prospector

Release 1.0

May 02, 2022

User Guide

Prospector is a package to conduct principled inference of stellar population properties from photometric and/or spectroscopic data using flexible models. Prospector allows you to:

- Infer high-dimensional stellar population properties, including nebular emission, from rest UV through Far-IR data (with nested or ensemble MCMC sampling.)
- Combine photometric and spectroscopic data rigorously using a flexible spectroscopic calibration model.
- Use spectra and/or photometry to constrain the linear combination of stellar population components that are present in a galaxy (e.g. non-parametric SFHs).
- Forward model many aspects of spectroscopic data analysis and calibration, including spectral resolution, spectrophotometric calibration, sky emission (coming soon), and wavelength solution, thus properly incorporating uncertainties in these components in the final parameter uncertainties.

CHAPTER 1

Requirements

Prospector works with Python3

You will also need:

- numpy and SciPy
- emcee and/or dynesty for inference (Please cite these packages in any publications)
- sedpy (for filter projections)
- HDF5 and h5py (If you have Enthought or Anaconda one or both of these may already be installed, or you can get HDF5 from homebrew or macports)

For modeling galaxies you will need:

- FSPS and python-FSPS (Please cite these packages in any publications)

You probably also need to have AstroPy for FITS file processing and cosmological calculations, please cite this package in any publications.

For parallel processing with emcee you will need:

- MPI (e.g. openMPI or mvapich2, available from homebrew, macports, or Anaconda) and mpi4py

CHAPTER 2

Installation

Prospector itself is is pure python. However, several other packages are required (see above.) To install Prospector and its dependencies to a conda environment, use the following procedure:

```
# change this if you want to install elsewhere;
# or, copy and run this script in the desired location
CODEDIR=$PWD

# Create and activate environment (named 'prospector')
cd $CODEDIR
git clone git@github.com:bd-j/prospector.git
cd prospector
conda env create -f environment.yml
conda activate prospector
cd ..

# Install FSPS from source
git clone git@github.com:cconroy20/fsps
export SPS_HOME="$PWD/fsps"
cd $SPS_HOME/src
make clean; make all

# Install other repos from source
repos=( dfm/python-fsps bd-j/sedpy )
for r in "${repos[@]}"; do
    git clone git@github.com:$r
    cd ${r##*/}
    python setup.py install
    cd ..
done

cd prospector
python setup.py install

echo "Add 'export SPS_HOME=$SPS_HOME' to your .bashrc"
```

To install just Prospector

```
cd <install_dir>
git clone https://github.com/bd-j/prospector
cd prospector
python setup.py install
```

Then in Python

```
import prospect
```

CHAPTER 3

User Interaction

The primary user interaction is through a **parameter file**, a python file in which several functions must be defined. These functions are described below and are used to build the ingredients for a fit (data, model, and noise model.) During execution any supplied command line options are parsed – includiing any user defined custom arguments – and the resulting set of arguments is passed to each of these functions before fitting begins.

Command line syntax calls the parameter file and is as follows for single thread execution:

```
python parameter_file.py --dynesty
```

Additional command line options can be given (see below) e.g.

```
python parameter_file.py --emcee --nwalkers=128
```

will cause a fit to be run using emcee with 128 walkers.

A description of the available command line options can be obtained with

```
python parameter_file.py --help
```

This syntax requires that the end of the parameter file have something like the following code block at the end.

```
if __name__ == "__main__":
    import time
    from prospect.fitting import fit_model
    from prospect.io import write_results as writer
            from prospect import prospect_args

    # Get the default argument parser
    parser = prospect_args.get_parser()
    # Add custom arguments that controll the build methods
    parser.add_argument("--custom_argument_1", ...)
    # Parse the supplied arguments, convert to a dictionary, and add this file for␣
↪logging purposes
    args = parser.parse_args()
    run_params = vars(args)
```

(continues on next page)

(continued from previous page)

```
    run_params["param_file"] = __file__

    # Set up an output file name, build fit ingredients, and run the fit
    ts = time.strftime("%y%b%d-%H.%M", time.localtime())
    hfile = "{0}_{1}_mcmc.h5".format(args.outfile, ts)
    obs, model, sps, noise = build_all(**run_params)
    output = fit_model(obs, model, sps, noise, **run_params)

    # Write results to output file
    writer.write_hdf5(hfile, run_params, model, obs,
                      output["sampling"][0], output["optimization"][0],
                      tsample=output["sampling"][1],
                      toptimize=output["optimization"][1],
                      sps=sps)
```

3.1 Command Line Options and Custom Arguments

A number of default command line options are included with prospector. These options can control the output filenames and format and some details of how the model is built and run. However, most of the default parameters control the fitting backends.

You can inspect the default set of arguments and their default values as follows:

```
from prospect import prospect_args
parser = prospect_args.get_parser()
parser.print_help()
```

In the typical **parameter file** the arguments are converted to a dictionary and passed as keyword arguments to all of the `build_*()` methods described below.

A user can add custom arguments that will further control the behavior of the model and data building methods. This is done by adding arguments to the parser in the executable part of the **parameter file**. See the argparse documentation for details on adding custom arguments.

3.2 Build methods

The required methods in a **parameter file** for building the data and model are:

1. `build_obs()`: This function will take the command line arguments dictionary as keyword arguments and returns on obs dictionary (see *Data Formats* .)
2. `build_model()`: This function will take the command line arguments dictionary dictionary as keyword arguments and return an instance of a `ProspectorParams` subclass, containing information about the parameters of the model (see *Models* .)
3. `build_sps()`: This function will take the command line arguments dictionary dictionary as keyword arguments and return an **sps** object, which must have the method `get_spectrum()` defined. This object generally includes all the spectral libraries and isochrones necessary to build a model, as well as much of the model building code and as such has a large memory footprint.
4. `build_noise()`: This function should return a `NoiseModel` object for the spectroscopy and/or photometry. Either or both can be ``None``(the default) in which case the likelihood will not include covariant noise or jitter and is equivalent to basic χ^2.

CHAPTER 4

Data Formats

4.1 The `build_obs()` function

The `build_obs()` function in the parameter file is written by the user. It should take a dictionary of command line arguments as keyword arguments. It should return an `obs` dictionary described below.

Other than that, the contents can be anything. Within this function you might open and read FITS files, ascii tables, HDF5 files, or query SQL databases. You could, using e.g. an `objid` parameter, dynamically load data (including filter sets) for different objects in a table. Feel free to import helper functions, modules, and packages (like astropy, h5py, sqlite, etc.)

The point of this function is that you don't *have* to externally convert your data format to be what Prospector expects and keep another version of files lying around: the conversion happens *within* the code itself. Again, the only requirement is that the function can take a `run_params` dictionary as keyword arguments and that it return an `obs` dictionary as described below.

4.2 The `obs` Dictionary & Data Units

Prospector expects the data in the form of a dictionary returned by py:meth:*build_obs*. This dictionary should have (at least) the following keys and values:

`"wavelength"` The wavelength vector for the spectrum, ndarray. Units are vacuum Angstroms. The model spectrum will be computed for each element of this vector. Set to `None` if you have no spectrum. If fitting observed frame photometry as well, then these should be observed frame wavelengths.

`"spectrum"` The flux vector for the spectrum, ndarray of same length as the wavelength vector. If absolute spectrophotometry is available, the units of this spectrum should be Janskies divided by 3631 (i.e. maggies). Also the `rescale_spectrum` run parameter should be False.

`"unc"` The uncertainty vector (sigma), in same units as `"spectrum"`, ndarray of same length as the wavelength vector.

`"mask"` A boolean array of same length as the wavelength vector, where `False` elements are ignored in the likelihood calculation.

"filters" A sequence of sedpy filter objects or filter names, used to calculate model magnitudes.

"maggies" An array of photometric flux densities, same length as `"filters"`. The units are *maggies*. Maggies are a linear flux density unit defined as $maggie = 10^{-0.4 m_{AB}}$ where m_{AB} is the AB apparent magnitude. That is, 1 maggie is the flux density in Janskys divided by 3631. Set to `None` if you have no photometric data.

"maggies_unc" An array of photometric flux uncertainties, same length as `"filters"`, that gives the photometric uncertainties in units of *maggies*

"phot_mask" Like `"mask"`, a boolean array, used to mask the photometric data during the likelihood calculation. Elements with `False` values are ignored in the likelihood calculation.

If you do not have spectral or photometric data, set `"wavelength": None` or `"maggies": None` respectively. Feel free to add keys that store other metadata, these will be stored on output. However, for ease of storage these keys should either be numpy arrays or basic python datatypes that are JSON serializable (e.g. strings, ints, and floats and lists, dicts, and tuples thereof.)

CHAPTER 5

Models

The modular nature of Prospector allows it to be applied to a variety of data types and different scientific questions. However, this flexibility requires the user to take care in defining the model, including prior distributions, which may be specific to a particular scientific question or to the data to be fit. Different models and prior beliefs may be more or less appropriate for different kinds of galaxies. Certain kinds of data may require particular model components in order to be described well. Prospector strives to make this model building process as straightforward as possible, but it is a step that cannot be skipped.

The choice of which parameters to include, which to let vary, and what prior distributions to use will depend on the data being fit (including the types of objects) and the goal of the inference. As examples, if no infrared data is available then it is not necessary to fit – or even include in the model at all – the dust emission parameters controlling the shape of the infrared SED. For globular clusters or completely quenched galaxies it may not be necessary to include nebular emission in the model, and one may wish to adjust the priors on population age or star formation history to be more appropriate for such objects. If spectroscopic data is being fit then it may be necessary to include velocity dispersion as a free parameter. Generating and fitting mock data can be an extremely useful tool for exploring the sensitivity of a given type of data to various parameters.

5.1 Parameter Specification

All model parameters require a specification in the **parameter file**. A dictionary of parameter specifications, keyed by parameter name, is used to instantiate and configure the model objects (instances of ProspectorParams or its subclasses.) For a single parameter the specification is a dictionary that must at minimum include several keys:

`"name"` The name of the parameter, string.

`"N"` An integer specifying the length of the parameter. For the common case of a scalar parameter, use `1`.

`"init"` The initial value of the parameter. If the parameter is free to vary, this is where optimization or will start from, or, if no optimization happens, this will be the center of the initial ball of *emcee* walkers. If using nested sampling then the value of `"init"` is not important (though a value must still be given.) If the parameter is not free, then this is the value that will be used throughout optimization and sampling.

`"isfree"` Boolean specifying whether a parameter is free to vary during optimization and sampling (`True`) or not (`False`).

For parameters with `"isfree": True` the following additional key is required:

`"prior"` An instance of a prior object, including parameters for the prior (e.g. `priors.TopHat(mini=10, maxi=12)`).

If using `emcee`, the following key can be useful to have:

`"init_disp"` The dispersion in this parameter to use when generating an `emcee` sampler ball. This is not technically required, as it defaults to 10% of the initial value. It is ignored if nested sampling is used.

It's also a good idea to have a `"units"` key, a string describing the units of the the parameter. So, in the end, this looks something like

```
mass = {"name": "mass",
                "N": 1,
                "init": 1e9,
                "init_disp": 1e8, # only important if using emcee sampling
                "units": "M$_\odot$ of stars formed.",
                "isfree": True,
                "prior": priors.LogUniform(mini=1e7, maxi=1e12)
                }
```

Nearly all parameters used by FSPS can become a model parameter. When fitting galaxies the default python-FSPS parameter values will be used unless specified in a fixed parameter, e.g. `imf_type` can be changed by including it as a fixed parameter with value given by `"init"`.

Parameters can also be used to control the Prospector-specific parts of the modeling code. These include things like spectral smoothing, wavelength calibration, spectrophotometric calibration, and any parameters of the noise model. Be warned though, if you include a parameter that does not affect the model the code will not complain, and if that parameter is free it will simply result in a posterior PDF that is the same as the prior (though optimization algorithms may fail).

5.2 Priors

Prior objects can be found in the *`prospect.models.priors`* module. It is recommended to use the objects instead of the functions, as they have some useful attributes and are suitable for all types of sampling. The prior functions by contrast will not work for nested sampling. When specifying a prior using an object, you can and should specify the parameters of that prior on initialization, e.g.

```
mass["prior"] = priors.ClippedNormal(mean=0.0, sigma=1.0, mini=0.0, maxi=3.0)
```

5.3 Transformations

Sometimes the native parameterization of stellar population models is not the most useful. In these cases parameter *transformations* can prove useful.

Transformations are useful to impose parameter limits that are a function of other parameters; for example, when fitting for redshift it can be useful to reparameterize the age of a population (say, in Gyr) into its age as a fraction of the age of the universe at that redshift. This avoids the problem of populations that are older than the age of the universe, or complicated joint priors on the population age and the redshift. A number of useful transformation functions are provided in Prospector and these may be easily supplemented with user defined functions.

This parameter transformation and dependency mechanism can be used to tie any number of parameters to any number of other parameters in the model, as long as the latter parameters are not *also* dependent on some parameter transformation. This mechanism may also be used to avoid joint priors. For example, if one wishes to place a prior on the ratio of two parameters (say, that it be less than one) then the ratio itself can be introduced as a new parameter, and one of the original parameters can be "fixed" but have its value at each parameter location depend on the other original parameter and the new ratio parameter.

