12 prospect.io
  12.1 prospect.io.read_results ................................................. 41
  12.2 prospect.io.write_results ................................................. 43

13 prospect.utils ............................................................ 45
  13.1 prospect.utils.smoothing ..................................................... 45
  13.2 prospect.utils.plotting ...................................................... 46

14 License and Attribution ...................................................... 49

Python Module Index ............................................................. 51

Index ......................................................................................... 53
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.
Prospector is developed on Python 2.7 but should work with Python 3.

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 may also wish 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
Prospector is pure python.

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

Then in Python

```
import prospect
```
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 – including 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
```

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__':
    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 control 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)
    run_params["param_file"] = __file__
```

(continues on next page)
# Set up an output file name, build fit ingredients, and run the fit
hfile = "{}_{1}_mcmc.h5".format(args.outfile, int(time.time()))
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])

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 file-names 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:

```python
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 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 and is equivalent to basic $\chi^2$.  

---

8 Chapter 3. User Interaction
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 say 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. 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 maggies, same length as "filters". Maggies are a linear flux density unit defined as  

\[ \text{maggie} = 10^{0.4(m_{AB})} \]  

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 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.)
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

```python
mass = {"name": "mass",
        "N": 1,
        "init": 1e9,
        "init_disp": 1e8, # only important if using emcee sampling
        "units": "kg"
}
```
(continues on next page)
"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.

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

## 5.3 Parameter Set Templates

A number of predefined sets of parameters (with priors) are available as dictionaries of model specifications from `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

```python
from prospect.models.templates import TemplateLibrary
# Show all pre-defined parameter sets
TemplateLibrary.show_contents()
# Show details on the "parametric" 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)
```

## 5.4 The `build_model()` Method

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

The `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 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.
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

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

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.

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

### 6.2 Pickles

The results pickle is relatively portable file, which is a serialization of a dictionary containing
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

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

The model pickle has the extension `<timestamp>_model`. It is a direct serialization of the model object used during fitting, and is thus extremely 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:

```python
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

Several methods for visualization of the results are included in the Prospector.io.read_results module.

First, the results file can be read into useful dictionaries and objects using

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

See the help for `prospect.io.read_results.results_from()` for a description of the returned objects.

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 is useful to check for convergence. It can be done easily for both `emcee` and `dynesty` results by

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

Another useful thing is to look at the “corner plot” of the parameters. If one has the `corner.py` (`https://github.com/dfm/corner.py`) package, then

```python
cornerfig = reader.subcorner(results, showpars=mod.theta_labels()[0:5])
```

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 `subcorner` method, which is a thin wrapper on `corner.py`, but they are documented (help(reader.subcorner))

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 (for 'emcee' results)
ind_max = results['lnprobability'].argmax()
walker, iteration = np.unravel_index(ind_max, results['lnprobability'].shape)
theta_max = results['chain'][walker, iteration, :]

# 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.mean_model(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="Data")
pl.plot(wave, spec, label="MAP model")
You can check out the Jupyter notebook demo at

- InteractiveDemo

Here is an example of inference from an increasing number of photometric bands. Model parameters and SEDs are inferred (in grey) from a changing number of mock photometric bands. The mock is generated at the parameters marked in blue. This shows how with a small amount of data most posteriors are determined by the prior (dashed red) but that as the number of bands increases, the data are more informative and the posterior distributions are narrower than the prior.
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>`

```bash
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 (`:py:function:`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.

```python
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.

```python
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 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

```
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`.

```python
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.

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.

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

If you want to get the maximum a posteriori values, 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):

```python
# 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'].ravel(order='C')
else:
    # dynesty
    theta_max = res['chain'][imax, :].copy()
    flatchain = res['chain']

# 16th, 50th, and 84th percentiles of the posterior
from prospect.utils.plotting import quantile
post_pcts = [quantile(flatchain[:, i], percents=[16, 50, 84],
                      weights=res.get("weights", None))
              for i in range(model.ndim)]
```

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.

