sampling.FileSample` object.

The `mtfit.parallel_algorithms` entry point is for algorithms to replace the standard Monte Carlo random sampling algorithm, which can be called and run in parallel to generate new samples - see `mtfit.inversion._random_sampling_forward()`.

The algorithm is initialised as:

```
algorithm = extension_algorithm(**kwargs)
```

where `kwargs` are the input arguments for the inversion object, and a few additional parameters such as the number of samples (`number_samples`), which is the number of samples per iteration, accounting for memory. Additional `kwargs` can be added using the *mtfit.cmd_opts* entry point.

The algorithm will be initialised, and expected to return the moment tensors to check in the forward model, and `end=True`:

```
mts, end = self.algorithm.initialise()
```

`end` is a boolean flag to determine whether the end of the search has been reached, and mts is the numpy array of moment tensors in the form:

```
mts = np.array([[m11, ...],
                [m22, ...],
                [m33, ...],
                [sqrt(2)*m12, ...],
                [sqrt(2)*m13, ...],
                [sqrt(2)*m23, ...]])
```

After initialisation, the results are returned from the `mtfit.inversion.ForwardTask` object as a dictionary which should be parsed using the `iterate()` function:

```
mts, end = self.algorithm.iterate({'moment_tensors': mts, 'ln_pdf': ln_p_total, 'n
↪': N})
```

The forward models can be run in parallel, either using `multiprocessing`[79] or using MPI to pass the `end` flag. Consequently, these algorithms have no ordering, so can not depend on previous samples - to add an algorithm that is, it is necessary to use the *mtfit.directed_algorithms* entry point.

14.14 mtfit.directed_algorithms

This extension provides an entry point for customising the search algorithm. This can be installed can be installed using `setuptools` by adding the `mtfit.directed_algorithms` entry point to the extension `setup.py` script:

```
setup(...
      entry_points = {
            'mtfit.directed_algorithms':
                ['my_new_algorithm = mymodule:my_new_algorithm_class']
            }
      ...)
```

The algorithm should inherit from `mtfit.algorithms.__base__._BaseAlgorithm`, or have the functions `initialise()`, `iterate()`, `__output__()` and attribute `pdf_sample` as a `mtfit.sampling.Sample` or `mtfit.sampling.FileSample` object.

The `mtfit.directed_algorithms` entry point is for algorithms to replace the Markov chain Monte Carlo sampling algorithms - see `mtfit.inversion._mcmc_sampling_forward()`, using an `mtfit.inversion.MCMCForwardTask` object

The algorithm is initialised as:

```
algorithm = extension_algorithm(**kwargs)
```

where `kwargs` are the input arguments for the inversion object, and a few additional parameters such as the number of samples (`number_samples`), which is the number of samples per iteration, accounting for memory. Additional `kwargs` can be added using the *mtfit.cmd_opts* entry point.

The algorithm will be initialised, and expected to return the moment tensors to check in the forward model, and `end=True`:

```
mts, end = self.algorithm.initialise()
```

`end` is a boolean flag to determine whether the end of the search has been reached, and `mts` is the numpy array of moment tensors in the form:

```
mts = np.array([[m11, ...],
                [m22, ...],
                [m33, ...],
                [sqrt(2)*m12, ...],
```

[79] https://docs.python.org/2/library/multiprocessing.html#module-multiprocessing

```
                [sqrt(2)*m13, ...],
                [sqrt(2)*m23, ...]])
```

After initialisation, the results are returned from the `mtfit.inversion.ForwardTask` object as a dictionary which should be parsed using the iterate function:

```
mts, end = self.algorithm.iterate({'moment_tensors': mts,'ln_pdf': ln_p_total, 'n
↪': N})
```

The forward models are run in order, so can depend on previous samples - to add an algorithm that does not need this, use the *mtfit.parallel_algorithms* entry point.

14.15 mtfit.sampling

This extension provides an entry point for customising the moment tensor sampling used by the search algorithm. This can be installed can be installed using `setuptools` by adding the `mtfit.sampling` entry point to the extension `setup.py` script:

```
setup(...
      entry_points = {
            'mtfit.sampling':
                ['my_extension_name = mymodule:my_source_sampling']
            }
      ...)
```

The function should return a numpy array or matrix of normalised moment tensor six vectors in the form:

```
mts = np.array([[m11, ...],
                [m22, ...],
                [m33, ...],
                [sqrt(2)*m12, ...],
                [sqrt(2)*m13, ...],
                [sqrt(2)*m23, ...]])
```

If an alternate sampling is desired for the McMC case (ie. a different model), it is necessary to extend the algorithm class using the `mtfit.directed_algorithms` entry point.

