AIMS Documentation

Release 2.2.0

Daniel R. Reese

Apr 30, 2025

CONTENTS

CHAPTER

ONE

CONTENTS

1.1 Project Summary

1.1.1 Description

Name: "Asteroseismic Inference on a Massive Scale" (*AIMS*)

Goals:

- estimate stellar parameters and credible intervals/error bars
- chose a representative set or sample of reference models
- be computationally efficient

Inputs:

- classic constraints and error bars (Teff, L, ...)
- seismic constraints and error bars (individual frequencies)

Requirements:

- a *precalculated* grid of models including:
  - the models themselves
  - parameters for the model (M, R, Teff, age, ...)
  - theoretical frequency spectra for the models

Methodology:

- applies an MCMC algorithm based on the python package emcee. Relevant articles include:
  - Bazot et al. (2012, MNRAS 427, 1847)
  - Gruberbauer et al. (2012, ApJ 749, 109)
- interpolates within the grid of models using Delaunay tessellation (from the scipy.spatial package which is based on the Qhull library)
- modular approach: facilitates including contributions from different people

1.1.2 Contributors

Author:

- Daniel R. Reese

Comments, corrections, suggestions, and contributions:

- Diego Bossini
- Gael Buldgen
- Tiago L. Campante
- William J. Chaplin
- Hugo R. Coelho
- Guy R. Davies
- Benoît D. C. P. Herbert
- James S. Kuszlewicz
- Yveline Lebreton
- Martin W. Long
- Mikkel N. Lund
- Andrea Miglio
- Ben Rendle

1.1.3 Supplementary material

- a more technical `overview` of AIMS
- a PDF version of this documentation may be downloaded `here`

1.1.4 Copyright information

- the AIMS project is distributed under the terms of the GNU General Public License, version 3
- a copy of of this license may be downloaded `here` and should also be included in `AIMS.tgz`

1.2 Acknowledgements

The "Asteroseismic Inference on a Massive Scale" (AIMS) project was developed at the University of Birmingham by Daniel R. Reese as one of the deliverables for the SPACEINN network. The SPACEINN network is funded by the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 312844.

1.2.1 Publications

If AIMS is used in any publication, the SPACEINN network kindly asks you to acknowledge the use of this software using a phrase such as the following:

> "This article made use of AIMS, a software for fitting stellar pulsation data, developed in the context of the SPACEINN network, funded by the European Commission's Seventh Framework Programme."

We also ask you to cite the article:

> Rendle et al., 2019, "AIMS - a new tool for stellar parameter determinations using asteroseismic constraints", MNRAS 484, 771.

1.3 Requirements

The following python packages are needed for AIMS:

- dill

- emcee
- ptemcee
  - this is only needed if you decide to do parallel tempering
  - note: This is a different version than the one available on PyPI as this version contains a bugfix for newer versions of python. It can be installed with

```
pip install ptemcee@git+https://github.com/waltervrossem/ptemcee@f6e91e7
```

- corner
  - note: this used to be called triangle in previous releases
- scipy
- numpy
- f2py
  - this is usually included with numpy
- matplotlib
- multiprocessing
  - this is already part of the standard python library
- tqdm
- lxml

For convenience, a *requirements.txt* file has been included. This allows the user to install the needed python packages via the command:

```
pip install -r requirements.txt
```

Note

This version of SPInS is compatible both with versions 2 and 3 of emcee, an MCMC package written in python (see Foreman-Mackey et al., 2013, PASP 125, 306).

However, as of version 3, emcee no longer supports parallel tempering. Therefore, SPInS now uses the separate package ptemcee for parallel tempering. This also has the added benefit of enabling the use of dynamic temperatures which may speed up convergence in some cases (see Vousden, Farr, and Mandel, 2016, MNRAS 455, 1919)

1.4 Download `AIMS`

- Click `here` to download AIMS.
- The contents of this file may then be extracted via the command:

```
tar -zxvf aims-master.tar.gz
```

- This will lead to the creation of a folder called `AIMS` and a subfolder called `AIMS/doc`.
  - the `AIMS` folder contains the AIMS program; it is from this folder that AIMS is launched.
  - the `AIMS\doc` folder is where the documentation is generated. Typing `make html` within this folder will generate this web page in `AIMS/doc/_build/html/`. Typing `make latexpdf` will generate a pdf version of this documentation in `AIMS/doc/_build/latex/AIMS.pdf`.

1.5 Installation

To avoid issues with other python installations, it is recommended to use a virtual environment and instsall the required packages there, activate the environment, and then compile it:

```
python -m venv .venv
source .venv/bin/activate
pip install -r src/requirements.txt
make --directory=src
```

As of version 1.2, a few strategic parts of the code have been rewritten in FORTRAN thus leading to a considerable speed up. These FORTRAN subroutines are then integrated into the AIMS code thanks to the f2py project. Accordingly, these FORTRAN subroutines need to be compiled before running AIMS. A Makefile has been provided for convenience. Hence, one simply needs to type the command:

```
make
```

The user may change the choice of FORTRAN compiler as well as the compilation options by editing the Makefile.

1.6 Usage

There are three different ways of using AIMS:

1. generating a binary file with the grid of models (including names, global parameters, and pulsation frequencies).

   Note

   This step must be carried out before the following two steps as these require the above binary file to function correctly.

2. carrying out tests to evaluate the accuracy of the interpolation for a given grid of models.
3. finding the properties of an observed star thanks to its classic and seismic parameters.

The way AIMS is used is decided by the values given in the `AIMS_configure.py` file, which also contains a number of other control parameters. Extensive comments are included in this file to help the user know how to set the various parameters.

1.6.1 Generating a binary grid

Requirements:

- a grid of models, including the pulsation frequencies; the formats for the files with the pulsation frequencies is described in *model.Model.read_file()*.
- a list with the paths and a set of global parameters for each model in the grid; the format this file is described in *model.Model_grid.read_model_list()*.

Relevant parameters in `AIMS_configure.py`:

- `mode`: set this to `"write_grid"` so that AIMS will write binary grid.
- `mode_format`: this specifies the format of the files with the pulsation frequencies.
- `list_grid`: set this to the filename of the file with the list of paths and global parameters.
- `binary_grid`: set this to the filename of the file which will contain the binary data.

- `grid_params`: specify the parameters relevant to the grid (excluding age, which is dealt with separately). Different options can be found in the source to *model.Model.string_to_param()*.
- `npositive`: set this to `True` to only save modes with $n \geq 0$ in the binary file.
- `agsm_cutoff`: set this to `True` to exclude modes above the cutoff frequency, as identified by the `icase` variable in agsm files from the ADIPLS pulsation code.

To run AIMS in this configuration, just type the following in a terminal window:

```
./AIMS.py
```

1.6.2 Testing the accuracy of the interpolation

Requirements:

- a binary grid of models as produced by AIMS

Relevant parameters in `AIMS_configure.py`:

- `mode`: set this to `"test_interpolation"` so that AIMS will carry out the interpolation tests.
- `interpolation_file`: specify the name of the file in which to write the results from the interpolation test in binary format. These results can be plotted using `plot_interpolation_test.py`.

To run AIMS in this configuration, just type the following in a terminal window:

```
./AIMS.py
```

1.6.3 Characterising an observed star

Requirements:

- a binary grid of models as produced by AIMS
- a file with the observational data; the format for this file is similar to the format used for the Asteroseismic Modeling Portal (AMP) with some simplifications and is described below. It will be read by *AIMS.Likelihood.read_constraints()*

Relevant parameters in `AIMS_configure.py`:

- `mode`: set this to `"fit_data"`
- most of the parameters in this file - see comments for details

To run AIMS in this configuration, just type the following in a terminal window:

```
./AIMS.py file_with_constraints
```

where `file_with_constraints` is the file with the observational constraints.

1.7 File formats

1.7.1 Format of a file with a list of models and properties:

Description:

- The first line is a header. It contains the root folder (including the final slash) with the grid of models and optionally, a suffix for the names of the files with the theoretical pulsation frequencies. For example:

```
/home/dreese/models_inversions/Grid_mesa_MS/   .freq
```

- Each of the following lines correspond to one model in the grid. They are composed of 9 or more columns with the following information:

  1. The second part of the path for the given model. When concatenated with the prefix on the first line, this should give the full path to the model. If, furthermore, the suffix from the first line is appended to it, it gives the name of the file with the frequencies.
  2. The stellar mass in g
  3. The stellar radius in cm
  4. The stellar luminosity in $\mathrm{g.cm^2.s^{-3}}$
  5. The metallicity
  6. The hydrogen content
  7. The stellar age in Myrs
  8. The effective temperature in K
  9. A dimensionless age parameter
  10. (user-defined) This and the following columns correspond to the parameters specified in the `user_params` variable given in `AIMS_configure.py`.

- Except for the first line, the order of the lines does not matter. AIMS will construct evolutionary tracks based on the parameters selected in the `grid_params` variable given in `AIMS_configure.py`, and sort them according to age.

Example:

Here's an example of a file read by AIMS (via the *model.Model_grid.read_model_list()* method):

```
/home/dreese/models_inversions/Grid_mesa_MS/   .freq
M0.80/LOGS_M0.80/M0.80Z0.0028Y0.2536/m0.80Y0.2536Z0.0028a1.8ovh0.2ovhe0_n1.
↪profile.FGONG   1.59136E+33   5.02248266E+10   2.33097993E+33   0.0028   0.
↪7436   1.00000000E-04   6000.94326   0.00000   7.432106E-01
M0.80/LOGS_M0.80/M0.80Z0.0028Y0.2536/m0.80Y0.2536Z0.0028a1.8ovh0.2ovhe0_n2.
↪profile.FGONG   1.59136E+33   4.86716596E+10   1.94640636E+33   0.0028   0.
↪7436   2.09332874E+02   5827.26021   0.01383   7.319698E-01
M0.80/LOGS_M0.80/M0.80Z0.0028Y0.2536/m0.80Y0.2536Z0.0028a1.8ovh0.2ovhe0_n3.
↪profile.FGONG   1.59136E+33   4.89176532E+10   1.97545563E+33   0.0028   0.
↪7436   4.34073185E+02   5834.15715   0.02868   7.188704E-01
```

It contains three models. The structure of the first model can be found in the following file:

```
/home/dreese/models_inversions/Grid_mesa_MS/M0.80/LOGS_M0.80/M0.80Z0.
↪0028Y0.2536/m0.80Y0.2536Z0.0028a1.8ovh0.2ovhe0_n1.profile.FGONG
```

and its frequencies in this file:

```
/home/dreese/models_inversions/Grid_mesa_MS/M0.80/LOGS_M0.80/M0.80Z0.
↪0028Y0.2536/m0.80Y0.2536Z0.0028a1.8ovh0.2ovhe0_n1.profile.FGONG.freq
```

The tenth column corresponds to the central hydrogen content, as specified by the contents of the `user_params` variable from `AIMS_configure.py`:

```
user_params = (("Xc", r'Central hydrogen, $%sX_c%s$'),)
```

1.7.2 Format of a file with theoretical frequencies:

AIMS is able to read files with the theoretical frequencies in various different text formats (see example below) as well as in the grand summary format from ADIPLS. This is a FORTRAN binary format described on pages 32 and 33 of the ADIPLS documentation. What follows is a description of one the text formats used by AIMS for files with frequencies:

Description:

- the first line is a header (and is skipped)
- the following lines contain five columns which correspond to l, n, frequency, a_value, inertia
  - the a_value column is ignored, so it could contain anything. `InversionKit` will typically put the difference between the numerical and variational frequencies in that column.