```python
# 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,
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, :]

# Get the modeled spectra and photometry.
# These have the same shape as the obs['spectrum'] and obs['maggies'] arrays.
spec, phot, mfrac = model.mean_model(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']
chiax.plot(wave, chi, 'o')
chiax.set_ylabel('Chi')
sedax.set_xlabel('wavelength')
chiax.set_xscale('log')
This module includes objects that store parameter specifications 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.

```python
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.

Initialization is via, e.g.,

```python
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 \( (\text{ndim},) \).
- **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 propagating the initial parameters into the params state dictionary, and store the initial 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.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, **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 key "logify_spectrum" which is True if the comparison to the model is to be made in the log.
• sps – An sps object to be used in the model generation. It must have the :py:method:`get_spectrum` method defined.

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

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

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, sps=None, **kwargs)
Given a theta vector, generate a spectrum, photometry, and any extras (e.g. stellar mass), *not* including any instrument calibration effects.

Parameters
• theta – ndarray of parameter values.
• sps – A StellarPopBasis object to be used in the model generation.
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 stellar mass formed.

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

```python
spec_calibration(theta=None, obs=None, **kwargs)
```
Implements a Chebyshev polynomial calibration model. This only occurs if "poly_coeffs" is present in the `params` dictionary, otherwise the value of `params['spec_norm']` is returned.

Parameters

- `theta` – (optional) If given, set `params` using this vector before calculating the calibration polynomial. `ndarray` of shape (ndim,)
- `obs` – A dictionary of observational data, must contain the key "wavelength"

Returns `cal` If `params['cal_type']` is "poly", a polynomial given by 'spec_norm':math:’ imes (1 + Sum_{m=1}^M''''poly_coeffs'[m-1]'':math:' imes T_n(x))’. Otherwise, the exponential of a Chebyshev polynomial.

```python
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

## 9.1 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.

```python
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.

```python
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 available 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..

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

```python
loc
```
This should be overridden.

```python
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.TopHat**(*parnames=[], name='', **kwargs*)

A simple uniform prior, described by two parameters

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

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

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

A simple gaussian prior.

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

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

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

A Gaussian prior clipped to some range.

- **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>`

**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>`
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>

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>

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>

9.2 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 metallicity parameter.
prospect.models.transforms.delogify_mass

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

This function transforms 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 \( \text{tburst} > \text{tage} \).

Parameters

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

Returns

\( \text{tburst} \)

The age of the host when the burst occurred (i.e. the FSPS \( \text{tburst} \) parameter)

prospect.models.transforms.tage_from_tuniv

This function calculates a galaxy age from the age of the universe at \( z_{\text{red}} \) and the age given as a fraction of the age of the universe. This allows for both \( z_{\text{red}} \) and \( \text{tage} \) parameters without \( \text{tage} \) exceeding the age of the universe.

Parameters

- **zred** – Cosmological redshift.
- **tage_tuniv** – The ratio of \( \text{tage} \) to the age of the universe at \( z_{\text{red}} \).

Returns

\( \text{tage} \)

The stellar population age, in Gyr

prospect.models.transforms.zred_to_agebins

Set the nonparameteric SFH age bins depending on the age of the universe at \( z_{\text{red}} \). 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

