

pySYD Documentation

Release 6.10.1a

Ashley Chontos and contributors

Nov 07, 2022

SOFTWARE

1	Getting started	3
1.1	Installation	3
1.2	Dependencies	4
1.3	Setup	5
1.4	Quickstart	6
1.5	Fun	7
2	Crashteroseismology	9
2.1	A quick timeout	9
2.2	A crash course in asteroseismology	9
2.3	Running your favorite star	17
3	pySYD library	19
3.1	Introduction	19
3.2	pySYD inputs	23
3.3	Software modes	27
3.4	Target class	31
3.5	Models & utilities	44
3.6	Saved outputs	54
3.7	What next?	59
3.8	TL;DR	60
4	User guide	61
4.1	Introduction	61
4.2	Single star applications	67
4.3	Star sample	82
4.4	Advanced options	82
4.5	Interactive usage	87
4.6	pySYD option glossary	101
4.7	et al.	111
5	Glossary of documentation terms	113
6	Vision of the pySYD project	117
6.1	Goals	117
7	Attribution	119
7.1	Citations	119
7.2	Projects w/ pySYD	120
8	Contributing	121

8.1	The pySYD team	121
8.2	Community guidelines	122
	Bibliography	125
	Python Module Index	127
	Index	129

Asteroseismology, or the study of stellar oscillations, is a powerful tool for studying the internal structure of stars and determining their fundamental properties. For stars similar to the Sun, turbulent near-surface convection excites sound waves that propagate within the stellar cavity, and hence provides powerful constraints on stellar interiors that are inaccessible by any other means. Asteroseismology is now widely-accepted as the gold standard for the characterization of fundamental stellar properties (e.g., masses, radii, ages, etc.). In an effort to make asteroseismology more accessible to the broader astronomy community, pySYD was developed as a Python package to automatically detect solar-like oscillations and characterize their global properties.

Important: The pySYD documentation is currently being revamped – we apologize in advance for any inconveniences this may cause but appreciate your understanding!

This package is being actively developed in a [public repository on GitHub](#) – we especially welcome and *encourage* any new contributions to help make pySYD better! Please see our [community guidelines](#) to find out how you can help. No contribution is too small!

GETTING STARTED

Jump to quickstart

1.1 Installation

There are three main ways you can install the software:

1. *Install via PyPI*
2. *Create an environment*
3. *Clone directly from GitHub*

Note: The recommended way to install this package is from PyPI via pip, since it will automatically enforce the proper dependencies and versions

1.1.1 Use pip

The pySYD package is available on the Python Package Index ([PyPI](#)) and therefore you can install the latest stable version directly using pip:

```
$ python -m pip install pysyd
```

The pysyd binary should have been automatically placed in your system's path via the pip command. To check the command-line installation, you can use the help command in a terminal window, which should display something similar to the following output:

```
$ pysyd --help

usage: pySYD [-h] [--version] {check,fun,load,parallel,plot,run,setup,test} ...

pySYD: automated measurements of global astero seismic parameters

options:
  -h, --help            show this help message and exit
  --version              Print version number and exit.
```

(continues on next page)

(continued from previous page)

```
pySYD modes:
{check,fun,load,parallel,plot,run,setup,test}
  check      Check data for a target or other relevant information
  fun        Print logo and exit
  load       Load in data for a given target
  parallel    Run pySYD in parallel
  plot       Create and show relevant figures
  run        Run the main pySYD pipeline
  setup      Easy setup of relevant directories and files
  test       Test current installation
```

If your system can not find the pysyd executable, change into the top-level pysyd directory and try running the following command:

```
$ python setup.py install
```

1.1.2 Create an environment

You can also use `conda` to create an environment. For this example, I'll call it 'astero'.

```
$ conda create -n astero numpy scipy pandas astropy matplotlib tqdm
```

See our complete list of dependencies (including versions) *below*. Then activate the environment and install pySYD:

```
$ conda activate astero
$ pip install git+https://github.com/ashleychontos/pySYD
```

1.1.3 Clone from GitHub

If you want to contribute, you can clone the latest development version from [GitHub](https://github.com/ashleychontos/pySYD) using `git`.

```
$ git clone git://github.com/ashleychontos/pySYD.git
```

The next step is to build and install the project:

```
$ python -m pip install .
```

which needs to be executed from the top-level directory inside the cloned pySYD repo.

1.2 Dependencies

This package has the following dependencies:

- [Python](#) (≥ 3)
- [Numpy](#)
- [pandas](#)
- [Astropy](#)

- `scipy`
- `Matplotlib`
- `tqdm`

Explicit version requirements are specified in the project `requirements.txt` and `setup.cfg`. However, using `pip` or `conda` should install and enforce these versions automatically.

1.3 Setup

The software package comes with a convenient setup feature which we **strongly encourage** you to do because it:

- downloads example data for three stars
- provides the properly-formatted [optional] input files *and*
- sets up the relative local directory structure

Note: this step is helpful *regardless* of how you intend to use the software package.

1.3.1 Make a local directory

We recommend to first create a new, local directory to keep all your pysyd-related data, information and results in a single, easy-to-find location. The folder or directory can be whatever is most convenient for you:

```
$ mkdir pysyd
$ cd pysyd
```

1.3.2 Initialize setup

Now all you need to do is change into that directory, run the following command and let pySYD do the rest of the work for you!

```
$ pysyd setup -v
```

We used the *verbose* command so you can see what is being downloaded and where it is being downloaded to.

```
Downloading relevant data from source directory:
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
   Dload  Upload  Total    Spent    Left     Speed
100    25    100    25     0     0    378      0  --:--:-- --:--:-- --:--:--   378
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
   Dload  Upload  Total    Spent    Left     Speed
100   810    100   810     0     0  11739      0  --:--:-- --:--:-- --:--:--  11739
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
   Dload  Upload  Total    Spent    Left     Speed
100  1518k    100  1518k     0     0  8930k      0  --:--:-- --:--:-- --:--:--  8930k
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
   Dload  Upload  Total    Spent    Left     Speed
100  3304k    100  3304k     0     0  11.4M      0  --:--:-- --:--:-- --:--:--  11.4M
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
```

(continues on next page)

(continued from previous page)

	Dload	Upload	Total	Spent	Left	Speed
100 1679k 100 1679k 0 0	9489k	0	--:--:--	--:--:--	--:--:--	9489k
% Total % Received % Xferd	Average Speed	Time	Time	Time	Current	
	Dload	Upload	Total	Spent	Left	Speed
100 3523k 100 3523k 0 0	13.0M	0	--:--:--	--:--:--	--:--:--	13.0M
% Total % Received % Xferd	Average Speed	Time	Time	Time	Current	
	Dload	Upload	Total	Spent	Left	Speed
100 1086k 100 1086k 0 0	7103k	0	--:--:--	--:--:--	--:--:--	7103k
% Total % Received % Xferd	Average Speed	Time	Time	Time	Current	
	Dload	Upload	Total	Spent	Left	Speed
100 2578k 100 2578k 0 0	10.2M	0	--:--:--	--:--:--	--:--:--	10.2M

Note(s):

- created `input` file directory at `/Users/ashleychontos/pysyd/info`
- saved an example of a star `list`
- saved an example `for` the star information file
- created data directory at `/Users/ashleychontos/pysyd/data`
- example data saved to data directory
- results will be saved to `/Users/ashleychontos/pysyd/results`

As shown above, example data and other relevant files were downloaded from the [public GitHub repo](#).

If you forget or accidentally happen to run this again (in the same directory), you will get the following *lovely* reminder:

```
$ pysyd setup -v
```

```
Looks like you've probably done this
before since you already have everything!
```

1.4 Quickstart

Use the following to get up and running right away:

```
$ python -m pip install pysyd
$ mkdir pysyd
$ cd pysyd
$ pysyd setup [optional]
```

The last command which will provide you with example data and files to immediately get going. This is essentially a summary of all the steps discussed on this page but a more consolidated version.

You are now ready to do some asteroseismology!

1.5 Fun

For some extra added fun and just because, type the following in your terminal or command prompt for a little surprise:

```
$ pysyd fun
```



CRASHTEROSEISMOLOGY

2.1 A quick timeout

The examples on this page assume that the user already has some basic-level experience with Python and therefore if not, we highly recommend that you first visit the Python website and going through some of [their tutorials](#) before attempting ours.

We will work through two examples – each demonstrating a different application of the software. The first example will run pySYD as a script from command line since this is what it was optimized for. We will break down each step of the software as much as possible with the hope that it will provide a nice introduction to both the software *and* science. For the second one, we will reconstruct everything in a more condensed version and show pySYD imported and used as a module.

If you have *any* questions, check out our [user guide](#) for more information. If this still does not address your question or problem, please do not hesitate to contact [Ashley](#) directly.

2.2 A crash course in asteroseismology

For purposes of this first example, we'll assume that we don't know anything about the star or its properties so that the software runs from start to finish on its own. In any normal circumstance, however, we can provide additional inputs like the center of the frequency range with the oscillations, or *numax* (ν_{\max}), that can bypass steps and save time.

2.2.1 Initialize script

When running pySYD from command line, you will likely use something similar to the following command:

```
pysyd run --star 1435467 -dv --ux 5000 --mc 200
```

which we will now deconstruct.

pysyd if you used `pip install`, the binary (or executable) should be available. In fact, the setup file defines the entry point for `pysyd`, which is accessed through the `pysyd.cli.main` script – where you can also find all available parsers and commands

run regardless of how you choose to use the software, the most common way you will likely implement pySYD is in run mode – which, just as it sounds, will process stars in order. This is saved to the `args` parser `Namespace` as the `mode`, which will run the pipeline by calling `pysyd.pipeline.run`. There are currently five available (tested) modes (with two more in development), all which are described in more detail [here](#)

- star 1435467** here we are running a single star, KIC 1435467. You can also provide multiple targets, where the stars will append to a list and then be processed consecutively. On the other hand, if no targets are provided, the program would default to reading in the star or todo list (via `info/todo.txt`). Again, this is because the software is optimized for running many stars.
- dv** adapting Linux-like behavior, we reserved the single hash options for booleans which can all be grouped together (as shown above). Here the `-d` and `-v` are short for display and verbose, respectively, and show the figures and verbose output. For a full list of options available, please see our command-line glossary. There are dozens of options to make your experience as customized as you'd like!
- ux 5000** this is an upper frequency limit for the first module that identifies the power eXcess due to solar-like oscillations. In this case, there are high frequency artefacts that we would like to ignore. *We actually made a special notebook tutorial specifically on how to address and fix this problem.* If you'd like to learn more about this or are having a similar issue, please visit this page.
- mc 200** last but certainly not least - the mc (for Monte Carlo-like) option sets the number of iterations the pipeline will run for. In this case, the pipeline will run for 200 steps, which allows us to bootstrap uncertainties on our derived properties.

Note: For a *complete* list of options which are currently available via command-line interface (CLI), see our special CLI *glossary*.

2.2.2 The steps

The software operates in roughly the following steps:

1. *Load in parameters and data*
2. *Search and estimate initial values*
3. *Select best-fit stellar background model*
4. *Fit global parameters*
5. *Estimate uncertainties*

For each step, we will first show the relevant block of printed (or *verbose*) output, then describe what the software is doing behind the scenes and if applicable, conclude with the section-specific results (i.e. files, figures, etc.).

Warning: Please make sure that all input data are in the correct units in order for the software to provide reliable results. If you are unsure, please visit [this page](#) for more information about formatting and input data.

1. Load in parameters and data

```
-----  
Target: 1435467  
-----
```

```
# LIGHT CURVE: 37919 lines of data read  
# Time series cadence: 59 seconds  
# POWER SPECTRUM: 99518 lines of data read  
# PS oversampled by a factor of 5  
# PS resolution: 0.426868 muHz  
-----
```

During this step, it will take the star name along with the command-line arguments and create an instance of the `pysyd.target.Target` object. Initialization of this class will automatically search for and load in data for the given star, as shown in the output above. Both the light curve and power spectrum were available for KIC 1435467 and as you can see in these cases, pySYD will use both arrays to compute additional information like the time series cadence, power spectrum resolution, etc.

If there are issues during the first step, pySYD will flag this and immediately halt any further execution of the code. If something seems questionable during this step but is not fatal for executing the pipeline, it will only return a warning. In fact, all `pysyd.target` class instances will have an `ok` attribute - literally meaning that the star is ‘ok’ to be processed. By default, the pipeline checks this attribute before moving on.

Since none of this happened, we can move on to the next step.

