Bagpipes Documentation

Release 0.8.1

Adam Carnall

Dec 02, 2019
## Contents

1 Source and installation 3
2 Citation 5
3 Getting started 7
4 Acknowledgements 9
Index 23
Bayesian Analysis of Galaxies for Physical Inference and Parameter ESTimation is a state of the art Python code for modelling galaxy spectra and fitting spectroscopic and photometric observations.

Bagpipes is currently in a state of active development in the run-up to the release of version 1. Please email me if you have any queries, or find things which don’t work as they’re supposed to.
Bagpipes is developed at GitHub, however the code cannot be installed from there, as the large model grid files aren’t included in the repository. The code should instead be installed with pip:

```
pip install bagpipes
```

All of the code’s Python dependencies will be automatically installed. The only non-Python dependency is the MultiNest nested sampling algorithm (used only for fitting). To install MultiNest see point 1 of the “on your own computer” section of the PyMultiNest installation instructions.

In my experience, the sequence of commands necessary to install MultiNest on a mac is as follows:

```
git clone https://github.com/JohannesBuchner/MultiNest
brew install gcc49
export DYLD_LIBRARY_PATH="/usr/local/bin/gcc-4.9:$DYLD_LIBRARY_PATH"
cd MultiNest/build
cmake ..
make
sudo make install
cd ..
rm -r MultiNest
```
Bagpipes is described in Section 3 of Carnall et al. 2018. If you make use of Bagpipes, please include a citation to this work in any publications. Please note development of the code has been ongoing since this work was published, so certain parts of the code are no longer as described.
The best place to get started is by looking at the iPython notebook examples. It’s a good idea to tackle them in order as the later examples build on the earlier ones. These documentation pages contain a more complete reference guide.

Bagpipes is structured around three core classes:

- `model_galaxy`: for generating model galaxy spectra
- `galaxy`: for loading observational data into Bagpipes
- `fit`: for fitting models to observational data.
A few of the excellent projects Bagpipes relies on are:

- The Bruzual & Charlot 2003 stellar population models.
- The Draine & Li 2007 dust emission models.
- The MultiNest nested sampling algorithm (Feroz et al. 2013)
- The PyMultiNest Python interface for Multinest (Buchner et al. 2014).
- The Cloudy photoionization code (Ferland et al. 2017).
- The Deepdish HDF5 loading/saving interface.

### 4.1 Making model spectra: model_galaxy

Model galaxy spectra and associated observables are created using the `model_galaxy` class. Check out the first iPython notebook example for a quick-start guide to making models.

```python
class bagpipes.model_galaxy(model_components, filt_list=None, spec_wavs=None, spec_units='ergscma', phot_units='ergscma', index_list=None)
```

Builds model galaxy spectra and calculates predictions for spectroscopic and photometric observables.

**Parameters**

- `model_components (dict)` – A dictionary containing information about the model you wish to generate.
- `filt_list (list - optional)` – A list of paths to filter curve files, which should contain a column of wavelengths in angstroms followed by a column of transmitted fraction values. Only required if photometric output is desired.
- `spec_wavs (array - optional)` – An array of wavelengths at which spectral fluxes should be returned. Only required of spectroscopic output is desired.
• **spec_units** *(str - optional)* – The units the output spectrum will be returned in. Default is “ergs/cm²” for ergs per second per centimetre squared per angstrom, can also be set to “μJy” for microjanskys.

• **phot_units** *(str - optional)* – The units the output spectrum will be returned in. Default is “ergs/cm²” for ergs per second per centimetre squared per angstrom, can also be set to “μJy” for microjanskys.

• **index_list** *(list - optional)* – list of dicts containing definitions for spectral indices.

**update**(model_components)

Update the model outputs to reflect new parameter values in the model_components dictionary. Note that only the changing of numerical values is supported.

### 4.1.1 The model_components dictionary

The first and most important argument passed to `model_galaxy` is the `model_components` dictionary. This contains all of the physical information about the model you wish to create. A complete guide to the `model_components` dictionary is provided [here](#).