As a simple example, we consider sampling in the log of the SF timescale instead of the timescale itself. The follwing code

```
def delogify(logtau=0, **extras):
    return 10**logtau

model_params["tau"]["isfree"] = False
model_params["tau"]["depends_on"] = delogify
model_params["logtau"] = dict(N=1, init=0, isfree=True, prior=priors.TopHat(mini=-1,␣
↪maxi=1))
```

could be used to set the value of `tau` using the free parameter `logtau` (i.e., sample in the log of a parameter, though setting a `prospect.models.priors.LogUnifrom` prior is equivalent in terms of the posterior).

This dependency function must take optional extra keywords (`**extras`) because the entire parameter dictionary will be passed to it. Then add the new parameter specification to the `model_params` dictionary for the parameter that can vary (and upon which the fixed parameter depends). In this example that new free parameter would be `logtau`.

This pattern can also be used to tie arbitrary parameters together (e.g. gas-phase and stellar metallicity) while still allowing them to vary. A parameter may depend on multiple other (free or fixed) parameters, and multiple parameters may depend on a single other (free or fixed) parameter. This mechanism is used extensively for the non-parametric SFHs, and is recommended for complex dust attenuation models.

It is important that any parameter with the `"depends_on"` key present is a fixed parameter. For portability and easy reconstruction of the model it is important that the `depends_on` function either be importable (e.g. one of the functions supplied in *`prospect.models.transforms`*) or defined within the parameter file.

5.4 Parameter Set Templates

A number of predefined sets of parameters (with priors) are available as dictionaries of model specifications from *prospect.models.templates.TemplateLibrary*, these can be a good starting place for building your model. To see the available parameter sets to inspect the free and fixed parameters in a given set, you can do something like

```
from prospect.models.templates import TemplateLibrary
# Show all pre-defined parameter sets
TemplateLibrary.show_contents()
# Show details on the "parameteric" set of parameters
TemplateLibrary.describe("parametric_sfh")
# Simply print all parameter specifications in "parametric_sfh"
print(TemplateLibrary["parametric_sfh"])
# Actually get a copy of one of the predefined sets
model_params = TemplateLibrary["parametric_sfh"]
# Instantiate a model object
from prospect.models import SedModel
model = SedModel(model_params)
```

5.5 The `build_model()` Method

This method in the **parameter file** should take the `run_params` dictionary as keyword arguments, and return an instance of a subclass of *`prospect.models.ProspectorParams`*.

The model object, a subclass of *prospect.models.ProspectorParams*, is initialized with a list or dictionary (keyed by parameter name) of each of the model parameter specifications described above. If using a list, the order of the list sets the order of the free parameters in the parameter vector. The free parameters will be varied by the code during the optimization and sampling phases. The initial value from which optimization is begun is set by the `"init"` values of each parameter. For fixed parameters the `"init"` value gives the value of that parameter to use throughout the optimization and sampling phases (unless the `"depends_on"` key is present, see advanced.)

The `run_params` dictionary of arguments (including command line modifications) can be used to change how the model parameters are specified within this method before the *prospect.models.ProspectorParams* model object is instantiated. For example, the value of a fixed parameter like `zred` can be set based on values in `run_params` or additional parameters related to dust or nebular emission can be optionally added based on switches in `run_params`.

Useful model objects include *prospect.models.SpecModel* and `prospect.models.PolySpecModel`. The latter includes tools for optimization of spectrophotometric calibration.

CHAPTER 6

Output format

By default the output of the code is an HDF5 file, with filename `<output>_<timestamp>_mcmc.h5`

Optionally several pickle files (pickle is Python's internal object serialization module), roughly equivalent to IDL SAVE files, can be output. These may be convenient, but are not very portable.

6.1 HDF5 output

The output HDF5 file contains `datasets` for the input observational data and the MCMC sampling chains. A significant amount of metadata is stored as JSON in dataset attributes. Anything that could not be JSON serialized during writing will have been pickled instead, with the pickle stored as string data in place of the JSON.

The HDF5 files can read back into python using

```
import prospect.io.read_results as reader
filename = "<outfilestring>_<timestamp>_mcmc.h5"
results, obs, model = reader.results_from(filename)
```

which gives a `results` dictionary, the `obs` dictionary containing the data to which the model was fit, and the `model` object used in the fitting. The `results` dictionary contains the production MCMC chains from *emcee* or the chains and weights from *dynesty*, basic descriptions of the model parameters, and the `run_params` dictionary. Some additional ancillary information is stored, such as code versions, runtimes, MCMC acceptance fractions, and model parameter positions at various phases of of the code. There is also a string version of the **parameter file** used. The results dictionary contains the information needed to regenerate the *sps* object used in generating SEDs.

```
sps = reader.get_sps(res)
```

It can sometimes be difficult to reconstitute the model object if it is complicated, for example if it was built by using or referencing files or data that are no longer available. For this reason it is suggested that references to filenames in parameter files be made command-line arguments

6.2 Pickles

It is possible to output prospector results as python "pickle" files. The results pickle is a serialization of the results dictionary, and has `<timestamp>_mcmc` appended onto the output file string specified when the code was run, where `timestamp` is in UT seconds. It uses only basic scientific python types (e.g. dictionaries, lists, and numpy arrays). It should therefore be readable on any system with Python and Numpy installed. This can be accomplished with

```
import pickle
filename = "<outfilestring>_<timestamp>_mcmc"
with open(filename, "rb") as f:
    result = pickle.load(f)
print(result.keys())
```

A second pickle file containing the model object has the extension `<timestamp>_model`. It is a direct serialization of the model object used during fitting, and is thus useful for regenerating posterior samples of the SED, or otherwise exploring properties of the model.

However, this requires Python and a working Prospector installation of a version compatible with the one used to generate the model pickle. If that is possible, then the following code will read the model pickle:

```
import pickle
model_file = "<outfilestring>_<timestamp>_model"
with open(model_file, 'rb') as mf:
    mod = pickle.load(mf)
print(type(mod))
```

If Powell optimization was performed, this pickle also contains the optimization results (as a list of Scipy Optimizer-Result objects).

6.3 Basic diagnostic plots

For detailed plotting, see the `prospect.plotting` module. Several methods for basic visualization of the results are also included in the *`prospect.io.read_results`* module.

First, the results file can be read into useful dictionaries and objects using :py:meth:prospect.io.read_results.results_from``

```
import prospect.io.read_results as reader
filename = "<outfilestring>_<timestamp>_mcmc"
results, obs, model = reader.results_from(filename)
```

It is often desirable to plot the parameter traces for the MCMC chains. That is, one wants to see the evolution of the parameter values as a function of MCMC iteration. This can be useful to check for convergence. It can be done easily for both *emcee* and *dynesty* results by

```
tracefig = reader.traceplot(results)
```

Another useful thing is to look at the "corner plot" of the parmeters. If one has the corner.py package, then

```
cornerfig = reader.subcorner(results, showpars=model.theta_labels()[:5])
```

will return a corner plot of the first 5 free parameters of the model. If `showpars` is omitted then all free parameters will be plotted. There are numerous other options to the *`prospect.io.read_results.subcorner()`* method, which is a thin wrapper on *corner.py*.

Finally, one often wants to look at posterior samples in the space of the data, or perhaps the maximum a posteriori parameter values. Taking the MAP as an example, this would be accomplished by

```
        import np

# Find the index of the maximum a posteriori sample
ind_max = results["lnprobability"].argmax()
if res["chain"].ndim > 2:
    # emcee
    walker, iteration = np.unravel_index(ind_max, results["lnprobability"].shape)
            theta_max = results["chain"][walker, iteration, :]
elif res["chain"].ndim == 2:
    # dynesty
    theta_max = results["chain"][indmax, :]

# We need the SPS object to generate a model
sps = reader.get_sps(results)
# now generate the SED for the max. a post. parameters
spec, phot, x = model.predict(theta_max, obs=obs, sps=sps)

# Plot the data and the MAP model on top of each other
import matplotlib.pyplot as pl
if obs['wave'] is None:
            wave = sps.wavelengths
else:
    wave = obs['wavelength']
pl.plot(wave, obs['spectrum'], label="Spec Data")
pl.plot(wave, spec, label="MAP model spectrum")
if obs['filters'] is not None:
    pwave = [f.wave_effective for f in obs["filters"]]
    pl.plot(pwave, obs['maggies'], label="Phot Data")
    pl.plot(pwave, phot, label="MAP model photometry")
```

However, if all you want is the MAP model this may be stored for you, without the need to regenerate the `sps` object

```
import matplotlib.pyplot as pl

        best = res["bestfit"]
a = model.params["zred"] + 1
pl.plot(a * best["restframe_wavelengths"], best['spectrum'], label="MAP spectrum")
if obs['filters'] is not None:
    pwave = [f.wave_effective for f in obs["filters"]]
    pl.plot(pwave, best['photometry'], label="MAP photometry")
```

CHAPTER 7

Demonstrations

You can check out the Jupyter notebook demo at

- InteractiveDemo

7.1 Interactive Figure

Also, below is an example of inference from an increasing number of photometric bands. Model parameters and SEDs are inferred (in blue) from a changing number of mock photometric bands (grey points). The mock is generated at the parameters and with the SED marked in black. This shows how with a small amount of data most posteriors are determined by the prior (dotted green) but that as the number of bands increases, the data are more infomative and the posterior distributions are narrower than the prior.

Click the buttons show the inference for different filter sets:

CHAPTER 8

Tutorial

Here is a guide to getting up and running with Prospector.

We assume you have installed Prospector and all its dependencies as laid out in the docs. The next thing you need to do is make a temporary work directory, `<workdir>`

```
cd <workdir>
cp <codedir>/demo/demo_* .
```

We now have a *parameter file* or two, and some data. Take a look at the `demo_photometry.dat` file in an editor, you'll see it is a simple ascii file, with a few rows and several columns. Each row is a different galaxy, each column is a different piece of information about that galaxy.

This is just an example. In practice Prospector can work with a wide variety of data types.

8.1 The parameter file

Open up `demo_params.py` in an editor, preferably one with syntax highlighting. You'll see that it's a python file. It includes some imports, a number of methods that build the ingredients for the fitting, and then an executable portion.

Executable Script

The executable portion of the parameter file that comes after the `if __name__ == "__main__"` line is run when the parameter file is called. Here the possible command line arguments and their default values are defined, including any custom arguments that you might add. In this example we have added several command line arguments that control how the data is read and how the The supplied command line arguments are then parsed and placed in a dictionary. This dictionary is passed to all the ingredient building methods (described below), which return the data dictionary and necessary model objects. The data dictionary and model objects are passed to a function that runs the prospector fit (`prospect.fitting.fit_model()`). Finally, the fit results are written to an output file.

Building the fit ingredients: build_model

Several methods must be defined in the parameter file to build the ingredients for the fit. The purpose of these functions and their required output are described here. You will want to modify some of these for your specific model and data. Note that each of these functions will be passed a dictionary of command line arguments. These command line arguments, including any you add to the command line parser in the executable portion of the script, can therefore be used to control the behaviour of the ingredient building functions. For example, a custom command line argument can be used to control the type of model that is fit, or how or from where the data is loaded.

First, the `build_model()` function is where the model that we will fit will be constructed. The specific model that you choose to construct depends on your data and your scientific question.

We have to specify a dictionary or list of model parameter specifications (see *Models*). Each specification is a dictionary that describes a single parameter. We can build the model by adjusting predefined sets of model parameter specifications, stored in the `models.templates.TemplateLibrary` dictionary-like object. In this example we choose the `"parametric_sfh"` set, which has the parameters necessary for a vasic delay-tau SFH fit with simple attenuation by a dust screen. This parameter set can be inspected in any of the following ways

```
from prospect.models.templates import TemplateLibrary, describe
# Show basic description of all pre-defined parameter sets
TemplateLibrary.show_contents()
# method 1: print the whole dictionary of dictionaries
model_params = TemplateLibrary["parametric_sfh"]
print(model_params)
# Method 2: show a prettier summary of the free and fixed parameters
print(describe(model_params))
```

You'll see that this model has 5 free parameters. Any parameter with `"isfree": True` in its specification will be varied during the fit. We have set priors on these parameters, visible as e.g. `model_params["mass"]["prior"]`. You may wish to change the default priors for your particular science case, using the prior objects in the `models.priors` module. An example of adjusting the priors for several parameters is given in the `build_model()` method in `demo_params.py`. Any free parameter *must* have an associated prior. Other parameters have their value set to the value of the `"init"` key, but do not vary during the fit. They can be made to vary by setting `"isfree": True` and specifying a prior. Parameters not listed here will be set to their default values. Typically this means default values in the `fsps.StellarPopulation` object; see python-fsps for details. Once you get a set of parameters from the `TemplateLibrary` you can modify or add parameter specifications. Since `model_params` is a dictionary (of dictionaries), you can update it with other parameter set dictionaries from the `TemplateLibrary`.

Finally, the `build_model()` function takes the `model_params` dictionary or list that you build and uses it to instantiate a `SedModel` object.

```
from prospect.models import SedModel
model_params = TemplateLibrary["parametric_sfh"]
# Turn on nebular emission and add associated parameters
model_params.update(TemplateLibrary["nebular"])
model_params["gas_logu"]["isfree"] = True
model = SedModel(model_params)
print(model)
```

If you wanted to change the specification of the model using custom command line arguments, you could do it in `build_model()` by allowing this function to take keyword arguments with the same name as the custom command line argument. This can be useful for example to set the initial value of the redshift `"zred"` on an object-by-object basis. Such an example is shown in `demo_params.py`, which also uses command line arguments to control whether nebular and/or dust emission parameters are added to the model.