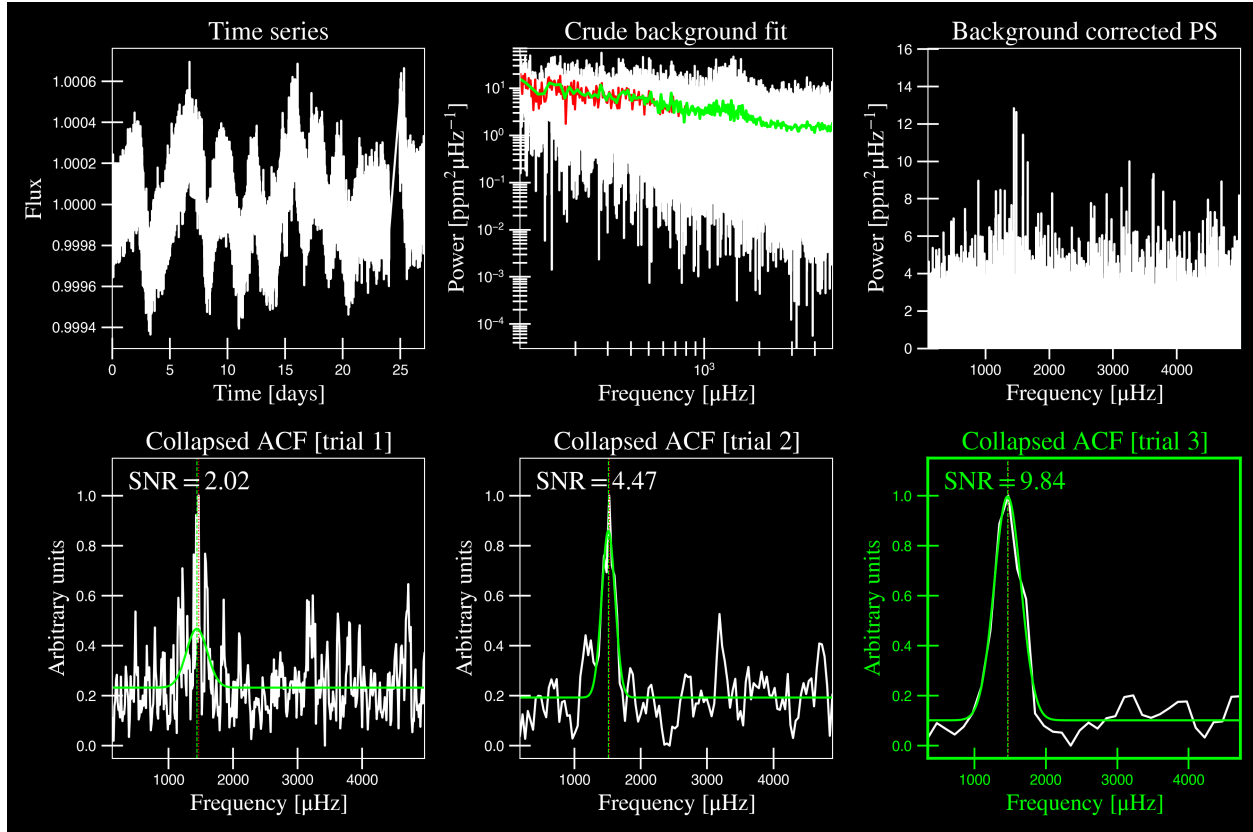
2. Search and estimate initial values

```
-----
PS binned to 228 datapoints

Numax estimates
-----
Numax estimate 1: 1440.07 +/- 81.33
S/N: 2.02
Numax estimate 2: 1513.00 +/- 50.26
S/N: 4.47
Numax estimate 3: 1466.28 +/- 94.06
S/N: 9.84
Selecting model 3
-----
```

The main thing we need to know before performing the global fit is an approximate starting point for the frequency corresponding to maximum power, or *numax* (ν_{\max}). Please read the next section for more information regarding this.

The software first makes a very rough approximation of the stellar background by binning the power spectrum in both log and linear spaces (think a very HEAVY smoothing filter), which the power spectrum is then divided by so that we are left with very little residual slope in the *PS*. The ‘Crude Background Fit’ is shown below in the second panel by the lime green line. The background-corrected power spectrum (*BCPS*) is shown in the panel to the right.



Next pySYD uses a “collapsed” autocorrelation function (*ACF*) technique with different bin sizes to identify localized power excess in the PS due to solar-like oscillations. By default, this is done three times (or trials) and hence, provides three different estimates - which is typically sufficient for these purposes. The bottom row in the above figure shows these three trials, highlighting the one that was selected, or the one with the highest signal-to-noise (S/N).

Finally, it saves the best estimates in a csv file for later use, which can be used to bypass this step the next time that the star is processed.

Table 1: 1435467 parameter estimates

stars	numax	dnu	snr
1435467	1466.27585610943	73.4338977674559	9.84295865829856

3. Select best-fit stellar background model

```

-----
GLOBAL FIT
-----
PS binned to 333 data points

Background model
-----
Comparing 6 different models:
Model 0: 0 Harvey-like component(s) + white noise fixed
  BIC = 981.66 | AIC = 2.95
Model 1: 0 Harvey-like component(s) + white noise term
  
```

(continues on next page)

(continued from previous page)

```

BIC = 1009.56 | AIC = 3.02
Model 2: 1 Harvey-like component(s) + white noise fixed
BIC = 80.27 | AIC = 0.22
Model 3: 1 Harvey-like component(s) + white noise term
BIC = 90.49 | AIC = 0.24
Model 4: 2 Harvey-like component(s) + white noise fixed
BIC = 81.46 | AIC = 0.20
Model 5: 2 Harvey-like component(s) + white noise term
BIC = 94.36 | AIC = 0.23
Based on BIC statistic: model 2
-----

```

A bulk of the heavy lifting is done in this main fitting routine, which is actually done in two separate steps: 1) modeling and characterizing the stellar background and 2) determining the global asteroseismic parameters. We do this *separately* in two steps because they have fairly different properties and we wouldn't want either of the estimates to be influenced by the other in any way.

Ultimately the stellar background has more of a “presence” in the power spectrum in that, dissimilar to solar-like oscillations that are observed over a small range of frequencies, the stellar background contribution is observed over all frequencies. Therefore by attempting to identify where the oscillations are in the power spectrum, we can mask them out to better characterize the background.

We should take a sidestep to explain something important that is happening behind the scenes. A major reason why the predecessor to pySYD, IDL-based SYD, was so successful was because it assumed that the estimated numax and granulation timescales could be scaled with the Sun – a fact that was not known at the time but greatly improved its ability to quickly and efficiently process stars. This is clearly demonstrated in the 2nd and 3rd panels in the figure below, where the initial guesses are strikingly similar to the fitted model.

While this scaling relation ensured great starting points for the background fit, SYD still required a lot fine-tuning by the user. Therefore we adapted the same approach but instead implemented an automated background model selection. After much trial and error, the *BIC* seems to perform better for our purposes - which is now the default metric used (but can easily be changed, if desired).

Measuring the granulation time scales is obviously limited by the total observation baseline of the time series but in general, we can resolve up to 3 Harvey-like components (or laws) at best (for now anyway). For more information about the Harvey model, please see the original paper¹ as well as its application in context .

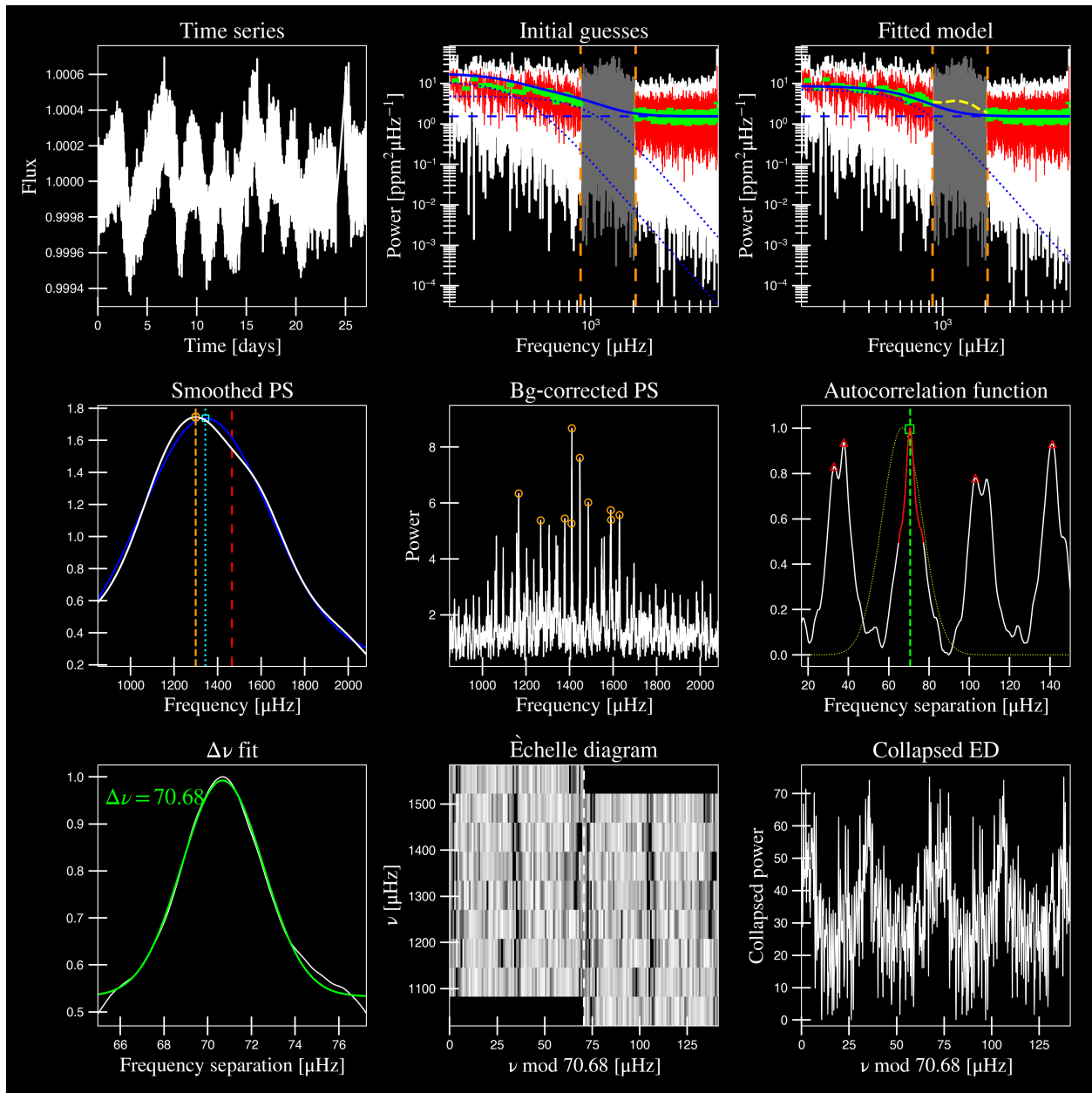
Therefore we use all this information to guess how many we should observe and end up with

$$n_{\text{models}} = 2 \cdot (n_{\text{laws}} + 1)$$

models for a given star. The fact of 2 is because we give the options to fix the white noise or for it to also be a free parameter. The +1 (times 2) is because we also want to consider the simplest model i.e. where we are not able to resolve any. From our perspective, the main purpose of implementing this was to try to identify null detections, since we do not expect to observe oscillations in every star. **However, this is a work in progress and we are still trying various methods to identify and quantify non-detections. Therefore if you have any ideas, please reach out to us!**

For this example we started with two Harvey-like components but the automated model selection preferred a simpler one consisting of a single Harvey law. In addition, the white noise was fixed and *not* a free parameter and hence, the final model had 3 less parameters than it started with. For posterity, we included the output if only a single iteration had been run (which we recommend by default when analyzing a star for the first time).

¹ Harvey (1985)



Note: For more information about what each panel is showing in any of these figures, please visit [this page](#).

4. Fit global parameters

If this was executed with its default mc setting (`== 1`, for a single iteration), the output parameters would look like that shown below. **In fact, we encourage folks to run new stars for a single step first (*ALWAYS*) before running it several iterations to make sure everything checks out.**

```
-----
Output parameters
-----
```

```
numax_smooth: 1299.81 muHz
A_smooth: 1.74 ppm^2/muHz
numax_gauss: 1344.46 muHz
A_gauss: 1.50 ppm^2/muHz
FWHM: 294.83 muHz
dnu: 70.68 muHz
tau_1: 234.10 s
sigma_1: 87.40 ppm
-----
```

- displaying figures
 - press RETURN to exit
 - combining results into single csv file
- ```

```

**Reminder:** the printed output above is for posterity. Please see the next section in the event that you are comparing outputs to test the software functionality.

The parameters are printed and saved in identical ways (sans the uncertainties).

