Starfish Documentation

Release 0.4.1

Ian Czekala

Feb 13, 2022

CONTENTS

1 Contents 3

2 Indices and tables 57

Python Module Index 59

Index 61

Starfish is a framework used for robust spectroscopic inference. While this package was designed around the need to infer stellar properties such as effective temperature $T_{\rm eff}$, surface gravity $\log(g)$, and metallicity $[{\rm Fe/H}]$ from high resolution spectra, the framework could easily be adapted to any type of model spectra: g alaxy spectra, supernovae spectra, or spectra of unresolved stellar clusters.

For more technical information, please see our paper. Also, please cite both the paper and the code if *Starfish* or any derivative of its work was used for yours!

CHAPTER

ONE

CONTENTS

1.1 Getting Started

Spectroscopic inference is typically a complicated process, requiring customization based upon the type of spectrum used. Therefore, *Starfish* is not a one-click solution but rather a framework of code that provides the building blocks for any spectroscopic inference code one may write. We provide a few example scripts that show how the *Starfish* code objects may be combined to solve a typical spectroscopic inference problem. This page summarizes the various components available for use and seeks to orient the user. More detailed information is provided at the end of each section.

1.1.1 Citation

If *Starfish* or any derivative of it was used for your work, please cite both the paper and the code. We provide a BibTeX formatted file here for your convenience. Thanks!

Papers

Some research that has made considerable usage of Starfish includes

- Czekala et al. 2015
- Gully-Santiago et al. 2017
- Greene et al. 2018
- Zhang et al. 2021b
- Zhang et al. 2021c

If you have used Starfish in your work, please let us know and we can add you to this list!

1.1.2 Installation

The source code and installation instructions can be found at the GitHub repository for Starfish, but it should be easy enough to run

```
pip install astrostarfish
```

If you prefer to play with some of our new features, check out the code directly from master

```
pip install git+https://github.com/iancze/Starfish.git#egg=astrostarfish
```

or if you prefer an editable version just add the `-e` flag to `pip install`

1.1.3 Obtaining model spectra

Because any stellar synthesis step is currently prohibitively expensive for the purposes of Markov Chain Monte Carlo (MCMC) exploration, *Starfish* relies upon model spectra provided as a synthetic library. However, if you do have a synthesis back-end that is fast enough, please feel free to swap out the synthetic library for your synthetic back-end.

First, you will need to download your synthetic spectral library of choice. What libraries are acceptable are dictated by the spectral range and resolution of your data. In general, it is preferable to start with a raw synthetic library that is sampled at least a factor of ~5 higher than your data. For our paper, we used the freely available PHOENIX library synthesized by T. O. Husser. Because the size of spectral libraries is typically measured in gigabytes, I would recommend starting the download process now, and then finish reading the documentation :)

More information about how to download raw spectra and use other synthetic spectra is available in *Grid Tools*. *Starfish* provides a few objects which interface to these spectral libraries.

1.1.4 The Spectral Emulator

For high signal-to-noise data, we found that any interpolation error can constitute a large fraction of the uncertainty budget (see the appendix of our paper). For lower quality data, it may be possible to live with this interpolation error and use a simpler (and faster) interpolation scheme, such as tri-linear interpolation. However, we found that for sources with $S/N \geq 100$ a smoother interpolation scheme was required, and so we developed a spectral emulator.

The spectral emulator works by reconstructing spectra from a linear combination of eigenspectra, where the weight for each eigenspectrum is a function of the model parameters. Therefore, the first step is to deconstruct your spectral library into a set of eigenspectra using principal component analysis (PCA). Thankfully, most of the heavy lifting is already implemented by the *scikit-learn* package.

The next step is training a Gaussian Process to model the reconstruction weights as a function of model parameters (e.g., effective temperature $T_{\rm eff}$, surface gravity $\log(g)$, and metallicity $[{\rm Fe/H}]$). Because the spectral emulator delivers a probability distribution over the many possible interpolated spectra, we can propagate interpolation uncertainty into our final parameter estimates. For more on setting up the emulator, see *Spectral Emulator*.

1.1.5 Spectrum data formats and runtime

High resolution spectra are frequently taken with echelle spectrographs, which have many separate spectral orders, or "chunks", of data. This chunking is convenient because the likelihood evaluation of each chunk is independent from the other chunks, meaning that the global likelihood evaluation for the entire spectrum can be parallelized on a computer with many cores.

The runtime of *Starfish* strongly scales with the number of pixels in each chunk. If instead of a chunked dataset, you have a single merged array of more than 3000 pixels, we strongly advise chunking the dataset up to speed computation time. As long as you have as many CPU cores as you do chunks, the evaluation time of *Starfish* is roughly independent of the number of chunks. Therefore, if you have access to a 64 core node of a cluster, *Starfish* can fit an entire ~50 order high-res echelle spectrum in about the same time as it would take to fit a single order. (For testing purposes, it may be wise to use only single order to start, however.)

Astronomical spectra come in a wide variety of formats. Although there is effort to simplify reading these formats, it is beyond the scope of this package to provide an interface that would suit everyone. *Starfish* requires that the user convert their spectra into one of two simple formats: *numpy* arrays or HDF5 files. For more about converting spectra to these data formats, see *Spectrum*.

1.1.6 The MCMC driver script

The main purpose of *Starfish* is to provide a framework for robustly deriving model parameters using spectra. The ability to self-consistently downweight model systematics resulting from incorrectly modeled spectral lines is accomplished by using a non-trivial covariance matrix as part of a multi-dimensional Gaussian likelihood function. In principle, one could use traditional non-linear optimization techniques to find the maximum of the posterior probability distribution with respect to the model parameters. However, because one is usually keenly interested in the *uncertainties* on the best-fitting parameters, we must use an optimization technique that explores the full posterior, such as Markov Chain Monte Carlo (MCMC).

1.1.7 Memory usage

In our testing, *Starfish* requires a moderate amount of RAM per process (~1 Gb) for a spectrum that has chunk sizes of ~3000 pixels.

1.2 Overview

1.2.1 Deriving Physical Parameters from Astronomical Spectra

Consider the following scenario: An astronomer returns from a successful observing trip with many high signal-to-noise, high resolution stellar spectra on her trusty USB thumbdrive. If she was observing with the type of echelle spectrograph common to most modern observatories, chances are that her data span a significant spectral range, perhaps the full optical (3700 to 9000 angstrom) or the full near-infrared (0.7 to 5 microns). Now that she's back at her home institution, she sets herself to the task of determining the stellar properties of her targets.

She is an expert in the many existing well-tested techniques for determining stellar properties, such as MOOG and SME. But the fact that these use only a small portion of her data—several well-chosen lines like Fe and Na—has stubbornly persisted in the back of her mind.

At the same time, the astronomer has been paying attention to the steady increase in availability of high quality synthetic spectra, produced by a variety of groups around the world. These libraries span a large range of the stellar parameters she cares about (effective temperature, surface gravity, and metallicity) with a tremendous spectral coverage from the UV to the near-infrared—fully covering her dataset. She wonders, "instead of choosing a subset of lines to study, what if I use these synthetic libraries to fit *all of my data*?"

She knows that it's not quite as simple as just fitting more spectral range. She knows that even though the synthetic spectral libraries are generally high quality and quite remarkable in their scope, it is still very hard to produce perfect synthetic spectra. This is primarily due to inaccuracies in atomic and molecular constants that are difficult to measure in the lab, making it difficult to ensure that all spectral lines are accurate over a wide swath of both stellar parameters and spectral range. The highest quality libraries tend to achieve their precision by focusing on a "sweet spot" of stellar parameters near those of the Sun, and by choosing a limited spectral range, where atomic constants can be meticulously vetted for accuracy.

The astronomer also knows that some of her stars may have non-solar ratios of elemental abundances, a behavior that is not captured by the limited set of adjustable parameters t hat specify a spectrum in a synthetic library. She's tried fitting the full spectrum of her stars using a simple χ^2 likelihood function, but she knows that ignoring these effects will lead to parameter estimates that are biased and have unrealistically small uncertainties. She wonders, "How can I fit my entire spectrum but avoid these pitfalls?"

1.2.2 Introducing *Starfish*: a General Purpose Framework for Robust Spectroscopic Inference

We have developed a framework for spectroscopic inference that fulfills the astronomer's dream of using all of the data, called Starfish. Our statistical framework attempts to overcome many of the difficulties that the astronomer noted. Principally, at high resolution and high sensitivity, *model systematics*—such as inaccuracies in the *strengths of particular lines*—will dominate the noise budget.

We address these problems by accounting for the covariant structure of the residuals that can result from fitting models to data in this high signal-to-noise, high spectral resolution regime. Using some of the *machinery* developed by the field of Gaussian processes, we can parameterize the covariant structure both due to general line mis-matches as well as specific "outlier" spectral lines due to pathological errors in the atomic and molecular line databases.

Besides alleviating the problem of systematic bias and spectral line outliers when *inferring stellar parameters*, this approach has many added benefits. By forward-modeling the data spectrum, we put the problem of spectroscopic inference on true probabilistic footing. Rather than iterating in an open loop between stellar spectroscopists and stellar modelers, whereby knowledge about the accuracies of line fits is communicated post-mortem, a probabilistic inference framework like Starfish delivers posterior distributions over the locations and strengths of outlier spectral lines. Combined with a suite of stellar spectra spanning a range of stellar parameters and a tunable list of atomic and molecular constants, a probabilistic framework like this provides a way to close the loop on improving both the stellar models and the stellar parameters inferred from them by comparing models to data directly, rather than mediating through a series of fits to selected spectral lines.

Lastly, using a forward model means that uncertainties about other non-stellar parameters, such as flux-calibration or interstellar reddening, can be built into the model and propagated forward. In a future version of Starfish we aim to include a parameterization for the accretion continuum that "veils" the spectra of young T Tauri stars.

1.2.3 Fitting Many Lines at Once

Here is a general example of what can happen when one attempts to fit data with synthetic spectra over a wide spectral range. This is an optical spectrum of WASP-14, an F star hosting a transiting exoplanet.

Fig. 1: A comparison of the data and a typical model fit, along with the corresponding residual spectrum. Notice that this residual spectrum does not look like pure white noise.

Fig. 2: A zoomed view of the gray band in the top panel, highlighting the mildly covariant residual structure that is produced by slight mismatches between the data and model spectra.

Fig. 3: The autocorrelation of the residual spectrum. Notice the substantial autocorrelation signal for offsets of 8 pixels or fewer, demonstrating clearly that the residuals are not well described by white (Poisson) noise alone.

1.2.4 Spectral Line Outliers

Here is a specific example of individual lines that are strongly discrepant from the data. There is substantial localized structure in the residuals due to "outlier" spectral lines in the model library. For any specific line, there might exist a set of model parameters that will improve the match with the data, but there is no single set of model parameters that will properly fit all of the lines at once.

1.2.5 Model the Covariance

In order to account for the covariant residual structure which results from model systematics, we derive a likelihood function with a non-trivial covariance matrix, which maps the covariances between pixels.

$$p(D|M) \propto |\det(C)|^{-1/2} \exp\left(-\frac{1}{2} R^T C^{-1} R\right)$$

We then parameterize this covariance matrix C using Gaussian process covariance kernels. This procedure is demonstrated in the following figure through the following decomposition of how the Gaussian process kernels contribute to the covariance matrix.

top panel: a typical comparison between the data and model spectra, along with the associated residual spectrum. The subsequent rows focus on the illustrative region shaded in gray.

The **left column** of panels shows the corresponding region of the covariance matrix C, decomposed into its primary contributions: (*top row*) the trivial noise matrix using just Poisson errors $\delta_{ij}\sigma_i$, (*middle row*) the trivial matrix combined with a "global" covariance kernel κ^G, and (*bottom row*) these matrices combined with a "local" covariance kernel κ^L to account for an outlier spectral line.

The **right column** of panels shows the zoomed-in residual spectrum with example random draws from the covariance matrix to the left. The shaded contours in orange represent the 1, 2, and 3 sigma dispersions of an ensemble of 200 random draws from the covariance matrix. Note that the trivial covariance matrix (*top row*) poorly reproduces both the scale and structure of the residual spectrum. The addition of a global kernel (*middle row*) more closely approximates the structure and amplitude of the residuals, but misses the outlier line at 5202.5 angstroms. Including a local kernel at that location (*bottom row*) results in a covariance matrix that does an excellent job of reproducing all the key residual features.

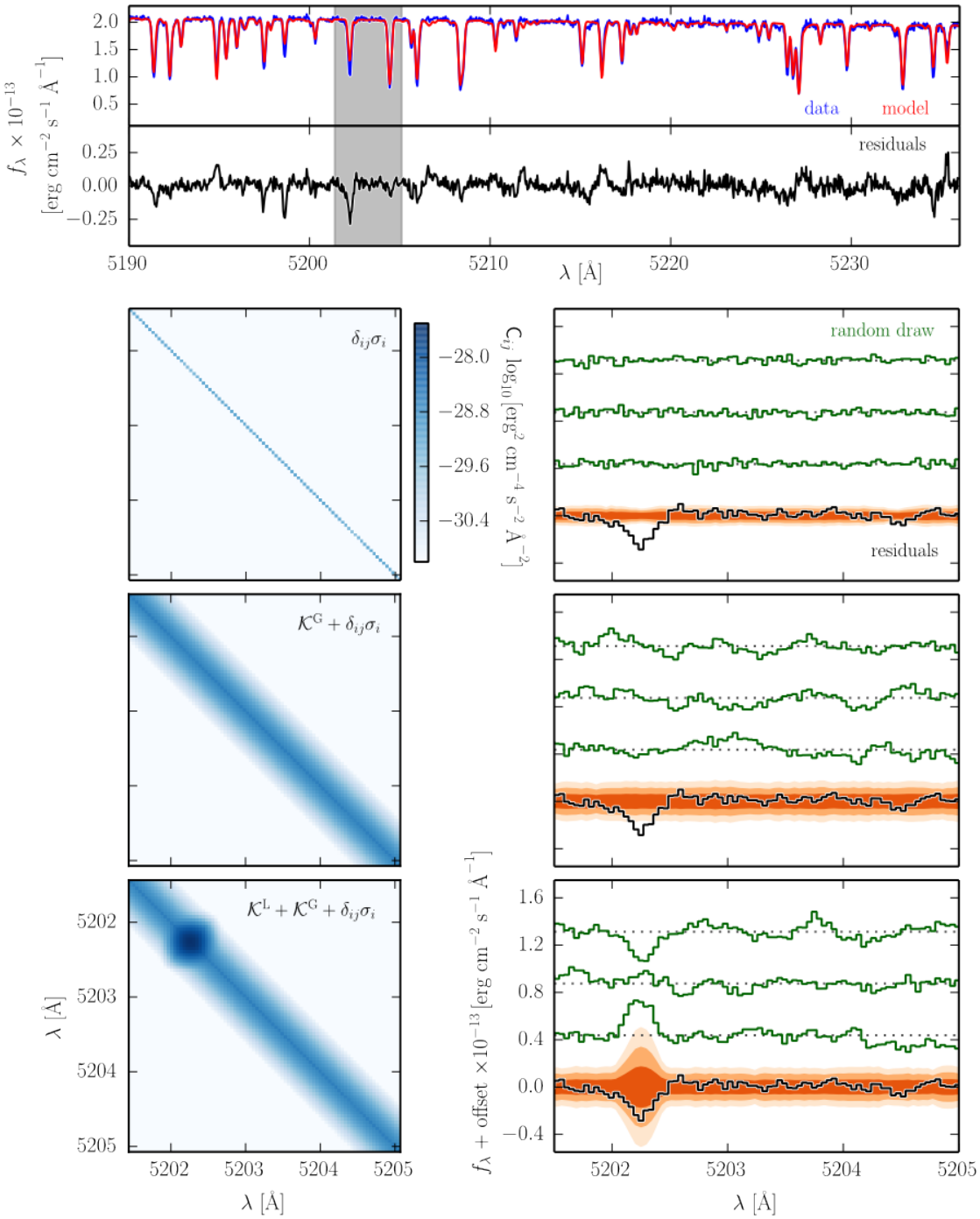
1.2.6 Robust to Outlier Spectral Lines

Starfish uses Markov Chain Monte Carlo (MCMC) to explore the full posterior probability distribution of the stellar parameters, including the noise parameters which describe the covariance of the residuals. By fitting all of the parameters simultaneously, we can be more confident that we have properly accounted for our uncertainty in these other parameters.

top A K-band SPEX spectrum of Gl 51 (an M5 dwarf) fit with a PHOENIX spectroscopic model. While the general agreement of the spectrum is excellent, the strength of the Na and Ca lines is underpredicted (also noted by Rojas-Ayala et al. 2012).

bottom The residual spectrum from this fit along with orange shading contours representing the distributions of a large number of random draws from the covariance matrix (showing 1, 2, and 3 sigma).