Building the fit ingredients: build_obs

The next thing to look at is the `build_obs()` function. This is where you take the data from whatever format you have and put it into the dictionary format required by Prospector for a single object. This means you will have to modify this function heavily for your own use. But it also means you can use your existing data formats.

Right now, the `build_obs()` function just reads ascii data from a file, picks out a row (corresponding to the photometry of a single galaxy), and then makes a dictionary using data in that row. You'll note that both the datafile name and the object number are keyword arguments to this function. That means they can be set at execution time on the command line, by also including those variables in the `run_params` dictionary. We'll see an example later.

When you write your own `build_obs()` function, you can add all sorts of keyword arguments that control its output (for example, an object name or ID number that can be used to choose or find a single object in your data file). You can also import helper functions and modules. These can be either things like astropy, h5py, and sqlite or your own project specific modules and functions. As long as the output dictionary is in the right format (see dataformat.rst), the body of this function can do anything.

Building the fit ingredients: the rest

Ok, now we go to the `build_sps()` function. This one is pretty straightforward, it simply instantiates our `sources.CSPSpecBasis` object. For nonparameteric fits one would use the `sources.FastStepBasis` object. These objects hold all the spectral libraries and produce an SED given a set of parameters. After that is `build_noise()`, which is for complexifying the noise model – ignore that for now.

8.2 Running a fit

There are two kinds of fitting packages that can be used with Prospector. The first is `emcee` which implements ensemble MCMC sampling, and the second is `dynesty`, which implements dynamic nested sampling. It is also possible to perform optimization. If `emcee` is used, the result of the optimization will be used to initalize the ensemble of walkers.

The choice of which fitting algorithms to use is based on command line flags (`--optimization`, `--emcee`, and `--dynesty`.) If no flags are set the model and data objects will be generated and stored in the output file, but no fitting will take place. To run the fit on object number 0 using `emcee` after an initial optimization, we would do the following at the command line

```
python demo_params.py --objid=0 --emcee --optimize \
--outfile=demo_obj0_emcee
```

If we wanted to change something about the MCMC parameters, or fit a different object, we could also do that at the command line

```
python demo_params.py --objid=1 --emcee --optimize \
--outfile=demo_obj1_emcee --nwalkers=32 --niter=1024
```

And if we want to use nested sampling with `dynesty` we would do the following

```
python demo_params.py --objid=0  --dynesty \
--outfile=demo_obj0_dynesty
```

Finally, it is sometimes useful to run the script from the interpreter to do some checks. This is best done with the IPython enhanced interactive python.

```
ipython
In [1]: %run demo_params.py --objid=0 --debug=True
```

You can then inspect the `obsdat` dictionary, the `model` object, and the `run_params` dictionary to make sure everything is working fine.

To see the full list of available command-line options, you can run the following

```
python demo_params.py --help
```

8.3 Working with the output

After the fit is completed we should have a file with a name like demo_obj0_<fitter>_<timestamp>_mcmc.h5. This is an HDF5 file containing sampling results and various configuration data, as well as the observational data that was fit. By setting run_params["output_pickles"]=True you can also output versions of this information in the less portable pickle format. We will read the HDF5 with python and make some plots using utilities in Prospector

To read the data back in from the output files that we've generated, use methods in prospect.io.read_results.

```
import prospect.io.read_results as reader
res, obs, model = reader.results_from("demo_obj_<fitter>_<timestamp>_mcmc.h5")
```

The res object is a dictionary containing various useful results. You can look at res.keys() to see a list of what it contains. The obs object is just the obs dictionary that was used in the fitting. The model object is the model object that was used in the fitting.

** Diagnostic plots**

There are also some methods in this module for basic diagnostic plots. The subcorner method requires that you have the corner package installed. It's possible now to examine the traces (i.e. the evolution of parameter value with MC iteration) and the posterior PDFs for the parameters.

```
# Trace plots
tfig = reader.traceplot(res)
# Corner figure of posterior PDFs
cfig = reader.subcorner(res)
```

Working with samples

If you want to get the *maximum a posteriori* sample, or percentiles of the posterior pdf, that can be done as follows (note that for dynesty the weights of each posterior sample must be taken into account when calculating quantiles) :

```
# Maximum posterior probability sample
imax = np.argmax(res['lnprobability'])
csz = res["chain"].shape
if res["chain"].ndim > 2:
    # emcee
    i, j = np.unravel_index(imax, res['lnprobability'].shape)
    theta_max = res['chain'][i, j, :].copy()
    flatchain = res["chain"].reshape(csz[0] * csz[1], csz[2])
        else:
    # dynesty
    theta_max = res['chain'][imax, :].copy()
    flatchain = res["chain"]

# 16th, 50th, and 84th percentiles of the posterior
from prospect.plotting.corner import quantile
weights = res.get("weights", None)
post_pcts = quantile(flatchain.T, q=[16, 50, 84], weights=weights)
```

Stored "best-fit" model

Further, the prediction of the data for the MAP posterior sample may be stored for you.

```
# Plot the stored maximum ln-probability sample
import matplotlib.pyplot as pl
```

(continues on next page)

(continued from previous page)

```
best = res["bestfit"]
a = model.params["zred"] + 1
pl.plot(a * best["restframe_wavelengths"], best['spectrum'], label="MAP spectrum")
if obs['filters'] is not None:
    pwave = [f.wave_effective for f in obs["filters"]]
    pl.plot(pwave, best['photometry'], label="MAP photometry")
    pl.set_title(best["parameter"])
```

This stored best-fit information is only available if the *sps* object was passed to the `write_hdf5()` after the fit is run. If it isn't available, you can regnerate the model predictions for the highest probability sample using the approach below.

Regenerating Model predictions

If necessary, one can regenerate models at any position in the posterior chain. This requires that we have the sps object used in the fitting to generate models, which we can regenerate using the `read_results.get_sps()` method.

```
# We need the correct sps object to generate models
sps = reader.get_sps(res)
```

Now we will choose a specific parameter value from the chain and plot what the observations and the model look like, as well as the uncertainty normalized residual. For `emcee` results we will use the last iteration of the first walker, while for `dynesty` results we will just use the last sample in the chain.

```
# Choose the walker and iteration number by hand.
walker, iteration = 0, -1
if res["chain"].ndim > 2:
    # if you used emcee for the inference
    theta = res['chain'][walker, iteration, :]
else:
    # if you used dynesty
    theta = res['chain'][iteration, :]

# Or get a fair sample from the posterior
from prospect.plotting.utils import sample_posterior
theta = sample_posterior(res["chain"], weights=res.get("weights", None), nsample=1)[0,
↪:]

# Get the modeled spectra and photometry.
# These have the same shape as the obs['spectrum'] and obs['maggies'] arrays.
spec, phot, mfrac = model.predict(theta, obs=res['obs'], sps=sps)
# mfrac is the ratio of the surviving stellar mass to the formed mass (the ``"mass"``␣
↪parameter).

# Plot the model SED
import matplotlib.pyplot as pl
wave = [f.wave_effective for f in res['obs']['filters']]
sedfig, sedax = pl.subplots()
sedax.plot(wave, res['obs']['maggies'], '-o', label='Observations')
sedax.plot(wave, phot, '-o', label='Model at {},{}'.format(walker, iteration))
sedax.set_ylabel("Maggies")
sedax.set_xlabel("wavelength")
sedax.set_xscale('log')

# Plot residuals for this walker and iteration
chifig, chiax = pl.subplots()
chi = (res['obs']['maggies'] - phot) / res['obs']['maggies_unc']
```

(continues on next page)

(continued from previous page)

```
chiax.plot(wave, chi, 'o')
chiax.set_ylabel("Chi")
chiax.set_xlabel("wavelength")
chiax.set_xscale('log')
```

CHAPTER 9

Frequently Asked Questions

9.1 How do I add filter transmission curves?

Many projects use particular filter systems that are not general or widely used. It is easy to add a set of custom filter curves. Filter projections are handled by the sedpy code. See the FAQ there <https:github.com/bd-j/sedpy/blob/main/docs/faq.rst> for detailed instructions on adding filter cirves.

9.2 What units?

Prospector natively uses *maggies* for both spectra and photometry, and it is easiest if you supply data in these units. maggies are a flux density (f_ν) unit defined as Janskys/3631. Wavelengths are *vacuum* Angstroms by default. By default, masses are in solar masses of stars *formed*, i.e. the integral of the SFH. Note that this is different than surviving solar masses (due to stellar mass loss).

9.3 How long will it take to fit my data?

That depends. Here are some typical timings for each likelihood call in various situations (macbook pro)

- 10 band photometry, no nebular emission, no dust emission: 0.004s
- photometry, including nebular emission: 0.04s
- spectroscopy including FFT smoothing: 0.05s

Note that the initial likelihood calls will be (much) longer than this. Under the hood, the first time it is called python-FSPS computes and caches many quantities that are reused in subsequent calls.

Many likelihood evaluations may be required for the sampling to converge. This depends on the kind of model you are fitting and its dimensionality, the sampling algorithm being used, and the kind of data that you have. Hours or even days per fit is not uncommon for more complex models.

9.4 Can I fit my spectrum too?

There are several extra considerations that come up when fitting spectroscopy

1) Wavelength range and resolution. Prospector is based on FSPS, which uses the MILES spectral library. These have a resolution of ~2.5A FWHM from 3750AA - 7200AA restframe, and much lower (R~200 or so, but not actually well defined) outside this range. Higher resolution data (after including both velocity dispersion and instrumental resolution) or spectral points outside this range cannot yet be fit.

2) Relatedly, line spread function. Prospector includes methods for FFT based smoothing of the spectra, assuming a gaussian LSF (in either wavelength or velocity space). There is also the possibility of FFT based smoothing for wavelength dependent gaussian dispersion (i.e. sigma_lambda = f(lambda) with f possibly a polynomial of lambda). In practice the smoothed spectra will be a combination of the library resolution plus whatever FFT smoothing is applied. Hopefully this can be made to match your actual data resolution, which is a combination of the physical velocity dispersion and the instrumental resolution. The smoothing is controlled by the parameters *sigma_smooth*, *smooth_type*, and *fftsmooth*

3) Nebular emission. While prospector/FSPS include self-consistent nebular emission, the treatment is probably not flexible enough at the moment to fit high S/N, high resolution data including nebular emission (e.g. due to deviations of line ratios from Cloudy predictions or to complicated gas kinematics that are different than stellar kinematics). Thus fitting nebular lines should take adavantage of the nebular line amplitude optimization/marginalization capabilities. For very low resolution data this is less of an issue.

4) Spectrophotometric calibration. There are various options for dealing with the spectral continuum shape depending on how well you think your spectra are calibrated and if you also fit photometry to tie down the continuum shape. You can optimize out a polynomial "calibration" vector, or simultaneously fit for a polynomial and marginalize over the polynomial coefficients (this allows you to place priors on the accuracy of the spectrophotometric calibration). Or you can just take the spectrum as perfectly calibrated.

9.5 How do I fit for redshift as well as other parameters?

The simplest way is to just let the parameter specification for `"zred"` indicate that it is free and specify a prior for it (see *Models*). If you don't include the `"lumdist"` parameter at all in the `model_params` dictionary, then the luminosity distance will be computed directly from the redshift assuming a WMAP9 cosmology.

The other thing to keep in mind when the redshift is free is that the SFH might be constrained by the age of the universe at that redshift (e.g. `"tage"` should always be less than `t_univ(zred)`). If this is a concern, you can use parameter transformations (again, see *Models*). Specifically for parametric SFHs one would make the `"tage"` parameter fixed but *depend on* `prospect.models.transforms.tage_from_tuniv()` and add a new `"tage_tuniv"` parameter that corresponds to `"tage"` as a fraction of the age of the universe (with values from 0-1).

9.6 What SFH parameters should I use?

That depends on the scientific question you are trying to answer, and to some extent on the data that you have.

9.7 How do I use the non-parametric SFHs?

Prospector is packaged with four families of non-parametric star formation histories. The simplest model fits for the (log or linear) mass formed in fixed time bins. This is the most flexible model but typically results in unphysical priors on age and specific star formation rates. Another model is the Dirichlet parameterization introduced in leja17, in which the fractional specific star formation rate for fixed each time bin follows a Dirichlet distribution. This model is moderately flexible and produces reasonable age and sSFR priors, but it still allows unphysical SFHs with sharp quenching and rejuvenation events. A third model, the continuity prior, trades some flexibility for accuracy by explicitly weighting against these (typically considered unphysical) sharply quenching and rejuvenating SFHs. This is done by placing a prior on the ratio of SFRs in adjacent time bins to ensure a smooth evolution of SFR(t). Finally, the flexible continuity prior retains this smoothness weighting but instead of fitting for the mass forms in fixed bins, fits for the size of time bins in which a fixed amount of mass is formed. This prior removes the discretization effect of the time bins in exchange for imposing a minimum mass resolution in the recovered SFH parameters. The performance of these different nonparametric models is compared and contrasted in detail in leja19.

In order to use these models, select the appropriate *parameter set template* to use in the `model_params` dictionary. You will also need to make sure to use the appropriate *source* object, `prospect.sources.FastStepBasis`.