Table 2: 1435467 global parameters

| parameter    | value            | uncertainty |
|--------------|------------------|-------------|
| numax_smooth | 1299.81293631    | —           |
| A_smooth     | 1.74435577479371 | —           |
| numax_gauss  | 1344.46209203309 | —           |
| A_gauss      | 1.49520571806361 | —           |
| FWHM         | 294.828524961042 | —           |
| dnu          | 70.6845197924864 | —           |
| tau_1        | 234.096929937095 | —           |
| sigma_1      | 87.4003388623678 | —           |

## 5. Estimate uncertainties

```

Sampling routine:
```

```
100%| 200/200 [00:21<00:00, 9.23it/s]

```

```
Output parameters

```

```
numax_smooth: 1299.81 +/- 56.64 muHz
A_smooth: 1.74 +/- 0.19 ppm^2/muHz
numax_gauss: 1344.46 +/- 41.16 muHz
A_gauss: 1.50 +/- 0.24 ppm^2/muHz
FWHM: 294.83 +/- 64.57 muHz
```

(continues on next page)

(continued from previous page)

```

dnu: 70.68 +/- 0.82 muHz
tau_1: 234.10 +/- 23.65 s
sigma_1: 87.40 +/- 2.81 ppm

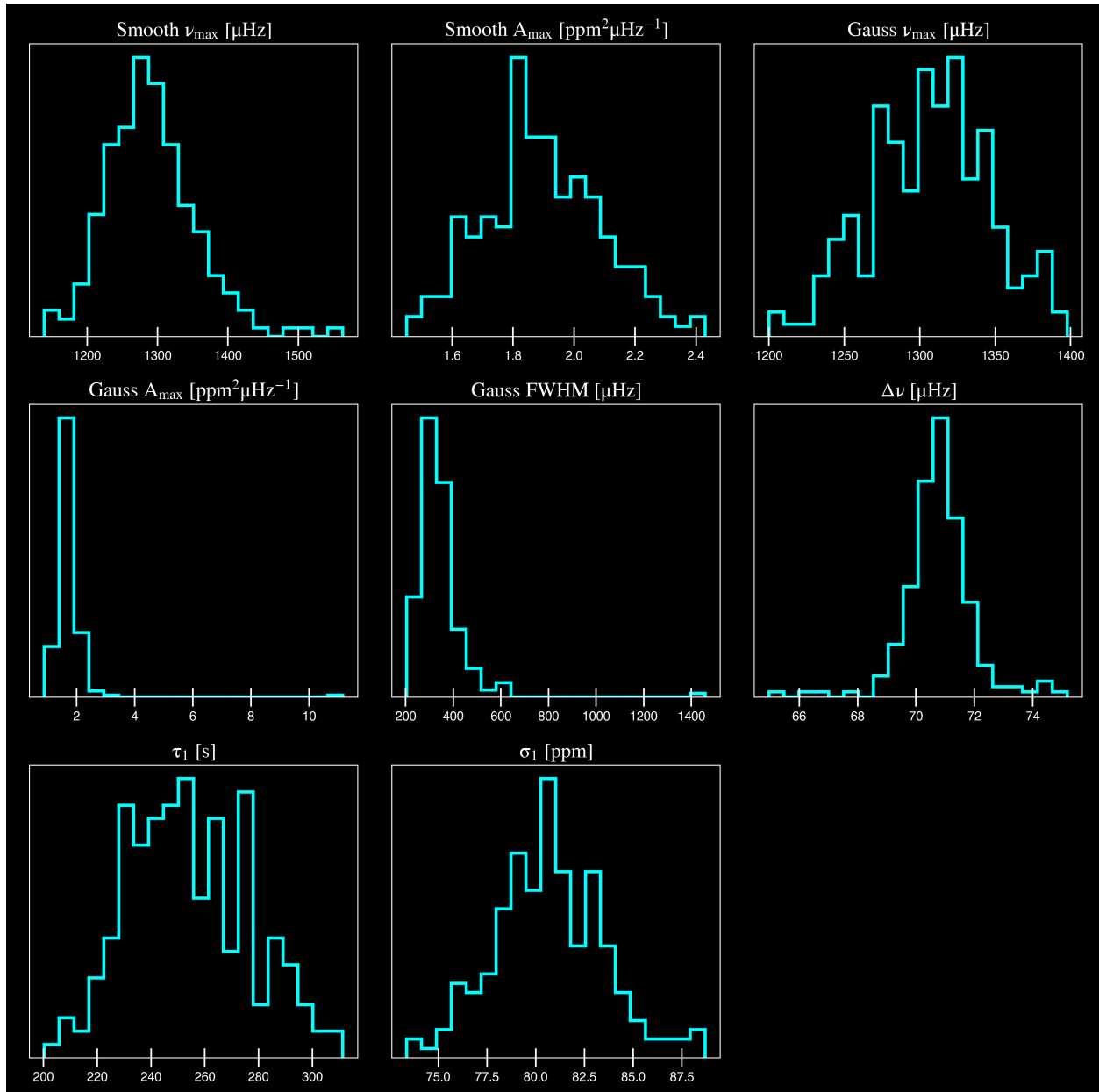
```

- ```

-----
- displaying figures
- press RETURN to exit
- combining results into single csv file
-----

```

Notice the difference in the printed parameters this time - which now have uncertainties!



^^ The figure above shows parameter posteriors for KIC 1435467. Sampling results can be saved by using the boolean flag `-z` or `--samples`, which will store the samples for the fitted parameters as comma-separated values using

pandas.

Table 3: 1435467 global parameters

parameter	value	uncertainty
numax_smooth	1299.81293631	56.642346824238
A_smooth	1.74435577479371	0.191605473120388
numax_gauss	1344.46209203309	41.160592041828
A_gauss	1.49520571806361	0.236092716197938
FWHM	294.828524961042	64.57265346103
dnu	70.6845197924864	0.821246814829682
tau_1	234.096929937095	23.6514289023765
sigma_1	87.4003388623678	2.81297225855344

- matches expected output for model 4 selection - notice how there is no white noise term

in the output. this is because the model preferred for this to be fixed

Note: While observations have shown that solar-like oscillations have an approximately Gaussian-like envelope, we have no reason to believe that they should behave exactly like that. This is why you will see two different estimates for ν_{\max} under the output parameters. **In fact for this methodology first demonstrated in Huber+2009, the smoothed numax value is what has been reported in the literature and should also be the adopted value here.**

2.3 Running your favorite star

Initially all defaults were set and saved from the command line parser but we recently extended the software capabilities – which means that it is more user-friendly and how you choose to use it is now completely up to you!

Alright so let's import the package for this example.

```
>>> from pysyd import utils
>>> name = '1435467'
>>> args = utils.Parameters(stars=[name])
>>> args
<PySYD Parameters>
```

Analogous to the command-line arguments, we have a container class (`pysyd.utils.Parameters`) that can easily be loaded in and modified to the user's needs. There are two keyword arguments that the Parameter object accepts – `args` and `stars` – both which are `None` by default. In fact, the `Parameters` class was also initialized in the first example but immediately knew it was executed as a script instead because `args` was *not* `None`.

As shown in the third line, we put the star list in list form **even though we are only processing a single star**. This is because both pySYD modes that process stars iterate through the lists, so we feed it a list that is iterable so it doesn't get confused.

Now that we have our parameters, let's create a pipeline `pysyd.target.Target` object to process.

```
>>> from pysyd.target import Target
>>> star = Target(name, args)
>>> star
<Star 1435467>
```

Instantiation of a `Target` star automatically searches for and loads in available data (based on the given ‘name’). This step will therefore flag anything that doesn’t seem right i.e., data is missing or paths are not correct.

Finally, before we process the star, we will need to adjust a couple settings so that it runs similarly to the first example (sans the boolean flags).

```
>>> star.params['upper_ex'] = 5000.  
>>> star.params['mc_iter'] = 200
```

Ok now that we have our desired settings and target, we can go ahead and process the star (which is fortunately a one-liner in this case):

```
>>> star.process_star()
```

And that’s it. If you ran it on the same star, the output figures and parameters should exactly match.

PYSYD LIBRARY

Thanks for stopping by the pySYD documentation and taking an interest in learning more about how it all works – we are so *thrilled* to share asteroseismology with you!

3.1 Introduction

pySYD was initially established as a one-to-one translation of the IDL-based SYD pipeline from Huber et al. (2009). In the *Kepler* days, SYD was extensively used to measure global asteroseismic parameters for many stars (e.g., [H2011]; [C2014]; [S2017a]; [Y2018]).

In order to process and analyze the enormous amounts of data from *Kepler* in real time, there were a *handful of other closed-source pipelines* developed around the same time that perform roughly similar types of analyses. In fact, there were several papers that compared results from each of these pipelines in order to ensure the reproducibility of science results from the *Kepler* legacy sample ([L2017]; [S2017b]).

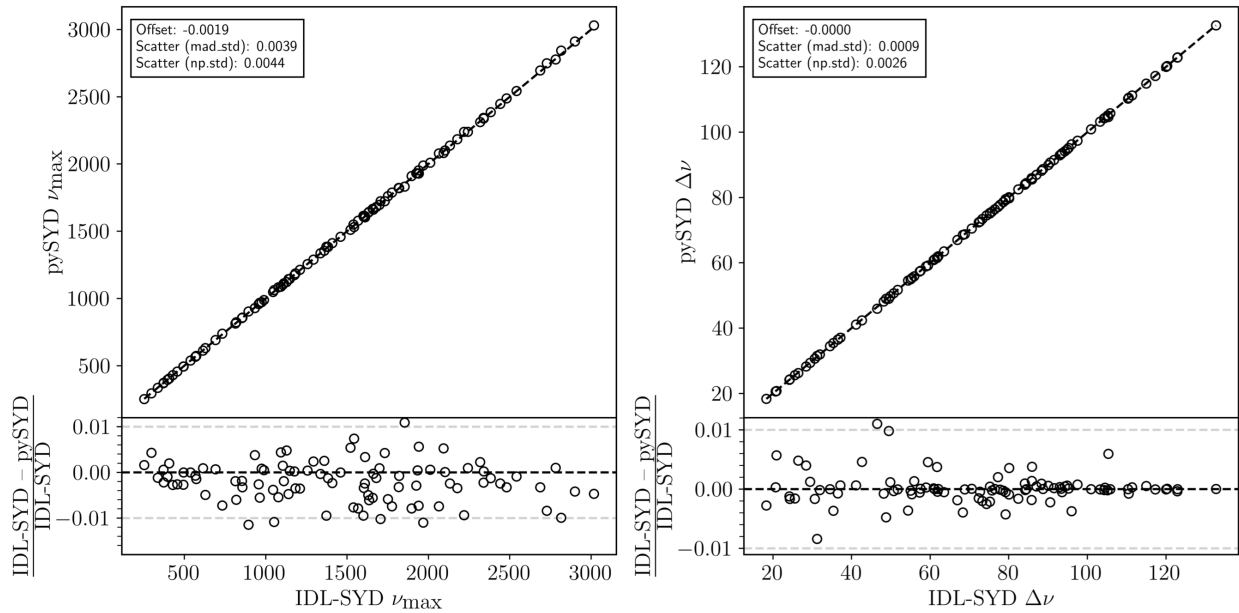
pySYD adapts the well-tested methodology from SYD while simultaneously improving these existing analyses and expanding upon numerous new features. Some improvements include:

- automated background model comparison and selection
- parallel processing and other easy compatibilities for running many stars
- easily customizable with command-line friendly interface
- modular and adaptable across different applications
- saves reproducible samples for future analyses (i.e. seeds)

3.1.1 Benchmarking to the *Kepler* legacy sample

We ran pySYD on ~100 *Kepler* legacy stars (defined [here](#)) observed in short-cadence and compared the output to SYD results from [S2017a]. The same time series and power spectra were used for both analyses, which are publicly available and hosted online c/o KASOC¹. The resulting values are compared for the two methods below for *numax* (ν_{\max} , left) and *dnu* ($\Delta\nu$, right).

¹ Kepler Asteroseismic Science Operations Center



The residuals show no strong systematics to within $<0.5\%$ in Dnu and $<\sim 1\%$ in numax , which is smaller than the typical random uncertainties. This confirms that the open-source Python package pySYD provides consistent results with the legacy IDL version that has been used extensively in the literature.

3.1.2 Related Tools

pySYD provides general purpose tools for performing asteroseismic analysis in the frequency domain. Several tools have been developed to solve related scientific and data analysis problems. We have compiled a list of software packages that perform similar or complementary analyses.

- **AMP:**
 - language:
 - reference:
 - documentation: no
 - publicly available: no
 - requires license: n/a
- **A2Z: determining global parameters of the oscillations of solar-like stars**
 - language: ?
 - reference: [yes](#)
 - documentation: no
 - publicly available: no
 - requires license: n/a
- **Background: an extension of DIAMONDS that fits the background signal of solar-like oscillators**
 - language: c++11

- reference: no
 - documentation: no
 - publicly available: [yes](#)
 - requires license: no
- **CAN: on the detection of Lorentzian profiles in a power spectrum**
 - language: ?
 - reference: [yes](#)
 - documentation: no
 - publicly available: no
 - requires license: n/a
- **COR: on detecting the large separation in the autocorrelation of stellar oscillation times series**
 - language: ?
 - reference: [yes](#)
 - documentation: no
 - publicly available: no
 - requires license: n/a
- **DIAMONDS: high-DImensional And multi-MOdal NESTE D Sampling**
 - language: c++11
 - reference: [yes](#)
 - documentation: [yes](#)
 - publicly available: [yes](#)
 - requires license: n/a
- **DLB:**
 - language: ?
 - reference: no
 - documentation: n/a
 - publicly available: no
 - requires license: n/a
- **FAMED: Fast & AutoMated pEakbagging with Diamonds**
 - language: IDL (currently being developed in Python)
 - reference: [yes](#)
 - documentation: [yes](#)
 - publicly available: [yes](#)
 - requires license: yes
- **Flicker Flipper?:**
 - language:

- reference:
 - documentation:
 - publicly available:
 - requires license: n/a
- **KAB: automated asteroseismic analysis of solar-type stars**
 - language: ?
 - reference: [yes](#)
 - documentation: no
 - publicly available: no
 - requires license: n/a
- **lightcurve: a friendly Python package for making discoveries with *Kepler* & TESS**
 - language: Python
 - reference: no
 - documentation: [yes](#)
 - publicly available: [yes](#)
 - requires license: no
- **OCT: automated pipeline for extracting oscillation parameters of solar-like main-sequence stars**
 - language: ?
 - reference: [yes](#)
 - documentation: no
 - publicly available: no
 - requires license: n/a
- **ORK: using the comb response function method to identify spacings**
 - language: ?
 - reference: [yes](#)
 - documentation: no
 - publicly available: no
 - requires license: n/a
- **QML: a power-spectrum autocorrelation technique to detect global asteroseismic parameters**
 - language: ?
 - reference: [yes](#)
 - documentation: no
 - publicly available: no
 - requires license: n/a
- **PBJam: a python package for automating asteroseismology of solar-like oscillators**
 - language: Python

- reference: [yes](#)
 - documentation: [yes](#)
 - publicly available: [yes](#)
 - requires license: no
- **SYD: automated extraction of oscillation parameters for *Kepler* observations of solar-type stars**
 - language: IDL
 - reference: [yes](#)
 - documentation: no
 - publicly available: no
 - requires license: yes

Important: If your software is not listed or if something listed is incorrect/missing, please open a pull request to add it, we aim to be inclusive of all *Kepler*-, K2- and TESS- related tools!