Notice how the outlier spectral line features are consistently identified and downweighted by the local covariance kernels. Because the parameters for the local kernels describing the spectral outliers are determined self-consistently along with the stellar parameters, we can be more confident that the influence of these outlier lines on the spectral fit is appropriately downweighted. This weighting approach is in contrast to a more traditional "sigma-clipping" procedure, which would discard these points from the fit. As noted by Mann et al. 2013, some mildly discrepant spectral regions


data
model
residuals
$f_\lambda \times 10^{-13}$ [erg cm^{-2} s^{-1} Å^{-1}]
λ [Å]
$\delta_{ij}\sigma_i$
$\mathcal{K}^{\rm G} + \delta_{ij}\sigma_i$
$\mathcal{K}^{\rm L} + \mathcal{K}^{\rm G} + \delta_{ij}\sigma_i$
C_{ij} log$_{10}$ [erg^2 cm^{-4} s^{-2} Å^{-2}]
random draw
residuals
f_λ + offset $\times 10^{-13}$ [erg cm^{-2} s^{-1} Å^{-1}]

actually contain significant spectral information about the stellar parameters, perhaps more information than spectral regions that are in excellent agreement with the data. Rather than simply discarding these discrepant regions, the appropriate step is then to determine the weighting by which these spectral regions should contribute to the total likelihood. These local kernels provide exactly such a weighting mechanism.

1.2.7 Marginalized Stellar Parameters

The forward modeling approach is unique in that the result is a posterior distribution over stellar parameters. Rather than yielding a simple metric of "best-fit" parameters, exploring the probability distribution with MCMC reveals any covariances between stellar parameters. For this star with the above K-band spectrum, the covariance between T_{eff} and $[Fe/H]$ is mild, but for stars of different spectral types the degeneracy can be severe.

Fig. 4: The posterior probability distribution of the interesting stellar parameters for Gl 51, marginalized over all of nuisance parameters including the covariance kernel hyperparameters. The contours are drawn at 1, 2, and 3 sigma levels for reference.

1.2.8 Spectral Emulator

For spectra with very high signal to noise, interpolation error from the synthetic library may constitute a significant portion of the noise budget. This error is due to the fact that stellar spectral synthesis is an inherently non-linear process requiring complex model atmospheres and radiative transfer. Unfortunately, we are not (yet) in an age where synthetic spectral synthesis over a large spectral range is fast enough to use within a MCMC call. Therefore, it is necessary to approximate an interpolated spectrum based upon spectra with similar stellar properties.

Following the techniques of Habib et al. 2007, we design a spectral emulator, which, rather than interpolating spectra, delivers a probability distribution over all probable interpolate spectra. Using this probability distribution, we can in our likelihood function analytically marginalize over all probable spectral interpolations, in effect forward propagating any uncertainty introduced by the interpolation process.

top The mean spectrum, standard deviation spectrum, and five eigenspectra that form the basis of the PHOENIX synthetic library used to model Gl 51, generated using a subset of the parameter space most relevant for M dwarfs.

bottom The original synthetic spectrum from the PHOENIX library ($T_{eff} = 3000$ K, $logg = 5.0$ dex, $[Fe/H] = 0.0$ dex) compared with a spectrum reconstructed from a linear combination of the derived eigenspectra, using the weights listed in the top panel.

1.3 Conversion from v0.2

There have been some significant changes to *Starfish* in the upgrades to version `0.3.0` and later. Below are some of the main changes, and we also recommend viewing some of the *Examples* to get a hang for the new workflow.

Warning: The current, updated code base does not have the framework for fitting multi-order Echelle spectra. We are working diligently to update the original functionality to match the updated API. For now, you will have to revert to Starfish `0.2.0`.

Note: Was there something in Starfish's utilities you used that was meaningfully removed? Open an issue request and we can work together to find a solution.

1.3.1 API-ification

One of the new goals for *Starfish* was to provide a more Pythonistic approach to its framework. This means instead of using configuration files and scripts the internals for *Starfish* are laid out and leave a lot more flexibility to the end-user without losing the functionality.

There are no more scripts

None of the previous scripts are included in version `0.3.0`. Instead, the functionality of the scripts is enocded into some of the examples, which should allow users a quick way to copy-and-paste their way into a working setup.

Analysis is made easier using other libraries

The previous analysis code for the MCMC chains left us with a decision to make: keep it baked in and locked to an exact MCMC library (*emcee*) or remove it from the project and let other libraries handle it. We chose the latter. Our recommendations for analyzing Bayesian MCMC chains is arviz.

There is no more config.yaml

This file has been eliminated as a byproduct of two endeavors: first is the elimination of the scripts- with a more interactive API in mind, we don't need to hardcode our values in a configuration file. Second is the smoothing of the consistency between the grid tools, the spectral emulator, and the statistical models. For instance, we don't need a configuration value for the grid parameter names because we can leave these as attributes in our GridInterfaces and propagate them upwards through the classes that use the interface.

The modularity has skyrocketed

One of the BIGGEST products of this rewrite is the simplification of the core of what *Starfish* provides: a statistical model for stellar spectra. If you have extra science you want to do, for example: binary star modelling, debris disk modeling, sun spot modeling, etc. we no longer lock down the full maximum likelihood estimation process. Because the new models provide, essentially, transformed stellar models and covariances, if we want to do our own science with the models beyond what *Starfish* already does, we can just plug-and-play! Here is some psuedo-code that exemplifies this behavior:

```
from Starfish.models import SpectrumModel
from Starfish.emulator import Emulator
from astropy.modeling import blackbody

emu = Emulator.load('emu.hdf5')
model = SpectrumModel(..., **initial_parameters)

flux, cov = model()
dust_flux = blackbody(model.data.waves, T_dust)
flux += dust_flux

# Continue with MLE using this composite flux
```

Overall, there are a lot of changes to the workflow for *Starfish*, too. So, again, I highly recommend looking through some *Examples* and browsing through the *API*.

1.3.2 Maintenenance

Clean up

Much of the bloat of the previous repository has been pruned. There still exists archived versions from the GitHub releases, but we've really tried to turn this into a much more professional-looking repository. If there were old files you were using or need to have a copy of, check out the archive.

CI Improvements

The continuous integration has also been improved to help limit the bugs we let through as well as vamp up some of the software development tools that are available to us. You'll see a variety of more pretty badges as well as a much-improved travis-ci matrix that allows us to test on multiple platforms and for multiple python versions

Cleaning up old Issues

Many issues are well outdated and will soon become irrelevant with version `0.3.0`. In an effort to remove some of the clutter we will be closing all issues older than 6 months old or that are solved with the new version. If you had an old issue and feel it was not resolved, feel free to reach out and reopen it so we can work on further improving *Starfish*.

1.4 API

Here you will find the documentation for the api methods and scripts that make up the core of *Starfish*. For even more in-depth reference, you may wish to dig through the source code at GitHub. Make sure you have followed the *installation instructions*

1.4.1 Grid Tools

`grid_tools` is a module to interface with and manipulate libraries of synthetic spectra.

Contents

- *Grid Tools*
  - *Downloading model spectra*
  - *Raw Grid Interfaces*
  - *HDF5 creators and Fast interfaces*
  - *Interpolators*
  - *Instruments*
  - *Utility Functions*

It defines many useful functions and objects that may be used in the modeling package `model`, such as *`Interpolator`*.

Downloading model spectra

Before you may begin any fitting, you must acquire a synthetic library of model spectra. If you will be fitting spectra of stars, there are many high quality synthetic and empirical spectral libraries available. In our paper, we use the freely available PHOENIX library synthesized by T.O. Husser. The library is available for download here: http://phoenix.astro.physik.uni-goettingen.de/. We provide a helper function *download_PHOENIX_models()* if you would prefer to use that.

Because spectral libraries are generally large (> 10 GB), please make sure you available disk space before beginning the download. Downloads may take a day or longer, so it is recommended to start the download ASAP.

You may store the spectra on disk in whatever directory structure you find convenient, provided you adjust the Starfish routines that read spectra from disk. To use the default settings for the PHOENIX grid, please create a `libraries` directory, a `raw` directory within `libraries`, and unpack the spectra in this format:

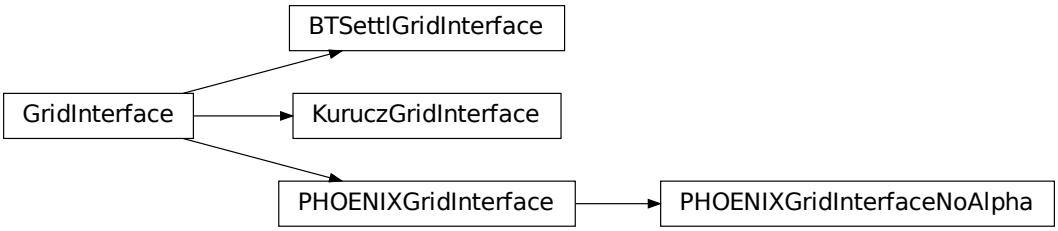
```
libraries/raw/
    PHOENIX/
        WAVE_PHOENIX-ACES-AGSS-COND-2011.fits
        Z+1.0/
        Z-0.0/
        Z-0.0.Alpha=+0.20/
        Z-0.0.Alpha=+0.40/
        Z-0.0.Alpha=+0.60/
        Z-0.0.Alpha=+0.80/
        Z-0.0.Alpha=-0.20/
        Z-0.5/
        Z-0.5.Alpha=+0.20/
        Z-0.5.Alpha=+0.40/
        Z-0.5.Alpha=+0.60/
        Z-0.5.Alpha=+0.80/
        Z-0.5.Alpha=-0.20/
        Z-1.0/
```

Raw Grid Interfaces

Grid interfaces are classes designed to abstract the interaction with the raw synthetic stellar libraries under a common interface. The *GridInterface* class is designed to be extended by the user to provide access to any new grids. Currently there are extensions for three main grids:

1. PHOENIX spectra by T.O. Husser et al 2013 *PHOENIXGridInterface*
2. Kurucz spectra by Laird and Morse (available to CfA internal only) *KuruczGridInterface*
3. PHOENIX BT-Settl spectra by France Allard *BTSettlGridInterface*

There are two interfaces provided to the PHOENIX/Husser grid: one that includes alpha enhancement and another which restricts access to 0 alpha enhancement.



Here and throughout the code, stellar spectra are referenced by a numpy array of parameter values, which corresponds to the parameters listed in the config file.

```
my_params = np.array([6000, 3.5, 0.0, 0.0])
```

Here we introduce the classes and their methods. Below is an example of how you might use the *PHOENIXGridInterface*.

class Starfish.grid_tools.**GridInterface**(*path*, *param_names*, *points*, *wave_units*, *flux_units*, *wl_range=None*, *air=True*, *name=None*)

A base class to handle interfacing with synthetic spectral libraries.

Parameters

- **path** (*str or path-like*) – path to the root of the files on disk.
- **param_names** (*list of str*) – The names of the parameters (dimensions) of the grid
- **points** (*array_like*) – the grid points at which spectra exist (assumes grid is square, not ragged, meaning that every combination of parameters specified exists in the grid).
- **wave_units** (*str*) – The units of the wavelengths. Preferably equivalent to an astropy unit string.
- **flux_units** (*str*) – The units of the model fluxes. Preferable equivalent to an astropy unit string.
- **wl_range** (*list [min, max], optional*) – the starting and ending wavelength ranges of the grid to truncate to. If None, will use whole available grid. Default is None.
- **air** (*bool, optional*) – Are the wavelengths measured in air? Default is True
- **name** (*str, optional*) – name of the spectral library, Default is None

check_params(*parameters*)

Determine if the specified parameters are allowed in the grid.

Parameters **parameters** (*array_like*) – parameter set to check

Raises **ValueError** – if the parameter values are outside of the grid bounds

Returns True if found in grid

Return type bool

load_flux(*parameters*, *header=False*, *norm=True*)

Load the flux and header information.

Parameters

- **parameters** (*array_like*) – stellar parameters
- **header** (*bool, optional*) – If True, will return the header alongside the flux. Default is False.
- **norm** (*bool, optional*) – If True, will normalize the flux to solar luminosity. Default is True.

Raises **ValueError** – if the file cannot be found on disk.

Returns

Return type numpy.ndarray if header is False, tuple of (numpy.ndarray, dict) if header is True

PHOENIX Interfaces

class Starfish.grid_tools.**PHOENIXGridInterface**(*path*, *air=True*, *wl_range=(3000, 54000)*)

Bases: Starfish.grid_tools.base_interfaces.GridInterface

An Interface to the PHOENIX/Husser synthetic library.

Note that the wavelengths in the spectra are in Angstrom and the flux are in F_λ as $erg/s/cm^2/cm$

Parameters

- **path** (*str or path-like*) – The path of the base of the PHOENIX library
- **air** (*bool, optional*) – Whether the wavelengths are measured in air or not. Default is True
- **wl_range** (*tuple, optional*) – The (min, max) of the wavelengths, in . Default is (3000, 54000), which is the full wavelength grid for PHOENIX.

check_params(*parameters*)

Determine if the specified parameters are allowed in the grid.

Parameters **parameters** (*array_like*) – parameter set to check

Raises **ValueError** – if the parameter values are outside of the grid bounds

Returns True if found in grid

Return type bool

load_flux(*parameters*, *header=False*, *norm=True*)

Load the flux and header information.

Parameters

- **parameters** (*array_like*) – stellar parameters
- **header** (*bool, optional*) – If True, will return the header alongside the flux. Default is False.
- **norm** (*bool, optional*) – If True, will normalize the flux to solar luminosity. Default is True.

Raises **ValueError** – if the file cannot be found on disk.

Returns

Return type numpy.ndarray if header is False, tuple of (numpy.ndarray, dict) if header is True

class Starfish.grid_tools.**PHOENIXGridInterfaceNoAlpha**(*path*, ***kwargs*)

Bases: Starfish.grid_tools.interfaces.PHOENIXGridInterface

An Interface to the PHOENIX/Husser synthetic library without any Alpha concentration doping.

Parameters **path** (*str or path-like*) – The path of the base of the PHOENIX library

Keyword Arguments **kwargs** (*dict*) – Any additional arguments will be passed to *PHOENIXGridInterface*'s constructor

See also:

PHOENIXGridInterface

check_params(*parameters*)

Determine if the specified parameters are allowed in the grid.

Parameters **parameters** (*array_like*) – parameter set to check

Raises **ValueError** – if the parameter values are outside of the grid bounds

Returns True if found in grid

Return type bool

In order to load a raw file from the PHOENIX grid, one would do

```
# if you downloaded the libraries elsewhere, be sure to include base="mydir"
import Starfish
from Starfish.grid_tools import PHOENIXGridInterfaceNoAlpha as PHOENIX
import numpy as np
mygrid = PHOENIX()
my_params = np.array([6000, 3.5, 0.0])
flux, hdr = mygrid.load_flux(my_params, header=True)

In [5]: flux
Out[5]:
array([ 4679672.5       ,  4595894.        ,  4203616.5       , ...,
          11033.5625    ,    11301.25585938,    11383.8828125 ], dtype=float32)

In [6]: hdr
Out[6]:
{'PHXDUST': False,
 'PHXLUM': 5.0287e+34,
 'PHXVER': '16.01.00B',
 'PHXREFF': 233350000000.0,
 'PHXEOS': 'ACES',
 'PHXALPHA': 0.0,
 'PHXLOGG': 3.5,
 'PHXTEFF': 6000.0,
 'PHXMASS': 2.5808e+33,
 'PHXXI_N': 1.49,
 'PHXXI_M': 1.49,
 'PHXXI_L': 1.49,
 'PHXMXLEN': 1.48701064748,
 'PHXM_H': 0.0,
 'PHXBUILD': '02/Aug/2010',
 'norm': True,
 'air': True}

In [7]: mygrid.wl
Out[7]:
```

(continues on next page)

(continued from previous page)

```
array([  3000.00133087,   3000.00732938,   3000.01332789, ...,
        53999.27587687,  53999.52580875,  53999.77574063])
```

There is also a provided helper function for downloading PHOENIX models

Starfish.grid_tools.**download_PHOENIX_models**(*path*, *ranges=None*, *parameters=None*)

Download the PHOENIX grid models from the Goettingen servers. This will skip over any ill-defined files or any files that already exist on disk in the given folder.

Parameters

- **path** (*str or path-like*) – The base directory to save the files in.
- **ranges** (*iterable of (min, max), optional*) – Each entry in ranges should be (min, max) for the associated parameter, in the order [Teff, logg, Z, (Alpha)]. Cannot be used with `parameters`. Default is None
- **parameters** (*iterable of iterables of length 3 or length 4, optional*) – The parameters to download. Should be a list of parameters where parameters can either be [Teff, logg, Z] or [Teff, logg, Z, Alpha]. All values should be floats or integers and not string. If no value provided, will download all models. Default is None

Raises **ValueError** – If both `parameters` and `ranges` are specified

Warning: This will create any directories if they do not exist

Warning: Please use this responsibly to avoid over-saturating the connection to the Gottingen servers.