### 4.1.2 Getting observables - photometry

In order to obtain predictions for photometric observations of a galaxy with the physical parameters defined in `model_components` it is necessary to define a list of filter curves through which observed fluxes should be calculated. This is done by defining a `filt_list`.

This is simply a list of paths (absolute or from the directory in which the code is being run) to the locations at which these filter curves are stored. The filter curve files should contain wavelengths in Angstroms in their first column and relative transmission values in their second.

Let’s look at a simple example of some code which creates predictions for photometry through a series of filter curves. For this to work you’d first need to put the filter curve files in the correct location. For sourcing filter curves I recommend the SVO Filter Profile Service.

```python
import bagpipes as pipes
import numpy as np

uvista_filt_list = [
    "uvista/CFHT_u.txt",
    "uvista/CFHT_g.txt",
    "uvista/CFHT_r.txt",
    "uvista/CFHT_i+i2.txt",
    "uvista/CFHT_z.txt",
    "uvista/subaru_z",
    "uvista/VISTA_Y.txt",
    "uvista/VISTA_J.txt",
    "uvista/VISTA_H.txt",
    "uvista/VISTA_K.txt",
    "uvista/VISTA_Ks.txt",
    "uvista/IRAC1",
    "uvista/IRAC2"
]

model = pipes.model_galaxy(model_components, filt_list=uvista_filt_list)
model.plot()
```

We now have a Bagpipes model galaxy! The final command generates a plot of the predicted fluxes.
Photometry is accessible as `model.photometry`, which is a 1D array of flux values in erg/s/cm^2/A in the same order as the filter curves are specified in `filt_list`. The output flux units can be converted to microJanskys using the `model_galaxy` keyword argument `phot_units="mujy"`.

### 4.1.3 Getting observables - spectroscopy

The process for obtaining model spectroscopy is simpler, just pass an array containing the desired wavelength sampling in Angstroms as the `spec_wavs` keyword argument.

```python
import bagpipes as pipes
import numpy as np

obs_wavs = np.arange(2500., 7500., 5.)
model = pipes.model_galaxy(model_components, spec_wavs=obs_wavs)
model.plot()
```

The output spectrum is stored as `model.spectrum` which is a two column array, containing wavelengths in Angstroms and spectral fluxes in erg/s/cm^2/A by default. The output flux units can be converted to microJanskys using the `model_galaxy` keyword argument `spec_units="mujy"`.

### 4.1.4 Getting observables - line fluxes

Emission line fluxes are stored in the `model_galaxy.line_fluxes` dictionary. The list of emission features is here. These are only non-zero if a nebular component is added to `model_components`. Emission line naming conventions are the same as in Cloudy. The names in the above file are the keys for the lines in `model_galaxy.line_fluxes`. For example, the Lyman alpha flux is under:

```python
model.line_fluxes["H  I  1215.67A"]
```

Emission line fluxes are returned in units of erg/s/cm^2.

### 4.1.5 Note on units at redshift zero

The units specified above apply at non-zero redshift, however at redshift zero the luminosity distance is zero which would lead to a division by zero error. At redshift zero the code instead returns luminosities, in erg/s/A for spectroscopy and photometry, and erg/s for emission lines.

### 4.1.6 Updating models

Creating a new `model_galaxy` is relatively slow, however changing parameter values in `model_components` and calling the `update` method of `model_galaxy` rapidly updates the output predictions described above.

It should be noted that the `update` method is designed to deal with changing numerical parameter values, not with adding or removing components of the model or changing non-numerical values such as the dust attenuation type.
4.2 The model_components dictionary

All of the physical parameters the user desires to specify for a model_galaxy object are passed in the model_components dictionary. This page will take you through all of the available options. For a quick introduction take a look at the first iPython notebook example.

Parameters are equal to mandatory if they must be specified, or equal to their default values if not.