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

This converts from an array of \( \log_{10}(\text{SFR}_j / \text{SFR}_{j+1}) \) and a value of \( \log_{10}(\sum_i M_i) \) 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 sfr 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.
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.

```python
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**
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 \texttt{mass\_fraction} Fraction of the formed stellar mass that still exists.

\texttt{update(**params)}
Update the \texttt{params} attribute, making parameters scalar if possible.

\texttt{update\_component(component\_index)}
Pass params that correspond to a single component through to the \texttt{fsps.StellarPopulation} object.

\textbf{Parameters}
\texttt{component\_index} – The index of the component for which to pull out individual parameters that are passed to the \texttt{fsps.StellarPopulation} object.

class \texttt{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 \texttt{SSPBasis} that is a faster way to do SSP models by letting FSPS do the weight calculations.

g\texttt{et\_galaxy\_spectrum(**params)}
Update parameters, then multiply SSP weights by SSP spectra and stellar masses, and sum.

\textbf{Returns}
\setlength{\itemindent}{1em}

\texttt{wave} Wavelength in angstroms.

\texttt{spectrum} Spectrum in units of Lsun/Hz/solar masses formed.

\texttt{mass\_fraction} Fraction of the formed stellar mass that still exists.

class \texttt{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 \texttt{SSPBasis} that implements a “nonparametric” (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:

\begin{itemize}
\item \texttt{agebins} - array of shape \texttt{((nbin, 2))} giving the younger and older (in lookback time) edges of each bin in log\texttt{10(years)}
\item \texttt{mass} - array of shape \texttt{(nbin,)} giving the total stellar mass (in solar masses) \texttt{formed} in each bin.
\end{itemize}

\texttt{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.

\textbf{Parameters}
\begin{itemize}
\item \texttt{agebins} – An array of bin edges, log\texttt{10(years)}. This method assumes that the upper edge of one bin is the same as the lower edge of another bin. \texttt{ndarray} of shape \texttt{(nbin, 2)}
\item \texttt{mformed} – The stellar mass formed in each bin. \texttt{ndarray} of shape \texttt{(nbin,)}
\item \texttt{epsilon} – (optional, default 1e-4) A small number used to define the fraction time separation of adjacent points at the bin edges.
\item \texttt{maxage} – (optional, default: \texttt{None}) A maximum age of stars in the population, in yrs. If \texttt{None} then the maximum value of \texttt{agebins} is used. Note that an error will occur if maxage < the maximum age in agebins.
\end{itemize}

\textbf{Returns}
\begin{itemize}
\item \texttt{time} The output time array for use with \texttt{sfh=3}, in Gyr. \texttt{ndarray} of shape \texttt{(2*N)}
\item \texttt{sfr} The output \texttt{sfr} array for use with \texttt{sfh=3}, in M\texttt{sun}/yr. \texttt{ndarray} of shape \texttt{(2*N)}
\item \texttt{maxage} The maximum valid age in the returned isochrone.
\end{itemize}

\texttt{get\_galaxy\_spectrum(**params)}
Construct the tabular SFH and feed it to the \texttt{ssp}.
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 to 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
prospect.fitting.run_emcee_sampler(lnprobfn, initial_center, model, verbose=True, postargs=[], postkwargs={}, nwalkers=None, nburn=[16], niter=32, walker_factor=4, prob0=None, storechain=True, pool=None, hdf5=None, interval=1, convergence_check_interval=None, **kwargs)

Run an emcee sampler, including iterations of burn-in and re-initialization. Returns the production sampler.

Parameters

- **lnprobfn** – The posterior probability function.
- **initial_center** – The initial center for the sampler ball.
- **model** – An instance of a models.ProspectorParams object.

Parameters

- **postargs** – Positional arguments for lnprobfn.
- **postkwargs** – Keyword arguments for lnprobfn.
- **nwalkers** – The number of walkers to use. If None, use the nearest power of two to \(\text{ndim} \times \text{walker_factor}\).
- **niter** – Number of iterations for the production run.
- **nburn** – List of the number of iterations to run in each round of burn-in (for removing stuck walkers.) E.g. \(\text{nburn}=[32, 64]\) will run the sampler for 32 iterations before reinitializing and then run the sampler for another 64 iterations before starting the production run.
- **storechain** – (default: True) If using HDF5 output, setting this to False will keep the chain from being held in memory by the sampler object.
- **pool** – (optional) A Pool object, either from multiprocessing or from emcee.mpi_pool.
• **hdf5** – (optional) H5py.File object that will be used to store the chain in the datasets "chain" and "lnprobability". If not set, the chain will instead be stored as a numpy array in the returned sampler object.

• **interval** – (optional, default: 1) Fraction of the full run at which to flush to disk, if using hdf5 for output.

• **convergence_check_interval** – How often to assess convergence, in number of iterations. If this is not None, then the KL convergence test is run.

**Parameters**

• **convergence_chunks** – The number of iterations to combine when creating the marginalized parameter probability functions.

• **convergence_stable_points_criteria** – The number of stable convergence checks that the chain must pass before being declared stable.
CHAPTER 12

prospect.io

12.1 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 sample_results A dictionary of various results including the sampling chain.

Returns obs The obs dictionary

Returns model The models.sedmodel() object.

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 fitting (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 fitting.

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, thin=1, 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 removing 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

```python
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`.