Examples

```
from Starfish.grid_tools import download_PHOENIX_models

ranges = [
    [5000, 5200] # T
    [4.0, 5.0] # logg
    [0, 0] # Z
]
download_PHOENIX_models(path='models', ranges=ranges)
```

or equivalently using `parameters` syntax

```
from itertools import product
from Starfish.grid_tools import download_PHOENIX_models

T = [6000, 6100, 6200]
logg = [4.0, 4.5, 5.0]
Z = [0]
params = product(T, logg, Z)
download_PHOENIX_models(path='models', parameters=params)
```

Other Library Interfaces

class Starfish.grid_tools.**KuruczGridInterface**(*path*, *air=True*, *wl_range=(5000*, *5400)*)
Bases: Starfish.grid_tools.base_interfaces.GridInterface

Kurucz grid interface.

Spectra are stored in f_nu in a filename like t03500g00m25ap00k2v070z1i00.fits, ap00 means zero alpha enhancement, and k2 is the microturbulence, while z1 is the macroturbulence. These particular values are roughly the ones appropriate for the Sun.

static get_wl_kurucz(*filename*)
The Kurucz grid is log-linear spaced.

load_flux(*parameters*, *header=False*, *norm=True*)
Load the flux and header information.

Parameters

- **parameters** (*array_like*) – stellar parameters
- **header** (*bool, optional*) – If True, will return the header alongside the flux. Default is False.
- **norm** (*bool, optional*) – If True, will normalize the flux to solar luminosity. Default is True.

Raises **ValueError** – if the file cannot be found on disk.

Returns

Return type numpy.ndarray if header is False, tuple of (numpy.ndarray, dict) if header is True

class Starfish.grid_tools.**BTSettlGridInterface**(*path*, *air=True*, *wl_range=(2999*, *13000)*)
Bases: Starfish.grid_tools.base_interfaces.GridInterface

BTSettl grid interface. Unlike the PHOENIX and Kurucz grids, the individual files of the BTSettl grid do not always have the same wavelength sampling. Therefore, each call of *load_flux()* will interpolate the flux onto a LogLambda spaced grid that ranges between *wl_range* and has a velocity spacing of 0.08 km/s or better.

If you have a choice, it's probably easier to use the Husser PHOENIX grid.

load_flux(*parameters*, *norm=True*)
Because of the crazy format of the BTSettl, we need to sort the wl to make sure everything is unique, and we're not screwing ourselves with the spline.

Parameters

- **parameters** (*dict*) – stellar parameters
- **norm** (*bool*) – If True, will normalize the spectrum to solar luminosity. Default is True

Creating your own interface

The *GridInterface* and subclasses exist solely to interface with the raw files on disk. At minimum, they should each define a `load_flux()` , which takes in a dictionary of parameters and returns a flux array and a dictionary of whatever information may be contained in the file header.

Under the hood, each of these is implemented differently depending on how the synthetic grid is created. In the case of the BTSettl grid, each file in the grid may actually have a flux array that has been sampled at separate wavelengths. Therefore, it is necessary to actually interpolate each spectrum to a new, common grid, since the wavelength axis of each spectrum is not always the same. Depending on your spectral library, you may need to do something similar.

HDF5 creators and Fast interfaces

While using the *Raw Grid Interfaces* may be useful for ordinary spectral reading, for fast read/write it is best to use HDF5 files to store only the data you need in a hierarchical binary data format. Let's be honest, we don't have all the time in the world to wait around for slow computations that carry around too much data. Before introducing the various ways to compress the spectral library, it might be worthwhile to review the section of the *Spectrum* documentation that discusses how spectra are sampled and resampled in log-linear coordinates.

If we will be fitting a star, there are generally three types of optimizations we can do to the spectral library to speed computation.

1. Use only a range of spectra that span the likely parameter space of your star. For example, if we know we have an F5 star, maybe we will only use spectra that have $5900 \text{ K} \leq T_{\text{eff}} \leq 6500 \text{ K}$.
2. Use only the part of the spectrum that overlaps your instrument's wavelength coverage. For example, if the range of our spectrograph is 4000 - 9000 angstroms, it makes sense to discard the UV and IR portions of the synthetic spectrum.
3. Resample the high resolution spectra to a lower resolution more suitably matched to the resolution of your spectrograph. For example, PHOENIX spectra are provided at $R \sim 500,000$, while the TRES spectrograph has a resolution of $R \sim 44,000$.

All of these reductions can be achieved using the *HDF5Creator* object.

HDF5Creator

class Starfish.grid_tools.**HDF5Creator**(*grid_interface*, *filename*, *instrument=None*, *wl_range=None*, *ranges=None*, *key_name=None*)

Create a HDF5 grid to store all of the spectra from a RawGridInterface, along with metadata.

Parameters

- **grid_interface** (*GridInterface*) – The raw grid interface to process while creating the HDF5 file
- **filename** (*str or path-like*) – Where to save the HDF5 file
- **instrument** (*Instrument*, optional) – If provided, the instrument to convolve/truncate the grid. If None, will maintain the grid's original wavelengths and resolution. Default is None
- **wl_range** (*list [min, max], optional*) – The wavelength range to truncate the grid to. Will be truncated to match grid wavelengths and instrument wavelengths if over or under specified. If set to None, will not truncate grid. Default is NOne

- **ranges** (*array_like, optional*) – lower and upper limits for each stellar parameter, in order to truncate the number of spectra in the grid. If None, will not restrict the range of the parameters. Default is None.
- **key_name** (*format str*) – formatting string that has keys for each of the parameter names to translate into a hash-able string. If set to None, will create a name by taking each parameter name followed by value with underscores delimiting parameters. Default is None.

Raises **ValueError** – if the wl_range is ill-specified or if the parameter range are completely disjoint from the grid points.

process_grid()
Run `process_flux()` for all of the spectra within the *ranges* and store the processed spectra in the HDF5 file.

Here is an example using the *HDF5Creator* to transform the raw spectral library into an HDF5 file with spectra that have the resolution of the *TRES* instrument. This process is also located in the `scripts/grid.py` if you are using the cookbook.

```
import Starfish
from Starfish.grid_tools import PHOENIXGridInterfaceNoAlpha as PHOENIX
from Starfish.grid_tools import HDF5Creator, TRES


mygrid = PHOENIX()
instrument = TRES()

creator = HDF5Creator(mygrid, instrument)
creator.process_grid()
```

HDF5Interface

Once you've made a grid, then you'll want to interface with it via *HDF5Interface*. The *HDF5Interface* provides `load_flux()` similar to that of the raw grid interfaces. It does not make any assumptions about how what resolution the spectra are stored, other than that the all spectra within the same HDF5 file share the same wavelength grid, which is stored in the HDF5 file as 'wl'. The flux files are stored within the HDF5 file, in a subfile called 'flux'.

class Starfish.grid_tools.**HDF5Interface**(*filename*)
Connect to an HDF5 file that stores spectra.

Parameters **filename** (*str or path-like*) – The path of the saved HDF5 file

property fluxes
Iterator to loop over all of the spectra stored in the grid, for PCA. Loops over parameters in the order specified by grid_points.

Returns

Return type Generator of numpy.ndarrays

load_flux(*parameters*, *header=False*)
Load just the flux from the grid, with possibly an index truncation.

parameters [array_like] the stellar parameters

header [bool, optional] If True, will return the header as well as the flux. Default is False

Returns

Return type numpy.ndarray if header is False, otherwise (numpy.ndarray, dict)

For example, to load a file from our recently-created HDF5 grid

```
import Starfish
from Starfish.grid_tools import HDF5Interface
import numpy as np

# Assumes you have already created and HDF5 grid
myHDF5 = HDF5Interface()
flux = myHDF5.load_flux(np.array([6100, 4.5, 0.0]))

In [4]: flux
Out[4]:
array([ 10249189.,  10543461.,  10742093., ...,   9639472.,   9868226.,
    10169717.], dtype=float32)
```

Interpolators

The interpolators are used to create spectra in between grid points, for example `[6114, 4.34, 0.12, 0.1]`.

class Starfish.grid_tools.**Interpolator**(*interface*, *wl_range=(0, inf)*, *cache_max=256*, *cache_dump=64*)

Quickly and efficiently interpolate a synthetic spectrum for use in an MCMC simulation. Caches spectra for easier memory load.

Parameters

- **interface** (*HDF5Interface* (recommended) or RawGridInterface) – The interface to the spectra
- **wl_range** (*tuple (min, max)*) – If provided, the data wavelength range of the region you are trying to fit. Used to truncate the grid for speed. Default is (0, np.inf)
- **cache_max** (*int*) – maximum number of spectra to hold in cache
- **cache_dump** (*int*) – how many spectra to purge from the cache once cache_max is reached

Warning: Interpolation causes degradation of information of the model spectra without properly forward propagating the errors from interpolation. We highly recommend using the *Spectral Emulator*

__call__(*parameters*)

Interpolate a spectrum

Parameters **parameters** (*numpy.ndarray or list*) – stellar parameters

Note: Automatically pops cache_dump items from cache if full.

interpolate(*parameters*)

Interpolate a spectrum without clearing cache. Recommended to use *__call__()* instead to take advantage of caching.

Parameters **parameters** (*numpy.ndarray or list*) – grid parameters

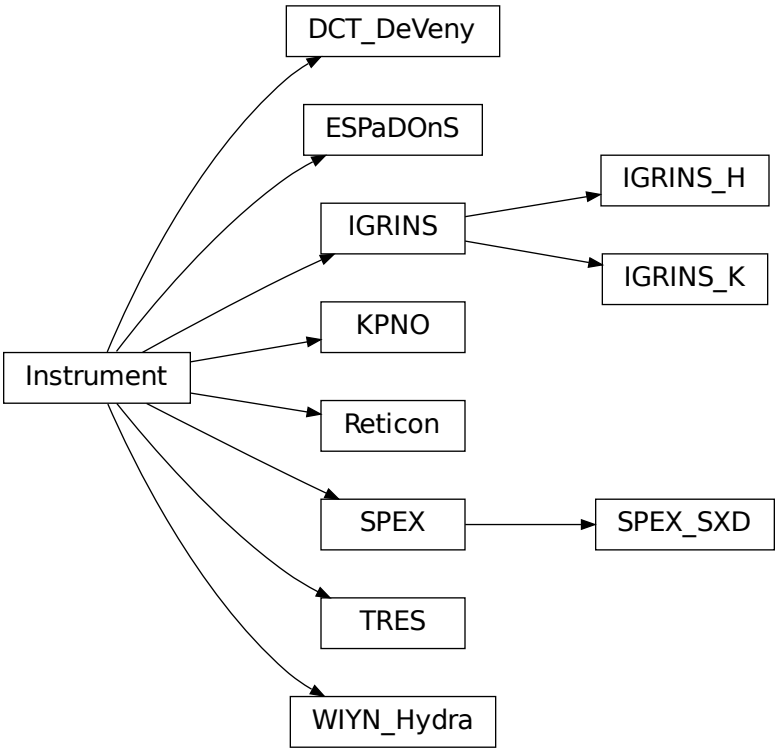
Raises **ValueError** – if parameters are out of bounds.

For example, if we would like to generate a spectrum with the aforementioned parameters, we would do

```
myInterpolator = Interpolator(myHDF5)
spec = myInterpolator([6114, 4.34, 0.12, 0.1])
```

Instruments

In order to take the theoretical synthetic stellar spectra and make meaningful comparisons to actual data, we need to convolve and resample the synthetic spectra to match the format of our data. *Instrument* s are a convenience object which store the relevant characteristics of a given instrument.



class Starfish.grid_tools.**Instrument**(*name: str, FWHM: float, wl_range: Tuple[float], oversampling: float = 4.0*)

Object describing an instrument. This will be used by other methods for processing raw synthetic spectra.

Parameters

- **name** (*string*) – name of the instrument
- **FWHM** (*float*) – the FWHM of the instrumental profile in km/s
- **wl_range** (*Tuple (low, high)*) – wavelength range of instrument
- **oversampling** (*float, optional*) – how many samples fit across the FWHM. Default is 4.0

__str__()
Prints the relevant properties of the instrument.

List of Instruments

It is quite easy to use the *Instrument* class for your own data, but we provide classes for most of the well-known spectrographs. If you have a spectrograph that you would like to add if you think it will be used by others, feel free to open a pull request following the same format.

class Starfish.grid_tools.**TRES**(*name='TRES'*, *FWHM=6.8*, *wl_range=(3500*, *9500)*)
Bases: Starfish.grid_tools.instruments.Instrument

TRES instrument

class Starfish.grid_tools.**KPNO**(*name='KPNO'*, *FWHM=14.4*, *wl_range=(6250*, *6650)*)
Bases: Starfish.grid_tools.instruments.Instrument

KNPO instrument

class Starfish.grid_tools.**Reticon**(*name='Reticon'*, *FWHM=8.5*, *wl_range=(5145*, *5250)*)
Bases: Starfish.grid_tools.instruments.Instrument

Reticon instrument

class Starfish.grid_tools.**SPEX**(*name='SPEX'*, *FWHM=150.0*, *wl_range=(7500*, *54000)*)
Bases: Starfish.grid_tools.instruments.Instrument

SPEX instrument at IRTF in Hawaii

class Starfish.grid_tools.**SPEX_SXD**(*name='SPEX_SXD'*)
Bases: Starfish.grid_tools.instruments.SPEX

SPEX instrument at IRTF in Hawaii short mode (reduced wavelength range)

class Starfish.grid_tools.**IGRINS_H**(*name='IGRINS_H'*, *wl_range=(14250*, *18400)*)
Bases: Starfish.grid_tools.instruments.IGRINS

IGRINS H band instrument

class Starfish.grid_tools.**IGRINS_K**(*name='IGRINS_K'*, *wl_range=(18500*, *25200)*)
Bases: Starfish.grid_tools.instruments.IGRINS

IGRINS K band instrument

class Starfish.grid_tools.**ESPaDOnS**(*name='ESPaDOnS'*, *FWHM=4.4*, *wl_range=(3700*, *10500)*)
Bases: Starfish.grid_tools.instruments.Instrument

ESPaDOnS instrument

class Starfish.grid_tools.**DCT_DeVeny**(*name='DCT_DeVeny'*, *FWHM=105.2*, *wl_range=(6000*, *10000)*)
Bases: Starfish.grid_tools.instruments.Instrument

DCT DeVeny spectrograph instrument.

class Starfish.grid_tools.**WIYN_Hydra**(*name='WIYN_Hydra'*, *FWHM=300.0*, *wl_range=(5500*, *10500)*)
Bases: Starfish.grid_tools.instruments.Instrument

WIYN Hydra spectrograph instrument.

Utility Functions

Starfish.grid_tools.**chunk_list**(*mylist*, *n=2*)
Divide a lengthy parameter list into chunks for parallel processing and backfill if necessary.

Parameters

- **mylist** (*1-D list*) – a lengthy list of parameter combinations
- **n** (*integer*) – number of chunks to divide list into. Default is `mp.cpu_count()`

Returns **chunks** (*2-D list* of shape (n, -1)) a list of chunked parameter lists.

Starfish.grid_tools.**determine_chunk_log**(*wl*, *wl_min*, *wl_max*)
Take in a wavelength array and then, given two minimum bounds, determine the boolean indices that will allow us to truncate this grid to near the requested bounds while forcing the wl length to be a power of 2.

Parameters

- **wl** (*np.ndarray*) – wavelength array
- **wl_min** (*float*) – minimum required wavelength
- **wl_max** (*float*) – maximum required wavelength

Returns numpy.ndarray boolean array

Wavelength conversions

Starfish.grid_tools.**vacuum_to_air**(*wl*)
Converts vacuum wavelengths to air wavelengths using the Ciddor 1996 formula.

Parameters **wl** (*numpy.ndarray*) – input vacuum wavelengths

Returns numpy.ndarray

Note: CA Prieto recommends this as more accurate than the IAU standard.

Starfish.grid_tools.**air_to_vacuum**(*wl*)
Convert air wavelengths to vacuum wavelengths.

Parameters **wl** (*np.array*) – input air wavelegths

Returns numpy.ndarray

Warning: It is generally not recommended to do this, as the function is imprecise.

Starfish.grid_tools.**calculate_n**(*wl*)
Calculate *n*, the refractive index of light at a given wavelength.

Parameters **wl** (*np.array*) – input wavelength (in vacuum)

Returns numpy.ndarray

1.4.2 Spectral Emulator

The spectral emulator can be likened to the engine behind *Starfish*. While the novelty of *Starfish* comes from using Gaussian processes to model and account for the covariances of spectral fits, we still need a way to produce model spectra by interpolating from our synthetic library. While we could interpolate spectra from the synthetic library using something like linear interpolation in each of the library parameters, it turns out that high signal-to-noise data requires something more sophisticated. This is because the error in any interpolation can constitute a significant portion of the error budget. This means that there is a chance that non-interpolated spectra (e.g., the parameters of the synthetic spectra in the library) might be given preference over any other interpolated spectra, and the posteriors will be peaked at the grid point locations. Because the spectral emulator returns a probability distribution over possible interpolated spectra, this interpolation error can be quantified and propagated forward into the likelihood calculation.