4.2.1 Global parameters

There are a few global parameters which can be inserted directly into model_components.

```
model_components = {}
model_components["redshift"] = mandatory  # Observed redshift
model_components["t_bc"] = 0.01        # Max age of birth clouds: Gyr
model_components["veldisp"] = 0.        # Velocity dispersion: km/s
model_components["sfh_comp"] = sfh_comp  # Dict containing SFH info
```

All other parameter values must first be placed within component dictionaries which are then inserted into model_components. Aside from observed redshift, the only other requirement is that at least one star-formation history component must be added to model_components for it to be valid.

4.2.2 Star-formation history components

Each SFH component is an individual parametric model for the SFH of the galaxy. Bagpipes can build up complex star-formation histories by superimposing multiple components. Components of the same type should be labelled sequentially in model_components e.g. burst1, burst2 etc.

All star-formation history components take the following keys:

```
sfh_comp = {}
sfh_comp["massformed"] = mandatory  # Log_10 total stellar mass formed: M_\rightarrow\text{Solar}
sfh_comp["metallicity"] = mandatory  # Metallicity: Z_{\odot} = 0.02
```

All SFH components also take one or more additional parameters describing their shape. All of the available options are listed below.

```
burst = {}  # Delta function burst
burst["age"] = mandatory  # Time since burst: Gyr
constant = {}  # tophat function
constant["age_max"] = mandatory  # Time since SF switched on: Gyr
constant["age_min"] = mandatory  # Time since SF switched off: Gyr
exponential = {}  # Tau model e^{-(t/\tau)}
exponential["age"] = mandatory  # Time since SFH began: Gyr
exponential["tau"] = mandatory  # Timescale of decrease: Gyr
delayed = {}  # Delayed Tau model t*e^{-(t/\tau)}
delayed["age"] = mandatory  # Time since SF began: Gyr
delayed["tau"] = mandatory  # Timescale of decrease: Gyr
```

(continues on next page)
If a custom SFH component is specified, the “history” key must contain either an array containing or a string giving the path to a file containing the star formation history. In both cases the format is a column of ages in years followed by a column of star formation rates in Solar masses per year.

### 4.2.3 Nebular component

The inclusion of the nebular component tells Bagpipes to include emission lines and nebular continuum emission in the model. These come from pre-computed Cloudy grids. The nebular emission model has only one free parameter, log_10 of the ionization parameter. The metallicity of the gas in the stellar birth clouds is assumed to be the same as the stars producing the ionizing flux.

```python
ebular = {}
ebular["logU"] = mandatory  # Log_10 of the ionization parameter.
```

### 4.2.4 Dust attenuation and emission component

The dust component governs attenuation and emission processes due to dust. Energy balance is assumed, such that all attenuated light is re-radiated.

Three dust attenuation models are implemented in Bagpipes, the Calzetti et al. (2000) model, the Cardelli et al. (1989) model, a model based on Charlot & Fall (2001) and the model of Salim et al. (2018). The dust emission models come from Draine & Li (2007).

```python
dust = {}
dust["type"] = mandatory  # Attenuation law: "Calzetti", "Cardelli", "CF00", "Salim"
dust["Av"] = mandatory  # Absolute attenuation in the V band: magnitudes
  # Multiplicative factor on Av for stars in birth clouds
  # Power-law slope of attenuation law ("CF00" only)
dust["n"] = 1.  # Deviation from Calzetti slope ("Salim" only)
dust["B"] = 0.  # 2175A bump strength ("Salim" only)

# Dust emission parameters
dust["qpah"] = 2.  # PAH mass fraction
  # Lower limit of starlight intensity distribution
  # Fraction of stars at umin
```

4.2. The model_components dictionary 13
4.3 Loading observational data: galaxy

This section will introduce you to loading observational data. This is stored in the galaxy object. Check out the second iPython notebook example for a quick-start guide to loading data.

4.3.1 API documentation: galaxy

class bagpipes.galaxy

\( ID, \) \( load\_data, \) \( spec\_units='ergscma', \) \( phot\_units='mujy', \) \( spec-\) 

\( spectrum\_exists=True, \) \( photometry\_exists=True, \) \( filt\_list=None, \) \( out\_units='ergscma', \) \( load\_indices=None, \) \( index\_list=None, \) \( index\_redshift=None, \) \( input\_spec\_cov\_matrix=False \)

A container for observational data loaded into Bagpipes.