Example:

Here's an example of a file with theoretical pulsation frequencies which can be read by AIMS (via the *model.Model.read_file()* method):

```
#l  n         nu_theo (muHz)  nu_var-nu_theo (muHz)              Inertia
0  15  3.225852209451052e+03  1.312960435370769e-03  3.233628965187502e-09
0  16  3.421699035498995e+03 -2.482639610207116e-03  2.229252226305757e-09
0  17  3.615805033992529e+03  3.993051574070705e-03  1.618154348529283e-09
0  18  3.809740380503104e+03  9.650666734160040e-04  1.250359548964621e-09
0  19  4.003716857281849e+03 -7.991676880010345e-03  1.033914933206195e-09
0  20  4.198691419457581e+03  1.742711681799847e-03  8.866985261874711e-10
1  15  3.316007619955153e+03  5.056100344972947e-03  2.715966891128009e-09
1  16  3.511258977705781e+03  1.855844971032639e-04  1.902147334986236e-09
1  17  3.705576731149742e+03 -2.505276897409203e-03  1.424266453221534e-09
1  18  3.899485457373566e+03  5.212276555539575e-03  1.134594720287415e-09
1  19  4.094401244305849e+03  6.020260397235688e-03  9.579611596023003e-10
1  20  4.289716814475406e+03 -1.019475706561934e-02  8.344804874142957e-10
2  15  3.399280335063532e+03 -8.466318249702454e-04  2.315947651745295e-09
2  16  3.594141943503532e+03  4.712417365681176e-03  1.665322627996223e-09
2  17  3.788792185755381e+03 -1.167229517704982e-03  1.277569745555387e-09
2  18  3.983271067684743e+03 -6.187409578615188e-03  1.048757367028520e-09
2  19  4.178866833517976e+03  6.893199766636826e-03  8.963691946280509e-10
2  20  4.374959711016754e+03  3.274638356742798e-03  7.911508926344487e-10
3  15  3.476224140192640e+03 -2.524210208321165e-03  2.009476926536794e-09
3  16  3.671438520072859e+03  2.351724720028869e-04  1.485336526791650e-09
3  17  3.866350877376991e+03  5.643782460992952e-03  1.167619144668003e-09
3  18  4.061929209725198e+03 -1.552865011490212e-03  9.789648655155361e-10
3  19  4.258077196700047e+03 -8.629839649984206e-03  8.472972126693386e-10
3  20  4.455063887754256e+03  1.484804296796938e-02  7.528069568152023e-10
```

1.7.3 Format of a file with observational constraints:

Description:

- a collection of lines with frequency data with either (l, freq, error_bar) or (l, n, freq, error_bar) (depending on the value of `read_n` in the `AIMS_configure.py` file). For example:

```
0 1503.5 0.16
```

or the following if specifying the radial order:

```
0 15 1503.5 0.16
```

- a collection of lines with classical constraints. These start with the name of the relevant parameter (see possible options in *model.Model.string_to_param()*) followed by a description of its probability distribution function. This probability distribution function is specified in two possible ways:
  - it is implicitly assumed to be Gaussian. In this situation it is only necessary to specify the mean value and the one sigma error bar. For example:

    ```
    Teff 6100 80
    ```

  - it is explicitly specified (different options are given in *AIMS.Distribution*):

    ```
    Teff Uniform 6000 6200
    ```

- anything following a # is a comment
- the order of the lines does not matter

Examples:

- example of a file where n is *not* specified:

```
0 1582.20 0.13  # this is a (useless) comment
0 1684.02 0.16
0 1785.57 0.15
1 1526.55 0.29
1 1628.90 0.30
1 1730.45 0.17
2 1575.49 0.82
2 1676.25 0.51
2 1777.62 0.27
Teff 6060.00 84.00
Fe_H -0.20 0.09
```

- example of a file where n is specified:

```
0 15 1582.20 0.13
0 16 1684.02 0.16
Teff 6060.00 84.00 # AIMS doesn't worry about the order of the lines
0 17 1785.57 0.15
1 14 1526.55 0.29
1 15 1628.90 0.30
1 16 1730.45 0.17
2 14 1575.49 0.82
2 15 1676.25 0.51
2 16 1777.62 0.27
Fe_H -0.20 0.09
```

Differences with AMP:

- the number of frequencies does not need to be specified (if this line contains supplementary parameters, than `AIMS.py` may confuse it with frequency data)
- there are no flags (one should adjust the parameters in `AIMS_configure.py` instead)
- the order of the lines is not important (one can mix the classic and seismic observables)
- it is possible to specify radial orders (depending on the value of `read_n` in the `AIMS_configure.py` file)
- the treatment of non-seismic constraints is more flexible
  - a larger variety of non-seismic constraints can be included (see possible options in *model.Model.string_to_param()*)
  - full parameter names are allowed (and preferred); for compatibility with AMP, the same one letter abbreviations are also allowed
  - it is possible to specify the probability distribution function

1.8 List of changes

Version	Changes
2.2.0	• fixed compilation error for compatibility with python>=3.12 and numpy>=2 • add option to plot a subset of params in corner plot • add option to make writing samples files optional • docs use version in `AIMS.py` • empty output folder before generating output • use `venv` in CI and get CI base `requirements.txt` from src/requirements.txt • Add option to add 'non-constraints' which are only used during plotting • use `AIMS_configure.py` from current working directory and a copy of it and the input file are saved in the output folder • fixed various deprecation warnings • use a version of ptemcee with a fix for indexing error in newer versions of numpy • fixed bug when reading gyre files which skipped the first line of data • use [M/H] calculation for [Fe/H] • also plot nu%dnu + dnu in echelle diagrams • replace remaining sys.exit with errors
2.1.0	• added compatibility with emcee3 and ptemcee • expanded unit test coverage • added ability to run AIMS without seismic constraints • various bugfixes
2.0.0	• modified storage for evolutionary tracks thus saving a lot of memory • a dimensionless age parameter is introduced for the purposes of interpolation between tracks (using the same approach as in SPInS) • new and more flexible implementation of frequency combinations • added a batch mode (without status bar)
1.3.0	• can run both in python2.x and python3.x • included default distributions for priors and tight ball ranges • can tolerate erroneous values of user_params when fitting observations • added status bar for iterations (thanks to B. Rendle) • added plots with evolution of walker percentiles
1.2.0	• removed extrapolation beyond grid limits • various subprograms rewritten in Fortran (thus accelerating the code)
1.1.0	• added extrapolation beyond grid limits
1.0.0	

1.9 The AIMS program

A module which contains the main program for AIMS as well as various classes which intervene when calculating the priors and likelihood function:

- *Distribution*: a class which represents a probability distribution
- *Prior_list*: a class with a list of priors
- *Mode*: a class used to represent observed modes
- *Combination*: a class used to represent a linear frequency combination
- *Combination_function*: a class used to represent functions of frequency combinations
- *Likelihood*: a class used to represent the likelihood function
- *Probability*: a class which groups the priors and likelihood function together

This module relies on the emcee package to apply an MCMC algorithm which will return a representative sample of models for a given set of seismic an classic constraints.

Warning

In various places in this module, for instance in the *Prior_list* and *Likelihood* classes, various methods return what is described as a χ^2 value. Technically, these are not χ^2 values, but rather $-\chi^2/2$, i.e. the argument of the exponential function which intervenes in the Gaussian probability distribution.

class AIMS.**Combination**

A class which contains indices and coefficients which represent a linear combination of frequencies.

add_coeff(*j*, *coeff*)

Append the given index and coefficient to the list of indices and coefficients.

Parameters

- **j** (*int*) – index of the mode
- **coeff** (*float*) – coefficient used in the frequency combination

coeff

Coefficients of the linear combination of frequencies.

find_values(*freq*)

Find the value associated with this frequency combination using the input frequencies.

Parameters
freq (*float array like*) – list of frequencies

index

Indices in of the linear combination of frequencies.

print_me()

Print frequency combination.

value

Value of the linear combination of frequencies.

class AIMS.**Combination_function**(*_name*, *_function*, *_gradient_function*, *_offset=0.0*)

A class which applies functions to linear frequency combinations.

add_combination(*combination*)

Append a frequency combination to this combination function.

Parameters
combination (*Combination*) – a linear frequency combination

combinations

The frequency combinations which intervene in the function.

find_values(*freq*)

Find the value and gradient of this frequency combination function for the input set of frequencies.

Parameters
freq (*float array like*) – list of frequencies

function

The function which is applied to the frequency combinations.

gradient

The gradient of the frequency combination function at the observed frequencies.

gradient_function

The gradient of the function which is applied to the frequency combinations.

name

Human-readable name of the frequency combination function

offset

Additive offset for the frequency combination function.

print_me()

Print frequency combination function.

value

Value of the frequency combination function at the observed frequencies.

class AIMS.**Distribution**(*_type*, *_values*)

A class which represents a probability distribution, and can yield its value for a given input parameter, or provide a random realisation.

Note

Derived from a class originally written by G. Davies.

Parameters

- **_type** (*string*) – type of probability function (current options include "Gaussian", "Truncated_gaussian", "Uniform", "IMF1", "IMF2", "Uninformative", "Above", "Below")
- **_values** (*list of floats*) – list of parameters relevant to the probability function

property error_bar

Returns an error bar based on the distribution. This does not necessarily correspond to the one-sigma value but rather to what is the most convenient value.

Returns
the error bar

Return type
float

`property mean`

Returns the mean value of the probability distribution.

Returns
the mean value of the probability distribution

Return type
float

`property nparams`

Return the number of relevant parameters for a given distribution.

Returns
the number of relevant parameters

Return type
int

`print_me()`

Print type and parameters of probability distribution.

`re_centre`(*value*)

Re-centre the probability distribution around the input value.

Parameters
value (*float*) – new value around which to centre the distribution

`re_normalise`(*value*)

Re-normalise the probability distribution so that its characteristic width corresponds to the input value.

Parameters
value (*float*) – new value around for the chacteristic width

`realisation`(*size=None*)

Return random values which statistically follow the probability distribution.

Parameters
size (*int or tuple of ints*) – shape of random variates

Returns
a set of random realisations

Return type
float

`to_string()`

Produce nice string representation of the distribution.

Returns
nice string representation of the distribution

Return type
string

`type`

Type of probability function (“Uniform”, “Gaussian”, “Truncated_gaussian”, “IMF1”, “IMF2”, “Uninformative”, “Above”, or “Below”)

values

List of parameters relevant to probability function

class AIMS.**Likelihood**

A class which described the likelihood function and allows users to evaluate it.

Rkkinv

Inverse of R matrix from the WhoSGlAd method (see Eqs. (D.5) and (D.10) from Farnir et al. (2019))

> **Note**
>
> The first index corresponds to the p vector whereas the second index corresponds to the normalised q vector.

add_combinations(*string*, *num_list*, *den_list=[]*, *target_ell=None*)

This finds the indices of modes which intervene in a frequency combination or ratio, as specified by the mandatory and optional arguments. These indices, the relevant coefficients, the numerator, the denominator, and the resultant value of the combination are stored in the *combination_functions* variable.

Parameters

- **string** (*string*) – name of the frequency combination
- **num_list** (*list of (int,int,float)*) – list of relative mode identifications and coefficients used to define a frequency combination or the numerator of a frequency ratio. This list contains tuples of the form (delta n, delta l, coeff).
- **den_list** (*list of (int,int,float)*) – list of relative mode identifications and coefficients used to define the denominator of a frequency ratio. If absent, then, it is assumed that a linear combination of frequencies is represented. The form is the same as for num_list.
- **target_ell** (*int*) – this is used to impose a specific l value on the first selected mode.

add_constraint(*constraint*)

Add a supplementary constraint to the list of constraints.

Parameters

constraint ((string, *Distribution*)) – supplementary constraint

add_dnu_constraint(*string*, *l_targets=[0]*)

Add the large frequency separation as a contraint. The coefficients are obtained via a least-squares approach. The approach taken here has two advantages:

1. Correlations between the large frequency separation and other seismic constraints will be taken into account.
2. The same modes will be used in the same way, both for the observations and the models.

Parameters

- **string** (*string*) – name of this seismic constraint
- **l_targets** (*list of int*) – specifies for which l values the large frequency separation is to be calculated. If None is supplied, all modes will be used.

> **Note**
>
> This uses an analytical approach and is therefore the prefered method.

`add_dnu_constraint_matrix`(*string*, *l_targets=[0]*)

Add the large frequency separation as a contraint. The coefficients are obtained via a least-squares approach. The approach taken here has two advantages:

1. Correlations between the large frequency separation and other seismic constraints will be taken into account.
2. The same modes will be used in the same way, both for the observations and the models.

Parameters

- **`string`** (*`string`*) – name of this seismic constraint
- **`l_targets`** (*`list of int`*) – specifies for which l values the large frequency separation is to be calculated. If `None` is supplied, all modes will be used.

> **Note**
>
> This uses a matrix approach and is therefore *not* the prefered method.

`add_nonconstraint`(*nonconstraint*)

Add a supplementary constraint to the list of non-constraints.