Eigenspectra decomposition

The first step of configuring the spectral emulator is to choose a subregion of the spectral library corresponding to the star that you will fit. Then, we want to decompose the information content in this subset of the spectral library into several *eigenspectra*. [Figure A.1 here].

The eigenspectra decomposition is performed via Principal Component Analysis (PCA). Thankfully, most of the heavy lifting is already implemented by the `sklearn` package.

`Emulator.from_grid()` allows easy creation of spectral emulators from an *`Starfish.grid_tools.HDF5Interface`*, which includes doing the initial PCA to create the eigenspectra.

```
>>> from Starfish.grid_tools import HDF5Interface
>>> from Starfish.emulator import Emulator
>>> emulator = Emulator.from_grid(HDF5Interface('grid.hdf5'))
```

Optimizing the emulator

Once the synthetic library is decomposed into a set of eigenspectra, the next step is to train the Gaussian Processes (GP) that will serve as interpolators. For more explanation about the choice of Gaussian Process covariance functions and the design of the emulator, see the appendix of our paper.

The optimization of the GP hyperparameters can be carried out by any maximum likelihood estimation framework, but we include a direct method that uses `scipy.optimize.minimize`.

To optimize the code, we will use the *`Emulator.train()`* routine.

Example optimizing using minimization optimizer

```
>>> from Starfish.grid_tools import HDF5Interface
>>> from Starfish.emulator import Emulator
>>> emulator = Emulator.from_grid(HDF5Interface('grid.hdf5'))
>>> emulator
Emulator
--------
Trained: False
lambda_xi: 2.718
Variances:
    10000.00
    10000.00
    10000.00
    10000.00
Lengthscales:
    [ 600.00  1.50  1.50 ]
```

(continues on next page)

(continued from previous page)

```
    [ 600.00  1.50  1.50 ]
    [ 600.00  1.50  1.50 ]
    [ 600.00  1.50  1.50 ]
Log Likelihood: -1412.00
>>> emulator.train()
>>> emulator
Emulator
--------
Trained: True
lambda_xi: 2.722
Variances:
    238363.85
    5618.02
    9358.09
    2853.22
Lengthscales:
    [ 1582.39  3.19  3.11 ]
    [ 730.81  1.61  2.14 ]
    [ 1239.45  3.71  2.78 ]
    [ 1127.40  1.63  4.46 ]
Log Likelihood: -1158.83
>>> emulator.save('trained_emulator.hdf5')
```

Note: The built in optimization target changes the state of the emulator, so even if the output of the minimizer has not converged, you can simply run *Emulator.train()* again.

If you want to perform MLE with a different method, feel free to make use of the general modeling framework provided by the function *Emulator.get_param_vector()*, *Emulator.set_param_vector()*, and *Emulator.log_likelihood()*.

Model spectrum reconstruction

Once the emulator has been optimized, we can finally use it as a means of interpolating spectra.

```
>>> from Starfish.emulator import Emulator
>>> emulator = Emulator.load('trained_emulator.hdf5')
>>> flux = emulator.load_flux([7054, 4.0324, 0.01])
>>> wl = emu.wl
```

If you want to take advantage of the emulator covariance matrix, you must use the interface via the *Emulator.__call__()* function

```
>>> from Starfish.emulator import Emulator
>>> emulator = Emulator.load('trained_emulator.hdf5')
>>> weights, cov = emulator([7054, 4.0324, 0.01])
>>> X = emulator.eigenspectra * emulator.flux_std
>>> flux = weights @ X + emulator.flux_mean
>>> emu_cov = X.T @ weights @ X
```

Lastly, if you want to process the model, it is useful to process the eigenspectra before reconstructing, especially if a resampling action has to occur. The *Emulator* provides the attribute *Emulator.bulk_fluxes* for such processing. For example

```
>>> from Starfish.emulator import Emulator
>>> from Starfish.transforms import instrumental_broaden
>>> emulator = Emulator.load('trained_emulator.hdf5')
>>> fluxes = emulator.bulk_fluxes
>>> fluxes = instrumental_broaden(emulator.wl, fluxes, 10)
>>> eigs = fluxes[:-2]
>>> flux_mean, flux_std = fluxes[-2:]
>>> weights, cov = emulator([7054, 4.0324, 0.01])
>>> X = emulator.eigenspectra * flux_std
>>> flux = weights @ X + flux_mean
>>> emu_cov = X.T @ weights @ X
```

Note: *Emulator.bulk_fluxes* provides a copy of the underlying arrays, so there is no change to the emulator when bulk processing.

Reference

Emulator

class Starfish.emulator.**Emulator**(*grid_points: nptyping.types._ndarray.NDArray, param_names: Sequence[str], wavelength: nptyping.types._ndarray.NDArray, weights: nptyping.types._ndarray.NDArray, eigenspectra: nptyping.types._ndarray.NDArray, w_hat: nptyping.types._ndarray.NDArray, flux_mean: nptyping.types._ndarray.NDArray, flux_std: nptyping.types._ndarray.NDArray, factors: nptyping.types._ndarray.NDArray, lambda_xi: float = 1.0, variances: Optional[nptyping.types._ndarray.NDArray] = None, lengthscales: Optional[nptyping.types._ndarray.NDArray] = None, name: Optional[str] = None*)

A Bayesian spectral emulator.

This emulator offers an interface to spectral libraries that offers interpolation while providing a variance-covariance matrix that can be forward-propagated in likelihood calculations. For more details, see the appendix from Czekala et al. (2015).

Parameters

- **grid_points** (*numpy.ndarray*) – The parameter space from the library.
- **param_names** (*array-like of str*) – The names of each parameter from the grid
- **wavelength** (*numpy.ndarray*) – The wavelength of the library models
- **weights** (*numpy.ndarray*) – The PCA weights for the original grid points
- **eigenspectra** (*numpy.ndarray*) – The PCA components from the decomposition
- **w_hat** (*numpy.ndarray*) – The best-fit weights estimator
- **flux_mean** (*numpy.ndarray*) – The mean flux spectrum
- **flux_std** (*numpy.ndarray*) – The standard deviation flux spectrum
- **lambda_xi** (*float, optional*) – The scaling parameter for the augmented covariance calculations, default is 1

- **variances** (*numpy.ndarray, optional*) – The variance parameters for each of Gaussian process, default is array of 1s
- **lengthscales** (*numpy.ndarray, optional*) – The lengthscales for each Gaussian process, each row should have length equal to number of library parameters, default is arrays of 3 * the max grid separation for the grid_points
- **name** (*str, optional*) – If provided, will give a name to the emulator; useful for keeping track of filenames. Default is None.

params
The underlying hyperparameter dictionary

Type dict

__call__ (*params: Sequence[float], full_cov: bool = True, reinterpret_batch: bool = False*) → Tuple[nptyping.types._ndarray.NDArray, nptyping.types._ndarray.NDArray]
Gets the mu and cov matrix for a given set of params

Parameters

- **params** (*array_like*) – The parameters to sample at. Should be consistent with the shapes of the original grid points.
- **full_cov** (*bool, optional*) – Return the full covariance or just the variance, default is True. This will have no effect of reinterpret_batch is true
- **reinterpret_batch** (*bool, optional*) – Will try and return a batch of output matrices if the input params are a list of params, default is False.

Returns

- **mu** (*numpy.ndarray (len(params),)*)
- **cov** (*numpy.ndarray (len(params), len(params))*)

Raises

- **ValueError** – If full_cov and reinterpret_batch are True
- **ValueError** – If querying the emulator outside of its trained grid points

property bulk_fluxes
A vertically concatenated vector of the eigenspectra, flux_mean, and flux_std (in that order). Used for bulk processing with the emulator.

Type numpy.ndarray

determine_chunk_log (*wavelength: Sequence[float], buffer: float = 50*)
Possibly truncate the wavelength and eigenspectra in response to some new wavelengths

Parameters

- **wavelength** (*array_like*) – The new wavelengths to truncate to
- **buffer** (*float, optional*) – The wavelength buffer, in Angstrom. Default is 50

See also:

`Starfish.grid_tools.utils.determine_chunk_log()`

classmethod from_grid (*grid*, ***pca_kwargs*)
Create an Emulator using PCA decomposition from a GridInterface.

Parameters

- **grid** (`GridInterface` or str) – The grid interface to decompose

- **pca_kwargs** (*dict, optional*) – The keyword arguments to pass to PCA. By default, *n_components=0.99* and *svd_solver='full'*.

See also:

`sklearn.decomposition.PCA()`

get_index(*params: Sequence[float]*) → int
Given a list of stellar parameters (corresponding to a grid point), deliver the index that corresponds to the entry in the fluxes, grid_points, and weights.

Parameters **params** (*array_like*) – The stellar parameters

Returns **index**

Return type int

get_param_dict() → dict
Gets the dictionary of parameters. This is the same as *Emulator.params*

Returns

Return type dict

get_param_vector() → nptyping.types._ndarray.NDArray
Get a vector of the current trainable parameters of the emulator

Returns

Return type numpy.ndarray

property lambda_xi
The tuning hyperparameter

Setter Sets the value.

Type float

property lengthscales
The lengthscales for each Gaussian process kernel.

Setter Sets the lengthscales given a 2d array

Type numpy.ndarray

classmethod load(*filename: Union[str, os.PathLike]*)
Load an emulator from and HDF5 file

Parameters **filename** (*str or path-like*) –

load_flux(*params: Union[Sequence[float], nptyping.types._ndarray.NDArray], norm=False*) → nptyping.types._ndarray.NDArray
Interpolate a model given any parameters within the grid's parameter range using eigenspectrum reconstruction by sampling from the weight distributions.

Parameters **params** (*array_like*) – The parameters to sample at.

Returns **flux**

Return type numpy.ndarray

log_likelihood() → float
Get the log likelihood of the emulator in its current state as calculated in the appendix of Czekala et al. (2015)

Returns

Return type float

Raises **scipy.linalg.LinAlgError** – If the Cholesky factorization fails

norm_factor(*params: Union[Sequence[float], nptyping.types._ndarray.NDArray]*) → float
Return the scaling factor for the absolute flux units in flux-normalized spectra

Parameters **params** (*array_like*) – The parameters to interpolate at

Returns **factor** – The multiplicative factor to normalize a spectrum to the model's absolute flux units

Return type float

save(*filename: Union[str, os.PathLike]*)
Save the emulator to an HDF5 file

Parameters **filename** (*str or path-like*) –

set_param_dict(*params: dict*)
Sets the parameters with a dictionary

Parameters **params** (*dict*) – The new parameters.

set_param_vector(*params: nptyping.types._ndarray.NDArray*)
Set the current trainable parameters given a vector. Must have the same form as *get_param_vector()*

Parameters **params** (*numpy.ndarray*) –

train(**opt_kwargs*)
Trains the emulator's hyperparameters using gradient descent. This is a light wrapper around *scipy.optimize.minimize*. If you are experiencing problems optimizing the emulator, consider implementing your own training loop, using this function as a template.

Parameters ****opt_kwargs** – Any arguments to pass to the optimizer. By default, *method='Nelder-Mead'* and *maxiter=10000*.

See also:

scipy.optimize.minimize()

property variances
The variances for each Gaussian process kernel.

Setter Sets the variances given an array.

Type numpy.ndarray

1.4.3 Spectrum

This module contains a few different routines for the manipulation of spectra.

Log lambda spacing

Throughout *Starfish*, we try to utilize log-lambda spaced spectra whenever possible. This is because this sampling preserves the Doppler content of the spectrum at the lowest possible sampling. A spectrum spaced linear in log lambda has equal-velocity pixels, meaning that

$$\frac{v}{c} = \frac{\Delta\lambda}{\lambda}$$

A log lambda spectrum is defined by the WCS keywords **CDELT1**, **CRVAL1**, and **NAXIS1**. They are related to the physical wavelengths by the following relationship

$$\lambda = 10^{\mathrm{CRVAL1} + \mathrm{CDELT1} \cdot i}$$

where i is the pixel index, with $i = 0$ referring to the first pixel and $i = (\mathrm{NAXIS1} - 1)$ referring to the last pixel.

The wavelength array and header keywords are often stored in a `wl_dict` dictionary, which looks like `{"wl":wl, "CRVAL1":CRVAL1, "CDELT1":CDELT1, "NAXIS1":NAXIS1}`.

These keywords are related to various wavelengths by

$$\frac{v}{c} = \frac{\Delta\lambda}{\lambda} = 10^{\mathrm{CDELT1}} - 1$$

$$\mathrm{CDELT1} = \log_{10}\left(\frac{v}{c} + 1\right)$$

$$\mathrm{CRVAL1} = \log_{10}(\lambda_{\mathrm{start}})$$

Many spectral routines utilize a keyword `dv`, which stands for Δv, or the velocity difference (measured in km/s) that corresponds to the width of one pixel.

$$\mathrm{dv} = c\frac{\Delta\lambda}{\lambda}$$

When resampling wavelength grids that are not log-lambda spaced (e.g., the raw synthetic spectrum from the library) onto a log-lambda grid, the `dv` must be calculated. Generally, `calculate_dv()` works by measuring the velocity difference of every pixel and choosing the smallest, that way no spectral information will be lost.

`Starfish.utils.`**`calculate_dv`**(*wave: Sequence*)

Given a wavelength array, calculate the minimum `dv` of the array.

Parameters **`wave`** (*`array-like`*) – The wavelength array

Returns delta-v in units of km/s

Return type float

`Starfish.utils.`**`create_log_lam_grid`**(*dv*, *start*, *end*)

Create a log lambda spaced grid with `N_points` equal to a power of 2 for ease of FFT.

Parameters

- **`dv`** (*`float`*) – Upper bound on the velocity spacing in km/s
- **`start`** (*`float`*) – starting wavelength (inclusive) in Angstrom
- **`end`** (*`float`*) – ending wavelength (inclusive) in Angstrom

Returns a wavelength dictionary containing the specified properties. Note that the returned dv will be less than or equal to the specified dv.

Return type dict

Raises

- **`ValueError`** – If starting wavelength is not less than ending wavelength
- **`ValueError`** – If any of the wavelengths are less than 0

`Starfish.utils.`**`calculate_dv_dict`**(*wave_dict*)

Given a `wave_dict`, calculate the velocity spacing.

Parameters **`wave_dict`** (*`dict`*) – wavelength dictionary

Returns delta-v in units of km/s

Return type float

Order

We organize our data into orders which are the building blocks of Echelle spectra. Each order has its own wavelength, flux, optional flux error, and optional mask.

Note: Typically, you will not be creating orders directly, but rather will be using them as part of a *Spectrum* object.

The way you interact with orders is generally using the properties `wave`, `flux`, and `sigma`, which will automatically apply the order's mask. If you want to reach the underlying arrays, say to create a new mask, use the appropriate _-prepended properties.

```
>>> order = Order(...)
>>> len(order)
3450
>>> new_mask = order.mask & (order._wave > 0.9e4) & (order._wave < 4.4e4)
>>> order.mask = new_mask
>>> len(order)
2752
```

API/Reference

class Starfish.spectrum.**Order**(*_wave: nptyping.types._ndarray.NDArray*, *_flux: nptyping.types._ndarray.NDArray*, *_sigma: Optional[nptyping.types._ndarray.NDArray] = None*, *mask: Optional[nptyping.types._ndarray.NDArray] = None*)

A data class to hold astronomical spectra orders.

Parameters

- **_wave** (*numpy.ndarray*) – The full wavelength array
- **_flux** (*numpy.ndarray*) – The full flux array
- **_sigma** (*numpy.ndarray, optional*) – The full sigma array. If None, will default to all 0s. Default is None
- **mask** (*numpy.ndarray, optional*) – The full mask. If None, will default to all Trues. Default is None

name

Type str

__len__()

property flux

The masked flux array

Type numpy.ndarray

mask: Optional[nptyping.types._ndarray.NDArray] = None

property sigma

The masked flux uncertainty array

Type numpy.ndarray

property wave

The masked wavelength array

Type numpy.ndarray

Spectrum

A *Spectrum* holds the many orders that make up your data. These orders, described by *Order*, are treated as rows in a two-dimensional array. We like to store these spectra in HDF5 files so we recommend creating a pre-processing method that may require any additional dependencies (e.g., *IRAF*) for getting your data into 2-d wavelength arrays calibrated to the same flux units as your spectral library models.

```
>>> waves, fluxes, sigmas = process_data("data.fits")
>>> data = Spectrum(waves, fluxes, sigmas, name="Data")
>>> data.save("data.hdf5")
```

Our HDF5 format is simple, with each dataset having shape (norders, npixels):

```
/
  waves
  fluxes
  sigmas
  masks
```

Whether you save your data to hdf5 or have an external process that saves into the same format above, you can then load the spectrum using

```
>>> data = Spectrum.load("data.hdf5")
```

When using HDF5 files, we highly recommended using a GUI program like HDF View to make it easer to see what's going on.