Parameters

- **ID** *(str)* – string denoting the ID of the object to be loaded. This will be passed to load_data.
- **load_data** *(function)* – User-defined function which should take ID as an argument and return spectroscopic and/or photometric data. Spectroscopy should come first and be an array containing a column of wavelengths in Angstroms, then a column of fluxes and finally a column of flux errors. Photometry should come second and be an array containing a column of fluxes and a column of flux errors.
- **filt_list** *(list - optional)* – A list of paths to filter curve files, which should contain a column of wavelengths in angstroms followed by a column of transmitted fraction values. Only needed for photometric data.
- **spectrum_exists** *(bool - optional)* – If you do not have a spectrum for this object, set this to False. In this case, load_data should only return photometry.
- **photometry_exists** *(bool - optional)* – If you do not have photometry for this object, set this to False. In this case, load_data should only return a spectrum.
- **spec_units** *(str - optional)* – Units of the input spectrum, defaults to ergs s^-1 cm^-2 A^-1, “ergscma”. Other units (microjanskys; mujy) will be converted to ergscma by default within the class (see out_units).
- **phot_units** *(str - optional)* – Units of the input photometry, defaults to microjanskys, “mujy”. The photometry will be converted to ergscma by default within the class (see out_units).
- **out_units** *(str - optional)* – Units to convert the inputs to within the class. Defaults to ergs s^-1 cm^-2 A^-1, “ergscma”.
- **index_list** *(list - optional)* – list of dicts containing definitions for spectral indices.
- **load_indices** *(function or str - optional)* – Load spectral index information for the galaxy. This can either be a function which takes the galaxy ID and returns index values in the same order as they are defined in index_list, or the str “from_spectrum”, in which case the code will measure the indices from the observed spectrum for the galaxy.
- **index_redshift** *(float - optional)* – Observed redshift for this galaxy. This is only ever used if the user requests the code to calculate spectral indices from the observed spectrum.
4.3.2 The load_data function

The most important argument passed to galaxy is the load_data function, which you will need to write to access your data files and return your data. load_data should take an ID string and return observed spectroscopic and/or photometric data in the correct format and units (see below).

For example:

```python
import bagpipes as pipes
eg_filt_list = ["list", "of", "filters"]

def load_data(ID):
    # Do some stuff to load up data for the object with the correct ID number
    return spectrum, photometry

ID_number = "0001"
galaxy = pipes.galaxy(ID_number, load_data, filt_list=eg_filt_list)
galaxy.plot()
```

This will plot the data returned by the load_data function.

By default, Bagpipes expects spectroscopic and photometric data to be returned by load_data in that order. If you do not have both, you must pass a keyword argument to galaxy, either spectrum_exists=False or photometry_exists=False.

The format of the spectrum returned by load_data should be a 2D array with three columns: wavelengths in Angstroms, fluxes in erg/s/cm^2/A and flux errors in the same units (can be changed to microJanskys with the spec_units keyword argument). These will be stored in galaxy.spectrum.

The format of the photometry returned by load_data should be a 2D array with a column of fluxes in microJanskys and a column of flux errors in the same units (can be changed to erg/s/cm^2/A with the phot_units keyword argument). The fluxes should be in the same order as the filters in your filt_list. Bagpipes will calculate effective wavelengths for each filter and store these along with the input data in galaxy.photometry.

4.4 Fitting observational data: fit

This section describes fitting observational data using the fit class. Check out the third iPython notebook example for a quick-start guide and the fourth for some more advanced options.

4.4.1 API documentation: fit

```python
bagpipes.fit(galaxy, fit_instructions, run=';', time_calls=False, n_posterior=500)
```

Top-level class for fitting models to observational data. Interfaces with MultiNest to sample from the posterior distribution of a fitted_model object. Performs loading and saving of results.