### 12.2 prospect.io.write_results

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

```python
prospect.io.write_results.write_hdf5(hfile, run_params, model, obs, sampler, optimize_result_list, tsample=0.0, toptimize=0.0, sampling_initial_center=[], **extras)
```

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

```python
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.
13.1 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 arbitrary 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.

## 13.2 prospect.utils.plotting

**prospect.utils.plotting.get_best**(res, **kwargs)
Get the maximum a posteriori parameters.

**prospect.utils.plotting.get_percentiles**(res, ptile=[16, 50, 84], start=0.0, thin=1, **extras)
Get get percentiles of the marginalized posterior for each parameter.

**Parameters**

- **res** – A results dictionary, containing a “chain” and “theta_labels” keys.

- **ptile** – (optional, default: [16, 50, 84]) A list of percentiles (integers 0 to 100) to return for each parameter.

- **start** – (optional, default: 0.5) How much of the beginning of chains to throw away before calculating percentiles, expressed as a fraction of the total number of iterations.

- **thin** – (optional, default: 10.0) Only use every thin iteration when calculating percentiles.

**Returns** **pcts** Dictionary with keys giving the parameter names and values giving the requested percentiles for that parameter.

**prospect.utils.plotting.get_stats**(res, pnames, **kwargs)
For selected parameters, get the truth (if known), the MAP value from the chain, and the percentiles.

**Parameters**
- **res** – A results dictionary, containing a “chain” and “theta_labels” keys.
- **pnames** – List of strings giving the names of the desired parameters.

```python
prospect.utils.plotting.posterior_samples(res, samples=[1.0], **kwargs)
```

Pull samples of theta from the MCMC chain

**Parameters**

- **res** – A results dictionary, containing a “chain” and “theta_labels” keys.
- **samples** – Iterable of random numbers between 0 and 1.
- **kwargs** – Extra keywords are passed to hist_samples.

**Returns** thetas A list of parameter vectors pulled at random from the chain, of same length as samples.

```python
prospect.utils.plotting.hist_samples(res, showpars=None, start=0, thin=1, return_improb=False, **extras)
```

Get posterior samples for the parameters listed in showpars. This can be done for different ending fractions of the (thinned) chain.

**Parameters**

- **res** – A results dictionary, containing a “chain” and “theta_labels” keys.
- **showpars** – A list of strings giving the desired parameters.
- **start** – (optional, default: 0.5) How much of the beginning of chains to throw away before calculating percentiles, expressed as a fraction of the total number of iterations.
- **thin** – (optional, default: 10.0) Only use every thin iteration when calculating percentiles.

```python
prospect.utils.plotting.joint_pdf(res, p1, p2, pmap={}, **kwargs)
```

Build a 2-dimensional array representing the binned joint PDF of 2 parameters, in terms of sigma or fraction of the total distribution.

For example, to plot contours of the joint PDF of parameters "parname1" and "parname2" from the last half of a chain with 30bins in each dimension:

```python
xb, yb, sigma = joint_pdf(res, parname1, parname2, nbins=30, start=0.5)
ax.contour(xb, yb, sigma, **plotting_kwargs)
```

**Parameters**

- **p1** – The name of the parameter for the x-axis
- **p2** – The name of the parameter for the y axis

**Returns** xb, yb, sigma The bins and the 2-d histogram

```python
prospect.utils.plotting.compute_sigma_level(trace1, trace2, nbins=30, weights=None, extents=None, **extras)
```

From a set of traces in two parameters, make a 2-d histogram of number of standard deviations. Following examples from J Vanderplas.

