Starfish Documentation

Release 0.3.0

Ian Czekala

Jun 23, 2019

Contents

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 1

Contents

1.1 Introduction

1.1.1 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?"

1.1.2 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.

1.1.3 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.

1.1.4 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.

1.1.5 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.

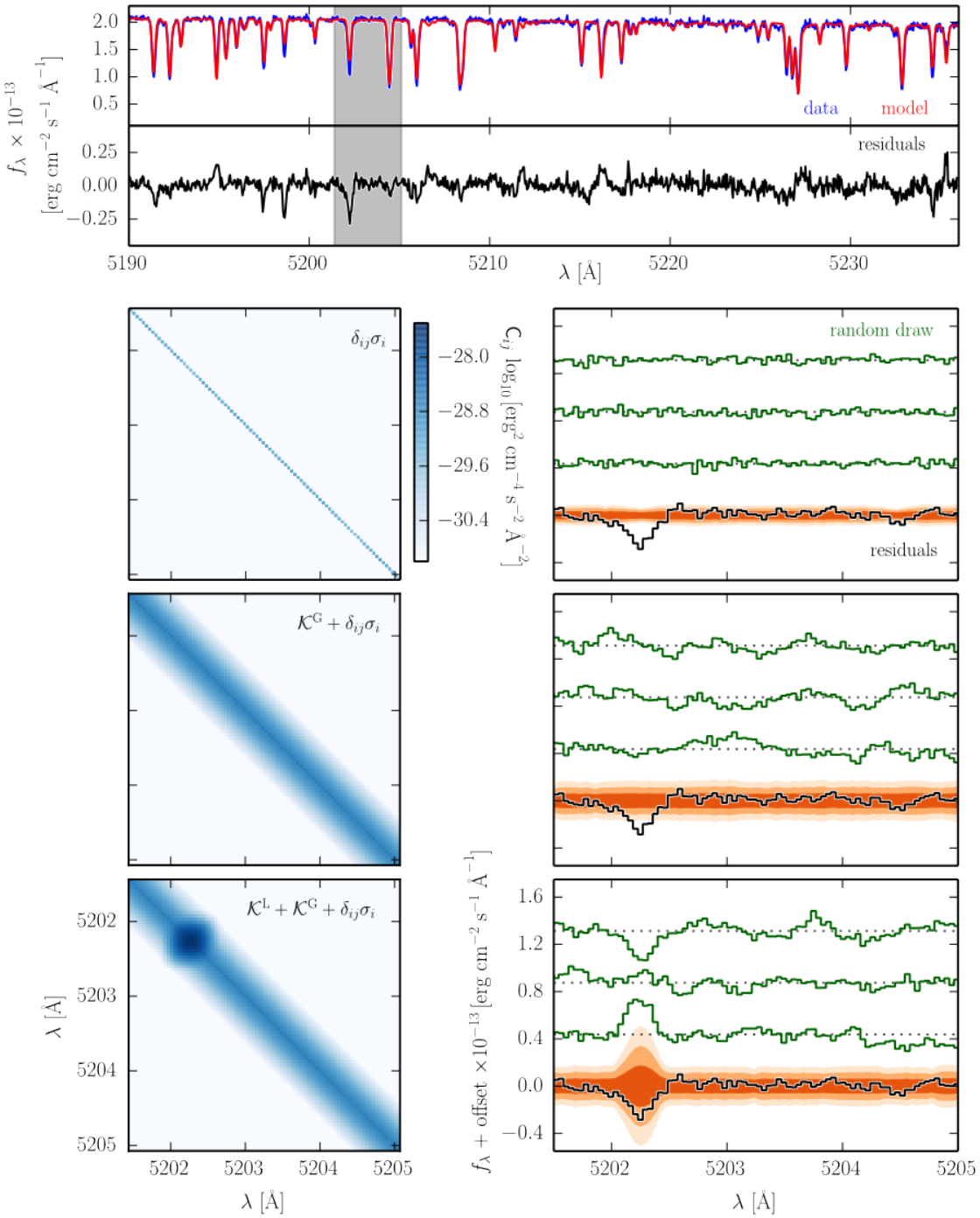
1.1.6 Robust to Outlier Spectral Lines

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 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.