Parameters

- **galaxy** (bagpipes.galaxy) – A galaxy object containing the photometric and/or spectroscopic data you wish to fit.
• **fit_instructions** *(dict)* – A dictionary containing instructions on the kind of model which should be fitted to the data.

• **run** *(string - optional)* – The subfolder into which outputs will be saved, useful e.g. for fitting more than one model configuration to the same data.

• **time_calls** *(bool - optional)* – Whether to print information on the average time taken for likelihood calls.

• **n_posterior** *(int - optional)* – How many equally weighted samples should be generated from the posterior once fitting is complete. Default is 500.

```python
fit (verbose=False, n_live=400, mpi_off=False)
```

Fit the specified model to the input galaxy data.

**Parameters**

• **verbose** *(bool - optional)* – Set to True to get progress updates from the sampler.

• **n_live** *(int - optional)* – Number of live points: reducing speeds up the code but may lead to unreliable results.

### 4.4.2 The fit_instructions dictionary

The two arguments passed to the `fit` class are a `galaxy` object (described in the *loading observational data* section) and the `fit_instructions` dictionary, which contains instructions on the model to be fitted to the data.

This is very similar to the `model_components` dictionary, however some additional options are available so that as well as being fixed, parameters can be fitted and prior probability density functions specified. A complete guide to the `fit_instructions` dictionary is provided [here](#).

### 4.4.3 Running the sampler

The MultiNest nested sampling algorithm can be run in order to sample from the posterior distribution using the `fit` method of the `fit` class. Nested sampling is similar to MCMC with a few key differences, for example no initial starting parameters are necessary.

### 4.4.4 Obtaining fitting results

The main output of the code is a set of samples from the posterior probability distribution for the model parameters. The code will also calculate samples for a series of derived quantities, e.g. the living stellar mass, ongoing star-formation rate etc. Samples are stored in the `fit.posterior.samples` dictionary. More information is available in the third ipython notebook example.

### 4.4.5 Saved outputs

The code saves basic output quantities needed to reconstruct the fit results without re-running the sampler as a hdf5 file under `pipes/posterior/<ID>.h5`. When the same fit is run again the results of the previous sampler run will be loaded by default, and you will not be able to re-fit the data. If you want to start over you’ll need to delete the saved file or change the run (see below).
4.4.6 Making output plots

Bagpipes can provide several standard plots. These are saved under the `pipes/plots/` folder.

These can be generated with:

```python
fit.plot_spectrum_posterior()  # Shows the input and fitted spectrum/
fit.plot_sfh_posterior()      # Shows the fitted star-formation history
fit.plot_1d_posterior()       # Shows 1d posterior probability distributions
fit.plot_corner()             # Shows 1d and 2d posterior probability
```

You may find some of the functions available under `pipes.plotting` helpful when generating your own custom plots.

4.4.7 The run keyword argument

Often we will want to fit a series of different models to data, changing star-formation histories, parameter limits, priors etc. In order to quickly switch between different fitting runs without deleting output posteriors we can specify the `run` keyword argument of `fit`. This will cause all outputs in `pipes/posterior/` and `pipes/plots/` to be saved into a further subdirectory with the name passed as `run`, e.g. `pipes/posterior/<run>/`.

4.5 The fit_instructions dictionary

The `fit_instructions` dictionary is similar to the `model_components` dictionary described in the `making model galaxies` section. Available options are the same, however as well as fixed values, the user can specify parameters to be fitted by defining a prior range and probability density function.

For example, a very simple model could be fitted with the following `fit_instructions` dictionary:

```python
burst = {}
burst["age"] = (0., 15.)       # Vary age from 0 to 15 Gyr
burst["metallicity"] = (0., 2.5) # Vary metallicity from 0 to 2.5 · Solar
burst["massformed"] = (0., 13.) # Vary log_10(mass formed) from 0 to 13

fit_instructions = {}
fit_instructions["burst"] = burst # Add the burst sfh component to
                                    # the fit
fit_instructions["redshift"] = (0., 10.) # Vary observed redshift from 0 to 10
```

Note that the code also automatically imposes a limit that no stars can form before the big bang, so the upper limit on the prior on `burst["age"]` will vary with observed redshift.