```python
prospect.utils.plotting.trim_walkers(res, threshold=-10000.0)
```

Remove walkers with probability below some threshold. Useful for removing stuck walkers

```python
prospect.utils.plotting.fill_between(x, y1, y2=0, ax=None, **kwargs)
```

Plot filled region between y1 and y2.
This function works exactly the same as matplotlib’s fill_between, except that it also plots a proxy artist (specifically, a rectangle of 0 size) so that it can be added it appears on a legend.

```
prospect.utils.plotting.figgrid(ny, nx, figsize=None, left=0.1, right=0.85, top=0.9, bottom=0.1, wspace=0.2, hspace=0.1)
```

Gridpars is left, right
CHAPTER 14

License and Attribution

Copyright 2014-2018 Benjamin D. Johnson and contributors.
This code is available under the MIT License.
p
prospect.fitting, 39
prospect.io.read_results, 41
prospect.io.write_results, 43
prospect.models, 27
prospect.models.priors, 30
prospect.models.transforms, 32
prospect.sources, 35
prospect.utils.plotting, 46
prospect.utils.smoothing, 45
Index

A

all_ssp_weights (prospect.sources.SSPBasis attribute), 35

distribution (prospect.models.priors.TopHat attribute), 31
dustratio_to_dust1() (in module prospect.models.transforms), 33

B

Beta (class in prospect.models.priors), 32

BlackBodyDustBasis (class in prospect.sources), 37

distribution (prospect.models.priors.ClippedNormal attribute), 31

distribution (prospect.models.priors.LogNormal attribute), 31
distribution (prospect.models.priors.LogUniform attribute), 32
distribution (prospect.models.priors.Normal attribute), 31
distribution (prospect.models.priors.SkewNormal attribute), 32
distribution (prospect.models.priors.StudentT attribute), 32

C

clipped_to_bounds() (prospect.models.ProspectorParams method), 27

ClippedNormal (class in prospect.models.priors), 31

configure() (prospect.models.ProspectorParams method), 27

convert_sfh() (prospect.sources.FastStepBasis method), 37

CSPSpecBasis (class in prospect.sources), 36

D

delogify_mass() (in module prospect.models.transforms), 33
distribution (prospect.models.priors.Beta attribute), 32
distribution (prospect.models.priors.ClippedNormal attribute), 31
distribution (prospect.models.priors.LogNormal attribute), 31
distribution (prospect.models.priors.LogUniform attribute), 32
distribution (prospect.models.priors.Normal attribute), 31
distribution (prospect.models.priors.SkewNormal attribute), 32
distribution (prospect.models.priors.StudentT attribute), 32

distribution (prospect.models.priors.TopHat attribute), 31

dustratio_to_dust1() (in module prospect.models.transforms), 33

E

eemcee_restarter() (in module prospect.io.read_results), 41

F

FastSSPBasis (class in prospect.sources), 37

FastStepBasis (class in prospect.sources), 37

figgrid() (in module prospect.utils.plotting), 48

fill_between() (in module prospect.utils.plotting), 47

fixed_params (prospect.models.ProspectorParams attribute), 28

free_params (prospect.models.ProspectorParams attribute), 28

G

galaxy_spectrum() (prospect.models.CSPSpecBasis method), 36

galaxy_spectrum() (prospect.models.FastSSPBasis method), 37

galaxy_spectrum() (prospect.models.FastStepBasis method), 37

galaxy_spectrum() (prospect.models.SSPBasis method), 36

galaxy_spectrum() (prospect.models.SSPBasis method), 36