Parameters

`constraint` ((string, *`Distribution`*)) – supplementary constraint

`add_nu_min_constraint`(*string*, *target_ell=0*, *pos=0*, *min_n=False*)

Add the minimun frequencies/modes of a specific ell value as a seismic constraint. Typically, such constraints are used as an "anchor" when combined with constraints based on frequency ratios.

Parameters

- **`string`** (*`string`*) – name of the seismic constraint
- **`target_ell`** (*`int`*) – ell value of the minimum frequency/mode
- **`pos`** (*`int`*) – position of desired frequency (0 = lowest, 1 = second lowest . . .)
- **`min_n`** (*`boolean`*) – if `False`, look for minimum observational frequency. If `True`, look for minimum radial order.

`add_seismic_constraint`(*string*)

Add seismic contraints based on the keyword given in `string`.

Parameters

`string` (*`string`*) – keyword which specifies the type of constraint to be added. Current options include:

- `nu`: individual frequencies
- `nu0`: individual frequencies (radial modes only)
- `nu_min0`: radial mode with minimum frequency
- `nu_min1`: radial mode with second lowest frequency

- `nu_min2`: radial mode with third lowest frequency
- `r02`: r_{02} frequency ratios
- `r01`: r_{01} frequency ratios
- `r10`: r_{10} frequency ratios
- `dnu`: individual large frequency separations (using all modes)
- `dnu0`: individual large frequency separations (using radial modes only)
- `d2nu`: second differences (using all modes)
- `avg_dnu`: average large frequency separation (using all modes)
- `avg_dnu0`: average large frequency separation (using radial modes only)
- `whosglad_dnu`: WhoSGlAd version of average large frequency separation (all modes)
- `whosglad_dnu0`: WhoSGlAd version of average large frequency separation (l=0 modes)
- `whosglad_dnu1`: WhoSGlAd version of average large frequency separation (l=1 modes)
- `whosglad_r01`: WhoSGlAd version of average r01 frequency ratio
- `whosglad_r02`: WhoSGlAd version of average r02 frequency ratio
- `whosglad_Delta01`: WhoSGlAd Delta01 seismic constraint
- `whosglad_Delta02`: WhoSGlAd Delta02 seismic constraint
- `whosglad_eps0`: WhoSGlAd average frequency offset (l=0 modes)
- `whosglad_eps1`: WhoSGlAd average frequency offset (l=1 modes)
- `whosglad_AHe`: WhoSGlAd amplitude indicator for helium content
- `whosglad_Abcz`: WhoSGlAd amplitude indicator for the base of the convection zone

add_whosglad_AHe_constraint(*string*, *maxpower=2*)

Add a seismic constraint which provides a normalised He glitch amplitude using the WhoSGlAd method (Farnir et al. 2019). This approach has the advantage of leading to (near) orthogonal seismic indicators.

Parameters

- **string** (*string*) – name of this seismic constraint
- **maxpower** (*int*) – maximum power in Pj(n)=n^j polynomial

add_whosglad_Abcz_constraint(*string*, *maxpower=2*)

Add a seismic constraint which provides a normalised glitch amplitude for the base of the convection zone using the WhoSGlAd method (Farnir et al. 2019). This approach has the advantage of leading to (near) orthogonal seismic indicators.

Parameters

- **string** (*string*) – name of this seismic constraint
- **maxpower** (*int*) – maximum power in Pj(n)=n^j polynomial

add_whosglad_Delta_constraint(*string*, *l_target*, *maxpower=2*)

Add a ratio of large frequency separations as a contraint using the WhoSGlAd method (Farnir et al. 2019). This approach has the advantage of leading to (near) orthogonal seismic indicators.

Parameters

- **string** (*string*) – name of this seismic constraint

- **l_target** (*int*) – specifies for which l to calculate the average Delta constraint.
- **maxpower** (*int*) – maximum power in Pj(n)=n^j polynomial

add_whosglad_dnu_constraint(*string*, *l_targets=[0]*, *maxpower=2*)

Add an average large frequency separation as a contraint using the WhoSGlAd method (Farnir et al. 2019). This approach has the advantage of leading to (near) orthogonal seismic indicators.

Parameters

- **string** (*string*) – name of this seismic constraint
- **l_targets** (*list of int*) – specifies for which l values the large frequency separation is to be calculated. If `None` is supplied, all modes will be used.
- **maxpower** (*int*) – maximum power in Pj(n)=n^j polynomial

add_whosglad_epsilon_constraint(*string*, *l_target*, *maxpower=2*)

Add an average frequency offset as a contraint using the WhoSGlAd method (Farnir et al. 2019). This approach has the advantage of leading to (near) orthogonal seismic indicators.

Parameters

- **string** (*string*) – name of this seismic constraint
- **l_target** (*int*) – specifies for which l to calculate the average epsilon offset.
- **maxpower** (*int*) – maximum power in Pj(n)=n^j polynomial

add_whosglad_ratio_constraint(*string*, *l_target*, *maxpower=2*)

Add an average frequency ratio as a contraint using the WhoSGlAd method (Farnir et al. 2019). This approach has the advantage of leading to (near) orthogonal seismic indicators.

Parameters

- **string** (*string*) – name of this seismic constraint
- **l_target** (*int*) – specifies for which l to calculate the average frequency ratio.
- **maxpower** (*int*) – maximum power in Pj(n)=n^j polynomial

apply_constraints(*my_model*)

Calculate a χ^2 value for the set of constraints (excluding seismic constraints based on mode frequencies).

Parameters
my_model (*model.Model*) – model for which the χ^2 value is being calculated

Returns
the χ^2 value deduced from classic constraints

Return type
float

assign_n(*my_model*)

Assign the radial orders based on proximity to theoretical frequencies from an input model.

Parameters
my_model (*model.Model*) – input model

classic_weight

Absolute weight to be applied to classic constraints (incl. nu_max constraint).

clear_seismic_constraints()

This clears the seismic constraints. Specifically, the list of seismic combinations, and associated covariance matrix and its inverse are reinitialised.

coeff

2D float array with the coefficients for each frequency combination. The indices are:

1. The index of the term
2. The cumulative index of the frequency combination

combination_functions

This contains functions of frequency combinations.

compare_frequency_combinations(*my_model*, *mode_map*, *a=[]*)

This finds a χ^2 value based on a comparison of frequency combination functions, as defined in the `combination_function` variable.

Parameters

- **my_model** (*model.Model*) – model for which the χ^2 value is being calculated
- **mode_map** (*list of int*) – a mapping which relates observed modes to theoretical ones
- **a** (*array-like*) – parameters of surface correction terms

Returns
the χ^2 value for the seismic constraints

Return type
float

Note

I'm assuming none of the modes are missing (i.e. that mode_map doesn't contain the value -1)

constraints

List of constraints which intervene in the likelihood function.

construct_whosglad_basis(*maxpower=2*, *glitchpower=-5*)

Construct orthogonal basis for the smooth component of a pulsation spectrum using the Gram-Schmidt orthonormalisation method as described in the WhoSGlAd method (Farnir et al. 2019).

Parameters

- **maxpower** (*int*) – maximum power in Pj(n)=n^j polynomial
- **glitchpower** (*int*) – lower power used in Helium glitch component

cov

Covariance matrix which intervenes when calculating frequency combinations.

create_combination_arrays()

Create array form of frequency combination functions to be used with a fortran based routine for calculating the seismic chi^2 value.

create_mode_arrays()

Create arrays with mode parameters (n, l, freq), which can be interfaced with fortran methods more easily.

dot_product_whosglad(*v1*, *v2*)

Dot product used for constructing an orthonormal set of seismic indicators using the WhoSGlAd method (Farnir et al. 2019).

Parameters

- **v1** (*float np.array*) – a vector of frequencies (or some polynomial of the radial order)
- **v2** (*float np.array*) – a vector of frequencies (or some polynomial of the radial order)

Returns
the dot product between v1 and v2

Return type
float

evaluate(*my_model*)

Calculate ln of likelihood function (i.e. a χ^2 value) for a given model.

Parameters
my_model (*model.Model*) – model for which the χ^2 value is being calculated

Returns
the χ^2 value, optionally the optimal surface amplitudes (depending on the value of `AIMS_configure.surface_option`), integers to indicate whether the model was rejected due to classic or seismic contraints

Return type
float, np.array (optional), int, int

> **Note**
>
> This avoids model interpolation and can be used to gain time.

find_covariance()

This prepares the covariance matrix and its inverse based on the frequency combinations in *combination_functions*.

> **Warning**
>
> This method should be called *after* all of the methods which add to the list of frequency combinations.

find_l_list(*l_targets*, *npowers=2*)

Find a list of l values with the following properties:

- each l value only occurs once
- each l value given in the parameter `l_targets` is in the result l list, except if there is less than npowers modes with this l value
- if the parameter `l_targets` is `None`, look for all l values with npowers or more modes associated with them

Parameters

- **l_targets** (*list of int*) – input list of l values
- **npowers** (*int*) – the number of powers in the fit

Returns
new list of l values with the above properties

Return type
list of int

`find_map`(*my_model*, *use_n*)

This finds a map which indicates the correspondance between observed modes and theoretical modes from `my_model`.

Parameters

- **my_model** – model for which the χ^2 value is being calculated
- **use_n** (*boolean*) – specify whether to use the radial order when finding the map from observed modes to theoretical modes. If `False`, the map is based on frequency proximity.

Returns
the correspondance between observed and theoretical modes from the above model, and the number of observed modes which weren't mapped onto theoretical modes

Return type
list of int, int

Note

- a value of -1 is used to indicate that no theoretical mode corresponds to a particular observed mode.
- only zero or one observed mode is allowed to correspond to a theoretical mode

`find_vec`(*a_combination_function*)

This finds a set of coefficients which intervene when constructing the coviance matrix for frequency combination functions.

Parameters
a_combination_function (*Combination_function*) – variable which specifies the frequency combination function.

Returns
the above set of coefficients

Return type
np.array

`find_weights`()

Find absolute weights for seismic and classic constraints based on options in `AIMS_configure.py`.

`find_weights_new`()

Find absolute weights for seismic and classic constraints based on options in `AIMS_configure.py`.

`fvalues`

Array with the observed frequencies

`get_optimal_surface_amplitudes`(*my_model*, *mode_map*)

Find optimal surface correction amplitude, for the surface correction specified by `surface_option`.

Parameters

- **my_model** (*model.Model*) – the model for which we're finding the surface correction amplitude
- **mode_map** (*list of int*) – a mapping which relates observed modes to theoretical ones

Returns
optimal surface correction amplitudes

Return type
np.array

guess_dnu(*with_n=False*)

Guess the large frequency separation based on the radial modes.

Parameters
with_n (*boolean*) – specifies whether to use the n values already stored with each mode, when calculating the large frequency separation.

Returns
the large frequency separation

Return type
float

guess_n()

Guess the radial order of the observed pulsations modes.

This method uses the large frequency separation, as calculated with *guess_dnu()*, to estimate the radial orders. These orders are subsequently adjusted to avoid multiple modes with the same identification. The resultant radial orders could be off by a constant offset, but this is not too problematic when computing frequency combinations or ratios.

indices

2D int array with the mode indices for each frequency combination. The indices are:

1. The index of the term
2. The cumulative index of the frequency combination

invcov

Inverse of covariance matrix, *Likelihood.cov*.

lvalues

Array with the l values of the observed modes

modes

List of pulsation modes (of type *Mode*).

ncoeff

The number of terms in each frequency combination. The index is a cumulative index that uniquely designates one of the frequency combinations from one of the frequency combination functions. It is also the same as the second index in the *coeff* and *indices* variables.

ncomb

1D int array which specifies the number of frequency combinations within each frequency combination function

nonconstraints

List of constraints which are only used during plotting and not used in the likelihood function.

nvalues

Array with the n values of the observed modes

qk

Normalised q vectors from the WhoSGlAd method (Farnir et al. 2019).

read_constraints(*filename*, *factor=1.0*)

Read a file with pulsation data and constraints.

Parameters

- **filename** (*string*) – name of file with pulsation data.
- **factor** (*float*) – multiplicative factor for pulsation frequencies. Can be used for conversions.

seismic_weight

Absolute weight to be applied to seismic constraints

sort_modes()

Sort the modes. The ordering will depend on the value of `use_n` from the `AIMS_configure.py` file.

class AIMS.**Mode**(*_n*, *_l*, *_freq*, *_dfreq*)

A class which describes an *observed* pulsation mode.