3.1.3 References

3.2 pySYD inputs

For what it's worth and if you haven't done so already, running the pySYD *setup feature* will conveniently provide *all* of files which are discussed in detail on this page.

3.2.1 Required

The only thing that's really *required* is the data.

For a given star ID, possible input data are its:

1. light curve ('ID_LC.txt') and/or
2. power spectrum ('ID_PS.txt').

Light curve: The *Kepler*, K2 & TESS missions have provided *billions* of stellar light curves, or a measure of the object's brightness (or flux) in time. Like most standard photometric data, we require that the time array is in units of days. **This is really important if the software is calculating the power spectrum for you!** The y-axis is less critical here – it can be anything from units of fraction flux or brightness as a function of time, along with any other normalization(s). **Units:** time (days) vs. normalized flux (ppm)

Power spectrum: the frequency series or *power spectrum* is what's most important for the asteroseismic analyses applied and performed in this software. Thanks to open-source languages like Python, we have many powerful community-driven libraries like *astropy* that can fortunately compute these things for us. **Units:** frequency (μHz) vs. power density ($\text{ppm}^2\mu\text{Hz}^{-1}$)

Cases

Therefore for a given star, there are four different scenarios that arise from a combination of these two inputs and we describe how the software handles each of these cases.

Additionally, we will list these in the recommended order, where the top is the most preferred and the bottom is the least.

Case 1: light curve *and* power spectrum

Here, everything can be inferred and/or calculated from the data when both are provided. This includes the time series *cadence*, which is relevant for the *nyquist frequency*, or how high our sampling rate is. The total duration of the time series sets an upper limit on the time scales we can measure and also sets the resolution of the power spectrum. Therefore from this, we can determine if the power spectrum is oversampled or critically-sampled and make the appropriate arrays for all input data.

The following are attributes saved to the `pysyd.target.Target` object in this scenario:

- Parameter(s):
 - time series cadence (`star.cadence`)
 - nyquist frequency (`star.nyquist`)
 - total time series length or baseline (`star.baseline`)
 - upper limit for granulation time scales (`star.tau_upper`)
 - frequency resolution (`star.resolution`)
 - oversampling factor (`star.oversampling_factor`)
- Array(s):
 - time series (`star.time` & `star.flux`)
 - power spectrum (`star.frequency` & `star.power`)
 - copy of input power spectrum (`star.freq_os` & `star.pow_os`)
 - critically-sampled power spectrum (`star.freq_cs` & `star.pow_cs`)

Issue(s)

1. the only problem that can arise from this case is if the power spectrum is not normalized correctly or in the proper units (i.e. frequency is in μHz and power is in $\text{ppm}^2\mu\text{Hz}^{-1}$). This is actually more common than you think so if this *might* be the case, we recommend trying CASE 2 instead

Case 2: light curve *only*

Again we can determine the baseline and cadence, which set important features in the frequency domain as well. Since the power spectrum is not yet calculated, we can control if it's oversampled or critically-sampled. So basically for this case, we can calculate all the same things as in Case 1 *but* we just have a few more steps that may take a little more time to do.

The following are attributes saved to the `pysyd.target.Target` object in this scenario:

- Parameter(s):
 - time series cadence (`star.cadence`)
 - nyquist frequency (`star.nyquist`)

- total time series length or baseline (`star.baseline`)
- upper limit for granulation time scales (`star.tau_upper`)
- frequency resolution (`star.resolution`)
- oversampling factor (`star.oversampling_factor`)
- Array(s):
 - time series (`star.time` & `star.flux`)
 - newly-computed power spectrum (`star.frequency` & `star.power`)
 - copy of oversampled power spectrum (`star.freq_os` & `star.pow_os`)
 - critically-sampled power spectrum (`star.freq_cs` & `star.pow_cs`)

Issue(s)

1.

Case 3: power spectrum *only*

This case can be *o-k*, so long as additional information is provided.

Calculation(s)

- Parameter(s):
- Array(s):

Issue(s)

1.

Issue(s): 1) if oversampling factor not provided

2) if not normalized properly

Case 4: no data

well, we all know what happens when zero input is provided... but just in case, this will raise a `PySYDInputError`

CASE 1: light curve and power spectrum - Summary: - Calculation(s):

- time series cadence (Δt)
- nyquist frequency (ν_{nyq})
- time series duration or baseline (ΔT)
- frequency resolution ($\Delta \text{frequency}$)
- oversampling factor (i.e. critically-sampled has an `of=1`)
- critically-sampled power spectrum
- **Issue(s):**
 - the only problem that can arise from this case is if the power spectrum is not normalized correctly or in the proper units (i.e. frequency is in μHz and power is in $\text{ppm}^2 \mu\text{Hz}^{-1}$). This is actually more common than you think so if this *might* be the case, we recommend trying CASE 2 instead.

CASE 2: light curve *only* - summary: Again we can determine the baseline and cadence, which set important features in the

frequency domain as well. Since the power spectrum is not yet calculated, we can control if it's oversampled or critically-sampled

CASE 3: power spectrum *only* This case *can* be alright, as long as additional information is provided. Issue(s): 1) if oversampling factor not provided

2) if not normalized properly

Important: For the saved power spectrum, the frequency array has units of μHz and the power array is power density, which has units of $\text{ppm}^2 \mu\text{Hz}^{-1}$. We normalize the power spectrum according to Parseval's Theorem, which loosely means that the fourier transform is unitary. This last bit is incredibly important for two main reasons, but both that tie to the noise properties in the power spectrum: 1) different instruments (e.g., *Kepler*, TESS) have different systematics and hence, noise properties, and 2) the amplitude of the noise becomes smaller as your time series gets longer. Therefore when we normalize the power spectrum, we can make direct comparisons between power spectra of not only different stars, but from different instruments as well!

3.2.2 Optional

There are two main information files that can be provided but both are optional – whether you choose to use them or not is ultimately up to you!

Target list

For example, providing a star list via a basic text file is convenient for running a large sample of stars. We provided an example with the rest of the setup, but essentially all it is is a list with one star ID per line. The star ID *must* match the same ID associated with the data.

```
$ cat todo.txt
11618103
2309595
1435467
```

Note: If no stars are specified via command line or in a notebook, pySYD will read in this text file and process the list of stars by default.

Star info

As suggested by the name of the file, this contains star information on an individual basis. Similar to the data, target IDs must *exactly* match the given name in order to be successfully crossmatched – but this also means that the information in this file need not be in any particular order.

Below is a snippet of what the csv would look like:

Table 1: Star info

stars	rs	logg	teff	numax	lower_se	upper_se	lower_bg
1435467					100.0	5000.0	100.0
2309595					100.0		100.0

Just like the input data, the stars *must* match their ID but also, the commands must adhere to a special format. In fact, the columns in this csv are exactly equal to the value (or destination) that the command-line parser saves each option to. Since there are a ton of available columns, we won't list them all here but there are a few ways you can view the columns for yourself.

The first is by visiting our special command-line glossary, which explicitly states how each of the variables is defined. You can also see them fairly easily by importing the `pysyd.utils.get_dict` module and doing a basic `print` statement.

```
>>> from pysyd import utils
>>> columns = utils.get_dict('columns')
>>> print(columns['all'])
['rs', 'rs_err', 'teff', 'teff_err', 'logg', 'logg_err', 'cli', 'inpdire',
 'infdire', 'outdir', 'overwrite', 'show', 'ret', 'save', 'test', 'verbose',
 'dnu', 'gap', 'info', 'ignore', 'kep_corr', 'lower_ff', 'lower_lc', 'lower_ps',
 'mode', 'notching', 'oversampling_factor', 'seed', 'stars', 'todo', 'upper_ff',
 'upper_lc', 'upper_ps', 'stitch', 'n_threads', 'ask', 'binning', 'bin_mode',
 'estimate', 'adjust', 'lower_se', 'n_trials', 'smooth_width', 'step',
 'upper_se', 'background', 'basis', 'box_filter', 'ind_width', 'n_laws',
 'lower_bg', 'metric', 'models', 'n_rms', 'upper_bg', 'fix_wn', 'functions',
 'cmap', 'clip_value', 'fft', 'globe', 'interp_ech', 'lower_osc', 'mc_iter',
 'nox', 'noy', 'npb', 'n_peaks', 'numax', 'osc_width', 'smooth_ech', 'sm_par',
 'smooth_ps', 'threshold', 'upper_osc', 'hey', 'samples']
>>> len(columns['all'])
77
```

Note: This file is *especially* helpful for running many stars with different options - you can make your experience as customized as you'd like!

3.3 Software modes

3.3.1 Running as a script

When running the software, pySYD will look in the following paths:

- INFDIR : '~/path/to/local/pysyd/directory/info'
- INPDIR : '~/path/to/local/pysyd/directory/data'
- OUTDIR : '~/path/to/local/pysyd/directory/results'

which by default, is the absolute path of the current working directory (or wherever you ran setup from). We will eventually add an option to save the example data and files as package data (since it is easier to load in)

Important: when running `pysyd` as a script, there is one positional argument for the pipeline “mode”.

3.3.2 Importing as a module

3.3.3 Pipeline overview

The software generally operates in four main steps:

1. Loads in parameters and data
 2. Gets initial values
 3. Fits global parameters
 4. Estimates uncertainties
-

Note: The software will process the pipeline on oversampled spectra for the first iterations but will *always* switch to critically-sampled spectra for estimating uncertainties. **Calculating uncertainties with oversampled spectra can produce unreliable results and uncertainties!**

A pySYD pipeline Target class object has two main function calls:

1. **The first module :**

- **Summary:** a crude, quick way to identify the power excess due to solar-like oscillations
- This uses a heavy smoothing filter to divide out the background and then implements a frequency-resolved, collapsed autocorrelation function (ACF) using 3 different box sizes
- The main purpose for this first module is to provide a good starting point for the second module. The output from this routine provides a rough estimate for numax, which is translated into a frequency range in the power spectrum that is believed to exhibit characteristics of p-mode oscillations

2. **The second module :**

- **Summary:** performs a more rigorous analysis to determine both the stellar background contribution as well as the global asteroseismic parameters.
- Given the frequency range determined by the first module, this region is masked out to model the white- and red-noise contributions present in the power spectrum. The fitting procedure will test a series of models and select the best-fit stellar background model based on the BIC.
- The power spectrum is corrected by dividing out this contribution, which also saves as an output text file.
- Now that the background has been removed, the global parameters can be more accurately estimated. Numax is estimated by using a smoothing filter, where the peak of the heavily smoothed, background-corrected power spectrum is the first and the second fits a Gaussian to this same power spectrum. The smoothed numax has typically been adopted as the default numax value reported in the literature since it makes no assumptions about the shape of the power excess.
- Using the masked power spectrum in the region centered around numax, an autocorrelation is computed to determine the large frequency spacing.

Note: By default, both modules will run and this is the recommended procedure if no other information is provided.

If stellar parameters like the radius, effective temperature and/or surface gravity are provided in the `info/star_info.csv` file, pySYD can estimate a value for numax using a scaling relation. Therefore the first module can be bypassed, and the second module will use the estimated numax as an initial starting point.

There is also an option to directly provide numax in the **info/star_info.csv** (or via command line, see advanced usage for more details), which will override the value found in the first module. This option is recommended if you think that the value found in the first module is inaccurate, or if you have a visual estimate of numax from the power spectrum.

3.3.4 Introduction

This was initially created to be used with the command-line tool but has been expanded to include functions compatible with Python notebooks as well. This is still a work in progress!

3.3.5 Imports

3.3.6 Pipeline modes

There are currently four operational pySYD modes (and two under development):

1. **setup** : Initializes `pysyd.pipeline.setup` for quick and easy setup of directories, files and examples. This mode only inherits higher level functionality and has limited CLI (see parent parser below). Using this feature will set up the paths and files consistent with what is recommended and discussed in more detail below.
2. **load** : Loads in data for a single target through `pysyd.pipeline.load`. Because this does handle data, this has full access to both the parent and main parser.
3. **run** : The main pySYD pipeline function is initialized through `pysyd.pipeline.run` and runs the two core modules (i.e. `find_excess` and `fit_background`) for each star consecutively. This mode operates using most CLI options, inheriting both the parent and main parser options.
4. **parallel** : Operates the same way as the previous mode, but processes stars simultaneously in parallel. Based on the number of threads available, stars are separated into groups (where the number of groups is exactly equal to the number of threads). This mode uses all CLI options, including the number of threads to use for parallelization (see here).
5. **display** : will primarily be used for development and testing purposes as well, but
6. **test** : Currently under development but intended for developers.