The parameter templates are set up to transform from the sampling parameters (e.g. `logsfr_ratios`) to the fundamental parameters of the non-parametric SFH, the temporal bins and vector of masses formed in each bin. To change the bin widths or number of bins, several related aspects of the parameter set including the length of several parameters and the priors must be changed simultaneously. See :py:meth‘prospect.models.templates.adjust_continuity_agebins‘ for an example.

9.8 What bins should I use for the non-parametric SFH?

Deciding on the "optimal" number of bins to use in such non-parametric SFHs is a difficult question. The pioneering work of ocvirk06 suggests approximately 10 independent components can be recovered from extremely high S/N R=10000 spectra (and perfect models). The fundamental problem is that the spectra of single age populations change slowly with age (or metallicity), so the contributions of each SSP to a composite spectrum are very highly degenerate and some degree of regularization or prior information is required. However, the ideal regularization depends on the (*a priori* unknown) SFH of the galaxy. For example, for a narrow burst one would want many narrow bins near the burst and wide bins away from it. Reducing the number of bins effectively amounts to collapsing the prior for the ratio of the SFR in two sub-bins to a delta-function at 1. Using too few bins can result in biases in the same way as the strong priors imposed by parametric models. Tests in **‘leja19 <<https://ui.adsabs.harvard.edu/abs/2019ApJ...873...44C/abstract>>‘_** suggest that ~5 bins are adequate to model covariances in basic parameters from photometry, but more bins are better to explore detailed constraints on SFHs.

9.9 So should I use *emcee*, *nestle*, or *dynesty* for posterior sampling?

We recommend using the *dynesty* nested sampling package.

In addition to the standard sampling phase which terminates based on the quality of the estimation of the Bayesian evidence, *dyensty* includes a subsequent dynamic sampling phase which, as implemented in Prospector, instead terminates based the quality of the posterior estimation. This permits the user to specify stopping criteria based directly on the quality of the posterior sampling with the `nested_posterior_thresh` keyword, providing direct control over the trade-off between posterior quality and computational time. A value of 0.02 for this keyword specifies high-quality posteriors, whereas a value of 0.05 will produce reasonable but approximate posteriors. Additionally, *dynesty* sampling can be parallelized in Prospector: this produces faster convergence time at the cost of lower computational efficiency (i.e., fewer model evaluations per unit computational time). It is best suited for fast evaluation of small samples of objects, whereas single-core fits produce more computationally efficient fits to large samples of objects.

9.10 What settings should I use for *dynesty*?

The default *dynesty* settings in Prospector are optimized for a low-dimensional (N=4-7) model. Higher-dimensional models with more complex likelihood spaces will likely require more advanced *dynesty* settings to ensure efficient and timely convergence. This often entails increasing the number of live points, changing to more robust sampling methodology (e.g., from uniform to a random walk), setting a maximum number of function calls, or altering the target evidence and posterior thresholds. More details can be found in speagle20 and the dynesty online documentation. The list of options and their default values can be seen with

```
from prospect.utils import prospect_args
prospect_args.show_default_args()
```

9.11 The chains did not converge when using *dynesty*, why?

It is likely that they did converge; note that the convergence for MC sampling of a posterior PDF is not defined by the samples all tending toward the a single value, but as the *distribution* of samples remaining stable. The samples for a poorly constrained parameter will remain widely dispersed, even if the MC sampling has converged to the correct *distribution*

9.12 How do I use *emcee* in Prospector?

For each parameter, an initial value (`"init"` in the parameter specification) must be given. The ensemble of walkers is initialized around this value, with a Gaussian spread that can be specified separately for each parameter. Each walker position is evolved at each iteration using parameter proposals derived from an ensemble of the other walkers. In order to speed up initial movement of the cloud of walkers to the region of parameter space containing most of the probability mass, multiple user defined rounds of burn-in may be performed. After each round the walker distribution in parameter space is re-initialized to a multivariate Gaussian derived from the best 50% of the walkers (where best is defined in terms of posterior probability at the last iteration). The iterations in these burn-in rounds are discarded before a final production run. It is important to ensure that the chain of walkers has converged to a stable *distribution* of parameter values. Diagnosing convergence is fraught; a number of indicators have been proposed sharma17 including the auto-correlation time of the chain goodman10. Comparing the results of separate chains can also provide a sanity check.

9.13 When should I use optimization?

Optimization can be performed before ensemble MCMC sampling, to decrease the burn-in time of the MCMC algorithm. Prospector currently supports Levenburg-Marquardt least-squares optimization and Powell's method, as implemented in SciPy. It is possible to start optimizations from a number of different parameter values, drawn from the prior parameter distribution, in order to mitigate the problems posed by local maxima.

9.14 How do I plot the best fit SED? How do I plot uncertainties on that?

Prospector can compute and store the SED prediction for the highest probability sample, in the `"bestfit"` group of the output HDF5 file.

Note that the highest probability sample is *not* the same as the maximum a posteriori (MAP) solution. The MAP solution inhabits a vanishingly small region of the prior parameter space; it is exceedingly unlikely that the MCMC sampler would visit exactly that location. Furthermore, when large degeneracies are present, the maximum a posteriori parameters may be only very slightly more likely than many solutions with very different parameters.

To plot uncertainties we recommend regenerating SED predictions for a fair sample from the posterior PDF and estimating quantiles of the flux at each wavelength.

9.15 How do I get the wavelength array for plotting spectra and/or photometry when fitting only photometry?

When fitting only photometry, the *restframe* wavelength array for the predicted spectrum can be found in the `wavelengths` attribute of *`prospect.sources.SSPBasis`*. The wavelengths of the filters can be obtained from the `wave_effective` attribute of each filter in the `obs["filters"]` list.

9.16 Should I fit spectra in the restframe or the observed frame?

You can do either if you are fitting only spectra. If fitting in the restframe then the distance has to be specified explicitly, otherwise it is inferred from the redshift.

If you are fitting photometry and spectroscopy then you should be fitting the observed frame spectra.

9.17 What priors should I use?

That depends on the scientific question and the objects under consideration. In general we recommend using informative priors (e.g. narrow `Normal` distributions) for parameters that you think might matter at all.

9.18 What happens if a parameter is not well constrained? When should I fix parameters?

If some parameter is completely unconstrained you will get back the prior. There are also (often) cases where you are “prior-dominated”, i.e. the posterior is mostly set by the prior but with a small perturbation due to small amounts of information supplied by the data. You can compare the posterior to the prior, e.g. using the Kullback-Liebler divergence between the two distributions, to see if you have learned anything about that parameter. Or just overplot the prior on the marginalized pPDFs

To be fully righteous you should only fix parameters if

- you are very sure of their values;
- or if you don’t think changing the parameter will have a noticeable effect on the model;
- or if a parameter is perfectly degenerate (in the space of the data) with another parameter.

In practice parameters that have only a small effect but take a great deal of time to vary are often fixed.

9.19 What do I do about upper limits?

9.20 What do I do with the chain? What values should I report?

This is a general question for MC sampling techniques.

9.21 Why isn't the posterior PDF centered on the highest posterior probability sample?

9.22 How do I interpret the *lnprobability* or *lnp* values? Why do I get *lnp > 0*?

9.23 How do I know if Prospector is "working"?

CHAPTER 10

prospect.models

This module includes objects that store parameter specfications and efficiently convert between parameter dictionaries and parameter vectors necessary for fitting algorithms. There are submodules for parameter priors, common parameter transformations, and pre-defined sets of parameter specifications.

class prospect.models.**ProspectorParams**(*configuration*, *verbose=True*, *param_order=None*, ***kwargs*)

This is the base model class that holds model parameters and information about them (e.g. priors, bounds, transforms, free vs fixed state). In addition to the documented methods, it contains several important attributes:

- `params`: model parameter state dictionary.
- `theta_index`: A dictionary that maps parameter names to indices (or rather slices) of the parameter vector `theta`.
- `config_dict`: Information about each parameter as a dictionary keyed by parameter name for easy access.
- `config_list`: Information about each parameter stored as a list.

Intitialization is via, e.g.,

```
model_dict = {"mass": {"N": 1, "isfree": False, "init": 1e10}}
model = ProspectorParams(model_dict, param_order=None)
```

Parameters **`configuration`** – A list or dictionary of model parameters specifications.

clip_to_bounds(*thetas*)

Clip a set of parameters theta to within the priors.

Parameters **`thetas`** – The parameter vector, ndarray of shape `(ndim,)`.

Returns thetas The input vector, clipped to the bounds of the priors.

configure(*reset=False*, ***kwargs*)

Use the `config_dict` to generate a `theta_index` mapping, and propogate the initial parameters into the `params` state dictionary, and store the intital theta vector thus implied.

Parameters

- **`kwargs`** – Keyword parameters can be used to override or add to the initial parameter values specified in `config_dict`
- **`reset`** – (default: False) If true, empty the params dictionary before re-reading the `config_dict`

`fixed_params`
A list of the names fixed model parameters that are specified in the `config_dict`.

`free_params`
A list of the names of the free model parameters.

`map_theta`()
Construct the mapping from parameter name to the index in the theta vector corresponding to the first element of that parameter. Called during configuration.

`prior_product`(*theta*, *nested=False*, ***extras*)
Public version of _prior_product to be overridden by subclasses.

Parameters

- **`theta`** – The parameter vector for which you want to calculate the prior. ndarray of shape `(..., ndim)`
- **`nested`** – If using nested sampling, this will only return 0 (or -inf). This behavior can be overridden if you want to include complicated priors that are not included in the unit prior cube based proposals (e.g. something that is difficult to transform from the unit cube.)

Returns lnp_prior The natural log of the prior probability at `theta`

`prior_transform`(*unit_coords*)
Go from unit cube to parameter space, for nested sampling.

Parameters **`unit_coords`** – Coordinates in the unit hyper-cube. ndarray of shape `(ndim,)`.

Returns theta The parameter vector corresponding to the location in prior CDF corresponding to `unit_coords`. ndarray of shape `(ndim,)`

`propagate_parameter_dependencies`()
Propogate any parameter dependecies. That is, for parameters whose value depends on another parameter, calculate those values and store them in the `self.params` dictionary.

`rectify_theta`(*theta*, *epsilon=1e-10*)
Replace zeros in a given theta vector with a small number epsilon.

`set_parameters`(*theta*)
Propagate theta into the model parameters `params` dictionary.

Parameters **`theta`** – A theta parameter vector containing the desired parameters. ndarray of shape `(ndim,)`

`theta`
The current value of the theta vector, pulled from the `params` state dictionary.

`theta_bounds`()
Get the bounds on each parameter from the prior.

Returns bounds A list of length `ndim` of tuples `(lo, hi)` giving the parameter bounds.

`theta_disp_floor`(*thetas=None*)
Get a vector of dispersions for each parameter to use as a floor for the emcee walker-calculated dispersions. This can be overridden by subclasses.

Returns disp_floor The minimum dispersion in the parameters to use for generating clouds of walkers (or minimizers.) ndarray of shape `(ndim,)`

`theta_disps`(*default_disp=0.1*, *fractional_disp=False*)
Get a vector of absolute dispersions for each parameter to use in generating sampler balls for emcee's Ensemble sampler. This can be overridden by subclasses if fractional dispersions are desired.

Parameters

- **`initial_disp`** – (default: 0.1) The default dispersion to use in case the `"init_disp"` key is not provided in the parameter configuration.
- **`fractional_disp`** – (default: False) Treat the dispersion values as fractional dispersions.

Returns disp The dispersion in the parameters to use for generating clouds of walkers (or minimizers.) ndarray of shape `(ndim,)`

`theta_labels`(*name_map={}*)
Using the theta_index parameter map, return a list of the model parameter names that has the same order as the sampling chain array.

Parameters **`name_map`** – A dictionary mapping model parameter names to output label names.

Returns labels A list of labels of the same length and order as the theta vector.

class `prospect.models.`**`SpecModel`**(*configuration*, *verbose=True*, *param_order=None*, ***kwargs*)
A subclass of *`ProspectorParams`* that passes the models through to an `sps` object and returns spectra and photometry, including optional spectroscopic calibration, and sky emission.

This class performs most of the conversion from intrinsic model spectrum to observed quantities, and additionally can compute MAP emission line values and penalties for marginalization over emission line amplitudes.

`cache_eline_parameters`(*obs*, *nsigma=5*)
This computes and caches a number of quantities that are relevant for predicting the emission lines, and computing the MAP values thereof, including

- `_ewave_obs` - Observed frame wavelengths (AA) of all emission lines.
- `_eline_sigma_kms` - Dispersion (in km/s) of all the emission lines
- `_elines_to_fit` - If fitting and marginalizing over emission lines, this stores indices of the lines to actually fit, as a boolean array. Only lines that are within `nsigma` of an observed wavelength points are included.
- `_eline_wavelength_mask` - A mask of the *_outwave* vector that indicates which pixels to use in the emission line fitting. Only pixels within `nsigma` of an emission line are used.

Can be subclassed to add more sophistication redshift - first looks for `eline_delta_zred`, and defaults to `zred` sigma - first looks for `eline_sigma`, defaults to 100 km/s

Parameters **`nsigma`** – (float, optional, default: 5.) Number of sigma from a line center to use for defining which lines to fit and useful spectral elements for the fitting. float.

`flux_norm`()
Compute the scaling required to go from Lsun/Hz/Msun to maggies. Note this includes the (1+z) factor required for flux densities.