Parameters

- **_n** (*int*) – radial order of observed mode
- **_l** (*int*) – harmonic degree of observed mode.
- **_freq** (*float*) – pulsation frequency (in μHz).
- **_dfreq** (*float*) – error bar on pulsation frequency (in μHz).

Warning

Negative values are not accepted for `_l`, `_freq`, or `_dfreq`.

dfreq

Error bar on pulsation frequency (in μHz).

freq

Pulsation frequency (in μHz).

l

Harmonic degree of observed mode.

match(*a_mode*)

Check to see if input mode has the same (n,l) values as the current mode.

Parameters

a_mode (*Mode*) – input mode which is being compared with current mode.

Returns

`True` if the input mode has the same (n,l) values as the current mode.

Return type

boolean

n

Radial order of observed mode.

print_me()

Print the mode in a human readable form.

class AIMS.**Prior_list**

A class which contains a list of priors as well as convenient methods for adding priors and for evaluating them.

add_prior(*aPrior*)

Add a prior to the list.

Parameters
aPrior (*Distribution*) – prior which is to be added to the list.

priors

A list of probability distributions which correspond to priors.

realisation(*size=None*)

Return an array with realisations for each prior. The last dimension will correspond to the different priors.

Parameters
size (*int or tuple of ints*) – shape of random variates (for each prior)

Returns
a set of realisations

Return type
numpy float array

class AIMS.**Probability**(*_priors*, *_likelihood*)

A class which combines the priors and likelihood function, and allows the the user to evalute ln of the product of these.

Parameters

- **_priors** (*Prior_list*) – input set of priors
- **_likelihood** (*Likelihood*) – input likelihood function

evaluate(*my_model*)

Evalulate the ln of the product of the priors and likelihood function, i.e. the probability, for a given model, to within an additive constant.

Parameters
my_model (*model.Model*) – input model

Returns
the ln of the probability, integers indicating if the model has bee rejected based on classic constraints, seismic constraints, and/or priors

Return type
float, int, int, int

Note

This avoids model interpolation and can be used to gain time.

is_outside(*params*)

Test to see if the given set of parameters lies outside the grid of models. This is done by evaluate the probability and seeing if the result indicates this.

Parameters
params (*array-like*) – input set of parameters

Returns
True if the set of parameters corresponds to a point outside the grid.

Return type
boolean

likelihood

The likelihood function.

priors

The set of priors.

AIMS.**accepted_parameters = []**

list of parameters associated with accepted models

AIMS.**append_osm_parameter**(*config_osm*, *name*, *value*, *step*, *rate*, *bounds*)

Add a parameter in xlm format in the file with the classic constraints for OSM.

Parameters

- **config_osm** (*lxml.etree._Element*) – XLM etree element to which to add the parameter
- **name** (*string*) – name of the parameter
- **value** (*float*) – value of the parameter
- **step** (*float*) – parameter step (this intervenes when numerically calculating derivatives with respect to this parameter)
- **rate** (*float*) – parameter rate (this corresponds to a tolerance on this parameter)
- **bounds** (*float tuple*) – bounds on the parameter

AIMS.**append_osm_surface_effects**(*modes_osm*, *name*, *numax*, *values*)

Add a method with which to calculate surface effects to the OSM contraint file.

Parameters

- **modes_osm** (*lxml.etree._Element*) – XML element to which to add the surface effects method
- **name** (*string*) – name of the method
- **numax** (*float*) – value of numax
- **values** (*float tuple*) – values which intervene in the method

AIMS.**autocorr_time = []**

integrated autocorrelelation time (this is useful for testing convergence)

AIMS.**best_MCMC_model = None**

best model from the MCMC run

AIMS.**best_MCMC_params = None**

parameters for the model *best_MCMC_model*

AIMS.**best_MCMC_result = -1e+300**

ln(probability) result for the model *best_MCMC_model*

AIMS.**best_age_range = 0.0**

Age range on track with best_model_model

AIMS.**best_grid_model = None**

best model from a scan of the entire grid

AIMS.**best_grid_params = None**

parameters for the model *best_grid_model*

AIMS.**best_grid_result = -1e+300**

ln(probability) result for the model *best_grid_model*

AIMS.**check_configuration**()

Test the version of the EMCEE package and set the mc2 variable accordingly.

Test the values of the variables in check_configuration to make sure they're acceptable. If an unacceptable value is found, then this will stop AIMS and explain what variable has an erroneous value.

AIMS.**find_a_blob**(*params*)

Find a blob (i.e. supplementary output parameters) for a given set of parameters (for one model). The blob also includes the log(P) value as a first entry.

Parameters
params (*array-like*) – input set of parameters

Returns
list of supplementary output parameters

Return type
list of floats

AIMS.**find_best_model**()

Scan through grid of models to find "best" model for a given probability function (i.e. the product of priors and a likelihood function).

AIMS.**find_best_model_in_track**(*ntrack*)

Scan through an evolutionary track to find "best" model for *prob*, the probability function (i.e. the product of priors and a likelihood function).

Parameters
ntrack (*int*) – number of the evolutionary track

Returns
the ln(probability) value, the "best" model, the parameters of accepted models, the parameters of rejected models, the age range of the track, and the numbers of models rejected due to classic constraints, seismic contraints, and priors

Return type
(float, *model.Model*, 2D float list, 2D float list, float, int, int, int)

AIMS.**find_blobs**(*samples*)

Find blobs (i.e. supplementary output parameters) from a set of samples (i.e. for multiple models).

Parameters
samples (*list/array of array-like*) – input set of samples

Returns
set of supplementary output parameters

Return type
np.array

AIMS.**grid = None**

grid of models

AIMS.**grid_params_MCMC = ()**

parameters used in the MCMC run (excluding surface correction parameters)

AIMS.**grid_params_MCMC_with_surf = ()**

parameters used in the MCMC run (including surface correction parameters)

AIMS.**init_walkers**()

Initialise the walkers used in emcee.

Returns
array of starting parameters

Return type
np.array

AIMS.**interpolation_tests**(*filename*)

Carry out various interpolation tests and write results in binary format to file.

Parameters
filename (*string*) – name of file in which to write test results.

Note

The contents of this file may be plotted using methods from `plot_interpolation_test.py`.

AIMS.**load_binary_data**(*filename*)

Read a binary file with a grid of models.

Parameters
filename (*string*) – name of file with grid in binary format

Returns
the grid of models

Return type
model.Model_grid

AIMS.**log0 = -1e+300**

a large negative value used to represent ln(0)

AIMS.**mc2 = None**

True if emcee's version is 2 or prior

AIMS.**my_map = None**

pointer to the map function (either the parallel or sequential versions)

AIMS.**ndims = 0**

number of dimensions for MCMC parameters (includes *nsurf*)

AIMS.**nreject_classic = 0**

Number of models rejected because of classic constraints

AIMS.**nreject_prior = 0**
Number of models rejected based on priors

AIMS.**nreject_seismic = 0**
Number of models rejected because of seismic constraints

AIMS.**nsurf = 0**
number of surface term parameters

AIMS.**output_folder = None**
folder in which to write the results

AIMS.**plot_distrib_iter**(*percentiles*, *labels*, *folder*)
Plot individual distribution of walkers as a function of iterations.

Parameters

- **percentiles** (*np.array*) – array with percentiles from emcee run
- **labels** (*list of strings*) – labels for the different dimensions in parameters space
- **folder** (*string*) – specify name of file in which to save plots of walkers.

Warning

This method must be applied before the samples are reshaped, and information on individual walkers lost.

AIMS.**plot_distrib_iter_old**(*samples*, *labels*, *folder*)
Plot individual distribution of walkers as a function of iterations.

Parameters

- **samples** (*np.array*) – samples from the emcee run
- **labels** (*list of strings*) – labels for the different dimensions in parameters space
- **folder** (*string*) – specify name of file in which to save plots of walkers.

Warning

This method must be applied before the samples are reshaped, and information on individual walkers lost.

AIMS.**plot_echelle_diagram**(*my_model*, *my_params*, *model_name*)
Produce an echelle diagram for input model.

Parameters

- **my_model** (*model.Model*) – model for which we're making an echelle diagram
- **my_params** (*array-like*) – parameters of the model
- **model_name** (*string*) – name used to describe this model. This is also used when naming the file with the echelle diagram.

AIMS.**plot_frequencies**(*grid*)
Plot frequencies along an evolutionary track. For debugging purposes only.

Parameters
grid (Model_grid) – grid of models

AIMS.**plot_frequency_diff**(*my_model*, *my_params*, *model_name*, *scaled=False*)

Make a diagram with frequency differences.

Parameters

- **my_model** (*model.Model*) – model for which we're plotting frequencie differences
- **my_params** (*array-like*) – parameters of the model
- **model_name** (*string*) – name used to describe this model. This is also used when naming the file with the plot.
- **scaled** (*boolean*) – if True, scale frequency differences by frequency error bars

AIMS.**plot_histograms**(*samples*, *names*, *fancy_names*, *truths=None*)

Plot a histogram based on a set of samples.

Parameters

- **samples** (*np.array*) – samples form the emcee run
- **names** (*list of strings*) – names of the quantities represented by the samples. This will be used when naming the file with the histogram
- **fancy_names** (*list of strings*) – name of the quantities represented by the samples. This will be used as the x-axis label in the histogram.
- **truths** (*list of floats*) – reference values (typically the true values or some other important values) to be added to the histograms as a vertical line

AIMS.**plot_walkers**(*samples*, *labels*, *filename*, *nw=3*)

Plot individual walkers.

Parameters

- **samples** (*np.array*) – samples from the emcee run
- **labels** (*list of strings*) – labels for the different dimensions in parameters space
- **filename** (*string*) – specify name of file in which to save plots of walkers.
- **nw** (*int*) – number of walkers to be plotted

> **Warning**
>
> This method must be applied before the samples are reshaped, and information on individual walkers lost.

AIMS.**pool = None**

pool from which to carry out parallel computations

AIMS.**prob = None**

Probability type object that represents the probability function which includes the likelihood and priors

AIMS.**rejected_parameters = []**

list of parameters associated with rejected models

AIMS.**run_emcee**(*p0*)

Run the emcee program.

Parameters

p0 (*np.array*) – the initial set of walkers

Returns
the emcee sampler for the MCMC run, array with walker percentiles as a function of iteration number

Return type
emcee sampler object, np.array

AIMS.**statistical_model = None**

model corresponding to statistical parameters

AIMS.**statistical_params = None**

parameters for the model *statistical_model*

AIMS.**statistical_result = -1e+300**

ln(probability) result for the model *statistical_model*

AIMS.**string_to_title**(*string*)

Create fancy title from string.

Parameters
string (*string*) – string from which the title is created.

Returns
the fancy string title

Return type
string

AIMS.**swap_dimensions**(*array*)

Swaps the two first dimensions of an array. This is useful for handling the different conventions used to store the samples in emcee3 and ptemcee.

Parameters
array (*np.array*) – input array

Returns
array with swapped dimensions

Return type
np.array

AIMS.**threshold = -1e+290**

threshold for "accepted" models. Needs to be greater than *log0*

AIMS.**tight_ball_distributions = None**

Prior_list type object with the distributions for the initial tight ball

AIMS.**write_LEGACY_summary**(*filename*, *KIC*, *labels*, *samples*)

Write a one line summary of the statistical properties based on a sequence of realisations to a file. The format matches that of the LEGACY project.

The results include:

- average values for each variable (statistical mean)
- error bars for each variable (standard mean deviation)

Parameters

- **filename** (*string*) – name of file in which to write the statistical properties
- **KIC** (*string*) – KIC number of the star

- **labels** (*list of strings*) – names of relevant variables
- **samples** (*np.array*) – samples for which statistical properties are calculated

AIMS.**write_SPInS_file_cgs**(*filename*)

Write list file compatible with SPInS. The quantities mass, luminosity, and radius are provided in cgs.

> **Note**
>
> The header in the output file might need to be edited by hand.

AIMS.**write_SPInS_file_solar**(*filename*)

Write list file compatible with SPInS. The quantities mass, luminosity, and radius are provided solar units.