3.3.7 Examples

3.3.8 pysyd.pipeline API

`pysyd.pipeline.check(args)`

Check target

This is intended to be a way to check a target before running it by plotting the times series data and/or power spectrum. This works in the most basic way but has not been tested otherwise

Parameters

args [argparse.Namespace] the command line arguments

Important: has not been extensively tested - also it is exactly the same as “load” so think through if this is actually needed and decided which is easier to understand

`pysyd.pipeline.fun(args)`

Get logo output

Parameters

args [argparse.Namespace] the command line arguments

`pysyd.pipeline.load(args)`

Load target

Load in a given target to check data or figures

Note: this does *not* load in a target or target data, this is purely information that is required to run any pySYD mode successfully (with the exception of `pysyd.pipeline.setup`)

Parameters

args [argparse.Namespace] the command line arguments

`pysyd.pipeline.parallel(args)`

Parallel execution

Run pySYD concurrently for a large number of stars

Parameters

args [argparse.Namespace] the command line arguments

Methods `pipe`

See also:

[`pysyd.pipeline.run`](#)

`pysyd.pipeline.plot(args)`

Make plots

Module to load in all relevant information and dictionaries required to run the pipeline

Note: this does *not* load in a target or target data, this is purely information that is required to run any pySYD mode successfully (with the exception of `pysyd.pipeline.setup`)

Parameters

args [argparse.Namespace] the command line arguments

`pysyd.pipeline.run(args)`

Run pySYD

Main function to initiate the pySYD pipeline for one or many stars (the latter is run consecutively *not* concurrently)

Parameters

args [argparse.Namespace] the command line arguments

Methods [`pysyd.utils.Parameters`](#) `pysyd.pipeline.pipe` `pysyd.utils._scrape_output`

See also:

[`pysyd.pipeline.parallel`](#)

`pysyd.pipeline.setup(args)`

Quick software setup

Running this after installation will create the appropriate directories in the current working directory as well as download example data and files to test your pySYD installation

Parameters

args [`argparse.Namespace`] the command line arguments

note [`str`, optional] suppressed (optional) verbose output

raw [`str`] path to download “raw” package data and examples from the pySYD source directory

3.4 Target class

The heart & soul of the pySYD package

3.4.1 Introduction

At some point, every given star will be processed as a `pysyd.target.Target` object. This is where a bulk of the scientific data analysis is done.

3.4.2 Imports

3.4.3 Usage

3.4.4 `pysyd.target` API

class `pysyd.target.Target(name, args)`

Main pipeline target object

Deprecated since version 1.6.0: `Target.ok` will be removed in pySYD 6.0.0, it is replaced by new error handling, that will instead raise exceptions or warnings

A new instance (or star) is created for each target that is processed. Instantiation copies the relevant, individual star dictionary (and the inherited constants) and will then load in data using the provided star name

Parameters

name [`str`] which target to load in and/or process

args [`pysyd.utils.Parameters`] container class of pysyd parameters

Attributes

params [`Dict`] copy of `args.params[name]` dictionary with pysyd parameters and options

check_numax(`columns=[numax, dnu, snr]`)

Check ν_{\max}

Checks if there is an initial starting point or estimate for *numax*

Parameters

columns [List[str]] saved columns if the `estimate_numax()` function was run

Raises

utils.InputError if an invalid value was provided as input for `numax`

utils.ProcessingError if it still cannot find any estimate for *numax*

collapse_ed(*n_trials=3*)

Get ridges

Optimizes the large frequency separation by determining which spacing creates the “best” ridges (but is currently under development) think similar to a step-echelle but quicker and more hands off?

Attributes

x [numpy.ndarray] x-axis for the collapsed ED $\sim [0, 2 \times \Delta\nu]$

y [numpy.ndarray] marginalized power along the y-axis (i.e. collapsed on to the x-axis)

Important: need to optimize this - currently does nothing really

compute_acf(*fft=True, smooth_ps=2.5*)

ACF

Compute the autocorrelation function (*ACF*) of the background-divided power spectrum (i.e. `bg_corr`), with an option to smooth the *BCPS* first

Parameters

fft [bool, default=True] if **True**, uses FFTs to compute the ACF, otherwise it will use `numpy.correlate`

smooth_ps [float, optional] convolve the background-corrected PS with a box filter of this width (μHz)

Attributes

bgcorr_smooth [numpy.ndarray] smoothed background-corrected power spectrum if `smooth_ps != 0` else copy of `bg_corr`

lag, auto [numpy.ndarray, numpy.ndarray] the autocorrelation of the “zoomed-in” power spectrum

compute_spectrum(*oversampling_factor=1, store=False*)

Compute power spectrum

NEW function to calculate a power spectrum given time series data, which will normalize the power spectrum to spectral density according to Parseval’s theorem

Parameters

oversampling_factor [int, default=1] the oversampling factor to use when computing the power spectrum

store [bool, default=False] if **True**, it will store the original data arrays for plotting purposes later

Yields

frequency, power [numpy.ndarray, numpy.ndarray] power spectrum computed from the input time series data (i.e. `time` & `flux`) using the `astropy.timeseries.LombScargle` module

Returns

frequency, power [numpy.ndarray, numpy.ndarray] the newly-computed and normalized power spectrum (in units of μHz vs. $\text{ppm}^2\mu\text{Hz}^{-1}$)

Important: If you are unsure if your power spectrum is in the proper units, we recommend using this new module to compute and normalize for you. This will ensure the accuracy of the results.

correct_background(metric='bic')

Correct background

Corrects for the stellar background contribution in the power spectrum by *both* dividing and subtracting this out, which also saves copies of each (i.e. `bg_div` *background-divided power spectrum* to ID_BDPS.txt and `bg_sub` background-subtracted power spectrum to :ref:`ID_BSPS.txt`). After this is done, a copy of the BDPS is saved to `bg_corr` and used for dnu calculations and the echelle diagram.

Parameters

metric [str, default='bic'] which metric to use (i.e. bic or aic) for model selection

Attributes

frequency, bg_div [numpy.ndarray, numpy.ndarray] background-divided power spectrum (BDPS -> higher S/N for echelle diagram)

frequency, bg_sub [numpy.ndarray, numpy.ndarray] background-subtracted power spectrum (BSPS -> preserves mode amplitudes)

frequency, bg_corr [numpy.ndarray, numpy.ndarray] background-corrected power spectrum, which is a copy of the *BDPS*

derive_parameters(mc_iter=1)

Derive parameters

Main function to derive the background and global asteroseismic parameters (including uncertainties when relevant), which does everything from finding the initial estimates to plotting/saving results

Parameters

mc_iter [int, default=1] the number of iterations to run

Methods

pysyd.target.Target.check_numax first checks to see if there is a valid estimate or input value provides for numax

pysyd.target.Target.initial_parameters if so, it will estimate the rest of the initial guesses required for the background and global fitting (primarily using solar scaling relations)

pysyd.target.Target.first_step the first iteration determines the best-fit background model and global properties

pysyd.target.Target.get_samples bootstrap uncertainties by attempting to recover the parameters from the first step

echelle_diagram(smooth_ech=None, nox=None, noy='0+0', hey=False, npb=10, nshift=0, clip_value=3.0)

Echelle diagram

Calculates everything required to plot an *echelle diagram* **Note:** this does not currently have the `get_ridges` method attached (i.e. not optimizing the spacing or stepchelle)

Parameters

smooth_ech [float, default=None] value to smooth (i.e. convolve) ED by
nox [int, default=0] number of grid points in x-axis of echelle diagram
noy [str, default='0+0'] number of orders (y-axis) to plot in echelle diagram
npb [int, default=10] option to provide the number of points per bin as opposed to an arbitrary value (calculated from spacing and frequency resolution)
nshift [int, default=0] number of orders to shift echelle diagram (i.e. + is up, - is down)
hey [bool, default=False] plugin for Dan Hey's echelle package (**not currently implemented**)
clip_value [float, default=3.0] to clip any peaks higher than Nx the median value

Attributes

ed [numpy.meshgrid] smoothed + summed 2d power for echelle diagram
extent [List[float]] bounding box for echelle diagram

`estimate_background(ind_width=20.0)`

Background estimates

Estimates initial guesses for the stellar background contributions for both the red and white noise components

Parameters

ind_width [float, default=20.0] the independent average smoothing width (μHz)

Attributes

bin_freq, bin_pow, bin_err [numpy.ndarray, numpy.ndarray, numpy.ndarray] binned power spectrum using the *ind_width* bin size

`estimate_numax(binning=0.005, bin_mode='mean', smooth_width=20.0, ask=False)`

Estimate numax

Automated routine to identify power excess due to solar-like oscillations and estimate an initial starting point for *numax* (ν_{max})

Parameters

binning [float, default=0.005] logarithmic binning width (i.e. evenly spaced in log space)

bin_mode [{ 'mean', 'median', 'gaussian' }] mode to use when binning

smooth_width: float, default=20.0 box filter width (in μHz) to smooth power spectrum

ask [bool, default=False] If **True**, it will ask which trial to use as the estimate for numax

Attributes

bin_freq, bin_pow [numpy.ndarray, numpy.ndarray] copy of the power spectrum (i.e. *freq* & *pow*) binned equally in logarithmic space

smooth_freq, smooth_pow [numpy.ndarray, numpy.ndarray] copy of the binned power spectrum (i.e. *bin_freq* & *bin_pow*) binned equally in linear space – *yes, this is doubly binned intentionally*

freq, interp_pow [numpy.ndarray, numpy.ndarray] the smoothed power spectrum (i.e. *smooth_freq* & *smooth_pow*) interpolated back to the original frequency array (also referred to as “crude background model”)

freq, bgcorr_pow [numpy.ndarray, numpy.ndarray] approximate *background-corrected power spectrum* computed by dividing the original PS (*pow*) by the interpolated PS (*interp_pow*)

Methods

- `pysyd.target.Target.collapsed_acf`

estimate_parameters(*estimate=True*)

Estimate parameters

Calls all methods related to the first module

Parameters

estimate [bool, default=True] if numax is already known, this will automatically be skipped since it is not needed

Methods

- `pysyd.target.Target.initial_estimates`
- `pysyd.target.Target.estimate_numax`
- `pysyd.utils._save_estimates`

first_step(*background=True, globe=True*)

First step

Processes a given target for the first step, which has extra steps for each of the two main parts of this method (i.e. background model and global fit):

1. **background model:** the automated best-fit model selection is only performed in the first step, the results which are saved for future purposes (including the background-corrected power spectrum)
2. **global fit:** while the *ACF* is computed for every iteration, a mask is created in the first step to prevent the estimate for dnu to latch on to a different (i.e. incorrect) peak, since this is a multi-modal parameter space

Parameters

background [bool, default=True] run the automated background-fitting routine

globe [bool, default=True] perform global asteroseismic analysis (really only relevant if interested in the background model *only*)

Methods

`pysyd.target.Target.estimate_background` estimates the amplitudes/levels of both correlated and frequency-independent noise properties from the input power spectrum

`pysyd.target.Target.model_background` automated best-fit background model selection that is a summed contribution of various white + red noise componenets

`pysyd.target.Target.global_fit` after correcting for the best-fit background model, this derives the global asteroseismic parameters

See also:

`pysyd.target.Target.single_step`

fix_data(*frequency, power, kep_corr=False, ech_mask=None*)

Fix frequency domain data

Applies frequency-domain tools to power spectra to “fix” (i.e. manipulate) the data. If no available options are used, it will simply return copies of the original arrays

Parameters

save [bool, default=True] save all data products

kep_corr [bool, default=False] correct for known *Kepler* short-cadence artefacts

ech_mask [List[lower_ech, upper_ech], default=None] corrects for dipole mixed modes if not `None`

frequency, power [numpy.ndarray, numpy.ndarray] input power spectrum to be corrected

Methods

`pysyd.target.Target.remove_artefact` mitigate known *Kepler* artefacts

`pysyd.target.Target.whiten_mixed` mitigate mixed modes

Returns

frequency, power [numpy.ndarray, numpy.ndarray] copy of the corrected power spectrum

frequency_spacing(*n_peaks*=10)

Estimate $\Delta\nu$

Estimates the large frequency separation (or $\Delta\nu$) by fitting a Gaussian to the peak of the ACF “cutout” using `scipy.curve_fit`.

Parameters

n_peaks [int, default=10] the number of peaks to identify in the ACF

Attributes

peaks_l, peaks_a [numpy.ndarray] the *n* highest peaks (*n_peaks*) in the ACF

zoom_lag, zoom_auto [numpy.ndarray] cutout from the ACF of the peak near *dnu*

Returns

converge [bool] returns `False` if a Gaussian could not be fit within the 1000 iterations

Raises

utils.ProcessingError if a Gaussian could not be fit to the provided peak

See also:

`pysyd.target.Target.acf_cutout`, `pysyd.target.Target.optimize_ridges`, `pysyd.target.Target.echelle_diagram`

Note: For the first step, a Gaussian weighting (centered on the expected value for *dnu*, or *exp_dnu*) is automatically computed and applied by the pipeline to prevent the fit from latching on to a peak that is a harmonic and not the actual spacing

get_background()

Get background

Attempts to recover background model parameters in later iterations by using the `scipy.curve_fit` module using the same best-fit background model settings

Returns

converge [bool] returns `False` if background model fails to converge

get_epsilon(*n_trials*=3)

Get ridges

Optimizes the large frequency separation by determining which spacing creates the “best” ridges (but is currently under development) think similar to a step-echelle but quicker and more hands off?

Attributes

x [numpy.ndarray] x-axis for the collapsed ED $\sim [0, 2 \times \Delta\nu]$

y [numpy.ndarray] marginalized power along the y-axis (i.e. collapsed on to the x-axis)

Important: need to optimize this - currently does nothing really

get_samples()

Get samples

Estimates uncertainties for parameters by randomizing the power spectrum and attempting to recover the same parameters by calling the [*pysyd.target.Target.single_step*](#)

Attributes

frequency, power [numpy.ndarray, numpy.ndarray] copy of the critically-sampled power spectrum (i.e. *freq_cs* & *pow_cs*) after applying the mask \sim [lower_bg, upper_bg]

pbar [tqdm.tqdm, optional] optional progress bar used with verbose output when running multiple iterations

Note: all iterations except for the first step are applied to the *critically-sampled power spectrum* and *not* the *oversampled power spectrum*

Important: if the verbose option is enabled, the *tqdm* package is required

global_fit()

Global fit

Fits global asteroseismic parameters ν_{\max} and $\Delta\nu$, where the former is estimated two different ways.