To access the data, you can either access the full 2-d data arrays (which will have the appropriate mask applied) or iterate order-by-order

```
>>> data = Spectrum(...)
>>> len(data)
4
>>> data.waves.shape
(4, 2752)
>>> num_points = 0
>>> for order in data:
...   num_points += len(order)
>>> num_points == np.prod(data.shape)
True
```

API/Reference

class Starfish.spectrum.**Spectrum**(*waves*, *fluxes*, *sigmas=None*, *masks=None*, *name='Spectrum'*)

Object to store astronomical spectra.

Parameters

- **waves** (*1D or 2D array-like*) – wavelength in Angtsrom
- **fluxes** (*1D or 2D array-like*) – flux (in f_lam)
- **sigmas** (*1D or 2D array-like, optional*) – Poisson noise (in f_lam). If not specified, will be zeros. Default is None

- **masks** (*1D or 2D array-like, optional*) – Mask to blot out bad pixels or emission regions. Must be castable to boolean. If None, will create a mask of all True. Default is None
- **name** (*str, optional*) – The name of this spectrum. Default is "Spectrum"

Note: If the waves, fluxes, and sigmas are provided as 1D arrays (say for a single order), they will be converted to 2D arrays with length 1 in the 0-axis.

Warning: For now, the Spectrum waves, fluxes, sigmas, and masks must be a rectangular grid. No ragged Echelle orders allowed.

name
The name of the spectrum

Type str

__getitem__(*index: int*)

__len__()

__setitem__(*index: int*, *order: Starfish.spectrum.Order*)

property fluxes
The 2 dimensional masked flux arrays

Type numpy.ndarray

classmethod load(*filename*)
Load a spectrum from an hdf5 file

Parameters **filename** (*str or path-like*) – The path to the HDF5 file.

See also:

save()

property masks
The full 2-dimensional boolean masks

Type np.ndarray

plot(*ax=None*, ***kwargs*)
Plot all the orders of the spectrum

Parameters **ax** (*matplotlib.Axes, optional*) – If provided, will plot on this axis. Otherwise, will create a new axis, by default None

Returns The axis that was plotted on

Return type matplotlib.Axes

reshape(*shape*)
Reshape the spectrum to the new shape. Obeys the same rules that numpy reshaping does. Note this is not done in-place.

Parameters **shape** (*tuple*) – The new shape of the spectrum. Must abide by numpy reshaping rules.

Returns The reshaped spectrum

Return type *Spectrum*

save (*filename*)
Takes the current DataSpectrum and writes it to an HDF5 file.

Parameters **filename** (*str or path-like*) – The filename to write to. Will not create any missing directories.

See also:

load()

property shape
The shape of the spectrum, *(norders, npixels)*

Setter Tries to reshape the data into a new arrangement of orders and pixels following numpy reshaping rules.

Type numpy.ndarray

property sigmas
The 2 dimensional masked flux uncertainty arrays

Type numpy.ndarray

property waves
The 2 dimensional masked wavelength arrays

Type numpy.ndarray

1.4.4 Transforms

These classes and functions are used to manipulate stellar spectra. Users are not expected to directly call these methods unless they are playing around with spectrums or creating custom methods.

Starfish.transforms.**resample** (*wave*, *flux*, *new_wave*)
Resample onto a new wavelength grid using k=5 spline interpolation

Parameters

- **wave** (*array_like*) – The original wavelength grid
- **flux** (*array_like*) – The fluxes to resample
- **new_wave** (*array_like*) – The new wavelength grid

Raises **ValueError** – If the new wavelength grid is not strictly increasing monotonic

Returns The resampled flux with the same 1st dimension as the input flux

Return type numpy.ndarray

Starfish.transforms.**rescale** (*flux*, *scale*)
Rescale the given flux via the following equation

$$f \cdot \Omega$$

Parameters

- **flux** (*array_like*) – The input fluxes
- **scale** (*float or array_like*) – The scaling factor. If an array, must have same shape as the batch dimension of `flux`

Returns The rescaled fluxes with the same shape as the input fluxes

Return type numpy.ndarray

Starfish.transforms.**renorm**(*wave*, *flux*, *reference_flux*)

Renormalize one spectrum to another

This uses the *rescale()* function with a `log_scale` of

$$\log \Omega = \int f^*(w)dw \Big/ \int f(w)dw$$

where f^* is the reference flux, f is the source flux, and the integrals are over a common wavelength grid

Parameters

- **wave** (*array_like*) – The wavelength grid for the source flux
- **flux** (*array_like*) – The flux for the source
- **reference_flux** (*array_like*) – The reference source to renormalize to

Returns The renormalized flux

Return type numpy.ndarray

Starfish.transforms.**doppler_shift**(*wave*, *vz*)

Doppler shift a spectrum according to the formula

$$\lambda \cdot \sqrt{\frac{c + v_z}{c - v_z}}$$

Parameters

- **wave** (*array_like*) – The unshifted wavelengths
- **vz** (*float*) – The doppler velocity in km/s

Returns Altered wavelengths with the same shape as the input wavelengths

Return type numpy.ndarray

Starfish.transforms.**instrumental_broaden**(*wave*, *flux*, *fwhm*)

Broadens given flux by convolving with a Gaussian kernel appropriate for a spectrograph's instrumental properties. Follows the given equation

$$f = f * \mathcal{F}_v^{\text{inst}}$$

$$\mathcal{F}_v^{\text{inst}} = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{1}{2}\left(\frac{v}{\sigma}\right)^2\right]$$

This is carried out by multiplication in the Fourier domain rather than using a convolution function.

Parameters

- **wave** (*array_like*) – The current wavelength grid
- **flux** (*array_like*) – The current flux
- **fwhm** (*float*) – The full width half-maximum of the instrument in km/s. Note that this is quivalent to $2.355 \cdot \sigma$

Raises **ValueError** – If the full width half maximum is negative.

Returns The broadened flux with the same shape as the input flux

Return type numpy.ndarray

Starfish.transforms.**rotational_broaden**(*wave*, *flux*, *vsini*)

Broadens flux according to a rotational broadening kernel from Gray (2005)[1]

[1] Gray, D. (2005). *The observation and Analysis of Stellar Photospheres*.

Parameters

- **wave** (*array_like*) – The current wavelength grid
- **flux** (*array_like*) – The current flux
- **vsini** (*float*) – The rotational velocity in km/s

Raises **ValueError** – if *vsini* is not positive

Returns The broadened flux with the same shape as the input flux

Return type numpy.ndarray

Cambridge: Cambridge University Press. doi:10.1017/CB09781316036570

Starfish.transforms.**extinct**(*wave*, *flux*, *Av*, *Rv=3.1*, *law='ccm89'*)

Extinct a spectrum following one of many empirical extinction laws. This makes use of the *extinction* package. In general, it follows the form

$$f \cdot 10^{-0.4 A_V \cdot A_\lambda(R_V)}$$

Parameters

- **wave** (*array_like*) – The input wavelengths in Angstrom
- **flux** (*array_like*) – The input fluxes
- **Av** (*float*) – The absolute attenuation
- **Rv** (*float, optional*) – The relative attenuation (the default is 3.1, which is the Milky Way average)
- **law** (*str, optional*) – The extinction law to use. One of *{'ccm89', 'odonnell94', 'calzetti00', 'fitzpatrick99', 'fm07'}* (the default is 'ccm89')

Raises

- **ValueError** – If *law* does not match one of the availabe laws
- **ValueError** – If Rv is not positive

Returns The extincted fluxes, with same shape as input fluxes.

Return type numpy.ndarray

Starfish.transforms.**chebyshev_correct**(*wave*, *flux*, *coeffs*)

Multiply the input flux by a Chebyshev series in order to correct for calibration-level discrepancies.

Parameters

- **wave** (*array-lioke*) – Input wavelengths
- **flux** (*array-like*) – Input flux
- **coeffs** (*array-like*) – The coefficients for the chebyshev series.

Returns The corrected flux

Return type numpy.ndarray

Raises **ValueError** – If only processing a single spectrum and the linear coefficient is not 1.

1.4.5 Models

SpectrumModel

The *SpectrumModel* is the main implementation of the Starfish methods for a single-order spectrum. It works by interfacing with both *Starfish.emulator.Emulator*, *Starfish.spectrum.Spectrum*, and the methods in *Starfish.transforms*. The spectral emulator provides an interface to spectral model libraries with a covariance matrix for each interpolated spectrum. The transforms provide the physics behind alterations to the light. For a given set of parameters, a transformed spectrum and covariance matrix are provided by

```
>>> from Starfish.models import SpectrumModel
>>> model = SpectrumModel(...)
>>> flux, cov = model()
```

It is also possible to optimize our parameters using the interfaces provided in *SpectrumModel.get_param_vector()*, *SpectrumModel.set_param_vector()*, and *SpectrumModel.log_likelihood()*. A very minimal example might be

```
>>> from Starfish.models import SpectrumModel
>>> from scipy.optimize import minimize
>>> model = SpectrumModel(...)
>>> def nll(P):
        model.set_param_vector(P)
        lnprob = model.log_likelihood()
        return -lnprob
>>> P0 = model.get_param_vector()
>>> soln = minimize(nll, P0, method='Nelder-Mead')
```

For a more thorough example, see the *Examples*.

Parametrization

This model uses a method of specifying parameters very similar to Dan Foreman-Mackey's George library. There exists an underlying dictionary of the model parameters, which define what transformations will be made. For example, if `vz` exists in a model's parameter dictionary, then doppler shifting will occur when calling the model.

It is possible to have a parameter that transforms the spectrum, but is not fittable. We call these *frozen* parameters. For instance, if my 3 model library parameters are T_{eff}, $\log g$, and $[Fe/H]$ (or `T`, `logg`, `Z` in the code), but I don't want to fit $log g$, I can freeze it:

```
>>> from Starfish.models import SpectrumModel
>>> model = SpectrumModel(...)
>>> model.freeze('logg')
```

When using this framework, you can see what transformations will occur by looking at *SpectrumModel.params* and what values are fittable by *SpectrumModel.get_param_dict()* (or the other getters for the parameters).

```
>>> model.params
{'T': 6020, 'logg': 4.2, 'Z': 0.0, 'vsini': 84.0, 'log_scale': -10.23}
>>> model.get_param_dict()
{'T': 6020, 'Z': 0.0, 'vsini': 84.0, 'log_scale': -10.23}
```

To undo this, simply thaw the frozen parameters

```
>>> model.thaw('logg')
>>> model.params == model.get_param_dict()
True
```

The underlying parameter dictionary is a special flat dictionary. Consider the nested dictionary

```
>>> cov = {
...     'log_amp': 0,
...     'log_ls': 6,
... }
>>> model['global_cov'] = cov
>>> model.params
{
    'T': 6020,
    'logg': 4.2,
    'Z': 0.0,
    'vsini': 84.0,
    'log_scale': -10.23,
    'global_cov:log_amp': 0,
    'global_cov:log_ls': 6
}
```

These values can be referenced normally or using its flat key

```
>>> model['global_cov:log_amp'] == model['global_cov']['log_amp']
True
```

API/Reference

class Starfish.models.**SpectrumModel**(*emulator: Union[str, Starfish.emulator.emulator.Emulator], data: Union[str, Starfish.spectrum.Spectrum], grid_params: Sequence[float], max_deque_len: int = 100, norm=False, name: str = 'SpectrumModel', **params*)

A single-order spectrum model.

Parameters

- **emulator** (Starfish.emulators.Emulator) – The emulator to use for this model.
- **data** (*Starfish.spectrum.Spectrum*) – The data to use for this model
- **grid_params** (*array-like*) – The parameters that are used with the associated emulator
- **max_deque_len** (*int, optional*) – The maximum number of residuals to retain in a deque of residuals. Default is 100
- **norm** (*bool, optional*) – If true, will rescale the model flux to the appropriate flux normalization according to the original spectral library. Default is *False*.
- **name** (*str, optional*) – A name for the model. Default is 'SpectrumModel'

Keyword Arguments **params** (*dict*) – Any remaining keyword arguments will be interpreted as parameters.

Here is a table describing the avialable parameters and their related functions

Parameter	Function
vsini	*rotational_broaden()*
vz	*doppler_shift()*
Av	*extinct()*
Rv	*extinct()*
log_scale	*rescale()*
cheb	*chebyshev_correct()*

Note: If `log_scale` is not specified, the model will use *renorm()* to automatically scale the spectrum to the data using the ratio of integrated fluxes.

Note: *cheb* corresponds to a list/array of coefficients, however we force the constant coefficient (*c0*) to be 1. This means *cheb* will correspond to *c1, c2, ...* The entire list can be retrieved like *model["cheb"]* and indiviual values can be retrieved with *model["cheb:1"]*.

The `global_cov` keyword arguments must be a dictionary definining the hyperparameters for the global covariance kernel, `kernels.global_covariance_matrix()`

Global Parameter	Description
log_amp	The natural logarithm of the amplitude of the Matern kernel
log_ls	The natural logarithm of the lengthscale of the Matern kernel

The `local_cov` keryword argument must be a list of dictionaries defining hyperparameters for many Gaussian kernels, , `kernels.local_covariance_matrix()`

Local Parameter	Description
log_amp	The natural logarithm of the amplitude of the kernel
mu	The location of the local kernel
log_sigma	The natural logarithm of the standard deviation of the kernel

params
The dictionary of parameters that are used for doing the modeling. (The Chebyshev coefficients are not stored in this structure)

Type dict

grid_params
The vector of parameters for the spectral emulator. Setter obeys frozen parameters.

Type ndarray

cheb
The vector of *c1, c2, . . .* Chebyshev coefficients. *c0* is fixed to 1 by definition. Setter obeys frozen parameters.

Type ndarray

frozen
A list of strings corresponding to frozen parameters

Type list

residuals
A deque containing residuals from calling *SpectrumModel.log_likelihood()*

Type deque

__call__()
Performs the transformations according to the parameters available in `self.params`

Returns **flux, cov** – The transformed flux and covariance matrix from the model

Return type tuple

property cheb
The Chebyshev polynomial coefficients used for the background model

Type numpy.ndarray

freeze(*names*)
Freeze the given parameter such that *get_param_dict()* and *get_param_vector()* no longer include this parameter, however it will still be used when calling the model.

Parameters **name** (*str or array-like*) – The parameter to freeze. If `'all'`, will freeze all parameters. If `'global_cov'` will freeze all global covariance parameters. If `'local_cov'` will freeze all local covariance parameters.

Raises **ValueError** – If the given parameter does not exist

See also:

thaw()

get_param_dict(*flat: bool = False*) → dict
Gets the dictionary of thawed parameters.

Parameters **flat** (*bool, optional*) – If True, returns the parameters completely flat. For example, `['local']['0']['mu']` would have the key `'local:0:mu'`. Default is False

Returns

Return type dict

See also:

set_param_dict()

get_param_vector()
Get a numpy array of the thawed parameters

Returns

Return type numpy.ndarray

See also:

set_param_vector()

property grid_params
The parameters used for the spectral emulator.

Setter Sets the values in the order of `Emulator.param_names`

Type numpy.ndarray

property labels
The thawed parameter names

Type tuple of str

load(*filename*)
Load a saved model state from a TOML file

Parameters **filename** (*str or path-like*) – The saved state to load

log_likelihood(*priors: Optional[dict] = None*) → float
Returns the log probability of a multivariate normal distribution

Parameters **priors** (*dict, optional*) – If provided, will use these priors in the MLE. Should contain keys that match the model's keys and values that have a *logpdf* method that takes one value (like `scipy.stats` distributions). Default is None.

Warning: No checks will be done on the `priors` for speed.

Returns

Return type float

plot(*axes=None*, *plot_kwargs=None*, *resid_kwargs=None*)
Plot the model.

This will create two subplots, one which shows the current model against the data, and another which shows the current residuals with 3:math:*sigma* contours from the diagonal of the covariance matrix. Note this requires matplotlib to be installed, which is not installed by default with Starfish.

Parameters

- **axes** (*iterable of matplotlib.Axes, optional*) – If provided, will use the first two axes to plot, otherwise will create new axes, by default None
- **plot_kwargs** (*dict, optional*) – If provided, will use these kwargs for the comparison plot, by default None
- **resid_kwargs** (*dict, optional*) – If provided, will use these kwargs for the residuals plot, by default None

Returns The returned axes, for the user to edit as they please

Return type list of matplotlib.Axes

save(*filename*, *metadata=None*)
Saves the model as a set of parameters into a TOML file

Parameters

- **filename** (*str or path-like*) – The TOML filename to save to.
- **metadata** (*dict, optional*) – If provided, will save the provided dictionary under a 'metadata' key. This will not be read in when loading models but provides a way of providing information in the actual TOML files. Default is None.

set_param_dict(*params*)
Sets the parameters with a dictionary. Note that this should not be used to add new parameters

Parameters **params** (*dict*) – The new parameters. If a key is present in `self.frozen` it will not be changed

See also:

get_param_dict()

set_param_vector(*params*)
Sets the parameters based on the current thawed state. The values will be inserted according to the order of *SpectrumModel.labels*.