> **Note**
>
> The header in the output file might need to be edited by hand.

AIMS.**write_binary_data**(*infile*, *outfile*)

Read an ascii file with a grid of models, and write corresponding binary file.

Parameters

- **infile** (*string*) – input ascii file name
- **outfile** (*string*) – output binary file name

AIMS.**write_combinations**(*filename*, *samples*)

Produce a list of linear combinations of grid models (based on interpolation) corresponding to the provided model parameters.

Parameters

- **filename** (*string*) – name of the file to which to write the model combinations
- **samples** (*np.array*) – set of model parameters for which we would like to obtain the grid models and interpolation coefficients

AIMS.**write_list_file**(*filename*)

Write list file from which to generate binary grid. Various filters can be included to reduce the number of models.

> **Note**
>
> This code is intended for developpers, not first time users.

AIMS.**write_model**(*my_model*, *my_params*, *my_result*, *model_name*, *extended=False*)

Write text file with caracteristics of input model.

Parameters

- **my_model** (*model.Model*) – model for which we're writing a text file
- **my_params** (*array-like*) – parameters of the model
- **my_result** (*float*) – ln(P) value obtained for the model

- **model_name** (*string*) – name used to describe this model. This is also used when naming the text file.
- **extended** – if set to True, all of the theoretical modes are saved in the text file, including those not matched to observations

AIMS.**write_osm_don**(*filename*, *my_model*)

Write file with choice of physical ingredients to be used by CESAM or CESTAM and OSM.

Parameters

- **filename** (*string*) – name of file which will contain the physical ingredients
- **my_model** (*model.Model*) – model from which which is derived various physical constraints/settings

Note

Written by B. Herbert.

AIMS.**write_osm_frequencies**(*filename*, *my_model*)

Write file with frequencies for Optimal Stellar Model (OSM), written by R. Samadi.

Parameters

- **filename** (*string*) – name of file which will contain the frequencies
- **my_model** (*model.Model*) – model from which are derived the radial orders

Note

Written by B. Herbert.

AIMS.**write_osm_xml**(*filename*, *my_params*, *my_model*)

Write file with classic constraints for OSM

Parameters

- **filename** (*string*) – name of file with classic constraints
- **my_model** (*model.Model*) – model used in deriving some of the constraints

Note

Originally written by B. Herbert. Includes some modifications.

AIMS.**write_percentiles**(*filename*, *labels*, *samples*)

Write percentiles based on a sequence of realisations to a file. The results include:

+/- 2 sigma error bars (using the 2.5th and 97.5th percentiles) +/- 1 sigma error bars (using the 16th and 84th percentiles) the median value (i.e. the 50th percentile)

Parameters

- **filename** (*string*) – name of file in which to write the percentiles
- **labels** (*list of strings*) – names of relevant variables

- **samples** (*np.array*) – samples for which statistical properties are calculated

AIMS.**write_readme**(*filename*, *elapsed_time*)

Write parameters relevant to this MCMC run.

Parameters

filename (*string*) – name of file in which to write the statistical properties

AIMS.**write_samples**(*filename*, *labels*, *samples*)

Write raw samples to a file.

Parameters

- **filename** (*string*) – name of file in which to write the samples
- **labels** (*list of strings*) – names of relevant variables (used to write a header)
- **samples** (*array-like*) – samples for which statistical properties are calculated

AIMS.**write_statistics**(*filename*, *labels*, *samples*)

Write statistical properties based on a sequence of realisations to a file. The results include:

- average values for each variable (statistical mean)
- error bars for each variable (standard mean deviation)
- correlation matrix between the different variables

Parameters

- **filename** (*string*) – name of file in which to write the statistical properties
- **labels** (*list of strings*) – names of relevant variables
- **samples** (*np.array*) – samples for which statistical properties are calculated

1.10 The model module

A module which contains various classes relevant to the grid of models:

- *Model*: a model
- *Track*: an evolutionary track
- *Model_grid*: a grid of models

These different classes allow the program to store a grid of models and perform a number of operations, such as:

- retrieving model properties
- interpolate within the grid models
- sort the models within a given evolutionary track
- ...

class model.**Model**(*_glb*, *_name=None*, *_modes=None*)

A class which contains a stellar model, including classical and seismic information.

Parameters

- **_glb** (*np.array*) – 1D array of global parameters for this model. Its dimension should be greater or equal to *nglb*
- **_name** (*string*) – name of the model (typically the second part of its path)

- **_modes** (*list of (int, int, float, float)*) – list of modes in the form of tuples (n,l,freq,inertia) which will be appended to the set of modes in the model.

property FeH

Find [Fe/H] value for model.

The conversion from (Xs,Zs) to [Fe/H] is performed using the following formula:

$$[\mathrm{Fe/H}] = \frac{[\mathrm{M/H}]}{\mathrm{A_{FeH}}} = \frac{1}{\mathrm{A_{FeH}}} \log_{10}\left(\frac{\mathrm{z/x}}{\mathrm{z_\odot/x_\odot}}\right)$$

Returns
the $[\mathrm{Fe/H}]$ value

Return type
float

> **Note**
>
> The relevant values are given in *constants*

property FeH0

Find initial [Fe/H] value for model.

The conversion from (X,Z) to [Fe/H] is performed using the following formula:

$$[\mathrm{Fe/H}] = \frac{[\mathrm{M/H}]}{\mathrm{A_{FeH}}} = \frac{1}{\mathrm{A_{FeH}}} \log_{10}\left(\frac{\mathrm{z/x}}{\mathrm{z_\odot/x_\odot}}\right)$$

Returns
the initial $[\mathrm{Fe/H}]$ value

Return type
float

> **Note**
>
> The relevant values are given in *constants*

property MH

Find [M/H] value for model.

The conversion from (Xs,Zs) to [M/H] is performed using the following formula:

$$[\mathrm{M/H}] = \log_{10}\left(\frac{\mathrm{z/x}}{\mathrm{z_\odot/x_\odot}}\right)$$

Returns
the $[\mathrm{M/H}]$ value

Return type
float

> **Note**
>
> The relevant values are given in *constants*

property MH0

Find initial [M/H] value for model.

The conversion from (X,Z) to [M/H] is performed using the following formula:

$$[\mathrm{M/H}] = \log_{10}\left(\frac{\mathrm{z/x}}{\mathrm{z_\odot/x_\odot}}\right)$$

Returns
the initial $[\mathrm{M/H}]$ value

Return type
float

> **Note**
>
> The relevant values are given in *constants*

append_modes(*modes*)

Append a list of modes to the model.

Parameters
modes (*list of (int, int, float, float)*) – list of modes which are in the form of tuples: (n,l,freq,inertia).

property b_Kjeldsen2008

Return the exponent for the Kjeldsen et al. (2008) surface correction recipe, as calculated based on the Sonoi et al. (2015) scaling relation.

Returns
the Kjeldsen et al. exponent

Return type
float

property beta_Sonoi2015

Return the exponent for the Sonoi et al. (2015) surface correction recipe, as calculated based on the Sonoi et al. (2015) scaling relation.

Returns
the Kjeldsen et al. exponent

Return type
float

property cutoff

Find $\nu_{\mathrm{cut-off}}$ for model.

The $\nu_{\mathrm{cut-off}}$ value is obtained from the following scaling relation:

$$\frac{\nu_{\mathrm{cut-off}}}{\nu_{\mathrm{cut-off},\odot}} = \left(\frac{M}{M_\odot}\right)\left(\frac{R}{R_\odot}\right)^2\left(\frac{T_{\mathrm{eff}}}{T_{\mathrm{eff},\odot}}\right)^{-1/2}$$

Returns
the $\nu_{\mathrm{cut-off}}$ value

Return type
float

> **Note**
>
> The relevant values are given in *constants*

`find_epsilon`(*ltarget*)

Find epsilon, the constant offset in a simplified version of Tassoul's asymptotic formula:

$\nu_n = \Delta\nu(n + \varepsilon)$

Parameters
ltarget (*int*) – target l value. Only modes with this l value will be used in obtaining epsilon.

Returns
the constant offset

Return type
float

`find_large_separation`()

Find large frequency separation using only radial modes.

Returns
the large frequency separation

Return type
float

`find_mode`(*ntarget*, *ltarget*)

Find a mode with specific n and l values.

Parameters

- **ntarget** (*int*) – target n value
- **ltarget** (*int*) – target l value

Returns
the frequency of the mode

Return type
float

`find_mode_range`()

Find n and l ranges of the modes in the model.

Returns
the n and l ranges of the modes

Return type
int, int, int, int

`freq_sorted`()

Check to see if the frequencies are in ascending order for each l value.

Returns
`True` if the frequencies are in ascending order.

Return type
boolean

`get_age()`

Return age of stellar model.

This is useful for sorting purposes.

Returns
the age of the model

Return type
float

`get_freq`(*surface_option=None*, *a=[]*)

Obtain model frequencies, with optional frequency corrections.

Parameters

- **`surface_option`** (*`string`*) – specifies the type of surface correction. Options include:
  - `None`: no corrections are applied
  - `"Kjeldsen2008"`: apply a correction based on Kjeldsen et al. (2008)
  - **`"Kjeldsen2008_scaling"`: apply a correction based on Kjeldsen et al. (2008).** The exponent is based on a scaling relation from Sonoi et al. (2015).
  - **`"Kjeldsen2008_2"`: apply a correction based on Kjeldsen et al. (2008).** The exponent is a free parameter.
  - `"Ball2014"`: apply a one-term correction based on Ball and Gizon (2014)
  - `"Ball2014_2"`: apply a two-term correction based on Ball and Gizon (2014)
  - `"Sonoi2015"`: apply a correction based on Sonoi et al. (2015)
  - **`"Sonoi2015_scaling"`: apply a correction based on Sonoi et al. (2015)** The exponent is based on a scaling relation from Sonoi et al. (2015).
  - **`"Sonoi2015_2"`: apply a correction based on Sonoi et al. (2015)** The exponent is a free parameter.
- **`a`** (*`array-like`*) – amplitude parameters which intervene in the surface correction

Returns
models frequencies (including surface corrections)

Return type
np.array

> **Note**
>
> If surface_option==None or a==[], the original frequencies are returned (hence modifying them modifies the *`Model`* object).

`get_surface_correction`(*surface_option*, *a*)

Obtain corrections on model frequencies (these corrections should be *added* to the *theorectical* frequencies).

Parameters

- **surface_option** (*string*) – specifies the type of surface correction. Options include:
  - None: no corrections are applied
  - "Kjeldsen2008": apply a correction based on Kjeldsen et al. (2008)
  - **"Kjeldsen2008_scaling": apply a correction based on Kjeldsen et al. (2008).** The exponent is based on a scaling relation from Sonoi et al. (2015).
  - **"Kjeldsen2008_2": apply a correction based on Kjeldsen et al. (2008).** The exponent is a free parameter.
  - "Ball2014": apply a one-term correction based on Ball and Gizon (2014)
  - "Ball2014_2": apply a two-term correction based on Ball and Gizon (2014)
  - "Sonoi2015": apply a correction based on Sonoi et al. (2015)
  - **"Sonoi2015_scaling": apply a correction based on Sonoi et al. (2015)** The exponent is based on a scaling relation from Sonoi et al. (2015).
  - **"Sonoi2015_2": apply a correction based on Sonoi et al. (2015)** The exponent is a free parameter.
- **a** (*array-like*) – parameters which intervene in the surface correction. According to the correction they take on the following meanings:
  - "Kjeldsen2008": a[0]*freq**b_Kjeldsen2008
  - "Kjeldsen2008_scaling": a[0]*freq**b_scaling
  - "Kjeldsen2008_2": a[0]*freq**a[1]
  - "Ball2014": a[0]*freq**3/I
  - "Ball2014_2": a[0]*freq**3/I + a[1]/(freq*I)
  - "Sonoi2015": a[0]*[1 - 1/(1 + (nu/numax)**beta_Sonoi2015)]
  - "Sonoi2015_scaling": a[0]*[1 - 1/(1 + (nu/numax)**beta_scaling)]
  - "Sonoi2015_2": a[0]*[1 - 1/(1 + (nu/numax)**a[1])]

Returns
surface corrections on the model frequencies

Return type
np.array

> **Note**
>
> The array operations lead to the creation of a new array with the result, which avoids modifications of the original frequencies and inertias.

glb

Array which will contain various global quantities

multiply_modes(*constant*)

Multiply the frequencies by constant.