Returns norm (float) The normalization factor, scalar float.

`get_el`(*obs*, *calibrated_spec*, *sigma_spec=None*)
Compute the maximum likelihood and, optionally, MAP emission line amplitudes for lines that fall within

the observed spectral range. Also compute and cache the analytic penalty to log-likelihood from marginalizing over the emission line amplitudes. This is cached as `_ln_eline_penalty`. The emission line amplitudes (in maggies) at *_eline_lums* are updated to the ML values for the fitted lines.

Parameters

- **`obs`** – A dictionary containing the `'spectrum'` and `'unc'` keys that are observed fluxes and uncertainties, both ndarrays of shape `(n_wave,)`
- **`calibrated_spec`** – The predicted observer-frame spectrum in the same units as the observed spectrum, ndarray of shape `(n_wave,)`
- **`sigma_spec`** – Spectral covariance matrix, if using a non-trivial noise model.

Returns el The maximum likelihood emission line flux densities. ndarray of shape `(n_wave_neb, n_fitted_lines)` where `n_wave_neb` is the number of wavelength elements within `nsigma` of a line, and `n_fitted_lines` is the number of lines that fall within `nsigma` of a wavelength pixel. Units are same as `calibrated_spec`

`get_eline_gaussians`(*lineidx=slice(None, None, None)*, *wave=None*)

Generate a set of unit normals with centers and widths given by the previously cached emission line observed-frame wavelengths and emission line widths.

Parameters

- **`lineidx`** – (optional) A boolean array or integer array used to subscript the cached lines. Gaussian vectors will only be constructed for the lines thus subscripted.
- **`wave`** – (optional) The wavelength array (in Angstroms) used to construct the gaussian vectors. If not given, the cached *_outwave* array will be used.

Returns gaussians The unit gaussians for each line, in units Lsun/Hz. ndarray of shape (n_wave, n_line)

`get_eline_spec`(*wave=None*)

Compute a complete model emission line spectrum. This should only be run after calling predict(), as it accesses cached information. Relatively slow, useful for display purposes

Parameters **`wave`** – (optional, default: `None`) The wavelength ndarray on which to compute the emission line spectrum. If not supplied, the `_outwave` vector is used.

Returns eline_spec An (n_line, n_wave) ndarray

`mean_model`(*theta*, *obs*, *sps=None*, *sigma=None*, ***extras*)

Legacy wrapper around predict()

`nebline_photometry`(*filters*)

Compute the emission line contribution to photometry. This requires several cached attributes:

- `_ewave_obs`
- `_eline_lum`

Parameters **`filters`** – List of `sedpy.observate.Filter` objects

Returns nebflux The flux of the emission line through the filters, in units of maggies. ndarray of shape `(len(filters),)`

`observed_wave`(*wave*, *do_wavecal=False*)

Convert the restframe wavelngth grid to the observed frame wavelength grid, optionally including wavelength calibration adjustments. Requires that the `_zred` attribute is already set.

Parameters **`wave`** – The wavelength array

predict (*theta*, *obs=None*, *sps=None*, *sigma_spec=None*, ***extras*)
Given a `theta` vector, generate a spectrum, photometry, and any extras (e.g. stellar mass), including any calibration effects.

Parameters

- **theta** – ndarray of parameter values, of shape `(ndim,)`
- **obs** – An observation dictionary, containing the output wavelength array, the photometric filter lists, and the observed fluxes and uncertainties thereon. Assumed to be the result of `utils.obsutils.rectify_obs()`
- **sps** – An *sps* object to be used in the model generation. It must have the `get_galaxy_spectrum()` method defined.
- **sigma_spec** – (optional) The covariance matrix for the spectral noise. It is only used for emission line marginalization.

Returns spec The model spectrum for these parameters, at the wavelengths specified by `obs['wavelength']`, including multiplication by the calibration vector. Units of maggies

Returns phot The model photometry for these parameters, for the filters specified in `obs['filters']`. Units of maggies.

Returns extras Any extra aspects of the model that are returned. Typically this will be *mfrac* the ratio of the surviving stellar mass to the stellar mass formed.

predict_phot (*filters*)
Generate a prediction for the observed photometry. This method assumes that the parameters have been set and that the following attributes are present and correct:

- `_wave` - The SPS restframe wavelength array
- `_zred` - Redshift
- `_norm_spec` - Observed frame spectral fluxes, in units of maggies.
- `_eline_wave` and `_eline_lum` - emission line parameters from the SPS model

Parameters filters – List of `sedpy.observate.Filter` objects. If there is no photometry, `None` should be supplied

Returns phot Observed frame photometry of the model SED through the given filters. ndarray of shape `(len(filters),)`, in units of maggies. If `filters` is None, this returns 0.0

predict_spec (*obs*, *sigma_spec*, ***extras*)
Generate a prediction for the observed spectrum. This method assumes that the parameters have been set and that the following attributes are present and correct:

- `_wave` - The SPS restframe wavelength array
- `_zred` - Redshift
- `_norm_spec` - Observed frame spectral fluxes, in units of maggies
- `_eline_wave` and `_eline_lum` - emission line parameters from the SPS model

It generates the following attributes

- `_outwave` - Wavelength grid (observed frame)
- `_speccal` - Calibration vector

- `_elinespec` - emission line spectrum
- `_sed` - Intrinsic spectrum (before cilbration vector applied)

And if emission line marginalization is being performed, numerous quantities related to the emission lines are also cached (see `get_el()` for details.)

Parameters

- **`obs`** – An observation dictionary, containing the output wavelength array, the photometric filter lists, and the observed fluxes and uncertainties thereon. Assumed to be the result of `utils.obsutils.rectify_obs()`
- **`sigma_spec`** – (optional) The covariance matrix for the spectral noise. It is only used for emission line marginalization.

Returns spec The prediction for the observed frame spectral flux these parameters, at the wavelengths specified by `obs['wavelength']`, including multiplication by the calibration vector. ndarray of shape `(nwave,)` in units of maggies.

`smoothspec`(*wave*, *spec*)
Smooth the spectrum. See *`prospect.utils.smoothing.smoothspec()`* for details.

`wave_to_x`(*wavelength=None*, *mask=slice(None*, *None*, *None)*, ***extras*)
Map unmasked wavelengths to the interval -1, 1 masked wavelengths may have x>1, x<-1

`class` `prospect.models.`**`SedModel`**(*configuration*, *verbose=True*, *param_order=None*, ***kwargs*)
A subclass of *`ProspectorParams`* that passes the models through to an `sps` object and returns spectra and photometry, including optional spectroscopic calibration and sky emission.

`mean_model`(*theta*, *obs*, *sps=None*, *sigma_spec=None*, ***extras*)
Legacy wrapper around predict()

`predict`(*theta*, *obs=None*, *sps=None*, ***extras*)
Given a `theta` vector, generate a spectrum, photometry, and any extras (e.g. stellar mass), including any calibration effects.

Parameters

- **`theta`** – ndarray of parameter values, of shape `(ndim,)`
- **`obs`** – An observation dictionary, containing the output wavelength array, the photometric filter lists, and the observed fluxes and uncertainties thereon. Assumed to be the result of `utils.obsutils.rectify_obs()`
- **`sps`** – An *sps* object to be used in the model generation. It must have the `get_spectrum()` method defined.
- **`sigma_spec`** – (optional, unused) The covariance matrix for the spectral noise. It is only used for emission line marginalization.

Returns spec The model spectrum for these parameters, at the wavelengths specified by `obs['wavelength']`, including multiplication by the calibration vector. Units of maggies

Returns phot The model photometry for these parameters, for the filters specified in `obs['filters']`. Units of maggies.

Returns extras Any extra aspects of the model that are returned. Typically this will be *mfrac* the ratio of the surviving stellar mass to the stellar mass formed.

`sed`(*theta*, *obs=None*, *sps=None*, ***kwargs*)
Given a vector of parameters `theta`, generate a spectrum, photometry, and any extras (e.g. surviving

mass fraction), ***not** including any instrument calibration effects. The intrinsic spectrum thus produced is cached in *_spec* attribute

Parameters

- **`theta`** – ndarray of parameter values.
- **`obs`** – An observation dictionary, containing the output wavelength array, the photometric filter lists, and the observed fluxes and uncertainties thereon. Assumed to be the result of `utils.obsutils.rectify_obs()`
- **`sps`** – An *sps* object to be used in the model generation. It must have the `get_spectrum()` method defined.

Returns spec The model spectrum for these parameters, at the wavelengths specified by `obs['wavelength']`. Default units are maggies, and the calibration vector is **not** applied.

Returns phot The model photometry for these parameters, for the filters specified in `obs['filters']`. Units are maggies.

Returns extras Any extra aspects of the model that are returned. Typically this will be *mfrac* the ratio of the surviving stellar mass to the steallr mass formed.

`sky`(*obs*)
Model for the *additive* sky emission/absorption

`spec_calibration`(*theta=None*, *obs=None*, ***kwargs*)
Implements an overall scaling of the spectrum, given by the parameter `'spec_norm'`

Returns cal (float) A scalar multiplicative factor that gives the ratio between the true spectrum and the observed spectrum

`wave_to_x`(*wavelength=None*, *mask=slice(None*, *None*, *None)*, ***extras*)
Map unmasked wavelengths to the interval (-1, 1). Masked wavelengths may have x>1, x<-1

Parameters

- **`wavelength`** – The input wavelengths. ndarray of shape `(nwave,)`
- **`mask`** – optional The mask. slice or boolean array with `True` for unmasked elements. The interval (-1, 1) will be defined only by unmasked wavelength points

Returns x The wavelength vector, remapped to the interval (-1, 1). ndarray of same shape as `wavelength`

10.1 prospect.models.priors

priors.py – This module contains various objects to be used as priors. When called these return the ln-prior-probability, and they can also be used to construct prior transforms (for nested sampling) and can be sampled from.

class `prospect.models.priors.`**`Prior`**(*parnames=[]*, *name="*, ***kwargs*)
Encapsulate the priors in an object. Each prior should have a distribution name and optional parameters specifying scale and location (e.g. min/max or mean/sigma). These can be aliased at instantiation using the `parnames` keyword. When called, the argument should be a variable and the object should return the ln-prior-probability of that value.

```
ln_prior_prob = Prior()(value)
```

Should be able to sample from the prior, and to get the gradient of the prior at any variable value. Methods should also be avilable to give a useful plotting range and, if there are bounds, to return them.

Parameters **`parnames`** – A list of names of the parameters, used to alias the intrinsic parameter names. This way different instances of the same Prior can have different parameter names, in case they are being fit for....

`inverse_unit_transform`(*x*, ***kwargs*)
Go from the parameter value to the unit coordinate using the cdf.

`loc`
This should be overridden.

`sample`(*nsample=None*, ***kwargs*)
Draw a sample from the prior distribution.

Parameters **`nsample`** – (optional) Unused

`scale`
This should be overridden.

`unit_transform`(*x*, ***kwargs*)
Go from a value of the CDF (between 0 and 1) to the corresponding parameter value.

Parameters **`x`** – A scalar or vector of same length as the Prior with values between zero and one corresponding to the value of the CDF.

Returns theta The parameter value corresponding to the value of the CDF given by *x*.

`update`(***kwargs*)
Update *params* values using alias.

`class` `prospect.models.priors.`**`Uniform`**(*parnames=[]*, *name=”*, ***kwargs*)
A simple uniform prior, described by two parameters

Parameters

- **`mini`** – Minimum of the distribution
- **`maxi`** – Maximum of the distribution

`distribution = <scipy.stats._continuous_distns.uniform_gen object>`

`loc`
This should be overridden.

`scale`
This should be overridden.

`class` `prospect.models.priors.`**`TopHat`**(*parnames=[]*, *name=”*, ***kwargs*)
Uniform distribution between two bounds, renamed for backwards compatibility :param mini:

Minimum of the distribution

Parameters **`maxi`** – Maximum of the distribution

`class` `prospect.models.priors.`**`Normal`**(*parnames=[]*, *name=”*, ***kwargs*)
A simple gaussian prior.

Parameters

- **`mean`** – Mean of the distribution
- **`sigma`** – Standard deviation of the distribution

`distribution = <scipy.stats._continuous_distns.norm_gen object>`

loc
This should be overridden.

scale
This should be overridden.

class prospect.models.priors.**ClippedNormal**(*parnames=[]*, *name=''*, ***kwargs*)
A Gaussian prior clipped to some range.

Parameters

- **mean** – Mean of the normal distribution
- **sigma** – Standard deviation of the normal distribution
- **mini** – Minimum of the distribution
- **maxi** – Maximum of the distribution

distribution = <scipy.stats._continuous_distns.truncnorm_gen object>

loc
This should be overridden.

scale
This should be overridden.

class prospect.models.priors.**LogNormal**(*parnames=[]*, *name=''*, ***kwargs*)
A log-normal prior, where the natural log of the variable is distributed normally. Useful for parameters that cannot be less than zero.

Note that `LogNormal(np.exp(mode) / f) == LogNormal(np.exp(mode) * f)` and `f = np.exp(sigma)` corresponds to "one sigma" from the peak.

Parameters

- **mode** – Natural log of the variable value at which the probability density is highest.
- **sigma** – Standard deviation of the distribution of the natural log of the variable.

distribution = <scipy.stats._continuous_distns.lognorm_gen object>

loc
This should be overridden.

scale
This should be overridden.

class prospect.models.priors.**LogUniform**(*parnames=[]*, *name=''*, ***kwargs*)
Like log-normal, but the distribution of natural log of the variable is distributed uniformly instead of normally.