14.16 mtfit.sampling_prior

This extension provides an entry point for customising the prior distribution of moment tensors used by the search algorithm. This can be installed can be installed using `setuptools` by adding the `mtfit.sampling_prior` entry point to the extension `setup.py` script:

```
setup(...
      entry_points = {
            'mtfit.sampling_prior':
                ['my_extension_name = mymodule:my_sampling_prior']
            }
      ...)
```

Different functions should be chosen for the Monte Carlo algorithms compared to the Markov chain Monte Carlo algorithms. In the Monte Carlo case, the prior is used to calculate the Bayesian evidence, and depends on the source type parameters. It must reflect the prior distribution on the source samples as a Monte Carlo type integration is used to calculate it, and should return a float from two input floats:

```
prior = prior_func(gamma, delta)
```

In the Markov chain Monte Carlo case, the function should return the prior of a sample, dependent on the selected model, again as a float. It is called as:

```
prior = uniform_prior(xi, dc=None, basic_cdc=False, max_poisson=0, min_poisson=0)
```

where xi is a dictionary of the sample parameters e.g.:

```
xi = {'gamma': 0.1, 'delta': 0.3, 'kappa': pi/2, 'h': 0.5, 'sigma': 0}
```

If an alternate sampling is desired for the Markov chain Monte Carlo case (ie. a different model), it is necessary to extend the algorithm class using the `mtfit.directed_algorithms` entry point.

14.17 mtfit.sample_distribution

This extension provides an entry point for customising the source sampling used by the Monte Carlo search algorithm. This can be installed can be installed using `setuptools` by adding the `mtfit.sample_distribution` entry point to the extension `setup.py` script:

```
setup(...
      entry_points = {
            'mtfit.sample_distribution':
                ['my_extension_name = mymodule:my_random_model_func']
            }
      ...)
```

The model must generate a random sample according in the form of a numpy matrix or array:

```
mts = np.array([[m11, ...],
                [m22, ...],
                [m33, ...],
                [sqrt(2)*m12, ...],
                [sqrt(2)*m13, ...],
                [sqrt(2)*m23, ...]])
```

If an alternate sampling is desired for the Markov chain Monte Carlo case (ie. a different model), it is necessary to extend the algorithm class using the `mtfit.directed_algorithms` entry point.

14.18 mtfit.plot

This extension provides an entry point for customising the plot type the for mtfit.plot.MTplot object. This can be installed can be installed using `setuptools` by adding the `mtfit.plot` entry point to the extension `setup.py` script:

```
setup(...
      entry_points = {
            'mtfit.plot':
                ['plottype = mymodule:my_plot_class']
            }
      ...)
```

The object should be a callable object which can accept the moment tensor 6-vector, matplotlib figure, matplotlib grid_spec and other arguments (see the `mtfit.plot.plot_classes._BasePlot` class for an example), with the `__call__()` function corresponding to plotting the moment tensor.

The plottype name in the setup.py script should be lower case with no spaces, hypens or underscores (these are removed in parsing the plottype).

14.19 mtfit.plot_read

This extension provides an entry point for customising the input file parser for reading data for the mtfit.plot.MTplot object. This can be installed can be installed using `setuptools` by adding the `mtfit.plot_read` entry point to the extension `setup.py` script:

```
setup(...
      entry_points = {
            'mtfit.plot_read':
                ['.file_extension = mymodule:my_read_function']
            }
      ...)
```

The function should accept an input filename and return a tuple of dicts for event and station data respectively

14.20 mtfit.documentation

This extension provides an entry point for customising the search algorithm. This can be installed can be installed using `setuptools` by adding the `mtfit.documentation` entry point to the extension `setup.py` script:

```
setup(...
      entry_points = {
            'mtfit.documentation':
                ['my_extension_name = mymodule:my_rst_docs']
            }
      ...)
```

The function should return a ReST string that can be written out when building the documentation using `sphinx`.

The name should be the extension name with _ replacing spaces. This will be capitalised into the link in the documentation.

14.21 mtfit.source_code

This extension provides an entry point for customising the search algorithm. This can be installed can be installed using `setuptools` by adding the `mtfit.source_code` entry point to the extension `setup.py` script:

```
setup(...
      entry_points = {
            'mtfit.source_code':
                ['my_extension_name = mymodule:my_rst_source_code_docs']
            }
      ...)
```

The function should return a `ReST` string that can be written out when building the documentation using `sphinx`.

The name should be the extension name with _ replacing spaces. This will be capitalised into the link in the documentation.

CHAPTER 15

References

For a more comprehensive set of references please see *Pugh (2015)*.