Parameters
constant (*float*) – constant by which the mode frequencies are multiplied

name

Name of the model, typically the second part of its path

property numax

Find $\nu_{\rm max}$ for model.

The $\nu_{\rm max}$ value is obtained from the following scaling relation:

$$\frac{\nu_{\rm max}}{\nu_{\rm max,\odot}} = \left(\frac{M}{M_\odot}\right)\left(\frac{R}{R_\odot}\right)^{-2}\left(\frac{T_{\rm eff}}{T_{\rm eff,\odot}}\right)^{-1/2}$$

Returns
the $\nu_{\rm max}$ value

Return type
float

Note

The relevant values are given in *constants*

print_me()

Print classical and seismic characteristics of the model to standard output.

read_file(*filename*)

Read in a set of modes from a file. This method will either call *read_file_CLES()*, *read_file_MESA()*, or *read_file_agsm()* according to the value of the `mode_format` variable in `AIMS_configure.py`.

Parameters
filename (*string*) – name of the file with the modes. The format of this file is decided by the `mode_format` variable in `AIMS_configure.py`.

Returns
`True` if at least one frequency has been discarded (see note below).

Return type
boolean

Note

At this stage the frequencies should be expressed in μHz. They will be non-dimensionalised in `read_model_list_standard()`.

read_file_CLES(*filename*)

Read in a set of modes from a file. This uses the "simple" or "CLES" format as specified in the `mode_format` variable in `AIMS_configure.py`.

Parameters
filename (*string*) – name of the file with the modes. The file should contain a one-line header followed by five columns which correspond to l, n, frequency, unused, inertia.

Returns
`True` if at least one frequency has been discarded (see note below).

Return type
boolean

> **Note**
>
> - The fourth column is discarded.
> - Frequencies above *config.cutoff* * $\nu_{\mathrm{cut-off}}$ are discarded.

read_file_CLES_Mod(*filename*)

Read in a set of modes from a file. This uses the "CLES_Mod" format as specified in the `mode_format` variable in `AIMS_configure.py`.

Parameters
filename (*string*) – name of the file with the modes. The file may contain a header (the header lines start with "#") followed by four columns which correspond to l, n, unused, frequency.

Returns
`True` if at least one frequency has been discarded (see note below).

Return type
boolean

> **Note**
>
> - The third column is discarded.
> - Frequencies above *config.cutoff* * $\nu_{\mathrm{cut-off}}$ are discarded.

read_file_MESA(*filename*)

Read in a set of modes from a file. This uses the GYRE (i.e. MESA) format as specified in the `mode_format` variable in `AIMS_configure.py`.

Parameters
filename (*string*) – name of the file with the modes. The file should contain a seven-line header followed by various columns which contain l, n, frequency, and inertia for each pulsation mode.

Returns
`True` if at least one frequency has been discarded (see note below).

Return type
boolean

> **Note**
>
> - Frequencies above *config.cutoff* * $\nu_{\mathrm{cut-off}}$ are discarded.

read_file_PLATO(*filename*)

Read in a set of modes from a file. This uses the "PLATO" format as specified in the `mode_format` variable in `AIMS_configure.py`.

Parameters
filename (*string*) – name of the file with the modes. The file may contain a header (the

header lines start with "#") followed by four columns which correspond to l, n, frequency, inertia.

Returns
`False`

Return type
boolean

> **Note**
>
> - No mode selection is carried out, hence the return value
> - np.loadtxt is used to gain speed (the grid is quite large)

read_file_agsm(*filename*)

Read in a set of modes from a file. This uses the "agsm" format as specified in the `mode_format` variable in `AIMS_configure.py`.

Parameters
filename (*string*) – name of the file with the modes. This file is a binary fortran "agsm" file produced by the ADIPLS code. See instructions to the ADIPLS code for a description of this format.

Returns
`True` if at least one frequency has been discarded (see note below).

Return type
boolean

> **Note**
>
> - Frequencies above *config.cutoff* * $\nu_{\mathrm{cut-off}}$ are discarded.

read_modes_Aldo(*norder*, *farray*)

Read in a set of modes from a table. This uses the Aldo format as specified in the `mode_format` variable in `AIMS_configure.py`.

Parameters

- **norder** (*2D numpy int array*) – table which gives the starting n values and number of modes per l value
- **farray** (*1D numpy float array*) – table with mode frequencies and inertias as follows: frequency1, inertia1, frequency2, inertia2 ...

Returns
`True` if at least one frequency has been discarded (see note below).

Return type
boolean

> **Note**
>
> - Frequencies above *config.cutoff* * $\nu_{\mathrm{cut-off}}$ are discarded.

`remove_duplicate_modes()`

Remove duplicate modes.

Modes are considered to be duplicate if they have the same l and n values (regardless of frequency).

Returns
`True` if at least one mode has been removed.

Return type
boolean

> **Warning**
>
> This method assumes the modes are sorted.

`sort_modes()`

Sort the modes by l, then n, then freq.

`string_to_param(string)`

Return a parameter for an input string.

Parameters
`string` (*`string`*) – string that indicates which parameter we're seeking

Returns
the value of the parameter

Return type
float

`write_file_simple(filename)`

Write a set of modes into a file using the "simple" format as described in `read_file_simple()`.

Parameters
`filename` (*`string`*) – name of the file where the modes should be written.

> **Note**
>
> - The output frequencies are expressed in μHz

`property zsx_0`

Find the Z0/X0 value

Returns
the Z0/X0 value

Return type
float

`property zsx_s`

Find the Zs/Xs value

Returns
the Zs/Xs value

Return type
float

class model.**Model_grid**

A grid of models.

check_age_adim()

This checks that all of the tracks are sorted according to dimensionless age parameter.

distort_grid()

Define distortion matrix with which to distort grid to break its Cartesian character prior to tessellation. This can cause find_simplex to run much faster.

distort_mat

Transformation matrix used to break the Cartesian character of the grid and reduce computation time

find_epsilons(*ltarget*)

Find epsilon values in models from the grid

Parameters
ltarget (*int*) – target l value for which epsilons are being obtained

Returns
the epsilon values

Return type
list of floats

find_partition()

Find a partition of the grid for use with *Model_grid.test_interpolation()*

Returns
a random partition of [0 . . . n-1] into two equal halves, where n is the number of tracks in the grid

Return type
two lists of int

grid

Array containing the grid parameters for each evolutionary track (excluding age).

grid_params

Set of parameters (excluding age) used to construct the grid and do interpolations.

Note

For best interpolation results, these parameters should be comparable.

ndim

Number of dimensions for the grid (excluding age), as based on the *Model_grid.grid_params* variable

ndx

List containing track indices

plot_tessellation()

Plot the grid tessellation.

> **Warning**
>
> This only works for two-dimensional tessellations.

postfix

Last part of the filenames which contain the model frequencies (default = ".freq").

prefix

Root folder with grid of models (including final slash).

range(*aParam*)

Find range of values for the input parameter.

Parameters
aParam (*str*) – name of the parameter for which to find the range

> **Warning**
>
> The input parameter can only be one of the grid parameters or an age/mHe parameter.

read_model_list(*filename*)

Read list of models from a file and construct a grid. If the mode format is Aldo, an entirely different strategy must be used.

Parameters
filename (*string*) – name of the file with the list.

read_model_list_Aldo(*filename*)

Read list of models in Aldo format from a file and construct a grid.

A track in Aldo format contains 3 parts:

1. A three line header which is discarded.
2. A set of four lines, for l=0 to 3, which describe the mode structure. The first columns gives the start value of n, whereas the second column gives the number of modes for that particular l value.
3. A table where each line corresponds to a model and the different columns correspond to the model ID, global quantities, and pulsation frequencies and inertias.

Parameters
filename (*string*) – name of the file with the list of track files.

read_model_list_standard(*filename*)

Read list of models from a file and construct a grid.

Parameters
filename (*string*) – name of the file with the list. The first line of this file should contain a prefix which is typically the root folder of the grid of models. This followed by a file with multiple columns. The first 9 contain the following information for each model:

1. the second part of the path. When concatenated with the prefix on the first line, this should give the full path to the model.
2. The stellar mass in g
3. The stellar radius in cm

4. The stellar luminosity in $\mathrm{g.cm^2.s^{-3}}$
5. The metallicity
6. The hydrogen content
7. The stellar age in Myrs
8. The effective temperature in K
9. A dimensionless age parameter

The following columns contain the parameters specified in the `AIMS_configure.user_params` variable.

remove_tracks(*nthreshold*)

Removes stellar evolution tracks with fewer than nthreshold models.

Parameters
nthreshold (*int*) – lower limit on number of models in a stellar evolutionary track

Returns
True if tracks have been removed and the grid needs to be retessellated

Return type
boolean

replace_age_adim()

This replaces the dimensionless ages in the tracks according to the *replace_age_adim* option chosen in `AIMS_configure.py`.

tessellate()

Apply Delauny triangulation to obtain the grid tessellation.

tessellation

Object containing the tessellation of the grid used for interpolation.

test_freq()

Test to see if frequencies in all of the models of the grid are in ascending order for each l value.

Returns

The following items are returned

- the effective temperatures of the models with frequencies out of order
- the luminosities of the models with frequencies out of order
- the effective temperatures of the models with sorted frequencies
- the luminosities of the models with sorted frequencies

Return type
four lists of floats

test_interpolation()

Test interpolation between different evolutionary tracks in a given grid.

Returns

The following four items are returned:

- the interpolation errors
- the first half of the partition (where the interpolation is tested)

- the second half of the partition (used to carry out the interpolation)
- the tessellation associated with the second half of the partition

Return type
np.array, list, list, tessellation object

tracks
List of evolutionary tracks contained in the grid.

user_params
The set of user parameters involved in the grid. This is to avoid having a different set of user parameters in *AIMS_configure.py*

class model.**Track**(*aModel*, *grid_params*)
An evolutionary track.

Parameters

- **aModel** (*Model*) – first model to be added to evolutionary track (it does not need to be the youngest model in an evolutionary sequence). This Model is used to obtain the relevant parameters for the evolutionary track (as given by the *grid_params* variable).
- **grid_params** (*list of strings*) – list of strings which are the names of the parameters which describe the evolutionary track.

age_adim_to_age()
Replace dimensionless age parameter by the physical age.

> **Warning**
> This method should only be applied after the track has been sorted (according to age).

age_adim_to_scale_Xc()
Replace dimensionless age parameter by the central hydrogen abundance, Xc, scaled from 0 to 1.

> **Warning**
> This method should only be applied after the track has been sorted (according to age).

age_adim_to_scale_age()
Replace dimensionless age parameter by the physical age scaled from 0 to 1.

> **Warning**
> This method should only be applied after the track has been sorted (according to age).

property age_lower
Provides the lowest age an evolutionary track.

property age_range
Provides the age range for an evolutionary track.

property age_upper

Provides the highest age for an evolutionary track.

append(*aModel*)

Append a model to the evolutionary track.

Parameters
aModel (*Model*) – model which is being appended to the track

append_track(*aTrack*)

Append a track to the current evolutionary track (i.e. combine the two tracks), and remove the track which has been appended. The resultant track is then sorted according to age.

Parameters
aModel (*Model*) – model which is being appended to the track

duplicate_ages()

Check to see if the evolutionary track contains models with duplicate ages.

Returns
`True` if there are duplicate age(s)

Return type
boolean

> **Warning**
>
> This method should only be applied after the track has been sorted.

find_combination(*age*, *coef*)

Return a model combination at a given age which is obtained using linear interpolation.

Parameters

- **age** (*float*) – age of desired model in Myrs
- **coef** (*float*) – coefficient which multiplies this combination

Returns
pairs composed of an interpolation coefficient and a model name

Return type
tuple of (float, string)

> **Warning**
>
> This method assumes the track is sorted, since it applies a binary search algorithm for increased efficiency.

find_epsilon(*imodel*, *ltarget*)

Find epsilon, the constant offset in a simplified version of Tassoul's asymptotic formula:

$\nu_n = \Delta\nu(n + \varepsilon)$

Parameters

- **imodel** (*ltarget*) – model number

- **ltarget** (*int*) – target l value. Only modes with this l value will be used in obtaining epsilon.