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}]

1.1.7 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.

1.1.8 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.

1.2 Conversion from 0.2

There have been some significant changes to *Starfish* in the upgrades to version 0.3. 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`.

1.2.1 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. 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.

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*.

1.2.2 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. 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*.

1.3 Overview

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.

1.3.1 Citation

If *Starfish* or any derivative of it was used for your work, please cite both the paper and the code. Thanks!

1.3.2 Installation

The source code and installation instructions can be found at the Github repository for Starfish at https://github.com/iancze/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 develop branch

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

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

1.3.3 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.

1.3.4 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*.

1.3.5 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*.

1.3.6 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).

1.3.7 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.

1.4 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

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`.

PHOENIX Interfaces

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

Other Library Interfaces

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

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


```

(continues on next page)

(continued from previous page)

```
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'.

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]`.

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.

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.

Utility Functions

Wavelength conversions

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
```

(continues on next page)

(continued from previous page)

```
    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: -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.models.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

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

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

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.

Data Spectrum

The `DataSpectrum` holds the data spectrum that you wish to fit. You may read your data into this object in a few ways. First let's introduce the object and then discuss the reading methods.

First, you can construct an instance using the traditional `__init__` method:

```
# Read waves, fluxes, and sigmas from your dataset
# as numpy arrays using your own method.
waves, fluxes, sigmas = myownmethod()

myspec = DataSpectrum(waves, fluxes, sigmas)
```

Since `myownmethod()` may require a bunch of additional dependencies (e.g, *IRAF*), for convenience you may want to first read your data using your own custom method but then save it to a different format, like `hdf5`. Since `HDF5` files are all the rage these days, you may want to use them to store your entire data set in a single binary file. If you store your spectra in an HDF5 file as `(norders, npix)` arrays:

```
/
/waves
/fluxes
/sigmas
/masks
```

Then can read your data in as:

```
myspec = DataSpectrum.load("myspec.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.

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.

Transformations

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.DataSpectrum`, and the

methods in *Starfish.models.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*.

Parameterization

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
```

API/Reference

1.5 Examples

Here are where the cookbook and examples live.

1.5.1 Setup

This guide will show you how to get up and running with the grid tools and interfaces provided by *Starfish*.

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

```
import itertools
from Starfish.grid_tools import download_PHOENIX_models

T = [5000, 5100, 5200]
logg = [4.0, 4.5, 5.0]
Z = [0]
params = itertools.product(T, logg, Z)
download_PHOENIX_models(params, base='PHOENIX')
```

We now want to set up a grid interface to work with these downloaded files!

```
from Starfish.grid_tools import PHOENIXGridInterfaceNoAlpha

grid = PHOENIXGridInterfaceNoAlpha(base='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 the TRES spectrograph. We provide many popular spectrographs in our grid tools, including TRES.

Let's also say that, for a given dataset (in our future examples we use WASP 14 so let's consider that), we want to only use a reasonable subset of our original model grid. WASP 14 is currently labeled as an F5V star, so let's create a subset around that classification.

```
from Starfish.grid_tools.instruments import TRES

# Parameters are Teff, logg, and Z
```

(continues on next page)

(continued from previous page)

```
ranges = [
    [5900, 7400],
    [3.0, 6.0],
    [-1.0, 1.0]
]
inst = TRES()
```

Now we can create and process our HDF5Interface

```
from Starfish.grid_tools import HDF5Creator

creator = HDF5Creator(grid, 'F_TRES_grid.hdf5', instrument=inst, ranges=ranges)
creator.process_grid()
```

Setting up the Spectral emulator

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

```
from Starfish.grid_tools import HDF5Interface
from Starfish.emulator import Emulator

grid = HDF5Interface('F_TRES_grid.hdf5')
emu = Emulator.from_grid(grid)
emu.train()
emu.save('F_TRES_emu.hdf5')
```

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.

Once we have our trained emulator, we can move on to the modeling steps for our data.

1.5.2 Example: Single-Order Spectrum

TODO

1.5.3 Example: 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 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

S