Combining this with the simple example from the `inputting observational data` section:

```python
import bagpipes as pipes

eg_filt_list = ["list", "of", "filters"]

def load_data(ID, filtlist):
    (continues on next page)```
# Do some stuff to load up data for the object with the correct ID number

```python
return spectrum, photometry
```

ID_number = "0001"

galaxy = pipes.galaxy(ID_number, load_data, filt_list=eg_filt_list)

```python
fit = pipes.fit(galaxy, fit_instructions)
```

There is no need to vary all of the parameters in `fit_instructions`. Parameters can still be fixed to single values just like in `model_components`. Additionally, parameters can be set to mirror other parameters which are fixed or fitted. For example:

```python
burst1 = {}  # A burst component
burst1["age"] = 0.1  # Fix age to 0.1 Gyr
burst1["metallicity"] = (0., 2.5)  # Vary metallicity from 0 to 2.5 Solar
burst1["massformed"] = (0., 13.)  # Vary log_10(mass formed) from 0 to 13
def burst2 = {}  # A second burst component
burst2["age"] = 1.0  # Fix the age to 1.0 Gyr
burst2["metallicity"] = "burst1:metallicity"  # Mirror burst1:metallicity
burst2["massformed"] = (0., 13.)  # Vary log_10(mass formed) from 0 to 13

```python
fit_instructions = {}
fit_instructions["burst1"] = burst1  # Add the burst1 sfh component to the fit
fit_instructions["burst2"] = burst2  # Add the burst2 sfh component to the fit
fit_instructions["redshift"] = (0., 10.)  # Vary observed redshift from 0 to 10
```

### 4.5.1 Adding priors

At the moment, all of the parameters in the above example are fitted with uniform priors by default. We can add further keys to the relevant dictionaries to specify different priors. For example, if we wanted the prior on stellar metallicity to be uniform in log_10 of the parameter:

```python
burst = {}
burst["age"] = (0., 13.)
burst["metallicity"] = (0.01, 5.)
burst["metallicity_prior"] = "log_10"
burst["massformed"] = (0., 13.)
```

The list of currently available priors is:

```python
component = {}
component["parameter_prior"] = "uniform"  # Uniform prior
component["parameter_prior"] = "log_10"  # Uniform in log_10(parameter)
component["parameter_prior"] = "log_e"  # Uniform in log_e(parameter)
```
component["parameter_prior"] = "pow_10"  # Uniform in 10**parameter
component["parameter_prior"] = "recip"  # Uniform in 1/parameter
component["parameter_prior"] = "recipsq" # Uniform in 1/parameter**2
component["parameter_prior"] = "Gaussian" # Gaussian, also requires:
component["parameter_prior_mu"] = 0.5  # Gaussian mean
component["parameter_prior_sigma"] = 0.1 # Gaussian standard dev.

The limits specified are still applied when a Gaussian prior is used, for example:

```python
fit_instructions["redshift"] = (0., 1.)
fit_instructions["redshift_prior"] = "Gaussian"
fit_instructions["redshift_prior_mu"] = 0.7
fit_instructions["redshift_prior_sigma"] = 0.2
```

will result in a Gaussian prior on redshift centred on 0.7 with standard deviation 0.2 but which is always constrained to be between 0 and 1.

4.6 Fitting multiple objects: fit_catalogue

This section describes fitting a catalogue of objects with the same model using the fit_catalogue class. Check out the sixth iPython notebook example for a quick-start guide.

4.6.1 API documentation: fit_catalogue

```python
class bagpipes.fit_catalogue(IDs, fit_instructions, load_data, spectrum_exists=True, photometry_exists=True, make_plots=False, cat_filt_list=None, vary_filt_list=False, redshifts=None, redshift_sigma=0.0, run='.', analysis_function=None, time_calls=False, n_posterior=500, full_catalogue=False)
```

Fit a model to a catalogue of galaxies.