Bayes, T and Price, R, 1763. **An Essay towards Solving a Problem in the Doctrine of Chances. By the Late Rev. Mr. Bayes F. R. S. Communicated by Mr. Price in a Letter to John Canton A. M. F. R. S.**, *Philosophical Transactions of the Royal Society of London*.

Bernth, H., and Chapman, C., 2011, **A comparison of the dispersion relations for anisotropic elastodynamic finite-difference grids**, *Geophysics*, 76(3), WA43–WA50.

Beyreuther, M., Barsch, R., Krischer, L., Megies, T., Behr, Y., & Wassermann, J., 2010. **ObsPy: a Python toolbox for seismology**, *SRL Electron. Seismol.*.

Chapman, C. H. and Leaney, W. S., 2011. **A new moment-tensor decomposition for seismic events in anisotropic media**, *GJI*, 188(1), 343-370.

Green, P J, 1995. **Reversible jump Markov chain Monte Carlo computation and Bayesian model determination**, *Biometrika*, 82(4), 711-732.

Greenfield, T. & White, R. S., 2015. **Building Icelandic Igneous Crust by Repeated Melt Injections**, *J. Geophys. Res.*, 120(11), 7771-7788.

Hardebeck, J. L. & Shearer, P. M., 2002. **A new method for determining first-motion focal mechanisms**, *Bull. Seismol. Soc. Am.*, 92(6), 2264-2276.

Hardebeck, J. L. & Shearer, P. M., 2003. **Using S / P amplitude ratios to constrain the focal mechanisms of small earthquakes**, *Bull. Seismol. Soc. Am.*, 93(6), 2434-2444.

Hastings, W K, 1970. **Monte Carlo sampling methods using Markov chains and their applications**, *Biometrika*, 57(1), 97-109.

Hinkley, D V, 1969. **On the ratio of two correlated normal random variables**, *Biometrika*, 56(3), 635-639.

Hudson, J. A., R. G. Pearce, and R. M. Rogers, 1989. **Source type plot for inversion of the moment tensor**, *J. Geophys. Res.*, 94(B1), 765–774.

Laplace, P S, 1812. **Théorie analytique des probabilités**.

Markov, A A, 1954. **Theory of Algorithms**, *Moscow Academy of Sciences of the USSR* [*Israel Program for Scientific Translations for the National Science Foundation, Washington, D.C., 1961*].

Metropolis, N, Rosenbluth, A W, Rosenbluth, M N, Teller, A H, & Teller, E, 1953. **Equation of State Calculations by Fast Computing Machines**, *The Journal of Chemical Physics* 21 pp 1087-1092.

Norris, J R, 1998. **Markov Chains**, *Cambridge University Press*.

Pugh, D J, 2015, **Bayesian Source Inversion of Microseismic Events**, *PhD Thesis, Department of Earth Sciences, University of Cambridge*.

Pugh, D J, White, R S and Christie, P A F, 2016a, **A Bayesian method for microseismic source inversion**, *GJI*, 206(2), 1009-1038.

Pugh, D J, White, R S and Christie, P A F, 2016b, **Automatic Bayesian polarity determination**, *GJI*, 206(1), 275-291.

Pugh, D J, White, R S and Christie, P A F, 2017 **MTfit: A Bayesian Approach to Moment Tensor Inversion**, *SRL Electron. Seismol.*, *in prep*.

Reasenberg, P. A. & Oppenheimer, D., 1985. **FPFIT, FPPLOT and FPPAGE: Fortran computer programs for calculating and displaying earthquake fault-plane solutions** *OFR 85-739, Tech. rep., USGS*.

Riedesel, M. A., and T. H. Jordan, 1989. **Display and assessment of seismic moment tensors**, *Bull. Seismol. Soc. Am.*, 79(1), 85–100

Sivia, D S, 2000. **Data Analysis: A Bayesian Tutorial**, *Oxford Univ. Press*.

Snoke, J. A., 2003. **FOCMEC: FOCal MEChanism determinations**, *Tech. rep.*.

Tape, C and Tape, W, 2012. **A geometric setting for moment tensors**, *GJI*, 190(1) 476-498.

Mildon, Z K, Pugh D J, Tarasewicz, J, White, R S, Brandsdóttir, B, 2015. **Closing crack earthquakes within the Krafla caldera, North Iceland**, *GJI*, 207, 1137-1141.

Wilks, M., Bradford, I., Williams, M., Rodriguez, I. V., & Pugh, D., 2015. **Combined use of a deep monitoring array and shallow borehole arrays for moment tensor inversion**, in *77th EAGE Conf. Exhib.*.