Methods *numax_smooth* *numax_gaussian* *compute_acf* *frequency_spacing*

initial_estimates(*lower_ex*=1.0, *upper_ex*=8000.0, *max_trials*=6)

Initial estimates

Prepares data and parameters associated with the first module that identifies solar-like oscillations and estimates *numax*

Parameters

lower_ex [float, default=1.0] the lower frequency limit of the PS used to estimate *numax*

upper_ex [float, default=8000.0] the upper frequency limit of the PS used to estimate *numax*

max_trials [int, default=6] (arbitrary) maximum number of “guesses” or trials to perform to estimate *numax*

Attributes

frequency, power [numpy.ndarray, numpy.ndarray] copy of the entire oversampled (or critically-sampled) power spectrum (i.e. *freq_os* & *pow_os*)

freq, pow [numpy.ndarray, numpy.ndarray] copy of the entire oversampled (or critically-sampled) power spectrum (i.e. `freq_os` & `pow_os`) after applying the mask~[`lower_ex`,`upper_ex`]

module [str, default='parameters'] which part of the pipeline is currently being used

initial_parameters(*lower_bg=1.0, upper_bg=8000.0*)

Initial guesses

Estimates initial guesses for background components (i.e. timescales and amplitudes) using solar scaling relations. This resets the power spectrum and has its own independent filter or bounds (via [`lower_bg`, `upper_bg`]) to use for this subroutine

Parameters

lower_bg [float, default=1.0] lower frequency limit of PS to use for the background fit

upper_bg [float, default=8000.0] upper frequency limit of PS to use for the background fit

Attributes

frequency, power [numpy.ndarray, numpy.ndarray] copy of the entire oversampled (or critically-sampled) power spectrum (i.e. `freq_os` & `pow_os`)

frequency, random_pow [numpy.ndarray, numpy.ndarray] copy of the entire oversampled (or critically-sampled) power spectrum (i.e. `freq_os` & `pow_os`) after applying the mask~[`lower_bg`,`upper_bg`]

module [str, default='parameters'] which part of the pipeline is currently being used

i [int, default=0] iteration number

Methods

pysyd.target.Target.solar_scaling uses multiple solar scaling relations to determine accurate initial guesses for many of the derived parameters

Warning: This is typically sufficient for most stars but may affect evolved stars and need to be adjusted!

load_file(*path*)

Load text file

Load a light curve or a power spectrum from a basic 2xN txt file and stores the data into the `x` (independent variable) and `y` (dependent variable) arrays, where N is the length of the series

Parameters

path [str] the file path of the data file

Returns

x, y [numpy.ndarray, numpy.ndarray] the independent and dependent variables, respectively

load_power_spectrum(*long=1000000*)

Load power spectrum

Loads in available power spectrum and computes relevant information – also checks for time series data and will raise a warning if there is none since it will have to assume a *critically-sampled power spectrum*

Attributes

note [str, optional] verbose output

ps [bool] **True** if star ID has an available (or newly-computed) power spectrum

Yields

frequency, power [numpy.ndarray, numpy.ndarray] input power spectrum

freq_os, pow_os [numpy.ndarray, numpy.ndarray] copy of the oversampled power spectrum (i.e. frequency & power)

freq_cs, pow_cs [numpy.ndarray, numpy.ndarray] copy of the critically-sampled power spectrum (i.e. frequency & power) iff the *oversampling_factor* is provided, otherwise these arrays are just copies of *freq_os* & *pow_os* since this factor isn't known and needs to be assumed

Raises

pysyd.utils.InputWarning if no information or time series data is provided (i.e. *has* to assume the PS is critically-sampled)

load_time_series (*save=True, stitch=False, oversampling_factor=None*)

Load light curve

Loads in time series data and calculates relevant parameters like the cadence and nyquist frequency

Parameters

save [bool, default=True] save all data products

stitch [bool, default=False] “stitches” together time series data with large “gaps”

oversampling_factor [int, optional] oversampling factor of input power spectrum

Attributes

note [str, optional] verbose output

lc [bool] *True* if star ID has light curve data available

cadence [int] median cadence of time series data (Δt)

nyquist [float] nyquist frequency of the power spectrum (calculated from time series cadence)

baseline [float] total time series duration (ΔT)

tau_upper [float] upper limit of the granulation time scales, which is set by the total duration of the time series (divided in half)

Yields

time, flux [numpy.ndarray, numpy.ndarray] input time series data

frequency, power [numpy.ndarray, numpy.ndarray] newly-computed frequency array using the time series array (i.e. *time* & *flux*)

freq_os, pow_os [numpy.ndarray, numpy.ndarray] copy of the oversampled power spectrum (i.e. frequency & power)

freq_cs, pow_cs [numpy.ndarray, numpy.ndarray] copy of the critically-sampled power spectrum (i.e. frequency & power)

Raises

pysyd.utils.InputWarning if the oversampling factor provided is different from that computed from the time series data and power spectrum

pysyd.utils.InputError if the oversampling factor calculated from the time series data and power spectrum is not an integer

model_background(*n_laws=None, fix_wn=False, basis='tau_sigma'*)

Model stellar background

If nothing is fixed, this method iterates through $2(n_{\text{laws}} + 1)$ models to determine the best-fit background model due to stellar granulation processes, which uses a solar scaling relation to estimate the number of Harvey-like component(s) (or *n_laws*)

Parameters

n_laws [int, default=None] specify number of Harvey-like components to use in background fit

fix_wn [bool, default=False] option to fix the white noise instead of it being an additional free parameter

basis [str, default='tau_sigma'] which basis to use for background fitting, e.g. {a,b} parametrization

TODO: not yet operational

Methods `pysyd.models.background` - `scipy.curve_fit` - `pysyd.models._compute_aic` - `pysyd.models._compute_bic` - `pysyd.target.Target.correct_background`

Returns

converge [bool] returns `False` if background model fails to converge

Raises

utils.ProcessingError if this failed to converge on a single model during the first iteration

numax_gaussian()

Gaussian ν_{max}

Estimate numax by fitting a Gaussian to the “zoomed-in” power spectrum (i.e. *region_freq* and *region_pow*) using `scipy.curve_fit`

Returns

converge [bool] returns `False` if background model fails to converge

Raises

utils.ProcessingError if the Gaussian fit does not converge for the first step

numax_smooth(*sm_par=None*)

Smooth ν_{max}

Estimate numax by taking the peak of the smoothed power spectrum

Parameters

sm_par [float, optional] smoothing width for power spectrum calculated from solar scaling relation (typically ~1-4)

Attributes

frequency, pssm [numpy.ndarray, numpy.ndarray] smoothed power spectrum

frequency, pssm_bgcorr [numpy.ndarray, numpy.ndarray] smoothed *background-subtracted power spectrum*

region_freq, region_pow [numpy.ndarray, numpy.ndarray] oscillation region of the power spectrum (“zoomed in”) by applying the mask~[lower_ps,upper_ps]

numax_smoo [float] the ‘observed’ numax (i.e. the peak of the smoothed power spectrum)

dnu_smoo [float] the ‘expected’ dnu based on a scaling relation using the numax_smoo

optimize_ridges(*n=50, res=0.01*)

Get ridges

Optimizes the large frequency separation by determining which spacing creates the “best” ridges (but is currently under development) think similar to a step-echelle but quicker and more hands off?

Attributes

x [numpy.ndarray] x-axis for the collapsed ED $\sim [0, 2 \times \Delta\nu]$

y [numpy.ndarray] marginalized power along the y-axis (i.e. collapsed on to the x-axis)

Important: need to optimize this - currently does nothing really

process_star()

Run pipeline

Processes a given star with pySYD

Methods

- `pysyd.target.Target.estimate_parameters`
- `pysyd.target.Target.derive_parameters`
- `pysyd.target.Target.show_results`

red_noise(*box_filter=1.0, n_rms=20*)

Estimate red noise

Estimates amplitudes of red noise components by using a smoothed version of the power spectrum with the power excess region masked out – which will take the mean of a specified number of points (via -nrms, default=20) for each Harvey-like component

Parameters

box_filter [float, default=1.0] the size of the 1D box smoothing filter

n_rms [int, default=20] number of data points to average over to estimate red noise amplitudes

Attributes

smooth_pow [numpy.ndarray] smoothed power spectrum after applying the box filter

remove_artefact(*freq, pow, lcp=566.4233312035817, lf_lower=[240.0, 500.0], lf_upper=[380.0, 530.0], hf_lower=[4530.0, 5011.0, 5097.0, 5575.0, 7020.0, 7440.0, 7864.0], hf_upper=[4534.0, 5020.0, 5099.0, 5585.0, 7030.0, 7450.0, 7867.0]*)

Remove *Kepler* artefacts

Module to remove artefacts found in *Kepler* data by replacing known frequency ranges with simulated noise

Parameters

lcp [float] long cadence period (in Msec)

lf_lower [List[float]] lower limits of low-frequency artefacts

lf_upper [List[float]] upper limits of low-frequency artefacts

hf_lower [List[float]] lower limit of high frequency artefact

hf_upper [List[float]] upper limit of high frequency artefact

freq, pow [numpy.ndarray, numpy.ndarray] input data that needs to be corrected

Returns

frequency, power [numpy.ndarray, numpy.ndarray] copy of the corrected power spectrum

Note:**Known *Kepler* artefacts include:**

1. long-cadence harmonics
 2. sharp, high-frequency artefacts ($> 5000\mu\text{Hz}$)
 3. low frequency artefacts 250-400 μHz (mostly present in Q0 and Q3 data)
-

show_results(*show=False, verbose=False*)

Parameters

show [bool, optional] show output figures and text

verbose [bool, optional] turn on verbose output

single_step()

Single step

Similar to the first step, this function calls the same methods but uses the selected best-fit background model from the first step to estimate the parameters

Attributes

converge [bool] removes any saved parameters if any fits did not converge (i.e. `False`)

Returns

converge [bool] returns `True` if all relevant fits converged

Methods

`pysyd.target.Target.estimate_background` estimates the amplitudes/levels of both correlated and frequency-independent noise properties from the input power spectrum

`pysyd.target.Target.get_background` unlike the first step, which iterated through several models and performed a best-fit model comparison, this only fits parameters from the selected model in the first step

`pysyd.target.Target.global_fit` after correcting for the background model, this derives the global asteroseismic parameters

solar_scaling(*numax=None, scaling='tau_sun_single', max_laws=3, ex_width=1.0, lower_ps=None, upper_ps=None*)

Initial values

Using the initial starting value for

τm

nu_{max} , estimates the rest of the parameters needed for *both* the background and global fits. Uses scaling relations from the Sun to:

1. estimate the width of the region of oscillations using numax
2. guess starting values for granulation time scales

Parameters

numax [float, default=None]

provide initial value for numax to bypass the first module

scaling [str, default='tau_sun_single'] which solar scaling relation to use

max_laws [int, default=3] the maximum number of resolvable Harvey-like components

ex_width [float, default=1.0] fractional width to use for power excess centered on *numax*

lower_ps [float, default=None] lower bound of power excess to use for *ACF* [in *rmμHz*]

upper_ps [float, default=None] upper bound of power excess to use for *ACF* [in *rmμHz*]

Attributes

converge [bool, default=True] *True* if all fitting converges

stitch_data(*gap=20*)

Stitch light curve

For computation purposes and for special cases that this does not affect the integrity of the results, this module ‘stitches’ a light curve together for time series data with large gaps. For stochastic p-mode oscillations, this is justified if the lifetimes of the modes are smaller than the gap.

Parameters

gap [int, default=20] how many consecutive missing cadences are considered a ‘gap’

Attributes

time [numpy.ndarray] original time series array to correct

new_time [numpy.ndarray] the corrected time series array

Raises

pysyd.utils.InputWarning when using this method since it’s technically not a great thing to do

Warning: USE THIS WITH CAUTION. This is technically not a great thing to do for primarily two reasons:

1. you lose phase information *and*
2. can be problematic if mode lifetimes are shorter than gaps (i.e. more evolved stars)

Note: temporary solution for handling very long gaps in TESS data – still need to figure out a better way to handle this

white_noise()

Estimate white noise

Estimate the white noise level by taking the mean of the last 10% of the power spectrum

whiten_mixed(*freq, pow, dnu=None, lower_ech=None, upper_ech=None, notching=False*)

Whiten mixed modes

Module to help reduce the effects of mixed modes random white noise in place of $\ell = 1$ for subgiants with mixed modes to better constrain the large frequency separation

Parameters

dnu [float, default=None] the so-called large frequency separation to fold the PS to

lower_ech [float, default=None] lower frequency limit of mask to “whiten”

upper_ech [float, default=None] upper frequency limit of mask to “whiten”

notching [bool, default=False] if **True**, uses notching instead of generating white noise

freq, pow [numpy.ndarray, numpy.ndarray] input data that needs to be corrected

folded_freq [numpy.ndarray] frequency array modulo dnu (i.e. folded to the large separation, $\Delta\nu$)

Returns

frequency, power [numpy.ndarray, numpy.ndarray] copy of the corrected power spectrum

3.5 Models & utilities

Container classes, parameter dictionaries, functions related to file loading and/or saving as well as various data manipulation methods (i.e. correcting artefacts, binning data, etc.).

3.5.1 Introduction

3.5.2 Imports

Any dependencies

3.5.3 Usage

Used during...