**Parameters**

- **IDs (list)** – A list of ID numbers for galaxies in the catalogue
- **fit_instructions (dict)** – A dictionary containing the details of the model to be fitted to the data.
- **load_data (function)** – Function which takes ID as an argument and returns the model spectrum and photometry. Spectrum should come first and be an array with a column of wavelengths in Angstroms, a column of fluxes in erg/s/cm^2/A and a column of flux errors in the same units. Photometry should come second and be an array with a column of fluxes in microjanskys and a column of flux errors in the same units.
- **spectrum_exists (bool – optional)** – If the objects do not have spectroscopic data set this to False. In this case, load_data should only return photometry.
- **photometry_exists (bool – optional)** – If the objects do not have photometric data set this to False. In this case, load_data should only return a spectrum.
- **run (string – optional)** – The subfolder into which outputs will be saved, useful e.g. for fitting more than one model configuration to the same data.
• **make_plots** (*bool – optional*) – Whether to make output plots for each object.

• **cat_filt_list** (*list – optional*) – The filt_list, or list of filt_lists for the catalogue.

• **vary_filt_list** (*bool – optional*) – If True, changes the filter list for each object. When True, each entry in cat_filt_list is expected to be a different filt_list corresponding to each object in the catalogue.

• **redshifts** (*list – optional*) – List of values for the redshift for each object to be fixed to.

• **redshift_sigma** (*float – optional*) – If this is set, the redshift for each object will be assigned a Gaussian prior centred on the value in redshifts with this standard deviation. Hard limits will be placed at 3 sigma.

• **analysis_function** (*function – optional*) – Specify some function to be run on each completed fit, must take the fit object as its only argument.

• **time_calls** (*bool – optional*) – Whether to print information on the average time taken for likelihood calls.

• **n_posterior** (*int – optional*) – How many equally weighted samples should be generated from the posterior once fitting is complete for each object. Default 500.

• **full_catalogue** (*bool – optional*) – Adds minimum chi-squared values and rest-frame UVJ mags to the output catalogue, takes extra time, default False.

```
fit (verbose=False, n_live=400, mpi_serial=False)
```

Run through the catalogue fitting each object.

**Parameters**

• **verbose** (*bool – optional*) – Set to True to get progress updates from the sampler.

• **n_live** (*int – optional*) – Number of live points: reducing speeds up the code but may lead to unreliable results.

• **mpi_serial** (*bool – optional*) – When running through mpirun/mpiexec, the default behaviour is to fit one object at a time, using all available cores. When mpi_serial=True, each core will fit different objects.

```
fit_mpi_serial (verbose=False, n_live=400)
```

Run through the catalogue fitting multiple objects at once on different cores.

### 4.6.2 Saving of output catalogues

**fit_catalogue** will generate an output catalogue of posterior percentiles for all fit parameters plus some basic derived parameters. This is saved in the `pipes/cats` folder as `<run>.fits`.

### 4.6.3 Parallelisation

Bagpipes now supports parallelisation with MPI using the python package mpi4py. You can run both fit or fit_catalogue with MPI, just do `mpirun/mpiexec -n nproc python fit_with_bagpipes.py`. The default behaviour is to fit one object at a time using all available cores, this is useful for complicated models (e.g. fitting spectroscopy).

For catalogue fitting an alternative approach is also available, in which multiple objects are fitted at once, each using one core. This option can be activated using the `mpi_serial` keyword argument.
of fit_catalogue. This is better for fitting relatively simple models to large catalogues of photometry. This option currently requires a slightly modified version of pymultinest, which can be downloaded from this github repository. Please get in touch if you're having difficulty getting this to work.
Index

F
fit (class in bagpipes), 15
fit () (bagpipes.fit method), 16
fit () (bagpipes.fit_catalogue method), 20
fit_catalogue (class in bagpipes), 19
fit_mpi_serial () (bagpipes.fit_catalogue method), 20

G
galaxy (class in bagpipes), 14

M
model_galaxy (class in bagpipes), 9

U
update () (bagpipes.model_galaxy method), 10