Returns
the constant offset

Return type
float

find_large_separation(*imodel*)

Find large frequency separation using only radial modes.

Parameters
imodel (*ltarget*) – model number

Returns
the large frequency separation

Return type
float

find_mode_range()

Find n and l ranges of modes in models

Returns
the n and l ranges

Return type
int, int, int, int

find_modes(*ntarget*, *ltarget*)

Return two lists, one with the ages of the models and the other with the mode non-dimensional frequencies corresponding to target n and l values.

This function is useful for seeing how the frequency of a particular mode changes with stellar age.

Parameters

- **ntarget** (*int*) – target n value
- **ltarget** (*int*) – target l value

Returns
lists of ages and frequencies

Return type
list, list

find_modes_dim(*ntarget*, *ltarget*)

Return two lists, one with the ages of the models and the other with the mode dimensional frequencies corresponding to target n and l values.

This function is useful for seeing how the frequency of a particular mode changes with stellar age.

Parameters

- **ntarget** (*int*) – target n value
- **ltarget** (*int*) – target l value

Returns
lists of ages and frequencies

Return type
list, list

freq_sorted(*imodel*)

Check to see if the frequencies of model i are in ascending order for each l value.

Parameters
imodel (*int*) – the number of the model to be checked

Returns
True if the frequencies are in ascending order.

Return type
boolean

glb

Global properties of the models

grid_params

Names of the parameters used to construct the grid

interpolate_model(*age*)

Return a model at a given age which is obtained using linear interpolation.

Parameters
age (*float*) – age of desired model in Myrs

Returns
the interpolated model

Return type
Model

Warning

This method assumes the track is sorted, since it applies a binary search algorithm for increased efficiency.

is_sorted()

Check to see of models are in ascending order according to age.

Returns
True if the models ar in order of increasing age

Return type
boolean

is_sorted_adim()

Check to see of models are in ascending order according to age.

Returns
True if the models ar in order of increasing age

Return type
boolean

matches(*aModel*, *params_tol=None*)

Check to see if a model matches the evolutionary track and can therefore be included in the track.

Parameters

- **aModel** (*Model*) – input model being tested

- **params_tol** (*float np.array*) – the tolerance on each parameter. `None` means no tolerance (i.e. the parameters have to be exactly the same).

Returns
`True` only if all of the parameters of the input model match those of the evolutionary track within the provided tolerances

Return type
boolean

mode_indices

starting indices in *Track.modes* array corresponding to a given model

modes

array containing the modes (n, l, freq, inertia) of all of the models

names

List of model names

nmodes

Total number pulsation modes from all of the models in this evolutionary track

params

Parameters which characterise this evolutionary track

remove_duplicate_ages()

Removes models with duplicate ages from the evolutionary track.

Returns
`True` if models with duplicate age(s) have been removed

Return type
boolean

> **Warning**
>
> This method should only be applied after the track has been sorted.

sort()

Sort models within evolutionary track according to age.

test_interpolation(*nincr*)

Test accuracy of interpolation along evolutionary track.

This method removes every other model and retrieves its frequencies by interpolation from neighbouring models. The accuracy of the interpolated frequencies and global parameters are tested by carrying out comparisons with the original models.

Parameters
nincr (*int*) – increment with which to carry out the interpolation. By comparing results for different values of `nincr`, one can evaluate how the interpolation error depends on the size of the interval over which the interpolation is carried out.

Returns
the interpolation errors

Return type
np.array

model.**age_adim_str = 'Age_adim'**

Name of the non-dimensional age parameter used in the interpolation

model.**age_str = 'Age'**

Name of the physical age parameter used in the interpolation

model.**combine_models**(*model1*, *coef1*, *model2*, *coef2*)

Do linear combination of this model with another.

This method returns a new model which is the weighted sum of two models for the purposes of model interpolation. The classical parameters are combined in a self-consistent way as are the frequencies.

Parameters

- **model1** (*Model*) – first model
- **coef1** (*float*) – weighting coefficient applied to first model
- **model2** (*Model*) – second model
- **coef2** (*float*) – weighting coefficient applied to second model

Returns

the combined model

Return type

Model

Warning

One should avoid negative or zero coefficients as these could lead to undefined results.

model.**compare_models**(*model1*, *model2*)

Compare two models and find the largest frequency different for radial and non-radial modes.

Parameters

- **model1** (*Model*) – first model
- **model2** (*Model*) – second model

Returns

a 1D array to be used in `plot_test_interpolation.py` with the following measurements of the differences between the two models:

- `result[0]` = maximum error on the radial modes
- `result[1]` = RMS error on the radial modes
- `result[2]` = RMS error on the radial modes near $\nu_{\max}$
- `result[3]` = maximum error on the non radial modes
- `result[4]` = RMS error on the non radial modes
- `result[5]` = RMS error on the non radial modes near $\nu_{\max}$
- `result[6+[0:nglb]]` = errors on the global parameters

Return type

np.array

`model.eps = 1e-06`
relative tolerance on parameters used for setting up evolutionary tracks

`model.find_ages`(*coefs*, *tracks*, *age*)
Find ages to which each track needs to be interpolated for a specified age. The variable `age_interpolation` in AIMS_configure.py decides between the following options:

1. `age_interpolation` = `age`: each track is simply interpolated to `age`.
2. `age_interpolation` = `scale_age`: the age of each model along each evolutionary track, including the interpolated track, is linearly mapped onto the interval [0,1]. A dimensionless parameter `eta` is obtained by interpolating `age` onto the interval [0,1], using the linear transformation associated with the interpolated track. Using the parameter eta, a corresponding age is obtained along each track.
3. `age_interpolation` = `age_adim`: each track is interpolated to the appropriate dimensionless age so that the interpolated physical age reproduces the input age. This dimensionless age parameter is calculated via a fortran subroutine.

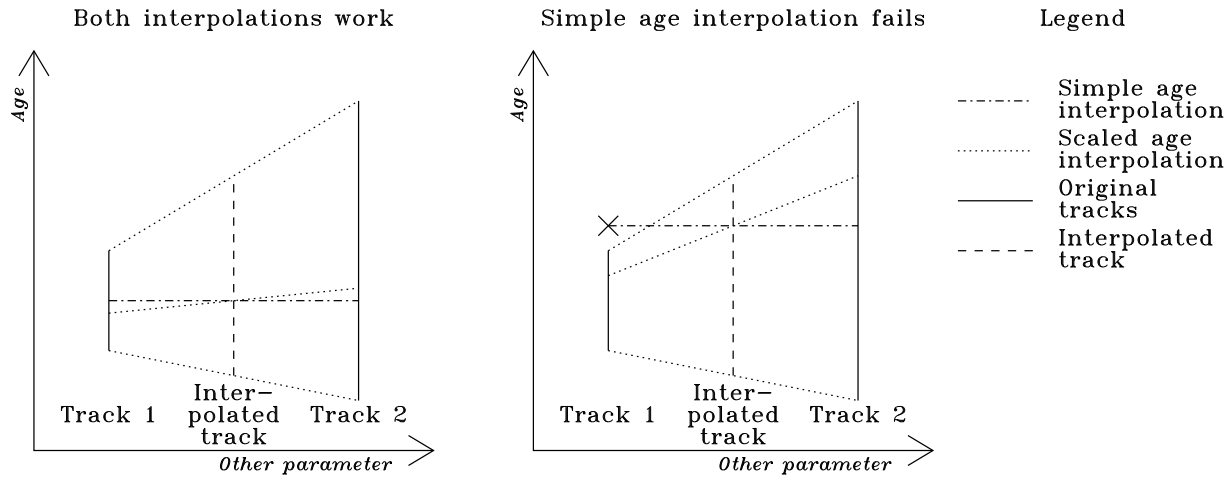

Fig. 1: This diagram illustrates age interpolation for the two first options, namely `age` and `scale_age`, and shows the advantages of selecting the latter.

Parameters

- **coefs** (*list of floats*) – interpolation coefficients used to weight each track.
- **tracks** (list of *Track*) – evolutionary tracks involved in the interpolation.
- **age** (*float*) – target age for the output interpolated model.

Returns
the relevant age for each track

Return type
list of floats

Note

- the interpolation coefficients should add up to 1.0
- there should be as many tracks as interpolation coefficients.

model.**find_combination**(*grid*, *pt*)

Find linear combination of models which corresponds to interpolating the model based on the provided parameters.

The interpolation is carried out using the same procedure as in *interpolate_model()*.

Parameters

- **grid** (*Model_grid*) – grid of models in which we're carrying out the interpolation
- **pt** (*array-like*) – set of parameters used for the interpolation. The first part contains the grid parameters, whereas the last element is the age. If the provided set of parameters lies outside the grid, then `None` is returned instead of an interpolated model.

Returns
pairs of coefficients and model names

Return type
tuple of (float,string)

model.**find_interpolation_coefficients**(*grid*, *pt*, *tessellation*, *ndx*)

Find interpolation weights from the corresponding simplex.

Linear interpolation weights are obtained with the simplex by finding the barycentric coordinates of the point given by `pt`.

Parameters

- **grid** (*Model_grid*) – grid of models in which we're carrying out the interpolation
- **pt** (*array-like*) – set of parameters used for finding the interpolation weights. The first part contains the grid parameters (relevant to this interpolation), whereas the last element is the age (not used here). If the provided set of parameters lies outside the grid, then `None` is returned instead of an interpolated model.
- **tessellation** (*Delaunay tessellation object*) – tessellation with which to carry out the interpolation.
- **ndx** (*list of int*) – indices of the grid points associated with the tessellation

Returns
lists of interpolation coefficients and tracks

Return type
list of floats, list of *Track*

model.**ftype**

type used for the frequencies

model.**get_surface_parameter_names**(*surface_option*)

Return the relevant parameter names for a given surface correction option.

Parameters
surface_option (*string*) – specifies the type of surface correction.

Returns
names for the surface parameters

Return type
tuple of strings

model.**gtype**

type used for grid data

model.**iage = 1**

index of the parameter corresponding to age in the *Model.glb* array

model.**iage_adim = 0**

index of the parameter corresponding to dimensionless age in the *Model.glb* array

model.**ifreq_ref = 12**

index of the parameter corresponding to the reference frequency (used to non-dimensionalise the pulsation frequencies of the model) in the *Model.glb* array

model.**iluminosity = 14**

index of the parameter corresponding to luminosity in the *Model.glb* array

model.**imass = 2**

index of the parameter corresponding to mass in the *Model.glb* array

model.**init_user_param_dict**()

Initialise the dictionaries which are related to user-defined parameters. For a given parameter, these dictionaries provide the appropriate index for for the *Model.glb* array as well as the appropriate latex name.

model.**interpolate_model**(*grid*, *pt*, *tessellation*, *ndx*)

Interpolate model in grid using provided parameters.

The interpolation is carried out in two steps. First, linear interpolation according to age is carried out on each node of the simplex containing the set of parameters. This interpolation is done using the *Track.interpolate_model* method. Then, linear interpolation is carried out within the simplex. This achieved by finding the barycentric coordinates of the model (i.e. the weights), before combining the age-interpolated models form the nodes using the *combine_models* method. In this manner, the weights are only calculated once, thereby increasing computational efficiency.

Parameters

- **grid** (*Model_grid*) – grid of models in which we're carrying out the interpolation
- **pt** (*array-like*) – set of parameters used for the interpolation. The first part contains the grid parameters, whereas the last element is the age. If the provided set of parameters lies outside the grid, then `None` is returned instead of an interpolated model.
- **tessellation** (*Delaunay tessellation object*) – tessellation with which to carry out the interpolation.
- **ndx** (*list of int*) – indices of the grid points associated with the tessellation

Returns

the interpolated model

Return type

Model

model.**iradius = 13**

index of the parameter corresponding to radius in the *Model.glb* array

model.**itemperature = 3**

index of the parameter corresponding to temperature in the *Model.glb* array

model.**ix0 = 5**

index of the parameter corresponding to the initial hydrogen content in the *Model.glb* array

model.**iz0 = 4**

index of the parameter corresponding to the initial metallicity the *Model.glb* array

model.**log0 = -1e+150**

value which is returned when log(0) is calculated (rather than causing an error)

model.**ltype**

type used for the l values

model.**make_distort_matrix**(*d*, *theta=0.157*)

Create a distortion matrix which can be used to make the grid more "tessellation-friendly", i.e. which leads to much shorter computation times for finding simplices.