Parameters

- **mini** – Minimum of the distribution
- **maxi** – Maximum of the distribution

distribution = <scipy.stats._continuous_distns.reciprocal_gen object>

class prospect.models.priors.**Beta**(*parnames=[]*, *name=''*, ***kwargs*)
A Beta distribution.

Parameters

- **mini** – Minimum of the distribution
- **maxi** – Maximum of the distribution

- **alpha** –
- **beta** –

distribution = <scipy.stats._continuous_distns.beta_gen object>

loc
This should be overridden.

scale
This should be overridden.

class prospect.models.priors.**StudentT**(*parnames=[]*, *name=''*, ***kwargs*)
A Student's T distribution

Parameters

- **mean** – Mean of the distribution
- **scale** – Size of the distribution, analogous to the standard deviation
- **df** – Number of degrees of freedom

distribution = <scipy.stats._continuous_distns.t_gen object>

loc
This should be overridden.

scale
This should be overridden.

class prospect.models.priors.**SkewNormal**(*parnames=[]*, *name=''*, ***kwargs*)
A normal distribution including a skew parameter

Parameters

- **location** – Center (*not* mean, mode, or median) of the distribution. The center will approach the mean as skew approaches zero.
- **sigma** – Standard deviation of the distribution
- **skew** – Skewness of the distribution

distribution = <scipy.stats._continuous_distns.skew_norm_gen object>

loc
This should be overridden.

scale
This should be overridden.

10.2 prospect.models.transforms

transforms.py – This module contains parameter transformations that may be useful to transform from parameters that are easier to _sample_ in to the parameters required for building SED models.

They can be used as "depends_on" entries in parameter specifications.

prospect.models.transforms.**stellar_logzsol**(*logzsol=0.0*, ***extras*)
Simple function that takes an argument list and returns the value of the *logzsol* argument (i.e. the stellar metallicity)

Parameters **logzsol** – FSPS stellar metaliicity parameter.

Returns logzsol The same.

`prospect.models.transforms.`**`delogify_mass`**(*logmass=0.0*, ***extras*)

Simple function that takes an argument list including a *logmass* parameter and returns the corresponding linear mass.

Parameters **`logmass`** – The log10(mass)

Returns mass The mass in linear units

`prospect.models.transforms.`**`tburst_from_fage`**(*tage=0.0*, *fage_burst=0.0*, ***extras*)

This function transfroms from a fractional age of a burst to an absolute age. With this transformation one can sample in `fage_burst` without worry about the case `tburst` > `tage`.

Parameters

- **`tage`** – The age of the host galaxy (Gyr)
- **`fage_burst`** – The fraction of the host age at which the burst occurred.

Returns tburst The age of the host when the burst occurred (i.e. the FSPS `tburst` parameter)

`prospect.models.transforms.`**`tage_from_tuniv`**(*zred=0.0*, *tage_tuniv=1.0*, ***extras*)

This function calculates a galaxy age from the age of the universe at `zred` and the age given as a fraction of the age of the universe. This allows for both `zred` and `tage` parameters without `tage` exceeding the age of the universe.

Parameters

- **`zred`** – Cosmological redshift.
- **`tage_tuniv`** – The ratio of `tage` to the age of the universe at `zred`.

Returns tage The stellar population age, in Gyr

`prospect.models.transforms.`**`zred_to_agebins`**(*zred=0.0*, *agebins=[]*, ***extras*)

Set the nonparameteric SFH age bins depending on the age of the universe at `zred`. The first bin is not altered and the last bin is always 15% of the upper edge of the oldest bin, but the intervening bins are evenly spaced in log(age).

Parameters

- **`zred`** – Cosmological redshift. This sets the age of the universe.
- **`agebins`** – The SFH bin edges in log10(years). ndarray of shape `(nbin, 2)`.

Returns agebins The new SFH bin edges.

`prospect.models.transforms.`**`dustratio_to_dust1`**(*dust2=0.0*, *dust_ratio=0.0*, ***extras*)

Set the value of dust1 from the value of dust2 and dust_ratio

Parameters

- **`dust2`** – The diffuse dust V-band optical depth (the FSPS `dust2` parameter.)
- **`dust_ratio`** – The ratio of the extra optical depth towards young stars to the diffuse optical depth affecting all stars.

Returns dust1 The extra optical depth towards young stars (the FSPS `dust1` parameter.)

`prospect.models.transforms.`**`logsfr_ratios_to_masses`**(*logmass=None*, *logsfr_ratios=None*, *agebins=None*, ***extras*)

This converts from an array of log_10(SFR_j / SFR_{j+1}) and a value of log10(Sum_i M_i) to values of M_i. j=0 is the most recent bin in lookback time.

prospect.models.transforms.**logsfr_ratios_to_sfrs**(*logmass=None*, *logsfr_ratios=None*, *agebins=None*, ***extras*)

Convenience function

prospect.models.transforms.**logsfr_ratios_to_agebins**(*logsfr_ratios=None*, *agebins=None*, ***extras*)

This transforms from SFR ratios to agebins by assuming a constant amount of mass forms in each bin agebins = np.array([NBINS,2])

use equation: delta(t1) = tuniv / (1 + SUM(n=1 to n=nbins-1) PROD(j=1 to j=n) Sn) where Sn = SFR(n) / SFR(n+1) and delta(t1) is width of youngest bin

prospect.models.transforms.**zfrac_to_masses**(*total_mass=None*, *z_fraction=None*, *agebins=None*, ***extras*)

This transforms from independent dimensionless z variables to sfr fractions and then to bin mass fractions. The transformation is such that sfr fractions are drawn from a Dirichlet prior. See Betancourt et al. 2010 and Leja et al. 2017

Parameters

- **total_mass** – The total mass formed over all bins in the SFH.
- **z_fraction** – latent variables drawn form a specific set of Beta distributions. (see Betancourt 2010)

Returns masses The stellar mass formed in each age bin.

prospect.models.transforms.**zfrac_to_sfrac**(*z_fraction=None*, ***extras*)

This transforms from independent dimensionless z variables to sfr fractions. The transformation is such that sfr fractions are drawn from a Dirichlet prior. See Betancourt et al. 2010 and Leja et al. 2017

Parameters **z_fraction** – latent variables drawn form a specific set of Beta distributions. (see Betancourt 2010)

Returns sfrac The star formation fractions (See Leja et al. 2017 for definition).

prospect.models.transforms.**zfrac_to_sfr**(*total_mass=None*, *z_fraction=None*, *agebins=None*, ***extras*)

This transforms from independent dimensionless z variables to SFRs.

Returns sfrs The SFR in each age bin (msun/yr).

prospect.models.transforms.**masses_to_zfrac**(*mass=None*, *agebins=None*, ***extras*)

The inverse of `zfrac_to_masses()`, for setting mock parameters based on mock bin masses.

Returns total_mass The total mass

Returns zfrac The dimensionless z variables used for sfr fraction parameterization.

CHAPTER 11

prospect.sources

Classes in the *prospect.sources* module are used to instantiate **sps** objects. They are defined by the presence of a `get_spectrum()` method that takes a wavelength array, a list of filter objects, and a parameter dictionary and return a spectrum, a set of broadband fluxes, and a blob of ancillary information.

Most of these classes are a wrapper on `fsps.StellarPopulation` objects, and as such have a significant memory footprint. The parameter dictionary can include any `fsps` parameter, as well as parameters used by these classes to control redshifting, spectral smoothing, wavelength calibration, and other aspects of the model.

class prospect.sources.**SSPBasis**(*zcontinuous=1, reserved_params=['tage', 'sigma_smooth'], interp_type='logarithmic', flux_interp='linear', mint_log=-3, compute_vega_mags=False, **kwargs*)

This is a class that wraps the fsps.StellarPopulation object, which is used for producing SSPs. The `fsps.StellarPopulation` object is accessed as `SSPBasis().ssp`.

This class allows for the custom calculation of relative SSP weights (by overriding `all_ssp_weights`) to produce spectra from arbitrary composite SFHs. Alternatively, the entire `get_galaxy_spectrum` method can be overridden to produce a galaxy spectrum in some other way, for example taking advantage of weight calculations within FSPS for tabular SFHs or for parameteric SFHs.

The base implementation here produces an SSP interpolated to the age given by `tage`, with initial mass given by `mass`. However, this is much slower than letting FSPS calculate the weights, as implemented in *FastSSPBasis*.

Furthermore, smoothing, redshifting, and filter projections are handled outside of FSPS, allowing for fast and more flexible algorithms.

Parameters **reserved_params** – These are parameters which have names like the FSPS parameters but will not be passed to the StellarPopulation object because we are overriding their functionality using (hopefully more efficient) custom algorithms.

all_ssp_weights

Weights for a single age population. This is a slow way to do this!

get_galaxy_elines()

Get the wavelengths and specific emission line luminosity of the nebular emission lines predicted by FSPS.

These lines are in units of Lsun/solar mass formed. This assumes that *get_galaxy_spectrum* has already been called.

Returns ewave The *restframe* wavelengths of the emission lines, AA

Returns elum Specific luminosities of the nebular emission lines, Lsun/stellar mass formed

`get_galaxy_spectrum`(***params*)

Update parameters, then multiply SSP weights by SSP spectra and stellar masses, and sum.

Returns wave Wavelength in angstroms.

Returns spectrum Spectrum in units of Lsun/Hz/solar masses formed.

Returns mass_fraction Fraction of the formed stellar mass that still exists.

`get_spectrum`(*outwave=None*, *filters=None*, *peraa=False*, ***params*)

Get a spectrum and SED for the given params.

Parameters

- **`outwave`** – (default: None) Desired *vacuum* wavelengths. Defaults to the values in *sps.wavelength*.
- **`peraa`** – (default: False) If *True*, return the spectrum in erg/s/cm^2/AA instead of AB maggies.
- **`filters`** – (default: None) A list of filter objects for which you'd like photometry to be calculated.
- **`**params`** – Optional keywords giving parameter values that will be used to generate the predicted spectrum.

Returns spec Observed frame spectrum in AB maggies, unless *peraa=True* in which case the units are erg/s/cm^2/AA.

Returns phot Observed frame photometry in AB maggies.

Returns mass_frac The ratio of the surviving stellar mass to the total mass formed.

`update`(***params*)

Update the parameters, passing the *unreserved* FSPS parameters through to the `fsps.StellarPopulation` object.

Parameters `params` – A parameter dictionary.

class `prospect.sources.`**`CSPSpecBasis`**(*zcontinuous=1*, *reserved_params=['sigma_smooth']*, *vactoair_flag=False*, *compute_vega_mags=False*, ***kwargs*)

A subclass of *`SSPBasis`* for combinations of N composite stellar populations (including single-age populations). The number of composite stellar populations is given by the length of the `"mass"` parameter. Other population properties can also be vectors of the same length as `"mass"` if they are independent for each component.

`get_galaxy_spectrum`(***params*)

Update parameters, then loop over each component getting a spectrum for each and sum with appropriate weights.

Parameters `params` – A parameter dictionary that gets passed to the `self.update` method and will generally include physical parameters that control the stellar population and output spectrum or SED.

Returns wave Wavelength in angstroms.

Returns spectrum Spectrum in units of Lsun/Hz/solar masses formed.

Returns mass_fraction Fraction of the formed stellar mass that still exists.

`update`(***params*)
Update the *params* attribute, making parameters scalar if possible.

`update_component`(*component_index*)
Pass params that correspond to a single component through to the fsps.StellarPopulation object.

Parameters `component_index` – The index of the component for which to pull out individual parameters that are passed to the fsps.StellarPopulation object.

`class prospect.sources.FastStepBasis`(*zcontinuous=1, reserved_params=['tage', 'sigma_smooth'], interp_type='logarithmic', flux_interp='linear', mint_log=-3, compute_vega_mags=False, **kwargs*)
Subclass of *SSPBasis* that implements a "nonparameteric" (i.e. binned) SFH. This is accomplished by generating a tabular SFH with the proper form to be passed to FSPS. The key parameters for this SFH are:

- `agebins` - array of shape ‘‘(nbin, 2)‘ giving the younger and older (in lookback time) edges of each bin in log10(years)
- `mass` - array of shape `(nbin,)` giving the total stellar mass (in solar masses) **formed** in each bin.

`convert_sfh`(*agebins*, *mformed*, *epsilon=0.0001*, *maxage=None*)
Given arrays of agebins and formed masses with each bin, calculate a tabular SFH. The resulting time vector has time points either side of each bin edge with a "closeness" defined by a parameter epsilon.

Parameters

- **agebins** – An array of bin edges, log(yrs). This method assumes that the upper edge of one bin is the same as the lower edge of another bin. ndarray of shape `(nbin, 2)`
- **mformed** – The stellar mass formed in each bin. ndarray of shape `(nbin,)`
- **epsilon** – (optional, default 1e-4) A small number used to define the fraction time separation of adjacent points at the bin edges.
- **maxage** – (optional, default: `None`) A maximum age of stars in the population, in yrs. If `None` then the maximum value of `agebins` is used. Note that an error will occur if maxage < the maximum age in agebins.

Returns time The output time array for use with sfh=3, in Gyr. ndarray of shape (2*N)

Returns sfr The output sfr array for use with sfh=3, in M_sun/yr. ndarray of shape (2*N)

Returns maxage The maximum valid age in the returned isochrone.