Parameters **params** (*array_like*) – The parameters to set in the model

Raises **ValueError** – If the *params* do not match the length of the current thawed parameters.

See also:

get_param_vector()

thaw(*names*)
Thaws the given parameter. Opposite of freezing

Parameters **name** (*str or array-like*) – The parameter to thaw. If `'all'`, will thaw all parameters. If `'global_cov'` will thaw all global covariance parameters. If `'local_cov'` will thaw all local covariance parameters.

Raises **ValueError** – If the given parameter does not exist.

See also:

freeze()

train(*priors: Optional[dict] = None*, ***kwargs*)
Given a *SpectrumModel* and a dictionary of priors, will perform maximum-likelihood estimation (MLE). This will use `scipy.optimize.minimize` to find the maximum a-posteriori (MAP) estimate of the current model state. Note that this alters the state of the model. This means that you can run this method multiple times until the optimization succeeds. By default, we use the "Nelder-Mead" method in *minimize* to avoid approximating any derivatives.

Parameters

- **priors** (*dict, optional*) – Priors to pass to *log_likelihood()*
- ****kwargs** (*dict, optional*) – These keyword arguments will be passed to *scipy.optimize.minimize*

Returns **soln** – The output of the minimization.

Return type *scipy.optimize.minimize_result*

Raises

- **ValueError** – If the priors are poorly specified
- **RuntimeError** – If any priors evaluate to non-finite values

See also:

log_likelihood()

Utils

There are some utilities that help with interfacing with the various Models

`Starfish.models.`**find_residual_peaks**(*model*, *num_residuals=100*, *threshold=4.0*, *buffer=2*, *wl_range=(0, inf)*)
Find the peaks of the most recent residual and return their properties to aid in setting up local kernels

Parameters

- **model** (*Model*) – The model to determine peaks from. Need only have a residuals array.

- **num_residuals** (*int, optional*) – The number of residuals to average together for determining peaks. By default 100.
- **threshold** (*float, optional*) – The sigma clipping threshold, by default 4.0
- **buffer** (*float, optional*) – The minimum distance between peaks, in Angstrom, by default 2.0
- **wl_range** (*2-tuple*) – The (min, max) wavelengths to consider. Default is (0, np.inf)

Returns **means** – The means of the found peaks, with the same units as model.data.wave

Return type list

Starfish.models.**optimize_residual_peaks**(*model*, *mus*, *threshold=0.1*, *sigma0=50*, *num_residuals=100*)

Optimize the local covariance parameters based on fitting the residual input means as Gaussians around the residuals

Parameters

- **model** (*Model*) – The model to determine peaks from. Need only have a residuals array.
- **mus** (*array-like*) – The means to instantiate Gaussians at and optimize.
- **threshold** (*float, optional*) – This is the threshold for restricting kernels; i.e. if a fit amplitude is less than threshold standard deviations then it will be thrown away. Default is 0.1
- **sigma0** (*float, optional*) – The initial standard deviation (in Angstrom) of each Gaussian. Default is 50 Angstrom.
- **num_residuals** (*int, optional*) – The number of residuals to average together for determining peaks. By default 100.

Returns A dictionary of optimized parameters ready to be plugged into model["local_cov"]

Return type dict

> **Warning:** I have had inconsistent results with this optimization, be mindful of your outputs and consider hand-tuning after optimizing.

Starfish.models.**covariance_debugger**(*cov: nptyping.types._ndarray.NDArray*)

Special debugging information for the covariance matrix decomposition.

1.5 Examples

Here you can find some examples generated from jupyter notebooks that show you how to get up and running with *Starfish*. The workflows presented offer a good starting point to begin diving into all that *Starfish* has to offer. Since the documents are generated from notebooks, you can download a copy and run them yourself with little to no fuss. The notebooks are hosted in the examples directory of the GitHub repository.

This page was generated from examples/setup.ipynb.

1.5.1 Setup

Here I will go over setting up our interfaces and emulators from a raw spectral library to prepare us for fitting some data in further examples.

Getting the Grid

To begin, we need a spectral model library that we will use for our fitting. One common example are the PHOENIX models, most recently computed by T.O. Husser. We provide many interfaces directly with different libraries, which can be viewed in *Raw Grid Interfaces*.

As a convenience, we provide a helper to download PHOENIX models from the Goettingen servers. Note this will skip any files already on disk.

[1]:
```
import numpy as np

from Starfish.grid_tools import download_PHOENIX_models

ranges = [[5700, 8600], [4.0, 6.0], [-0.5, 0.5]]  # T, logg, Z

download_PHOENIX_models(path="PHOENIX", ranges=ranges)
```

```
lte08600-6.00+0.5.PHOENIX-ACES-AGSS-COND-2011-HiRes.fits: 100%|| 330/330 [00:00<00:00,
↪ 1285.27it/s]
```

Now that we have the files downloaded, let's set up a grid interface

[2]:
```
from Starfish.grid_tools import PHOENIXGridInterfaceNoAlpha

grid = PHOENIXGridInterfaceNoAlpha(path="PHOENIX")
```

From here, we will want to set up our HDF5 interface that will allow us to go on to using the spectral emulator, but first we need to determine our model subset and instrument.

Setting up the HDF5 Interface

We set up an HDF5 interface in order to allow much quicker reading and writing than compared to loading FITS files over and over again. In addition, when considering the application to our likelihood methods, we know that for a given dataset, any effects characteristic of the instrument can be pre-applied to our models, saving on computation time during the maximum likelihood estimation.

Looking towards our fitting examples, we know we will try fitting some data from TRES Spectrograph. This instrument is available in our grid tools, but if yours isn't, you can always supply the FWHM in km/s. The FWHM (Γ) can be found using the resolving power, R

$$\Gamma = \frac{c}{R}$$

with c in km/s. Let's also say that, for a given dataset, we want to only use a reasonable subset of our original model grid. The data provided in future examples is a ~F3V star, so we will limit our model parameter ranges appropriately.

[3]:
```
from Starfish.grid_tools.instruments import SPEX
from Starfish.grid_tools import HDF5Creator

creator = HDF5Creator(
    grid, "F_SPEX_grid.hdf5", instrument=SPEX(), wl_range=(0.9e4, np.inf),␣
↪ranges=ranges
```

(continues on next page)

(continued from previous page)

```
)
creator.process_grid()
```

```
Processing [8.6e+03 6.0e+00 5.0e-01]: 100%|| 330/330 [06:33<00:00,  1.19s/it]
```

Setting up the Spectral Emulator

Once we have our pre-processed grid, we can make our spectral emulator and train its Gaussian process hyperparameters.

[4]:
```
from Starfish.emulator import Emulator

# can load from string or HDF5Interface
emu = Emulator.from_grid("F_SPEX_grid.hdf5")
emu
```

[4]:
```
Emulator
--------
Trained: False
lambda_xi: 1.000
Variances:
        10000.00
        10000.00
        10000.00
        10000.00
Lengthscales:
        [ 600.00  1.50  1.50 ]
        [ 600.00  1.50  1.50 ]
        [ 600.00  1.50  1.50 ]
        [ 600.00  1.50  1.50 ]
Log Likelihood: -1272.34
```

[5]:
```
%time emu.train(options=dict(maxiter=1e5))
emu
```

```
CPU times: user 17min 39s, sys: 1min 58s, total: 19min 38s
Wall time: 4min 55s
```

[5]:
```
Emulator
--------
Trained: True
lambda_xi: 1.010
Variances:
        176330.09
        1681.55
        1364.83
        433.88
Lengthscales:
        [ 2039.21  16.11  3.21 ]
        [ 1313.27  1.49  1.86 ]
        [ 2122.93  2.58  2.21 ]
        [ 1009.99  1.20  3.26 ]
Log Likelihood: -778.44
```

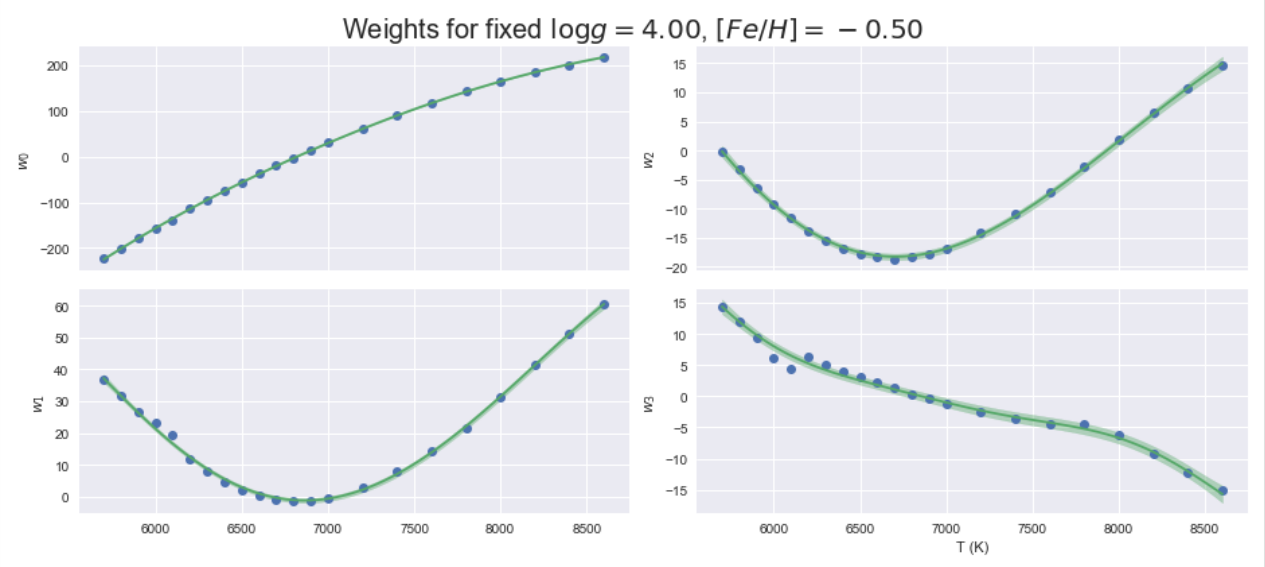
Note: If the emulator does not optimize the first time you use `train`, just run it again. You can also tweak the

arguments passed to `scipy.optimize.minimize` by passing them as keyword arguments to the call.

Warning: Training the emulator will take on the order of minutes to complete. The more eigenspectra that are used as well as the resolution of the spectrograph will mainly dominate this runtime.

We can do a sanity check on the optimization by looking at slice of the emulator's parameter space and the corresponding Gaussian process fit. We should see a smooth line connecting all the parameter values with some uncertainty that grows with large gaps or turbulent weights.

[6]:
```
%matplotlib inline
from Starfish.emulator.plotting import plot_emulator

plot_emulator(emu)
```



If we are satisfied, let's save this emulator and move on to fitting some data.

[7]:
```
emu.save("F_SPEX_emu.hdf5")
```

This page was generated from examples/single.ipynb.

1.5.2 Single-Order Spectrum

This will show how to fit a single-order spectrum using our *previous setup* on some ~*mysterious*~ IRTF SpeX data. The spectrum is available for download here.

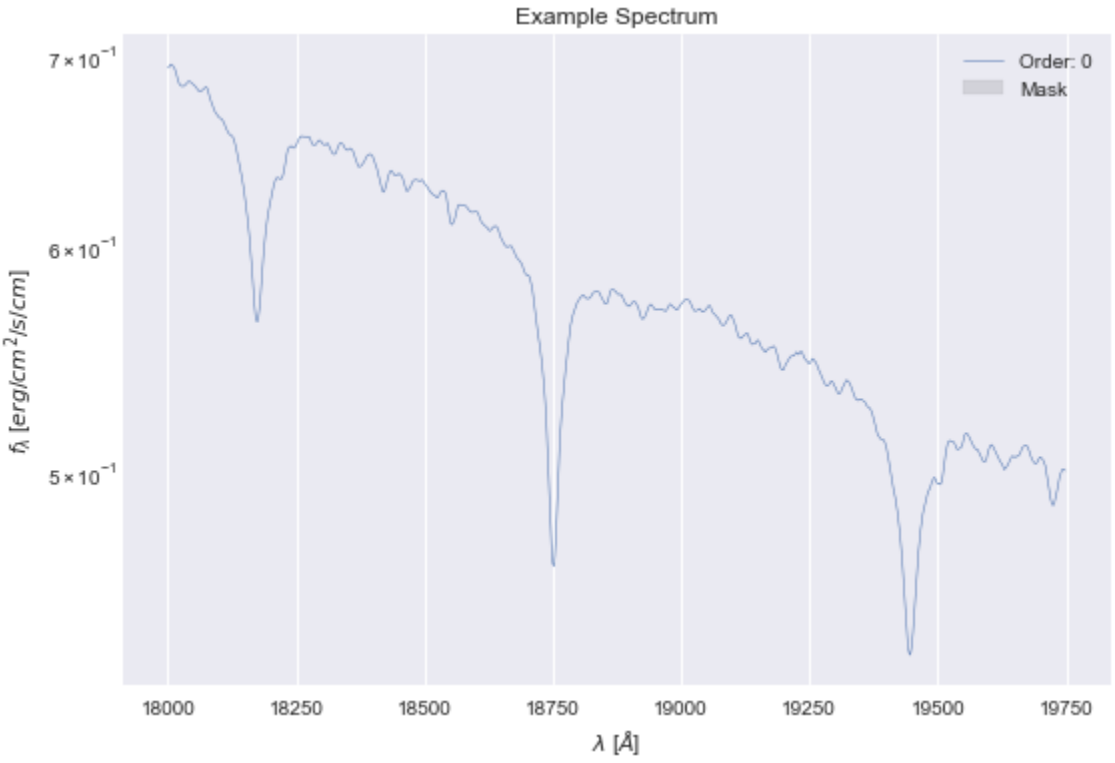
Note: This documentation is not meant to be an exhaustive tour of *Starfish*'s features, but rather a simple example showing a workflow typical of fitting data.

Preprocessing

Normally, you would pre-process your data. This includes loading the fits files, separating out the wavelengths, fluxes, uncertainties, and any masks. In addition, you would need to convert your data into the same units as your emulator. In our case, the PHOENIX emulator uses A and $erg/cm^2/s/cm$. For this example, though, I've already created a spectrum that you can load directly.

[1]:
```
%matplotlib inline
import matplotlib.pyplot as plt

plt.style.use("seaborn")
```

[2]:
```
from Starfish.spectrum import Spectrum

data = Spectrum.load("example_spec.hdf5")

data.plot();
```



Setting up the model

Now we can set up our initial model. We need, at minimum, an emulator, our data, and a set of the library grid parameters. Every extra keyword argument we add is added to our list of parameters. For more information on what parameters are available and what effect they have, see the *SpectrumModel documentation*.

Some of these parameters are based on guesses or pre-existing knowledge. In particular, if you want to fit `log_scale`, you should spend some time tuning it by eye, first. We also want our `global_cov:log_amp` to be reasonable, so pay attention to the σ-contours in the residuals plots, too.

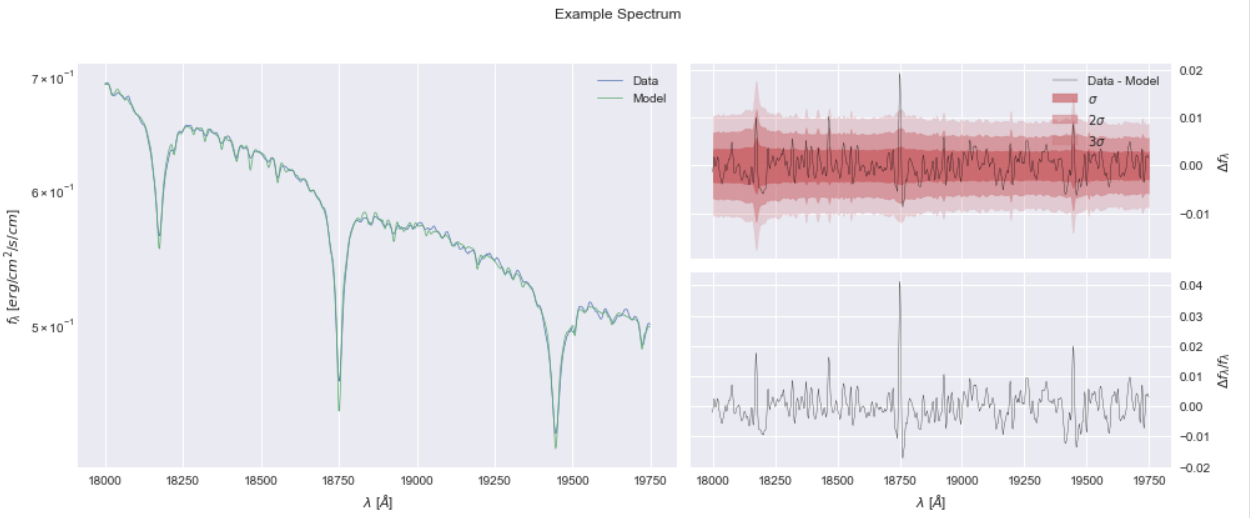
There aren't any previous in-depth works on this star, so we will start with some values based on the spectral type alone.