3.5.4 Examples

3.5.5 Models

`pysyd.models.background(frequency, guesses, mode='regular', ab=False, noise=None)`

The main model for the stellar background fitting

Parameters

frequency [numpy.ndarray] the frequency of the power spectrum

guesses [list] the parameters of the Harvey model

mode [{ 'regular', 'second', 'fourth' }] the mode of which Harvey model parametrization to use. Default mode is **regular**. The 'regular' mode is when both the second and fourth order terms are added in the denominator whereas, 'second' only adds the second order term and 'fourth' only adds the fourth order term.

total [bool] If **True**, returns the summed model over multiple components. This is deprecated.

ab [bool, optional] If **True**, changes to the traditional a, b parametrization as opposed to the SYD

noise [None, optional] If not **None**, it will fix the white noise to this value and not model it, reducing the dimension of the problem/model

Returns

model [np.ndarray] the stellar background model

TODO option to fix the white noise (i.e. **noise** option) option to change the parametrization (i.e. **ab** option)
option to add power law

`pysyd.models.gaussian(frequency, offset, amplitude, center, width)`

Gaussian model

Observed solar-like oscillations have a Gaussian-like profile and therefore, detections are modeled as a Gaussian distribution.

Parameters

frequency [numpy.ndarray] the frequency array

offset [float] the vertical offset

amplitude [float] amplitude of the Gaussian

center [float] center of the Gaussian

width [float] the width of the Gaussian

Returns

result [np.ndarray] the Gaussian distribution

`pysyd.models.harvey_fit(frequency, tau, sigma, exponent, mode='regular', ab=False)`

Testing

`pysyd.models.harvey_fourth(frequency, tau, sigma, mode='regular', ab=False)`

Testing

`pysyd.models.harvey_none(frequency, white_noise, ab=False)`

No Harvey model

Stellar background model that does not contain any Harvey-like components i.e. this is the simplest model of all - consisting of a single white-noise component. This was added with the hopes that it would be preferred in the model selection for non-detections.

Warning: check if this is working for null detections

Parameters

frequency [numpy.ndarray] the frequency array

white_noise [float] the white noise component

Returns

model [numpy.ndarray] the no-Harvey (white noise) model

`pysyd.models.harvey_one(frequency, tau_1, sigma_1, white_noise, ab=False)`

One Harvey model

Stellar background model consisting of a single Harvey-like component

Parameters

frequency [numpy.ndarray] the frequency array

tau_1 [float] timescale of the first harvey component

sigma_1 [float] amplitude of the first harvey component

white_noise [float] the white noise component

Returns

model [numpy.ndarray] the one-Harvey model

`pysyd.models.harvey_regular(frequency, tau, sigma, mode='regular', ab=False)`

Testing

`pysyd.models.harvey_second(frequency, tau, sigma, mode='regular', ab=False)`

Testing

`pysyd.models.harvey_three(frequency, tau_1, sigma_1, tau_2, sigma_2, tau_3, sigma_3, white_noise, ab=False)`

Three Harvey model

Stellar background model consisting of three Harvey-like components

Parameters

frequency [numpy.ndarray] the frequency array

tau_1 [float] timescale of the first harvey component

sigma_1 [float] amplitude of the first harvey component

tau_2 [float] timescale of the second harvey component

sigma_2 [float] amplitude of the second harvey component

tau_3 [float] timescale of the third harvey component

sigma_3 [float] amplitude of the third harvey component

white_noise [float] the white noise component

Returns

model [numpy.ndarray] the three-Harvey model

`pysyd.models.harvey_two(frequency, tau_1, sigma_1, tau_2, sigma_2, white_noise, ab=False)`

Two Harvey model

Stellar background model consisting of two Harvey-like components

Parameters

frequency [numpy.ndarray] the frequency array

tau_1 [float] timescale of the first harvey component

sigma_1 [float] amplitude of the first harvey component

tau_2 [float] timescale of the second harvey component

sigma_2 [float] amplitude of the second harvey component

white_noise [float] the white noise component

Returns

model [numpy.ndarray] the two-Harvey model

`pysyd.models.power(frequency, coefficient, exponent)`

Power law

Power law distribution used to model traditional “red” noise contributions i.e. the rise in power at low frequencies typically corresponding to long-term stellar variability

Parameters

frequency [numpy.ndarray] the frequency array

coefficient [float] the power law coefficient

exponent [float] the power law exponent

Returns

result [np.ndarray] the power law distribution

`pysyd.models.white(frequency, white_noise)`

Testing

class `pysyd.utils.Constants`

Container class for constants and known values – which is primarily solar asteroseismic values for our purposes.

UNITS ARE IN THE SUPERIOR CGS COME AT ME

exception `pysyd.utils.InputError(error, width=60)`

Class for pySYD user input errors (i.e., halts execution).

exception `pysyd.utils.InputWarning(warning, width=60)`

Class for pySYD user input warnings.

class `pysyd.utils.Parameters(args=None)`

Container class for pySYD parameters

Calls super method to inherit all relevant constants and then stores the default values for all pysyd modules

Methods

add_cli(args)

Add CLI

Save any non-default parameters provided via command line but skips over any keys in the override columns since those are star specific and have a given length – it will come back to this

Parameters

args [argparse.Namespace] the command line arguments

add_targets(stars=None)

Add targets

This was mostly added for non-command-line users, since this makes API usage easier.

check_cli(args, max_laws=3, override=['numax', 'dnu', 'lower_ex', 'upper_ex', 'lower_bg', 'upper_bg', 'lower_ps', 'upper_ps', 'lower_ech', 'upper_ech'])

Check CLI

Make sure that any command-line inputs are the proper lengths, types, etc.

Parameters

args [argparse.Namespace] the command line arguments

max_laws [int] maximum number of Harvey laws to be fit

Asserts

- the length of each array provided (in override) is equal
- the oversampling factor is an integer (if applicable)
- the number of Harvey laws to “force” is an integer (if applicable)

get_background(*background=True, basis='tau_sigma', box_filter=1.0, ind_width=20.0, n_rms=20, n_laws=None, fix_wn=False, metric='bic', lower_bg=None, upper_bg=None, functions=None*)

Background parameters

Gets parameters used during the automated background-fitting analysis

Attributes

params [Dict[str,Dict[,]]] the updated parameters

get_data(*info='info/star_info.csv', todo='info/todo.txt', stars=None, mode='run', gap=20, stitch=False, oversampling_factor=None, kep_corr=False, notching=False, dnu=None, lower_ech=None, upper_ech=None*)

Get data parser

Load parameters available in the data parser, which is mostly related to initial data loading and manipulation

Attributes

params [Dict[str,Dict[,]]] the updated parameters

get_defaults()

Load defaults

Gets default pySYD parameters by calling functions which are analogous to available command-line parsers and arguments

Attributes

params [Dict[str[Dict[,]]]] container class for pySYD parameters

Calls

- [`pysyd.utils.Parameters.get_parent`](#)
- [`pysyd.utils.Parameters.get_data`](#)
- [`pysyd.utils.Parameters.get_main`](#)
- [`pysyd.utils.Parameters.get_plot`](#)

get_estimate(*estimate=True, smooth_width=20.0, binning=0.005, bin_mode='mean', step=0.25, n_trials=3, ask=False, lower_ex=None, upper_ex=None*)

Search and estimate parameters

Get parameters relevant for the optional first module that looks for and identifies power excess due to solar-like oscillations and then estimates its properties

Attributes

params [Dict[str,Dict[,]]] the updated parameters

get_global(*globe=True, numax=None, lower_ps=None, upper_ps=None, ex_width=1.0, sm_par=None, smooth_ps=2.5, fft=True, threshold=1.0, n_peaks=5*)

Global fitting parameters

Get default parameters that are relevant for deriving global asteroseismic parameters ν_{\max} and $\Delta\nu$

Attributes

params [Dict[str,Dict[,]]] the updated parameters

get_main()

Get main parser

Load parameters available in the main parser i.e. core software functionality

Attributes

params [Dict[str,Dict[,]]] the updated parameters

Calls

- `pysyd.utils.Parameters.get_estimate`
- `pysyd.utils.Parameters.get_background`
- `pysyd.utils.Parameters.get_global`
- `pysyd.utils.Parameters.get_sampling`

get_parent(*inpdire='data', infdire='info', outdire='results', save=True, test=False, verbose=False, overwrite=False, warnings=False, cli=True, notebook=False*)

Get parent parser

Load parameters available in the parent parser i.e. higher-level software functionality

Attributes

params [Dict[str,Dict[,]]] the updated parameters

get_plot(*show_all=False, show=False, cmap='binary', hey=False, clip_value=3.0, interp_ech=False, nox=None, noy='0+0', npb=10, ridges=False, smooth_ech=None*)

Get plot parser

Save all parameters related to any of the output figures

Attributes

params [Dict[str,Dict[,]]] the updated parameters

get_sampling(*mc_iter=1, seed=None, samples=False, n_threads=0*)

Sampling parameters

Get parameters relevant for the sampling steps i.e. estimating uncertainties

Attributes

params [Dict[str,Dict[,]]] the updated parameters

exception `pysyd.utils.ProcessingError`(*error, width=60*)

Class for pySYD processing errors (i.e., halts execution).

exception `pysyd.utils.ProcessingWarning`(*warning, width=60*)

Class for pySYD user input warnings.

`pysyd.utils.delta_nu`(*numax*)

$\Delta\nu$

Estimates the large frequency separation using the numax scaling relation (add citation?)

Parameters

numax [float] the frequency corresponding to maximum power or numax (ν_{\max})

Returns

dnu [float] the approximated frequency spacing or dnu ($\Delta\nu$)

pysyd.utils.**get_dict**(type='params')

Get dictionary

Quick+convenient utility function to read in longer dictionaries that are used throughout the software

Parameters

type [{ 'columns', 'params', 'plots', 'tests', 'functions' }, default='params'] which dictionary to load in – which *MUST* match their relevant filenames

Returns

result [Dict[str,Dict[,]]] the relevant (type) dictionary

Important: 'functions' cannot be saved and loaded in like the other dictionary because it points to modules loaded in from another file

pysyd.utils.**get_infdir**(args, dl_dict, note,
source='https://raw.githubusercontent.com/ashleychontos/pySYD/master/dev/')

Create info directory

Parameters

args [argparse.Namespace] command-line arguments

note [str] verbose output

dl_dict [Dict[str,str]] dictionary to keep track of files that need to be downloaded

source [str] path to pysyd source directory on github

Returns

dl_dict [Dict[str,str]] dictionary of files to download for setup

note [str] updated verbose output

pysyd.utils.**get_inpdir**(args, dl_dict, note, save=False, examples=['1435467', '2309595', '11618103'],
exts=['LC', 'PS'],
source='https://raw.githubusercontent.com/ashleychontos/pySYD/master/dev/')

Create data (i.e. input) directory

Parameters

args [argparse.Namespace] command-line arguments

note [str] verbose output

dl_dict [Dict[str,str]] dictionary to keep track of files that need to be downloaded

source [str] path to pysyd source directory on github

examples [List[str]] KIC IDs for 3 example stars

exts [List[str]] data types to download for each star

Returns

dl_dict [Dict[str,str]] dictionary of files to download for setup

note [str] updated verbose output

`pysyd.utils.get_outdir(args, note)`

Create results directory

Parameters

args [argparse.Namespace] command-line arguments

note [str] verbose output

Returns

note [str] updated verbose output

`pysyd.utils.get_output(fun=False)`

Print logo output

Used within test mode when current installation is successfully tested.

Parameters

fun [bool, False] if calling module for 'fun', only prints logo but doesn't test software

`pysyd.utils.setup_dirs(args, note="", dl_dict={})`

Setup pySYD directories

Primarily most of pipeline.setup functionality to keep the pipeline script from getting too long. Still calls/downloads things in the same way: 1) info directory, 2) input + data directory and 3) results directory.

Parameters

args [argparse.Namespace] command-line arguments

note [str] verbose output

dl_dict [Dict[str,str]] dictionary to keep track of files that need to be downloaded

Returns

dl_dict [Dict[str,str]] dictionary of files to download for setup

note [str] updated verbose output

Calls

- `pysyd.utils.get_infdir`
- `pysyd.utils.get_inpdir`
- `pysyd.utils.get_outdir`

3.5.6 Utilities

`class pysyd.utils.Parameters(args=None)`

Container class for pySYD parameters

Calls super method to inherit all relevant constants and then stores the default values for all pysyd modules

Methods

add_cli(args)

Add CLI

Save any non-default parameters provided via command line but skips over any keys in the override columns since those are star specific and have a given length – it will come back to this

Parameters

args [argparse.Namespace] the command line arguments

add_targets(*stars=None*)

Add targets

This was mostly added for non-command-line users, since this makes API usage easier.

check_cli(*args, max_laws=3, override=['numax', 'dnu', 'lower_ex', 'upper_ex', 'lower_bg', 'upper_bg', 'lower_ps', 'upper_ps', 'lower_ech', 'upper_ech']*)

Check CLI

Make sure that any command-line inputs are the proper lengths, types, etc.