get_best() (in module prospect.utils.plotting), 46

galaxy_elines() (prospect.sources.SSPBasis method), 35

galaxy_spectrum() (prospect.models.CSPSpecBasis method), 36

galaxy_spectrum() (prospect.models.FastSSPBasis method), 37

galaxy_spectrum() (prospect.models.FastStepBasis method), 37

galaxy_spectrum() (prospect.models.SSPBasis method), 36

get_model() (in module prospect.io.read_results), 42

get_percentiles() (in module prospect.utils.plotting), 46

galaxy_spectrum() (prospect.sources.BlackBodyDustBasis method), 38
<table>
<thead>
<tr>
<th>Function</th>
<th>Module/Class/Method</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_spectrum()</td>
<td>prospect.sources.SSPBasis method</td>
<td>36</td>
</tr>
<tr>
<td>get_sps()</td>
<td>in module prospect.io.read_results</td>
<td>42</td>
</tr>
<tr>
<td>get_stats()</td>
<td>in module prospect.utils.plotting</td>
<td>46</td>
</tr>
<tr>
<td>hist_samples()</td>
<td>in module prospect.utils.plotting</td>
<td>47</td>
</tr>
<tr>
<td>inverse_unit_transform()</td>
<td>prospect.models.priors.Prior method</td>
<td>30</td>
</tr>
<tr>
<td>joint_pdf()</td>
<td>in module prospect.utils.plotting</td>
<td>47</td>
</tr>
<tr>
<td>loc</td>
<td>prospect.models.priors.Prior attribute</td>
<td>30</td>
</tr>
<tr>
<td>LogNormal</td>
<td>class in prospect.models.priors</td>
<td>31</td>
</tr>
<tr>
<td>logsfr_ratios_to_agebins()</td>
<td>in module prospect.models.transforms</td>
<td>34</td>
</tr>
<tr>
<td>logsfr_ratios_to_masses()</td>
<td>in module prospect.models.transforms</td>
<td>33</td>
</tr>
<tr>
<td>logsfr_ratios_to_sfrs()</td>
<td>in module prospect.models.transforms</td>
<td>33</td>
</tr>
<tr>
<td>LogUniform</td>
<td>class in prospect.models.priors</td>
<td>31</td>
</tr>
<tr>
<td>map_theta()</td>
<td>prospect.models.ProspectorParams method</td>
<td>28</td>
</tr>
<tr>
<td>masses_to_zfrac()</td>
<td>in module prospect.models.transforms</td>
<td>34</td>
</tr>
<tr>
<td>mean_model()</td>
<td>prospect.models.SedModel method</td>
<td>29</td>
</tr>
<tr>
<td>Normal</td>
<td>class in prospect.models.priors</td>
<td>31</td>
</tr>
<tr>
<td>normalization()</td>
<td>prospect.sources.BlackBodyDustBasis method</td>
<td>38</td>
</tr>
<tr>
<td>one_sed()</td>
<td>prospect.sources.BlackBodyDustBasis method</td>
<td>38</td>
</tr>
<tr>
<td>posterior_samples()</td>
<td>in module prospect.utils.plotting</td>
<td>47</td>
</tr>
<tr>
<td>Prior</td>
<td>class in prospect.models.priors</td>
<td>30</td>
</tr>
<tr>
<td>prior_product()</td>
<td>prospect.models.ProspectorParams method</td>
<td>28</td>
</tr>
<tr>
<td>prior_transform()</td>
<td>prospect.models.ProspectorParams method</td>
<td>28</td>
</tr>
<tr>
<td>propagate_parameter_dependencies()</td>
<td>prospect.models.ProspectorParams method</td>
<td>28</td>
</tr>
<tr>
<td>prospect.fitting</td>
<td>module</td>
<td>39</td>
</tr>
<tr>
<td>prospect.io.read_results</td>
<td>module</td>
<td>41</td>
</tr>
<tr>
<td>prospect.io.write_results</td>
<td>module</td>
<td>43</td>
</tr>
<tr>
<td>prospect.models</td>
<td>module</td>
<td>27</td>
</tr>
<tr>
<td>prospect.models.priors</td>
<td>module</td>
<td>30</td>
</tr>
<tr>
<td>prospect.models.transforms</td>
<td>module</td>
<td>32</td>
</tr>
<tr>
<td>prospect.models.sources</td>