[3]:
```
from Starfish.models import SpectrumModel

model = SpectrumModel(
    "F_SPEX_emu.hdf5",
    data,
    grid_params=[6800, 4.2, 0],
    Av=0,
    global_cov=dict(log_amp=38, log_ls=2),
)
model
```

[3]:
```
SpectrumModel
-------------
Data: Example Spectrum
Emulator: F_SPEX_emu
Log Likelihood: None

Parameters
  Av: 0
  global_cov:
    log_amp: 38
    log_ls: 2
  T: 6800
  logg: 4.2
  Z: 0
  log_scale: -0.020519147372786654 (fit)
```

In this plot, we can see the data and model in the left pane, the absolute errors (residuals) along with the diagonal of the covariance matrix as σ contours in the top-right, and the relative errors (residuals / flux) in the bottom-right

[4]:
```
model.plot();
```



Numerical Optimization

Now lets do a *maximum a posteriori* (MAP) point estimate for our data.

Here we freeze `logg` here because the PHOENIX models' response to `logg` compared to our data are relatively flat, so we fix the value using the *freeze* mechanics. This is equivalent to applying a δ-function prior.

[5]:
```
model.freeze("logg")
model.labels  # These are the fittable parameters
```

[5]:
```
('Av', 'global_cov:log_amp', 'global_cov:log_ls', 'T', 'Z')
```

Here we specify some priors using `scipy.stats` classes. If you have a custom distribution you want to use, create a class and make sure it has a `logpdf` member function.

[6]:
```
import scipy.stats as st

priors = {
    "T": st.norm(6800, 100),
    "Z": st.uniform(-0.5, 0.5),
    "Av": st.halfnorm(0, 0.2),
    "global_cov:log_amp": st.norm(38, 1),
    "global_cov:log_ls": st.uniform(0, 10),
}
```

Using the above priors, we can do our MAP optimization using `scipy.optimize.minimze`, which is usefully baked into the `train` method of our model. This should give us a good starting point for our MCMC sampling later.

[7]:
```
%time model.train(priors)
```

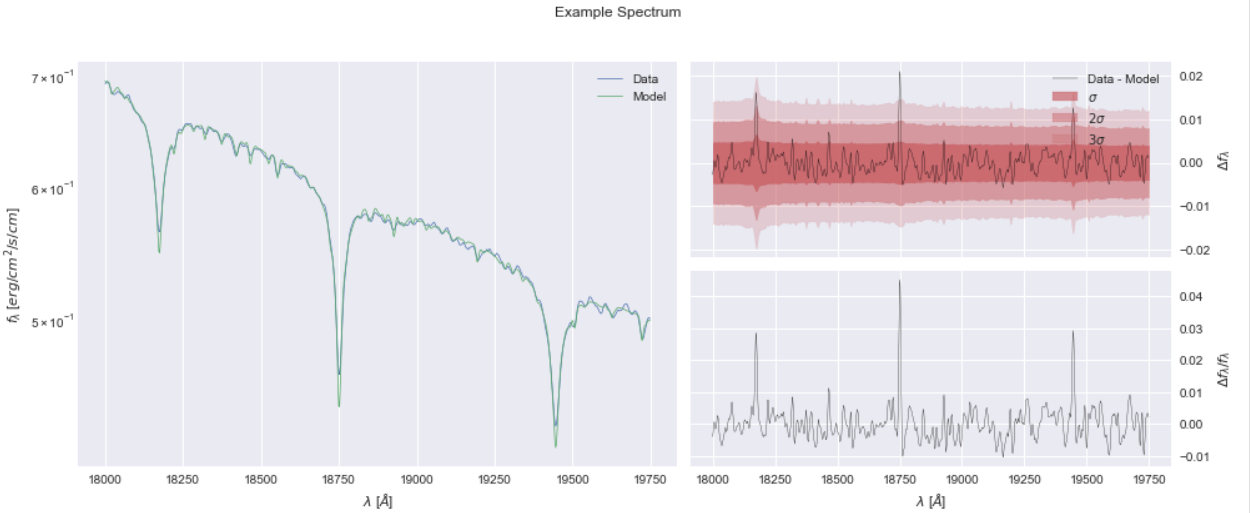
```
CPU times: user 5min 1s, sys: 1min 10s, total: 6min 11s
Wall time: 2min 11s
```

[7]:
```
 final_simplex: (array([[ 4.35997017e-14,  3.88150634e+01,  4.13042481e+00,
         7.01344658e+03, -3.39321728e-03],
       [ 1.31419453e-13,  3.88150632e+01,  4.13042476e+00,
         7.01344655e+03, -3.39321733e-03],
       [ 1.03637001e-13,  3.88150631e+01,  4.13042471e+00,
         7.01344650e+03, -3.39321746e-03],
       [ 9.47232580e-14,  3.88150633e+01,  4.13042477e+00,
         7.01344654e+03, -3.39321740e-03],
       [ 8.68158683e-14,  3.88150632e+01,  4.13042476e+00,
         7.01344655e+03, -3.39321733e-03],
       [ 6.46583923e-14,  3.88150633e+01,  4.13042476e+00,
         7.01344655e+03, -3.39321734e-03]]), array([-5875.45984265, -5875.45984265, -
↪5875.45984265, -5875.45984265,
       -5875.45984265, -5875.45984265]))
           fun: -5875.4598426469365
       message: 'Optimization terminated successfully.'
          nfev: 960
           nit: 575
        status: 0
       success: True
             x: array([ 4.35997017e-14,  3.88150634e+01,  4.13042481e+00,  7.
↪01344658e+03,
       -3.39321728e-03])
```

[8]:
```
model
```

[8]:
```
SpectrumModel
-------------
Data: Example Spectrum
Emulator: F_SPEX_emu
Log Likelihood: 5884.738816795258

Parameters
  Av: 4.3599701716468095e-14
  global_cov:
    log_amp: 38.81506339478135
    log_ls: 4.13042481437768
  T: 7013.446581077344
  Z: -0.0033932172800382084
  log_scale: 0.007658916985542291 (fit)

Frozen Parameters
  logg: 4.2
```

[9]:
```
model.plot();
```



[10]:
```
model.save("example_MAP.toml")
```

MCMC Sampling

Now, we will sample from our model. Note the flexibility we provide with *Starfish* in order to allow sampler front-end that allows blackbox likelihood methods. In our case, we will continue with emcee, which provides an ensemble sampler. We are using pre-release of version `3.0`. This document serves only as an example, and details about emcee's usage should be sought after in its documentation.

For this basic example, I will freeze both the global and local covariance parameters, so we are only sampling over `T`, `Z`, and `Av`.

[11]:
```
import emcee

emcee.__version__
```

[11]:
```
'3.0.2'
```

[12]:
```
model.load("example_MAP.toml")
model.freeze("global_cov")
model.labels
```

[12]:
```
('Av', 'T', 'Z')
```

[13]:
```
import numpy as np

# Set our walkers and dimensionality
nwalkers = 50
ndim = len(model.labels)

# Initialize gaussian ball for starting point of walkers
scales = {"T": 1, "Av": 0.01, "Z": 0.01}

ball = np.random.randn(nwalkers, ndim)

for i, key in enumerate(model.labels):
    ball[:, i] *= scales[key]
    ball[:, i] += model[key]
```

[14]:
```
# our objective to maximize
def log_prob(P, priors):
    model.set_param_vector(P)
    return model.log_likelihood(priors)


# Set up our backend and sampler
backend = emcee.backends.HDFBackend("example_chain.hdf5")
backend.reset(nwalkers, ndim)
sampler = emcee.EnsembleSampler(
    nwalkers, ndim, log_prob, args=(priors,), backend=backend
)
```

here we start our sampler, and following this example we check every 10 steps for convergence, with a max burn-in of 1000 samples.

Warning: This process *can* take a long time to finish. In cases with high resolution spectra or fully evaluating each nuisance covariance parameter, we recommend running on a remote machine. A setup I recommend is a remote jupyter server, so you don't have to create any scripts and can keeping working in notebooks.

[16]:
```
max_n = 1000

# We'll track how the average autocorrelation time estimate changes
index = 0
autocorr = np.empty(max_n)

# This will be useful to testing convergence
old_tau = np.inf


# Now we'll sample for up to max_n steps
for sample in sampler.sample(ball, iterations=max_n, progress=True):
    # Only check convergence every 10 steps
    if sampler.iteration % 10:
```

(continues on next page)

(continued from previous page)

```
        continue

    # Compute the autocorrelation time so far
    # Using tol=0 means that we'll always get an estimate even
    # if it isn't trustworthy
    tau = sampler.get_autocorr_time(tol=0)
    autocorr[index] = np.mean(tau)
    index += 1
    # skip math if it's just going to yell at us
    if np.isnan(tau).any() or (tau == 0).any():
        continue
    # Check convergence
    converged = np.all(tau * 10 < sampler.iteration)
    converged &= np.all(np.abs(old_tau - tau) / tau < 0.01)
    if converged:
        print(f"Converged at sample {sampler.iteration}")
        break
    old_tau = tau
```

```
  2%|         | 20/1000 [01:18<1:22:41,  5.06s/it]/Users/miles/.pyenv/versions/3.7.4/
↪Python.framework/Versions/3.7/lib/python3.7/site-packages/ipykernel_launcher.py:28:␣
↪RuntimeWarning: invalid value encountered in less
 40%|      | 400/1000 [37:01<55:31,  5.55s/it]
```

```
Converged at sample 410
```

After our model has converged, let's take a few extra samples to make sure we have clean chains. Remember, we have 50 walkers, so 100 samples ends up becoming 5000 across each chain!

[17]:
```
sampler.run_mcmc(backend.get_last_sample(), 100, progress=True);
```

```
100%|| 100/100 [40:52<00:00, 24.52s/it]
```

MCMC Chain Analysis

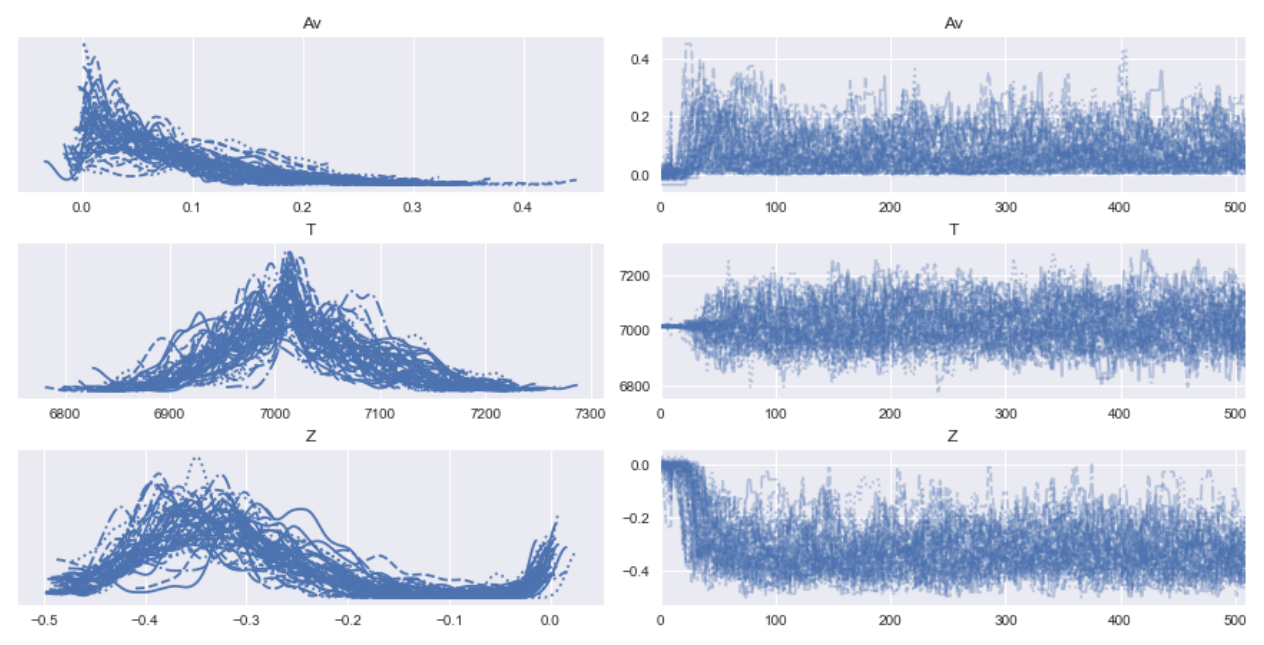
Chain analysis is a very broad topic that is mostly out of the scope of this example. For our analysis, we like using ArviZ with a simple corner plot as well.

[18]:
```
import arviz as az
import corner

print(az.__version__, corner.__version__)
```

```
0.11.0 2.1.0
```

[19]:
```
reader = emcee.backends.HDFBackend("example_chain.hdf5")
full_data = az.from_emcee(reader, var_names=model.labels)
```

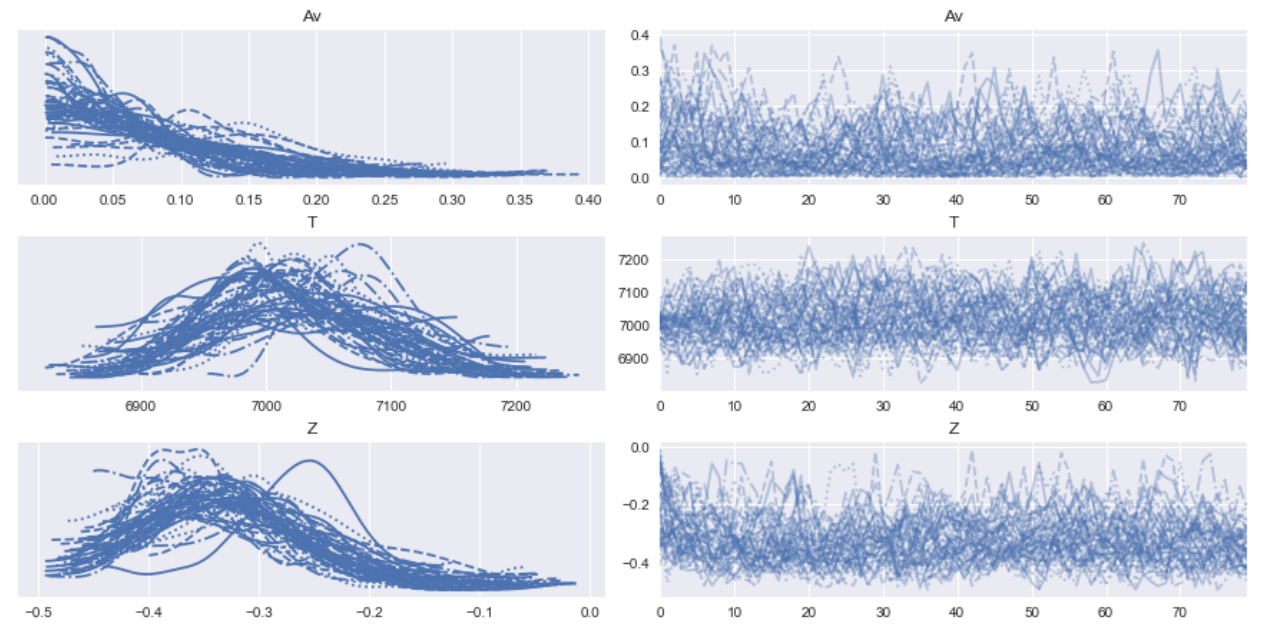
[20]:
```
az.plot_trace(full_data);
```



After seeing our full traces, let's discard and thin some of the burn-in

[21]:
```
tau = reader.get_autocorr_time(tol=0)
burnin = int(tau.max())
thin = int(0.3 * np.min(tau))
burn_samples = reader.get_chain(discard=burnin, thin=thin)
log_prob_samples = reader.get_log_prob(discard=burnin, thin=thin)
log_prior_samples = reader.get_blobs(discard=burnin, thin=thin)

dd = dict(zip(model.labels, burn_samples.T))
burn_data = az.from_dict(dd)
```