`get_galaxy_spectrum`(***params*)
Construct the tabular SFH and feed it to the `ssp`.

`class prospect.sources.FastSSPBasis`(*zcontinuous=1, reserved_params=['tage', 'sigma_smooth'], interp_type='logarithmic', flux_interp='linear', mint_log=-3, compute_vega_mags=False, **kwargs*)
A subclass of *SSPBasis* that is a faster way to do SSP models by letting FSPS do the weight calculations.

`get_galaxy_spectrum`(***params*)
Update parameters, then multiply SSP weights by SSP spectra and stellar masses, and sum.

Returns wave Wavelength in angstroms.

Returns spectrum Spectrum in units of Lsun/Hz/solar masses formed.

Returns mass_fraction Fraction of the formed stellar mass that still exists.

class prospect.sources.**BlackBodyDustBasis**(***kwargs*)

get_spectrum(*outwave=None*, *filters=None*, ***params*)
Given a params dictionary, generate spectroscopy, photometry and any extras (e.g. stellar mass).

Parameters

- **outwave** – The output wavelength vector.
- **filters** – A list of sedpy filter objects.
- ****params** – Keywords forming the parameter set.

Returns spec The restframe spectrum in units of erg/s/cm^2/AA

Returns phot The apparent (redshifted) maggies in each of the filters.

Returns extras A list of None type objects, only included for consistency with the SedModel class.

normalization()
This method computes the normalization (due do distance dimming, unit conversions, etc.) based on the content of the params dictionary.

one_sed(*icomp=0*, *wave=None*, *filters=None*, ***extras*)
Pull out individual component parameters from the param dictionary and generate spectra for those components

CHAPTER 12

prospect.fitting

`prospect.fitting.`**`lnprobfn`**(*theta*, *model=None*, *obs=None*, *sps=None*, *noise=(None, None)*, *residuals=False*, *nested=False*, *verbose=False*)

Given a parameter vector and optionally a dictionary of observational ata and a model object, return the matural log of the posterior. This requires that an sps object (and if using spectra and gaussian processes, a NoiseModel) be instantiated.

Parameters

- **`theta`** – Input parameter vector, ndarray of shape (ndim,)
- **`model`** – SedModel model object, with attributes including `params`, a dictionary of model parameter state. It must also have `prior_product()`, and `predict()` methods defined.
- **`obs`** –

  A dictionary of observational data. The keys should be

  – `"wavelength"` (angstroms)

  – `"spectrum"` (maggies)

  – `"unc"` (maggies)

  – `"maggies"` (photometry in maggies)

  – `"maggies_unc"` (photometry uncertainty in maggies)

  – `"filters"` (iterable of `sedpy.observate.Filter`)

  – and optional spectroscopic `"mask"` and `"phot_mask"` (same length as `spectrum` and `maggies` respectively, True means use the data points)
- **`sps`** – A *`prospect.sources.SSPBasis`* object or subclass thereof, or any object with a `get_spectrum` method that will take a dictionary of model parameters and return a spectrum, photometry, and ancillary information.
- **`noise`** – (optional, default: (None, None)) A 2-element tuple of `prospect.likelihood.NoiseModel` objects.

- **`residuals`** – (optional, default: False) A switch to allow vectors of χ values to be returned instead of a scalar posterior probability. This can be useful for least-squares optimization methods. Note that prior probabilities are not included in this calculation.
- **`nested`** – (optional, default: False) If `True`, do not add the ln-prior probability to the ln-likelihood when computing the ln-posterior. For nested sampling algorithms the prior probability is incorporated in the way samples are drawn, so should not be included here.

Returns lnp Ln posterior probability, unless `residuals=True` in which case a vector of χ values is returned.

`prospect.fitting.`**`fit_model`**(*obs*, *model*, *sps*, *noise=(None*, *None)*, *lnprobfn=<function lnprobfn>*, *optimize=False*, *emcee=False*, *dynesty=True*, ***kwargs*)

Fit a model to observations using a number of different methods

Parameters

- **`obs`** – The `obs` dictionary containing the data to fit to, which will be passed to `lnprobfn`.
- **`model`** – An instance of the *`prospect.models.SedModel`* class containing the model parameterization and parameter state. It will be passed to `lnprobfn`.
- **`sps`** – An instance of a *`prospect.sources.SSPBasis`* (sub-)class. Alternatively, anything with a compatible `get_spectrum()` can be used here. It will be passed to `lnprobfn`
- **`noise`** – (optional, default: (None, None)) A tuple of NoiseModel objects for the spectroscopy and photometry respectively. Can also be (None, None) in which case simple chi-square will be used.
- **`lnprobfn`** – (optional, default: lnprobfn) A posterior probability function that can take `obs`, `model`, `sps`, and `noise` as keywords. By default use the *`lnprobfn()`* defined above.
- **`optimize`** – (optional, default: False) If `True`, conduct a round of optimization before sampling from the posterior. The model state will be set to the best value at the end of optimization before continuing on to sampling or returning. Parameters controlling the optimization can be passed via `kwargs`, including
  - `min_method`: 'lm' | 'powell'
  - `nmin`: number of minimizations to do. Beyond the first, minimizations will be started from draws from the prior.
  - `min_opts`: dictionary of minimization options passed to the scipy.optimize.minimize method.

  See `run_minimize()` for details.
- **`emcee`** – (optional, default: False) If `True`, sample from the posterior using emcee. Additonal parameters controlling emcee can be passed via `**kwargs`. These include
  - `initial_positions`: A set of initial positions for the walkers
  - `hfile`: an open h5py.File file handle for writing result incrementally

  Many additional emcee parameters can be provided here, see `run_emcee()` for details.
- **`dynesty`** – If `True`, sample from the posterior using dynesty. Additonal parameters controlling dynesty can be passed via `**kwargs`. See `run_dynesty()` for details.

Returns output A dictionary with two keys, 'optimization' and 'sampling'. The value of each of these is a 2-tuple with results in the first element and durations (in seconds) in the second element.

CHAPTER 13

prospect.io

13.1 prospect.io.read_results

`prospect.io.read_results.`**`results_from`**(*filename*, *model_file=None*, *dangerous=True*, ***kwargs*)

Read a results file with stored model and MCMC chains.

Parameters

- **`filename`** – Name and path to the file holding the results. If `filename` ends in "h5" then it is assumed that this is an HDF5 file, otherwise it is assumed to be a pickle.
- **`dangerous`** – (default, True) If True, use the stored paramfile text to import the parameter file and reconstitute the model object. This executes code in the stored paramfile text during import, and is therefore dangerous.

Returns results

A dictionary of various results including:

- *"chain"* - Samples from the posterior probability (ndarray).
- *"lnprobability"* - The posterior probability of each sample.
- *"weights"* - The weight of each sample, if *dynesty* was used.
- *"theta_labels"* - List of strings describing free parameters.
- *"bestfit"* - The prediction of the data for the posterior sample with the highest *"lnprobability"*, as a dictionary.
- *"run_params"* - A dictionary of arguments supplied to prospector at the time of the fit.
- *"paramfile_text"* - Text of the file used to run prospector, string

Returns obs The obs dictionary

Returns model The models.SedModel() object, if it could be regenerated from the stored *"paramfile_text"*. Otherwise, *None*.

`prospect.io.read_results.`**`emcee_restarter`**(*restart_from=”*, *niter=32*, ***kwargs*)

Get the obs, model, and sps objects from a previous run, as well as the run_params and initial positions (which are determined from the end of the last run, and inserted into the run_params dictionary)

Parameters

- **`restart_from`** – Name of the file to restart the sampling from. An error is raised if this does not include an emcee style chain of shape (nwalker, niter, ndim)
- **`niter`** – (default: 32) Number of additional iterations to do (added toi run_params)

Returns obs The *obs* dictionary used in the last run.

Returns model The model object used in the last run.

Returns sps The *sps* object used in the last run.

Returns noise A tuple of (None, None), since it is assumed the noise model in the last run was trivial.

Returns run_params A dictionary of parameters controlling the operation. This is the same as used in the last run, but with the “niter” key changed, and a new “initial_positions” key that gives the ending positions of the emcee walkers from the last run. The filename from which the run is restarted is also stored in the “restart_from” key.

`prospect.io.read_results.`**`get_sps`**(*res*)

This gets exactly the SPS object used in the fiting (modulo any changes to FSPS itself).

It (scarily) imports the paramfile (stored as text in the results dictionary) as a module and then uses the *load_sps* method defined in the paramfile module.

Parameters **`res`** – A results dictionary (the output of *results_from()*)

Returns sps An sps object (i.e. from prospect.sources)

`prospect.io.read_results.`**`get_model`**(*res*)

This gets exactly the model object used in the fiting.

It (scarily) imports the paramfile (stored as text in the results dictionary) as a module and then uses the *load_model* method defined in the paramfile module, with *run_params* dictionary passed to it.

Parameters **`res`** – A results dictionary (the output of *results_from()*)

Returns model A prospect.models.SedModel object

`prospect.io.read_results.`**`traceplot`**(*results*, *showpars=None*, *start=0*, *chains=slice(None, None, None)*, *figsize=None*, *truths=None*, ***plot_kwargs*)

Plot the evolution of each parameter value with iteration #, for each walker in the chain.

Parameters

- **`results`** – A Prospector results dictionary, usually the output of `results_from('resultfile')`.
- **`showpars`** – (optional) A list of strings of the parameters to show. Defaults to all parameters in the `"theta_labels"` key of the `sample_results` dictionary.
- **`chains`** – If results are from an ensemble sampler, setting *chain* to an integer array of walker indices will cause only those walkers to be used in generating the plot. Useful for to keep the plot from getting too cluttered.
- **`start`** – (optional, default: 0) Integer giving the iteration number from which to start plotting.
- **`**plot_kwargs`** – Extra keywords are passed to the `matplotlib.axes._subplots.AxesSubplot.plot()` method.

Returns tracefig A multipaneled Figure object that shows the evolution of walker positions in the parameters given by `showpars`, as well as ln(posterior probability)

`prospect.io.read_results.`**`subcorner`**(*results, showpars=None, truths=None, start=0, thin=1, chains=slice(None, None, None), logify=['mass', 'tau'], **kwargs*)

Make a triangle plot of the (thinned, latter) samples of the posterior parameter space. Optionally make the plot only for a supplied subset of the parameters.

Parameters

- **`showpars`** – (optional) List of string names of parameters to include in the corner plot.
- **`truths`** – (optional) List of truth values for the chosen parameters.
- **`start`** – (optional, default: 0) The iteration number to start with when drawing samples to plot.
- **`thin`** – (optional, default: 1) The thinning of each chain to perform when drawing samples to plot.
- **`chains`** – (optional) If results are from an ensemble sampler, setting *chain* to an integer array of walker indices will cause only those walkers to be used in generating the plot. Useful for emoving stuck walkers.
- **`kwargs`** – Remaining keywords are passed to the `corner` plotting package.
- **`logify`** – A list of parameter names to plot in *log10(parameter)* instead of *parameter*

`prospect.io.read_results.`**`compare_paramfile`**(*res*, *filename*)

Compare the runtime parameter file text stored in the *res* dictionary to the text of some existing file with fully qualified path *filename*.

13.2 prospect.io.write_results

write_results.py - Methods for writing prospector ingredients and outputs to HDF5 files as well as to pickles.

`prospect.io.write_results.`**`write_hdf5`**(*hfile*, *run_params*, *model*, *obs*, *sampler=None*, *optimize_result_list=None*, *tsample=0.0*, *toptimize=0.0*, *sampling_initial_center=[]*, *sps=None*, ***extras*)

Write output and information to an HDF5 file object (or group).

Parameters

- **`hfile`** – File to which results will be written. Can be a string name or an *h5py.File* object handle.
- **`run_params`** – The dictionary of arguments used to build and fit a model.
- **`model`** – The *prospect.models.SedModel* object.
- **`obs`** – The dictionary of observations that were fit.
- **`sampler`** – The *emcee* or *dynesty* sampler object used to draw posterior samples. Can be *None* if only optimization was performed.
- **`optimize_result_list`** – A list of *scipy.optimize.OptimizationResult* objects generated during the optimization stage. Can be *None* if no optimization is performed

param sps: (optional, default: None) If a *prospect.sources.SSPBasis* object is supplied, it will be used to generate and store `prospect.io.write_results.`**`write_pickles`**(*run_params*, *model*, *obs*, *sampler*, *powell_results*, *outroot=None*, *tsample=None*, *toptimize=None*, *post_burnin_center=None*, *post_burnin_prob=None*, *sampling_initial_center=None*, *simpleout=False*, ***extras*)

Write results to two different pickle files. One (`*_mcmc`) contains only lists, dictionaries, and numpy arrays and is therefore robust to changes in object definitions. The other (`*_model`) contains the actual model object (and minimization result objects) and is therefore more fragile.

CHAPTER 14

prospect.plotting

figuremaker.py - module containing a class with basic plotting functionality and convenience methods for prospector results files.

class prospect.plotting.figuremaker.**FigureMaker**(*results_file=”*, *show=None*, *nufnu=False*, *microns=True*, *n_seds=-1*, *prior_samples=10000*, ***extras*)

A class for making figures from prospector results files. Usually you’ll want to subclass this and add specific plot making methods, But this class contains useful methods for generating and caching posterior SED predictions etc.