<td>module</td>
<td>35</td>
</tr>
<tr>
<td>prospect.utils.plotting</td>
<td>module</td>
<td>46</td>
</tr>
<tr>
<td>prospect.utils.smoothing</td>
<td>module</td>
<td>45</td>
</tr>
<tr>
<td>ProspectorParams</td>
<td>class in prospect.models</td>
<td>27</td>
</tr>
<tr>
<td>rectify_theta()</td>
<td>prospect.models.ProspectorParams method</td>
<td>28</td>
</tr>
<tr>
<td>results_from()</td>
<td>in module prospect.io.read_results</td>
<td>41</td>
</tr>
<tr>
<td>run_emcee_sampler()</td>
<td>in module prospect.fitting</td>
<td>39</td>
</tr>
<tr>
<td>sample()</td>
<td>prospect.models.priors.Prior method</td>
<td>30</td>
</tr>
<tr>
<td>scale</td>
<td>prospect.models.priors.Prior attribute</td>
<td>31</td>
</tr>
<tr>
<td>sed()</td>
<td>prospect.models.SedModel method</td>
<td>29</td>
</tr>
<tr>
<td>SedModel</td>
<td>class in prospect.models</td>
<td>29</td>
</tr>
<tr>
<td>set_parameters()</td>
<td>prospect.models.ProspectorParams method</td>
<td>28</td>
</tr>
<tr>
<td>SkewNormal</td>
<td>class in prospect.models.priors</td>
<td>32</td>
</tr>
<tr>
<td>sky()</td>
<td>prospect.models.SedModel method</td>
<td>30</td>
</tr>
<tr>
<td>smoothspec()</td>
<td>in module prospect.utils.smoothing</td>
<td>45</td>
</tr>
<tr>
<td>spec_calibration()</td>
<td>prospect.models.SedModel method</td>
<td>30</td>
</tr>
<tr>
<td>SSPBasis</td>
<td>class in prospect.sources</td>
<td>35</td>
</tr>
<tr>
<td>stellar_logzsol()</td>
<td>in module prospect.models.transforms</td>
<td>32</td>
</tr>
<tr>
<td>StudentT</td>
<td>class in prospect.models.priors</td>
<td>32</td>
</tr>
<tr>
<td>subcorner()</td>
<td>in module prospect.io.read_results</td>
<td>42</td>
</tr>
<tr>
<td>tage_from_tuniv()</td>
<td>in module prospect.models.transforms</td>
<td>33</td>
</tr>
<tr>
<td>tburst_from_fage()</td>
<td>in module prospect.models.transforms</td>
<td>33</td>
</tr>
<tr>
<td>theta</td>
<td>prospect.models.ProspectorParams attribute</td>
<td>28</td>
</tr>
<tr>
<td>theta_bounds()</td>
<td>prospect.models.ProspectorParams method</td>
<td>28</td>
</tr>
<tr>
<td>theta_disp_floor()</td>
<td>prospect.models.ProspectorParams method</td>
<td>28</td>
</tr>
</tbody>
</table>
theta_disps() \textit{(prospect.models.ProspectorParams method)}, 29
theta_labels() \textit{(prospect.models.ProspectorParams method)}, 29
TopHat \textit{(class in prospect.models.priors)}, 31
traceplot() \textit{(in module prospect.io.read_results)}, 42
trim_walkers() \textit{(in module prospect.utils.plotting)}, 47

U
unit_transform() \textit{(prospect.models.priors.Prior method)}, 31
update() \textit{(prospect.models.priors.Prior method)}, 31
update() \textit{(prospect.sources.CSPSpecBasis method)}, 37
update() \textit{(prospect.sources.SSPBasis method)}, 36
update_component()
\textit{(prospect.sources.CSPSpecBasis method)}, 37

W
wave_to_x() \textit{(prospect.models.SedModel method)}, 30
write_hdf5() \textit{(in module prospect.io.write_results)}, 43
write_pickles() \textit{(in module prospect.io.write_results)}, 43

Z
zfrac_to_masses() \textit{(in module prospect.models.transforms)}, 34
zfrac_to_sfr() \textit{(in module prospect.models.transforms)}, 34
zfrac_to_sfrac() \textit{(in module prospect.models.transforms)}, 34
zred_to_agebins() \textit{(in module prospect.models.transforms)}, 33