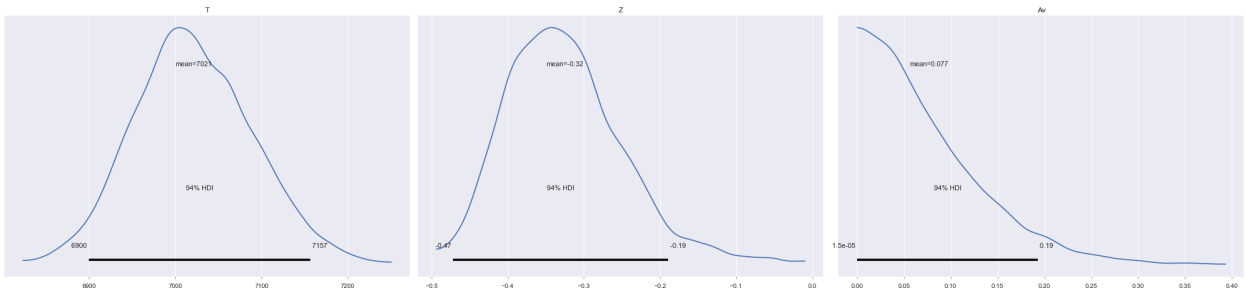
[22]:
```
az.plot_trace(burn_data);
```



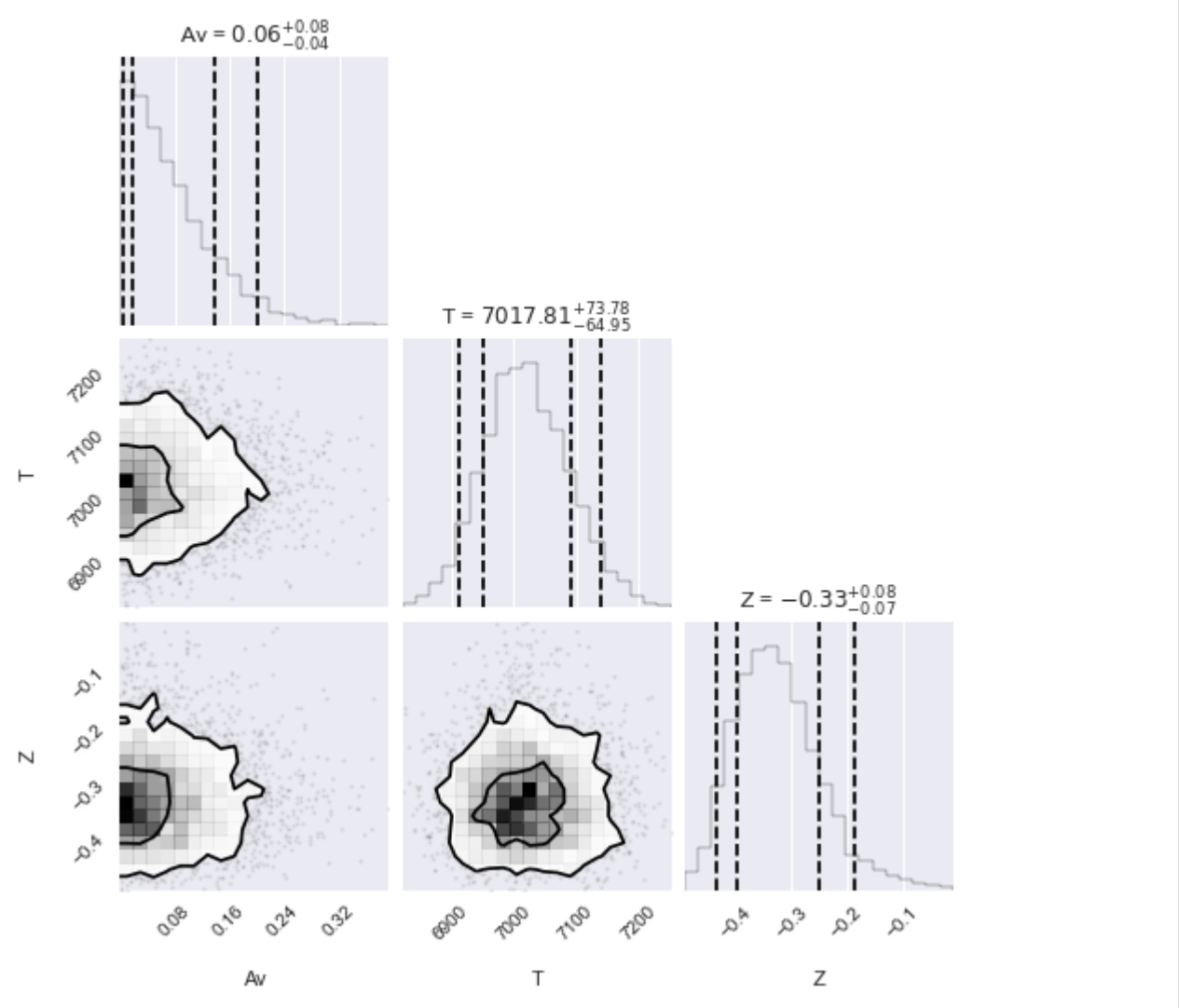
[23]:
```
az.summary(burn_data)
```

[23]:
```
        mean      sd    hdi_3%   hdi_97%  mcse_mean  mcse_sd  ess_mean  \
Av     0.077   0.063     0.000     0.193      0.003    0.002     396.0
T   7021.219  68.688  6899.632  7156.687      3.331    2.358     425.0
Z     -0.325   0.078    -0.472    -0.189      0.003    0.002     514.0

    ess_sd  ess_bulk  ess_tail  r_hat
Av   396.0     437.0    1599.0   1.09
T    425.0     434.0    1733.0   1.09
Z    514.0     537.0    1298.0   1.07
```

[24]:
```
az.plot_posterior(burn_data, ["T", "Z", "Av"]);
```



[25]:
```
# See https://corner.readthedocs.io/en/latest/pages/sigmas.html#a-note-about-sigmas
sigmas = ((1 - np.exp(-0.5)), (1 - np.exp(-2)))
corner.corner(
    burn_samples.reshape((-1, 3)),
    labels=model.labels,
    quantiles=(0.05, 0.16, 0.84, 0.95),
    levels=sigmas,
    show_titles=True,
);
```



After looking at our posteriors, let's look at our fit

[26]:
```
best_fit = dict(az.summary(burn_data)["mean"])
model.set_param_dict(best_fit)
model
```

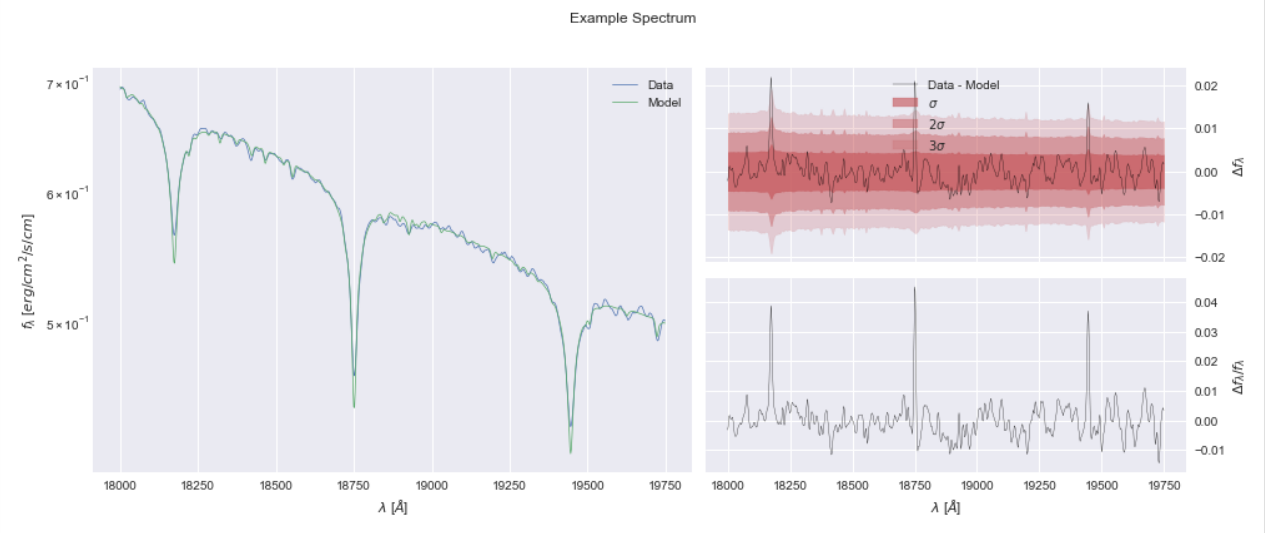
[26]:
```
SpectrumModel
-------------
Data: Example Spectrum
Emulator: F_SPEX_emu
Log Likelihood: 5883.673006463088

Parameters
  Av: 0.077
  T: 7021.219
  Z: -0.325
  log_scale: 0.011439395318924 84 (fit)

Frozen Parameters
  logg: 4.2
```

(continues on next page)

(continued from previous page)

```
  global_cov:log_amp: 38.81506339478135
  global_cov:log_ls: 4.13042481437768
```

[27]:
```
model.plot();
```



and finally, we can save our best fit.

[28]:
```
model.save("example_sampled.toml")
```

Now, on to the next star!

1.5.3 Multi-Order Spectrum

Warning: The current, updated code base does not have the framework for fitting multi-order Echelle spectra. We are working diligently to update the original functionality to match the updated API. For now, you will have to revert to Starfish `0.2`.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

Starfish.emulator, 24
Starfish.grid_tools, 11
Starfish.models, 37
Starfish.spectrum, 29
Starfish.transforms, 34

INDEX

Symbols

__call__() (*Starfish.emulator.Emulator method*), 27
__call__() (*Starfish.grid_tools.Interpolator method*), 20
__call__() (*Starfish.models.SpectrumModel method*), 40
__getitem__() (*Starfish.spectrum.Spectrum method*), 33
__len__() (*Starfish.spectrum.Order method*), 31
__len__() (*Starfish.spectrum.Spectrum method*), 33
__setitem__() (*Starfish.spectrum.Spectrum method*), 33
__str__() (*Starfish.grid_tools.Instrument method*), 21

A

air_to_vacuum() (*in module Starfish.grid_tools*), 23

B

BTSettlGridInterface (*class in Starfish.grid_tools*), 17
bulk_fluxes() (*Starfish.emulator.Emulator property*), 27

C

calculate_dv() (*in module Starfish.utils*), 30
calculate_dv_dict() (*in module Starfish.utils*), 30
calculate_n() (*in module Starfish.grid_tools*), 23
cheb (*Starfish.models.SpectrumModel attribute*), 39
cheb() (*Starfish.models.SpectrumModel property*), 40
chebyshev_correct() (*in module Starfish.transforms*), 36
check_params() (*Starfish.grid_tools.GridInterface method*), 13
check_params() (*Starfish.grid_tools.PHOENIXGridInterface method*), 14
check_params() (*Starfish.grid_tools.PHOENIXGridInterfaceNoAlpha method*), 15
chunk_list() (*in module Starfish.grid_tools*), 23
covariance_debugger() (*in module Starfish.models*), 43
create_log_lam_grid() (*in module Starfish.utils*), 30

D

DCT_DeVeny (*class in Starfish.grid_tools*), 22
determine_chunk_log() (*in module Starfish.grid_tools*), 23
determine_chunk_log() (*Starfish.emulator.Emulator method*), 27
doppler_shift() (*in module Starfish.transforms*), 35
download_PHOENIX_models() (*in module Starfish.grid_tools*), 16

E

Emulator (*class in Starfish.emulator*), 26
ESPaDOnS (*class in Starfish.grid_tools*), 22
extinct() (*in module Starfish.transforms*), 36

F

find_residual_peaks() (*in module Starfish.models*), 42
flux() (*Starfish.spectrum.Order property*), 31
fluxes() (*Starfish.grid_tools.HDF5Interface property*), 19
fluxes() (*Starfish.spectrum.Spectrum property*), 33
freeze() (*Starfish.models.SpectrumModel method*), 40
from_grid() (*Starfish.emulator.Emulator class method*), 27
frozen (*Starfish.models.SpectrumModel attribute*), 39

G

get_index() (*Starfish.emulator.Emulator method*), 28
get_param_dict() (*Starfish.emulator.Emulator method*), 28
get_param_dict() (*Starfish.models.SpectrumModel method*), 40
get_param_vector() (*Starfish.emulator.Emulator method*), 28
get_param_vector() (*Starfish.models.SpectrumModel method*), 40
get_wl_kurucz() (*Starfish.grid_tools.KuruczGridInterface static method*), 17

grid_params (*Starfish.models.SpectrumModel attribute*), 39
grid_params() (*Starfish.models.SpectrumModel property*), 40
GridInterface (*class in Starfish.grid_tools*), 13

H

HDF5Creator (*class in Starfish.grid_tools*), 18
HDF5Interface (*class in Starfish.grid_tools*), 19

I

IGRINS_H (*class in Starfish.grid_tools*), 22
IGRINS_K (*class in Starfish.grid_tools*), 22
Instrument (*class in Starfish.grid_tools*), 21
instrumental_broaden() (*in module Starfish.transforms*), 35
interpolate() (*Starfish.grid_tools.Interpolator method*), 20
Interpolator (*class in Starfish.grid_tools*), 20

K

KPNO (*class in Starfish.grid_tools*), 22
KuruczGridInterface (*class in Starfish.grid_tools*), 17

L

labels() (*Starfish.models.SpectrumModel property*), 40
lambda_xi() (*Starfish.emulator.Emulator property*), 28
lengthscales() (*Starfish.emulator.Emulator property*), 28
load() (*Starfish.emulator.Emulator class method*), 28
load() (*Starfish.models.SpectrumModel method*), 41
load() (*Starfish.spectrum.Spectrum class method*), 33
load_flux() (*Starfish.emulator.Emulator method*), 28
load_flux() (*Starfish.grid_tools.BTSettlGridInterface method*), 17
load_flux() (*Starfish.grid_tools.GridInterface method*), 13
load_flux() (*Starfish.grid_tools.HDF5Interface method*), 19
load_flux() (*Starfish.grid_tools.KuruczGridInterface method*), 17
load_flux() (*Starfish.grid_tools.PHOENIXGridInterface method*), 14
log_likelihood() (*Starfish.emulator.Emulator method*), 28
log_likelihood() (*Starfish.models.SpectrumModel method*), 41

M

mask (*Starfish.spectrum.Order attribute*), 31
masks() (*Starfish.spectrum.Spectrum property*), 33

N

name (*Starfish.spectrum.Order attribute*), 31
name (*Starfish.spectrum.Spectrum attribute*), 33
norm_factor() (*Starfish.emulator.Emulator method*), 29

O

optimize_residual_peaks() (*in module Starfish.models*), 43
Order (*class in Starfish.spectrum*), 31

P

params (*Starfish.emulator.Emulator attribute*), 27
params (*Starfish.models.SpectrumModel attribute*), 39
PHOENIXGridInterface (*class in Starfish.grid_tools*), 14
PHOENIXGridInterfaceNoAlpha (*class in Starfish.grid_tools*), 14
plot() (*Starfish.models.SpectrumModel method*), 41
plot() (*Starfish.spectrum.Spectrum method*), 33
process_grid() (*Starfish.grid_tools.HDF5Creator method*), 19

R

renorm() (*in module Starfish.transforms*), 34
resample() (*in module Starfish.transforms*), 34
rescale() (*in module Starfish.transforms*), 34
reshape() (*Starfish.spectrum.Spectrum method*), 33
residuals (*Starfish.models.SpectrumModel attribute*), 39
Reticon (*class in Starfish.grid_tools*), 22
rotational_broaden() (*in module Starfish.transforms*), 35

S

save() (*Starfish.emulator.Emulator method*), 29
save() (*Starfish.models.SpectrumModel method*), 41
save() (*Starfish.spectrum.Spectrum method*), 33
set_param_dict() (*Starfish.emulator.Emulator method*), 29
set_param_dict() (*Starfish.models.SpectrumModel method*), 41
set_param_vector() (*Starfish.emulator.Emulator method*), 29
set_param_vector() (*Starfish.models.SpectrumModel method*), 41
shape() (*Starfish.spectrum.Spectrum property*), 34
sigma() (*Starfish.spectrum.Order property*), 31
sigmas() (*Starfish.spectrum.Spectrum property*), 34
Spectrum (*class in Starfish.spectrum*), 32

SpectrumModel (*class in Starfish.models*), 38
SPEX (*class in Starfish.grid_tools*), 22
SPEX_SXD (*class in Starfish.grid_tools*), 22
Starfish.emulator (*module*), 24
Starfish.grid_tools (*module*), 11
Starfish.models (*module*), 37
Starfish.spectrum (*module*), 29
Starfish.transforms (*module*), 34

T

thaw() (*Starfish.models.SpectrumModel method*), 42
train() (*Starfish.emulator.Emulator method*), 29
train() (*Starfish.models.SpectrumModel method*), 42
TRES (*class in Starfish.grid_tools*), 22

V

vacuum_to_air() (*in module Starfish.grid_tools*), 23
variances() (*Starfish.emulator.Emulator property*), 29

W

wave() (*Starfish.spectrum.Order property*), 31
waves() (*Starfish.spectrum.Spectrum property*), 34
WIYN_Hydra (*class in Starfish.grid_tools*), 22