Parameters

- **show** – list of strings The names of the parameters (or transformed parameters) to show in posterior or corner plots
- **results_file** – string
- **n_seds** – int Number of SED samples to generate. if < 1, generate no SEDs. If 0, only generate the SEDs for the most probably sample
- **prior_samples** – int Number of prior samples to tale when computing the prior probability distributions numerically
- **nufnu** – bool Whether to plot fluxes in nufnu
- **microns** – bool Whether to plot wavelength as mictons

build_sps()

Build the SPS object and assign it to the *sps* attribute. This can be overridden by subclasses if necessary.

convert(*chain*)

Can make parameter transormations on the chain with this method, which you will want to subclass for your particular needs. For example, this method could be used to compute the mass-weighted age for every posterior sample and include that in the output structured array as the *mwa* column.

Parameters chain – structured ndarray of shape (nsample,) The structured ndarray of parameter values, with column names (and datatypes) given by the model free parameters.

Returns parchain structured ndarray of shape (nsample,) The structured ndarray of transformed parameter values.

draw_seds(*n_seds*, *dummy=None*)
Draw a number of samples from the posterior chain, and generate spectra and photometry for them.

Parameters

- **n_seds** – int Number of samples to draw and generate SEDs for
- **dummy** – dict, optional If given, use this dictionary as the obs dictionary when generating the intrinsic SED vector. Useful for generating an SED over a large wavelength range

make_art()
Make a dictionary of artists corresponding to the plotting styles that can be used for making legends

make_axes()
Make a set of axes and assign them to the object.

make_seds(*full=False*)
Generate and cache the best fit model spectrum and photometry. Optionally generate the spectrum and photometry for a number of posterior samples.

Populates the attributes **_best* and **_samples* where * is: * spec * phot * sed * cal

Parameters full – bool, optional If true, generate the intrinsic spextrum (*sed_**) over the entire wavelength range. The (restframe) wavelength vector will be given by *self.sps.wavelengths*

plot_all()
Main plotting function; makes axes, plotting styles, and then a corner plot and an SED (and residual) plot.

plot_corner(*caxes*, ***extras*)
Example to make a corner plot of the posterior PDFs for the parameters listed in *show*.

Parameters caxes – ndarray of axes of shape (nshow, nshow)

plot_sed(*sedax*, *residax=None*, *normalize=False*, *nufnu=True*, *microns=False*)
A very basic plot of the observed photometry and the best fit photometry and spectrum.

read_in(*results_file*)
Read a prospector results file, cache important components, and do any parameter transformations. The cached attributes are:

- *obs* - The *obs* dictionary ised for the fit
- *model* - The model used for the fit, if it could be reconstructed.
- *chain* - Structured array of parameter vector samples
- *weights* - Corresponding weights for each sample
- *ind_best* - Index of the sample with the highest posterior probability
- ***parchain* - The chain transformed to the desired derived parameters via** the *convert* method.

Parameters results_file – string full path of the file with the prospector results.

restframe_axis(*ax*, *microns=True*, *fontsize=16*, *ticksize=12*)
Add a second (top) x-axis with rest-frame wavelength

show_priors(*diagonals*, *spans*, *smooth=0.05*, *color='g'*, *peak=0.96*, ***linekwargs*)
Show priors, either using simple calls or using transformed samples. These will be overplotted on the supplied axes.

Parameters **diagonals** – ndarray of shape (nshow,) The axes on which to plot prior distributions, same order as *show*

show_transcurves(*ax*, *height=0.2*, *logify=True*, *linekwargs={'alpha': 0.7, 'color': '0.3', 'lw': 1.5}*)
Overplot transmission curves on an axis. The hight of the curves is computed as a (logarithmic) fraction of the current plot limits.

styles(*colorcycle=['royalblue', 'firebrick', 'indigo', 'darkorange', 'seagreen']*)
Define a set of plotting styles for use throughout the figure.

14.1 prospect.plotting.utils

prospect.plotting.utils.**sample_prior**(*model*, *nsample=1000000.0*)
Generate samples from the prior.

Parameters

- **model** – A ProspectorParams instance.
- **nsample** – (int, optional, default: 1000000) Number of samples to take

Returns samples ndarray of shape(nsample, ndim) Samples from the prior

Returns labels list of strings The names of the free parameters.

prospect.plotting.utils.**sample_posterior**(*chain*, *weights=None*, *nsample=10000*, *start=0*, *thin=1*, *extra=None*)

Parameters

- **chain** – ndarray of shape (niter, ndim) or (niter, nwalker, ndim)
- **weights** – weights for each sample, of shape (niter,)
- **nsample** – (optional, default: 10000) Number of samples to take
- **start** – (optional, default: 0.) Fraction of the beginning of the chain to throw away, expressed as a float in the range [0,1]
- **thin** – (optional, default: 1.) Thinning to apply to the chain before sampling (why would you do that?)
- **extra** – (optional, default: None) Array of extra values to sample along with the parameters of the chain. ndarray of shape (niter, …)

14.2 prospect.plotting.sfh

prospect.plotting.sfh.**sfh_quantiles**(*tvec, bins, sfrs, weights=None, q=[16, 50, 84]*)
Compute quantiles of a binned SFH

Parameters

- **tvec** – shape (ntime,) Vector of lookback times onto which the SFH will be interpolated.
- **bins** – shape (nsamples, nbin, 2) The age bins, in linear untis, same units as tvec
- **sfrs** – shape (nsamples, nbin) The SFR in each bin

Returns sfh_q shape(ntime, nq) The quantiles of the SFHs at each lookback time in *tvec* `prospect.plotting.sfh.`**`parametric_sfr`**(*tau=4*, *tage=13.7*, *power=1*, *mass=None*, *logmass=None*, *times=None*, ***extras*)

Return the SFR (Msun/yr) for the given parameters of an exponential or delayed exponential SFH. Does not account for burst, constant components, or truncations.

Parameters

- **`tau`** – float Exponential timescale, Gyr
- **`tage`** – float Lookback time of the earliest SF
- **`mass`** – float The total *formed* mass of the SFH up to *tage*.
- **`power`** – (optional, default: 1) Use 0 for exponential decline, and 1 for te^{-t} (delayed exponential decline)
- **`times`** – (optional, ndarray) If given, a set of *lookback* times where you want to calculate the sfr, same units as *tau* and *tage*

Returns sfr SFR in M_sun/year either for the lookback times given by *times* or at lookback time 0 if no times are given

`prospect.plotting.sfh.`**`nonpar_recent_sfr`**(*logmass*, *logsfr_ratios*, *agebins*, *sfr_period=0.1*)

vectorized

`prospect.plotting.sfh.`**`parametric_mwa`**(*tau=4*, *tage=13.7*, *power=1*)

Compute Mass-weighted age. This is done analytically

Parameters **`power`** – (optional, default: 1) Use 0 for exponential decline, and 1 for te^{-t} (delayed exponential decline)

`prospect.plotting.sfh.`**`nonpar_mwa`**(*logmass*, *logsfr_ratios*, *agebins*)

mass-weighted age, vectorized

14.3 prospect.plotting.corner

`prospect.plotting.corner.`**`quantile`**(*xarr*, *q*, *weights=None*)

Compute (weighted) quantiles from an input set of samples.

Parameters

- **`x`** – *~numpy.darray* with shape (nvar, nsamples) The input array to compute quantiles of.
- **`q`** – list of quantiles, from [0., 1.]
- **`weights`** – shape (nsamples)

Returns quants ndarray of shape (nvar, nq) The quantiles of each varaible.

`prospect.plotting.corner.`**`marginal`**(*x*, *ax=None*, *weights=None*, *span=None*, *smooth=0.02*, *color='black'*, *peak=None*, ***hist_kwargs*)

Compute a marginalized (weighted) histogram, with smoothing.

`prospect.plotting.corner.`**`corner`**(*samples*, *paxes*, *weights=None*, *span=None*, *smooth=0.02*, *color='black'*, *hist_kwargs={}*, *hist2d_kwargs={}*)

Make a smoothed cornerplot.

Parameters

- **`samples`** – *~numpy.ndarray* of shape (ndim, nsample) The samples from which to construct histograms.

- **`paxes`** – ndarray of pyplot.Axes of shape(ndim, ndim) Axes into which to plot the histograms.
- **`weights`** – ndarray of shape (nsample,), optional Weights associated with each sample.
- **`span`** – iterable with shape (ndim,), optional A list where each element is either a length-2 tuple containing lower and upper bounds or a float from *(0., 1.]* giving the fraction of (weighted) samples to include. If a fraction is provided, the bounds are chosen to be equal-tailed. An example would be:

  ```
  span = [(0., 10.), 0.95, (5., 6.)]
  ```

  Default is *0.999999426697* (5-sigma credible interval).

:param smooth [float or iterable with shape (ndim,), optional] The standard deviation (either a single value or a different value for each subplot) for the Gaussian kernel used to smooth the 1-D and 2-D marginalized posteriors, expressed as a fraction of the span. Default is *0.02* (2% smoothing). If an integer is provided instead, this will instead default to a simple (weighted) histogram with *bins=smooth*.

Parameters

- **`color`** – str or iterable with shape (ndim,), optional A *~matplotlib*-style color (either a single color or a different value for each subplot) used when plotting the histograms. Default is *'black'*.
- **`hist_kwargs`** – dict, optional Extra keyword arguments to send to the 1-D (smoothed) histograms.
- **`hist2d_kwargs`** – dict, optional Extra keyword arguments to send to the 2-D (smoothed) histograms.

Returns paxes

CHAPTER 15

prospect.utils

15.1 prospect.utils.smoothing

`prospect.utils.smoothing.`**`smoothspec`**`(`*wave*, *spec*, *resolution=None*, *outwave=None*, *smoothtype='vel'*, *fftsmooth=True*, *min_wave_smooth=0*, *max_wave_smooth=inf*, ***kwargs*`)`

Parameters

- **`wave`** – The wavelength vector of the input spectrum, ndarray. Assumed angstroms.
- **`spec`** – The flux vector of the input spectrum, ndarray
- **`resolution`** – The smoothing parameter. Units depend on `smoothtype`.
- **`outwave`** – The output wavelength vector. If `None` then the input wavelength vector will be assumed, though if `min_wave_smooth` or `max_wave_smooth` are also specified, then the output spectrum may have different length than `spec` or `wave`, or the convolution may be strange outside of `min_wave_smooth` and `max_wave_smooth`. Basically, always set `outwave` to be safe.
- **`smoothtype`** – (optional default: "vel") The type of smoothing to do. One of:
  - "vel" - velocity smoothing, `resolution` units are in km/s (dispersion not FWHM)
  - "R" - resolution smoothing, `resolution` is in units of lambda/ sigma(lambda) (where sigma(lambda) is dispersion, not FWHM)
  - "lambda" - wavelength smoothing. `resolution` is in units of AA
  - "lsf" - line-spread function. Use an aribitrary line spread function, which can be given as a vector the same length as `wave` that gives the dispersion (in AA) at each wavelength. Alternatively, if `resolution` is `None` then a line-spread function must be present as an additional `lsf` keyword. In this case all additional keywords as well as the `wave` vector will be passed to this `lsf` function.

- **fftsmooth** – (optional, default: True) Switch to use FFTs to do the smoothing, usually resulting in massive speedups of all algorithms.
- **min_wave_smooth** – (optional default: 0) The minimum wavelength of the input vector to consider when smoothing the spectrum. If `None` then it is determined from the output wavelength vector and padded by some multiple of the desired resolution.
- **max_wave_smooth** – (optional default: Inf) The maximum wavelength of the input vector to consider when smoothing the spectrum. If None then it is determined from the output wavelength vector and padded by some multiple of the desired resolution.
- **inres** – (optional) If given, this parameter specifies the resolution of the input. This resolution is subtracted in quadrature from the target output resolution before the kernel is formed.

  In certain cases this can be used to properly switch from resolution that is constant in velocity to one that is constant in wavelength, taking into account the wavelength dependence of the input resolution when defined in terms of lambda. This is possible iff: * `fftsmooth` is False * `smoothtype` is `"lambda"` * The optional `in_vel` parameter is supplied and True.

  The units of `inres` should be the same as the units of `resolution`, except in the case of switching from velocity to wavelength resolution, in which case the units of `inres` should be in units of lambda/sigma_lambda.
- **in_vel** – (optional) If supplied and True, the `inres` parameter is assumed to be in units of lambda/sigma_lambda. This parameter is ignored **unless** the `smoothtype` is `"lambda"` and `fftsmooth` is False.

Returns flux The smoothed spectrum on the *outwave* grid, ndarray.

CHAPTER 16

License and Attribution

If you use this code, please reference

```
@MISC{2019ascl.soft05025J,
    author = {{Johnson}, Benjamin D. and {Leja}, Joel L. and {Conroy}, Charlie and
        {Speagle}, Joshua S.},
        title = "{Prospector: Stellar population inference from spectra and SEDs}",
    keywords = {Software},
        year = 2019,
        month = may,
        eid = {ascl:1905.025},
        pages = {ascl:1905.025},
archivePrefix = {ascl},
    eprint = {1905.025},
    adsurl = {https://ui.adsabs.harvard.edu/abs/2019ascl.soft05025J},
    adsnote = {Provided by the SAO/NASA Astrophysics Data System}
}
```

Python Module Index

Index