Parameters

- **d** (*int*) – number of dimensions
- **theta** (*float*) – a small angle

Returns
a distortion matrix

Return type
2D float array

model.**make_scale_matrix**(*grid*)

Create a distortion matrix which can be used to make the grid more "tessellation-friendly", i.e. which leads to much shorter computation times for finding simplices.

Parameters
grid – set of points used in the construction of the tessellation

Returns
a distortion matrix

Return type
2D float array

model.**modetype = [('n', <class 'numpy.int16'>), ('l', <class 'numpy.int8'>), ('freq', <class 'numpy.float64'>), ('inertia', <class 'numpy.float64'>)]**

structure for modes

model.**nglb = 15**

total number of global quantities in a model (see *Model.glb*).

model.**nlin = 12**

total number of global quantities which are interpolated in a linear way (see *combine_models()*). These quantities are numbered 0:nlin-1

model.**ntype**

type used for the n values

model.**string_to_latex**(*string*, *prefix=''*, *postfix=''*)

Return a fancy latex name for an input string.

Parameters

- **string** (*string*) – string that indicates for which parameter we're seeking a latex name
- **prefix** (*string*) – optional prefix to add to the string
- **postfix** (*string*) – optional postfix to add to the string

Returns
a fancy latex name

Return type
string

> **Note**
>
> This also works for the names of the amplitude parameters for surface corrections.

`model.tol = 1e-06`
tolerance level for points slightly outside the grid

`model.user_params_index = {}`
dictionary which will supply the appropriate index for the user-defined parameters

`model.user_params_latex = {}`
dictionary which will supply the appropriate latex name for the user-defined parameters

1.11 The constants module

A module which contains the following physical constants:

Name of variable	Quantity it describes	Units
solar_radius	the solar radius	cm
solar_mass	the solar mass	g
solar_luminosity	the solar luminosity	$\mathrm{g.cm^2.s^{-3}}$
solar_temperature	the solar effective temperature	K
solar_dnu	the solar large frequency separation	μHz
solar_numax	the solar frequency at maximum power	μHz
solar_cutoff	the solar cutoff frequency	μHz
G	the gravitational constant	$\mathrm{cm^3.g^{-1}.s^{-2}}$
solar_x	the solar hydrogen content	dimensionless
solar_z	the solar metallicity content	dimensionless
Yp	primordial helium	dimensionless
A_FeH	multiplicative constant in $[\mathrm{M/H}] = \mathrm{A_{FeH}}[\mathrm{Fe/H}]$	dimensionless

> **Note**
>
> These values can be edited according to the latest discoveries. As good practise, it is helpful to include the relevant reference.

`constants.A_FeH = 1.0`
multiplicative constant which intervenes in the conversion from metal content to iron content

`constants.G = 6.67428e-08`
the gravitational constant in $\mathrm{cm^3.g^{-1}.s^{-2}}$

`constants.Yp = 0.248`
Primordial helium

`constants.solar_cutoff = 5300.0`
the solar cut-off frequency separation in μHz

constants.**solar_dnu = 135.1**

the solar large frequency separation in μHz

constants.**solar_luminosity = 3.8418e+33**

the solar luminosity in $\mathrm{g.cm^2.s^{-3}}$

constants.**solar_mass = 1.9892e+33**

the solar mass in g

constants.**solar_numax = 3090.0**

the solar frequency at maximum power in μHz

constants.**solar_radius = 69598000000.0**

the solar radius in cm

constants.**solar_temperature = 5777.0**

the solar temperature in K

constants.**solar_x = 0.7345**

the solar hydrogen content

constants.**solar_z = 0.0169**

the solar metallicity content

1.12 The functions module

A module which contains various reference functions which are useful for frequency combination functions.

functions.**identity**(*vec*)

Apply the identity function to the input value (x).

We recall that the identify function is expressed as follows: $f(x) = x$

Parameters

vec (*float array like*) – the input vector

Warning

No verifications are carried out within the function so the user must make sure the input vector only has one component, as implicitely assumed.

functions.**identity_gradient**(*vec*)

Apply the gradient of the identity function to the input value (x).

We recall that the gradient of the identify function is expressed as follows: $\vec{\Delta} f(x) = 1$

Parameters

vec (*float array like*) – the input vector

Warning

No verifications are carried out within the function so the user must make sure the input vector only has one component, as implicitely assumed.

functions.**norm**(*vec*)

Return the norm of an input vector.

We recall that the norm is expressed as follows: $\|(\vec{v})\| = \sqrt{\sum_i v_i^2}$

Parameters
vec (*float array like*) – the input vector

functions.**norm_gradient**(*vec*)

Return the gradient of the norm function for an input vector.

We recall that the gradient of the norm function takes on the following expression: $\vec{\Delta}\|(\vec{v})\| = \frac{\vec{v}}{\|\vec{v}\|}$

Parameters
vec (*float array like*) – the input vector

functions.**ratio**(*vec*)

Return the ratio for input values (x,y).

We recall that the ratio function is expressed as follows: $f(x, y) = \frac{x}{y}$

Parameters
vec (*float array like*) – the input vector

> **Warning**
>
> No verifications are carried out within the function so the user must make sure the input vector only has two components, as implicitely assumed.

functions.**ratio_gradient**(*vec*)

Return the gradient of the ratio function for input values (x,y).

We recall that gradient of the ratio function is expressed as follows: $\vec{\Delta} f(x, y) = \left(\frac{1}{y}, -\frac{x}{y^2}\right)$

Parameters
vec (*float array like*) – the input vector

> **Warning**
>
> No verifications are carried out within the function so the user must make sure the input vector only has two components, as implicitely assumed.

1.13 The `utilities` module

A module which contains various utility methods for handling strings and floats.

utilities.**is_number**(*s*)

Test a string to see if it is a number.

Parameters
s (*string*) – string which is being tested

Returns
`True` if `s` is a number, and `False` otherwise

Return type
boolean

> **Note**
>
> This method allows "d" and "D" as an exponent (i.e. for Fortran style numbers).

`utilities.`**`my_input`**`()`

Select appropriate input function depending on whether python2 or python3 is being used.

`utilities.`**`my_map`**`(`*fct*, *lst*`)`

Systematically applies a function to a list of items. This deals with the python3 behaviour of map which returns a map object rather than a list.

Parameters

- **`fct`** (*`function`*) – the function to be applied to each element of a list
- **`lst`** (*`list`*) – the list to which is applied the function

`utilities.`**`sparse_print`**`(`*filename*, *mat*`)`

Print a sparse matrix (for debug purposes only):

Parameters

- **`filename`** (*`string`*) – name of the file in which to print the matrix
- **`mat`** (*`numpy array`*) – the matrix to be printed

`utilities.`**`to_float`**`(`*s*`)`

Convert a string to a float.

Parameters
`s` (*`string`*) – string which will be converted to a float

Returns
the corresponding float

Return type
float

> **Note**
>
> This method allows "d" and "D" as an exponent (i.e. for Fortran style numbers).

`utilities.`**`trim`**`(`*s*`)`

Return a string with comments (starting with "#") removed.

Parameters
`s` (*`string`*) – the string for which we would like to remove comments.

Returns
the string without comments

Return type
string

1.14 The `plot_interpolation_test` tool

An interactive utility which plots various forms of interpolation error, stored in a binary file produced by `AIMS.test_interpolation()`. It specifically tests the errors from two types of interpolation:

- age interpolation: this is interpolation along a given evolutionary track
- track interpolation: this is interpolation between different evolutionary tracks

This utility allows various types of plots:

- 3D plots of interpolation errors as a function of grid structural parameters
- 2D slices which show interpolation errors as a function of age for a given evolutionary track
- interactive plots which allow you to select 2D slices

Note

Interpolation errors for models in a given evolutionary track are typically stored in arrays as follows:

- `result[model_number,ndim+0]` = maximum error on the radial modes
- `result[model_number,ndim+1]` = RMS error on the radial modes
- `result[model_number,ndim+2]` = RMS error on the radial modes near $\nu_{\rm max}$
- `result[model_number,ndim+3]` = maximum error on the non radial modes
- `result[model_number,ndim+4]` = RMS error on the non radial modes
- `result[model_number,ndim+5]` = RMS error on the non radial modes near $\nu_{\rm max}$
- `result[model_number,ndim+6+[0:nglb]]` = errors on the global parameters

where:

- `result` = the array which containts the interpolation errors
- `model_numer` = an index which represents the model (not necessarily the number of the model along the evolutionary track
- `ndim` = the number of dimensions in the grid (including age)
- `nglb` = the number of global parameters for stellar models in the grid

Warning

This plot utility only works with 3 dimensional grids (incl. the age dimension).

`plot_interpolation_test.`**`all_nan`**(*array*)

Test to see if all of the elements of an array are nan's.

Parameters
`array` (*`np.array`*) – array in which we're checking to see if all elements are nan's.

Returns
`True` if all the elements of `array` are nan's, and `False` otherwise.

Return type
boolean

plot_interpolation_test.**ndim = 0**

number of dimension in grid (including age)

plot_interpolation_test.**nglb = 0**

number of global parameters

plot_interpolation_test.**onpick_age**(*event*)

Event catcher for the grid plot (which shows the positions of the evolutionary tracks as a function of the grid parameters, excluding age).

Parameters:

Parameters

event – event caught by the grid plot.

plot_interpolation_test.**onpick_track**(*event*)

Event catcher for the partition tessellation plot (associated with tests of track interpolation).

Parameters

event – event caught by the partition tessellation plot.

plot_interpolation_test.**plot3D**(*results*, *error_ndx*, *tpe='max'*, *title=None*, *truncate=0*)

Create 3D plot showing the error as a function of the two first grid parameters.

Parameters

- **results** (*list of np.arrays*) – list of 2D arrays which contain various types of errors as a function of the model number along a given evolutionary track.
- **error_ndx** (*int*) – value which specifies the type of error to be plotted.
- **tpe** (*string*) – specifies how to combine errors along the evolutionary track. Options include:
  - "max": corresponds to taking the maximum value.
  - "avg": takes the root mean-square value.
- **title** (*string*) – the title of the plot
- **truncate** (*int*) – (default = 0): specifies how many models should be omitted on both ends of the track. This is useful for comparing results from tests involing different sizes of increments.

Note

See above introductory description for a more detailled description of the indices which intervene in the 2D arrays contained in `results` and of the relevant values for `error_ndx`.

plot_interpolation_test.**plot_grid**(*grid*)

Make an interactive plot of the grid. Clicking on the blue dots will produce a 2D slice showing age interpolation errors for the associated evolutionary track.

Parameters

grid (*np.array*) – array containing basic grid parameters (excluding age)

Warning

This only works for two-dimensional grids (excluding the age dimension).

plot_interpolation_test.**plot_partition_tessellation**(*grid*, *ndx1*, *ndx2*, *tessellation*)

Make an interactive tessellation plot based on the supplied partition on the grid. Clicking on the blue dots will produce a 2D slice showing track interpolation errors for the associated evolutionary track.

Parameters

- **grid** (*np.array*) – array containing basic grid parameters (excluding age)
- **ndx1** (*list of int*) – list with the indices of the first part of the partition.
- **ndx2** (*list of int*) – list with the indices of the second part of the partition.
- **tessellation** – grid tessellation associated with ndx2

Warning

This only works for two-dimensional tessellations.

plot_interpolation_test.**plot_slice_age**(*pos*)

Plot age interpolation error as a function of age for a given track.

Parameters

pos (*int*) – index of the relevant track.

Note

This *pos* index applies to results_age, i.e., it is based on the original track indices.

plot_interpolation_test.**plot_slice_track**(*pos*)

Plot track interpolation error as a function of age for a given track.

Parameters

pos (*int*) – index of the relevant track.

Note

This *pos* index applies to results_track, i.e., it is based on the indices deduced from the grid partition.

plot_interpolation_test.**results_age = None**

list which contains the arrays with the errors from age interpolation

plot_interpolation_test.**results_track = None**

list which contains the arrays with the errors from track interpolation

plot_interpolation_test.**titles = None**

the grid quantities, which will serve as axis labels

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

Y

Z