Parameters

args [argparse.Namespace] the command line arguments

max_laws [int] maximum number of Harvey laws to be fit

Asserts

- the length of each array provided (in override) is equal
- the oversampling factor is an integer (if applicable)
- the number of Harvey laws to “force” is an integer (if applicable)

get_background(*background=True, basis='tau_sigma', box_filter=1.0, ind_width=20.0, n_rms=20, n_laws=None, fix_wn=False, metric='bic', lower_bg=None, upper_bg=None, functions=None*)

Background parameters

Gets parameters used during the automated background-fitting analysis

Attributes

params [Dict[str,Dict[,]]] the updated parameters

get_data(*info='info/star_info.csv', todo='info/todo.txt', stars=None, mode='run', gap=20, stitch=False, oversampling_factor=None, kep_corr=False, notching=False, dnu=None, lower_ech=None, upper_ech=None*)

Get data parser

Load parameters available in the data parser, which is mostly related to initial data loading and manipulation

Attributes

params [Dict[str,Dict[,]]] the updated parameters

get_defaults()

Load defaults

Gets default pySYD parameters by calling functions which are analogous to available command-line parsers and arguments

Attributes

params [Dict[str[Dict[,]]]] container class for pySYD parameters

Calls

- `pysyd.utils.Parameters.get_parent`
- `pysyd.utils.Parameters.get_data`

- `pysyd.utils.Parameters.get_main`
- `pysyd.utils.Parameters.get_plot`

get_estimate(*estimate=True, smooth_width=20.0, binning=0.005, bin_mode='mean', step=0.25, n_trials=3, ask=False, lower_ex=None, upper_ex=None*)

Search and estimate parameters

Get parameters relevant for the optional first module that looks for and identifies power excess due to solar-like oscillations and then estimates its properties

Attributes

params [Dict[str,Dict[,]]] the updated parameters

get_global(*globe=True, numax=None, lower_ps=None, upper_ps=None, ex_width=1.0, sm_par=None, smooth_ps=2.5, fft=True, threshold=1.0, n_peaks=5*)

Global fitting parameters

Get default parameters that are relevant for deriving global asteroseismic parameters ν_{\max} and $\Delta\nu$

Attributes

params [Dict[str,Dict[,]]] the updated parameters

get_main()

Get main parser

Load parameters available in the main parser i.e. core software functionality

Attributes

params [Dict[str,Dict[,]]] the updated parameters

Calls

- `pysyd.utils.Parameters.get_estimate`
- `pysyd.utils.Parameters.get_background`
- `pysyd.utils.Parameters.get_global`
- `pysyd.utils.Parameters.get_sampling`

get_parent(*inpdire='data', infdir='info', outdir='results', save=True, test=False, verbose=False, overwrite=False, warnings=False, cli=True, notebook=False*)

Get parent parser

Load parameters available in the parent parser i.e. higher-level software functionality

Attributes

params [Dict[str,Dict[,]]] the updated parameters

get_plot(*show_all=False, show=False, cmap='binary', hey=False, clip_value=3.0, interp_ech=False, nox=None, noy='0+0', npb=10, ridges=False, smooth_ech=None*)

Get plot parser

Save all parameters related to any of the output figures

Attributes

params [Dict[str,Dict[,]]] the updated parameters

```
get_sampling(mc_iter=1, seed=None, samples=False, n_threads=0)
```

Sampling parameters

Get parameters relevant for the sampling steps i.e. estimating uncertainties

Attributes

params [Dict[str,Dict[,]]] the updated parameters

3.6 Saved outputs

We have shown examples applied to stars of various sample sizes, for different stellar types, of varying SNR detections, both single star and many star we will not include any additional examples on this page but instead, list and describe each of the output files. Therefore we refer the reader to check out this page, the command-line examples or the [notebook tutorials](#) if more examples are desired.

So while saving output files and figures is totally optional, we wanted to document them on this page since there's a lot of information to unpack.

Subdirectories are automatically created for each star that is processed. Based on the way you use pySYD, there are a number of different outputs which are saved by default. Here we will list and describe them all.

We will reserve this page solely for saved outputs and hence, please see our [crashteroseismology](#) example if you'd like more information about the printed verbose output.

3.6.1 Files

Listed are all the possible output files:

1. *ID_PS.txt*
2. ID_BSPS.txt
3. ID_BDPS.txt
4. *estimates.csv*
5. *global.csv*
6. *samples.csv*

which we describe in more detail below, including the frequency and likely scenarios they arise from.

1. ID_PS.txt

(special cases)

This file is created in the case where *only* the time series data was provided for a target and pySYD computed a power spectrum. This optional, extra step is important to make sure that the power spectrum used through the analyzes is both normalized correctly and has the proper units – this *ensures* accurate and reliable results.

Note: unlike every other output file, this is instead saved to the data (or input directory) so that the software can find it in later runs, which will save some time down the road. Of course you can always copy and paste it to the specific star's result directory if you'd like.

2. ID_BSPS.txt

(all cases)

After the best-fit background model is selected and saved, the model is generated and then subtracted from the power spectrum to remove all noise components present in a power spectrum. Therefore, there should be little to no residual slope left in the power spectrum after this step. This is saved as a basic text file in the star's output directory, where the first column is frequency (in μHz) and the second column is power density, with units of $\text{ppm}^2 \mu\text{Hz}^{-1}$ (i.e. this file has the same units as the power spectrum).

In fact to take a step back, it might be helpful to understand the application and importance of the background-corrected power spectrum (*BCPS*). The BCPS is used in subsequent steps such as computing global parameters (ν_{max} and $\Delta\nu$) and for constructing the *echelle diagram*. Therefore, we thought it might be useful to have a copy of this!

3. ID_BDPS.txt

(all cases)

Since we use both *BCPS*, we figured we'd clear up the muddy waters here (but also provide both copies to be used for their specific needs).

4. estimates.csv

(most cases)

By default, a module will run to estimate an initial value for the frequency corresponding to maximum power, or ν_{max} . The module selects the trial with the highest signal-to-noise (SNR) and saves the comma-separated values for three basic variables associated with the selected trial: *numax*, *dnu*, and the SNR.

The file is saved to the star's output directory, where both *numax* and *dnu* have frequency units in μHz and the SNR is unitless. Remember, these are just estimates though and adapted results should come from the other csv file called *global.csv*.

This module can be bypassed a few different ways, primarily by directly providing the estimate yourself. In the cases where this estimating routine is skipped, this file will not be saved.

Note: The *numax* estimate is *important* for the main fitting routine.

4. global.csv

(all cases)

5. samples.csv

(special cases)

If the monte-carlo sampling is used to estimate uncertainties, an optional feature is available (i.e. `-sampling`) to save the samples if desired.

Note: there is a *new* feature that saves and sets a random seed any time you are running a target for the first time and therefore, you should be able to reproduce the samples in the event that you forget to save the samples.

3.6.2 Figures

Listed are all possible output figures for a given star (in alphabetical order):

1. *background_only.png*
2. *bgmodel_fits.png*
3. *global_fit.png*
4. *power_spectrum.png*
5. *samples.png*
6. *search_&_estimate.png*
7. *time_series.png*

and similar to the *file section* above, we describe each in more detail below.

1. background_only.png

(rare cases)

This figure is produced when the user is interested in determining the stellar background model *only* and not the global asteroseismic properties. For example, detecting solar-like oscillations in cool stars is extremely difficult to do but we can still characterize other properties like their convective time scales, etc.

2. bgmodel_fits.png

(optional cases)

This figure is generated when the `--show`

3. global_fit.png

(almost all cases)

Top left: Original time series.

Top middle: Original power spectrum (white), lightly smoothed power spectrum (red), and binned power spectrum (green). Blue lines show initial guesses of the fit to the granulation background. The grey region is excluded from the background fit based on the `numax` estimate provided to the module.

Top right: Same as top middle but now showing the best fit background model (blue) and a heavily smoothed version of the power spectrum (yellow)

Center left: Background corrected, heavily smoothed power spectrum (white). The blue line shows a Gaussian fit to the data (used to calculate `numax_gaussian`) and the red square is the peak of the smoothed, background corrected power excess (`numax_smoothed`).

Center: Lightly smoothed, background corrected power spectrum centered on `numax`.

Center right: Autocorrelation function of the data in the center panel. The red dotted line shows the estimate `Dnu` value given the input `numax` value, and the red region shows the extracted ACF peak that will be used to measure `Dnu`. The yellow line shows the Gaussian weighting function used to define the red region.

Bottom left: ACF peak extracted in the center right panel (white) and a Gaussian fit to that peak (green). The center of the Gaussian is the estimate of `Dnu`.

Bottom middle: Echelle diagram of the background corrected power spectrum using the measured `Dnu` value.

Bottom right: Echelle diagram collapsed along the frequency direction.

4. power_spectrum.png

(special cases)

This is still in its developmental stage but the idea is that one is supposed to “check” a target before attempting to process the pipeline on any data. That means checking the input data for sketchy looking features. For example, *Kepler* short-cadence data has known artefacts present near the nyquist frequency for *Kepler* long-cadence data ($\sim 270\mu\text{Hz}$). In these cases, we have special frequency-domain tools that are meant to help mitigate such things (e.g., see `pysyd.target.Target.remove_artefact`)

5. samples.png

(many cases)

Each panel shows the samples of parameter estimates from Monte-Carlo simulations. Reported uncertainties on each parameter are calculated by taking the robust standard deviation of each distribution.

6. search_&_estimate.png

(most cases)

Top left: Original time series.

Top middle: Original power spectrum (white) and heavily smoothed power spectrum (green). The latter is used as an initial (crude) background fit to search for oscillations.

Top right: Power spectrum after correcting the crude background fit.

Bottom left: Frequency-resolved, collapsed autocorrelation function of the background-corrected power spectrum using a small step size. This step size is optimized for low-frequency oscillators. The green line is a Gaussian fit to the data, which provides the initial numax estimate.

Bottom middle: Same as bottom left but for the medium step size (optimized for subgiant stars).

Bottom right: Same as bottom left but for the large step size (optimized for main-sequence stars).

7. time_series.png

(special cases)

3.6.3 Takeaway

As we’ve said many times before, the software is optimized for running an ensemble of stars. Therefore, the utility function `pysyd.utils.scrape_output` will automatically concatenate the results for each of the main modules into a single csv in the parent results directory so that it’s easy to find and compare.

3.6.4 API

`pysyd.plots.check_data(star, args, show=True)`

Plot input data for a target

`pysyd.plots.create_benchmark_plot(filename='comparison.png', variables=['numax', 'dnu'], show=False, save=True, overwrite=False, npanels=2)`

Compare ensemble results between the pySYD and SYD pipelines for the *Kepler* legacy sample

`pysyd.plots.make_plots(star, show_all=False)`

Make plots

Function that establishes the default plotting parameters and then calls each of the relevant plotting routines

Parameters

star [`pysyd.target.Target`] the pySYD pipeline object

showall [bool, optional] option to plot, save and show the different background models (default=`False`)

Calls `pysyd.plots.plot_estimates` `pysyd.plots.plot_parameters` `pysyd.plots.plot_bgfits`
[optional] `pysyd.plots.plot_samples`

`pysyd.plots.plot_1d_ed(star, filename='1d_ed.png', npanels=1)`

Plot collapsed ED

Parameters

star [`target.Target`] the pySYD pipeline object

filename [str] the path or extension to save the figure to

npanels [int] number of panels in this figure (default=`1`)

`pysyd.plots.plot_bgfits(star, filename='bgmodel_fits.png', highlight=True)`

Comparison of the background model fits

Parameters

star [`target.Target`] the pySYD pipeline object

filename [str] the path or extension to save the figure to

highlight [bool, optional] if `True`, highlights the selected model

`pysyd.plots.plot_estimates(star, filename='search_&_estimate.png', highlight=True, n=0)`

Plot estimates

Creates a plot summarizing the results of the find excess routine.

Parameters

star [`pysyd.target.Target`] the pySYD pipeline object

filename [str] the path or extension to save the figure to

highlight [bool, default=`True`] option to highlight the selected estimate

`pysyd.plots.plot_light_curve(star, args, filename='time_series.png', npanels=1)`

Plot light curve data

Parameters

star [`target.Target`] the pySYD pipeline object

filename [str] the path or extension to save the figure to

npanels [int] number of panels in this figure (default=`1`)

`pysyd.plots.plot_parameters(star, subfilename='background_only.png', filename='global_fit.png', n=0)`

Plot parameters

Creates a plot summarizing all derived parameters

Parameters

star [pysyd.target.Target] the main pipeline Target class object

subfilename [str] separate filename in the event that only the background is being fit

filename [str] the path or extension to save the figure to

`pysyd.plots.plot_power_spectrum(star, args, filename='power_spectrum.png', npanels=1)`

Plot power spectrum

Parameters

star [target.Target] the pySYD pipeline object

filename [str] the path or extension to save the figure to

npanels [int] number of panels in this figure (default=`1`)

`pysyd.plots.plot_samples(star, filename='samples.png')`

Plot results of the Monte-Carlo sampling

Parameters

star [target.Target] the pySYD pipeline object

filename [str] the path or extension to save the figure to

`pysyd.plots.select_trial(star)`

Select trial

This is called when `--ask` is `True` (i.e. select which trial to use for

rm

nu_{max}) This feature used to be called as part of a method in the `pysyd.target.Target` class but left a stale figure open – this way it can be closed after the value is selected

Parameters

star [pysyd.target.Target] the pySYD pipeline object

Returns

value [int or float] depending on which `trial` was selected, this can be of integer or float type

3.7 What next?

You may be asking yourself, well what do I do with this information? (and that is a totally valid question to be asking)

3.8 TL;DR

If you do not have time to go through the entire user guide, we have summarized a couple important tidbits that we think you should know before using the software.

- The first is that the userbase for the initial pySYD release was intended for non-expert astronomers. **With this in mind, the software was originally developed to be as hands-off as possible – as a **strictly** command-line end-to-end tool.** However since then, the software has become more modular in recent updates, thus enabling broader capabilities that can be used across other applications (e.g., Jupyter notebooks).
 - In addition to being a command-line tool, the software is optimized for running many stars. This means that many of the options that one would typically use or prefer, such as printing output information and displaying figures, is `False` by default. For our purposes here though, we will invoke them to better understand how the software operates.
-

USER GUIDE

There is a healthy mixture of extra details, more examples, (probably uninteresting) definitions and different applications.

We have been actively developing these broader capabilities so we are very excited to share the software with you!

4.1 Introduction

As we have alluded to throughout the documentation, pySYD was intended to be used through its command-line interface (CLI) – which means that the software is specifically optimized for this usage and therefore most options probably have the best defaults already set. Here, “best” just means that the defaults work *best* for most stars.

However, that does not necessarily mean that your star(s) or setting(s) are expected to conform or adhere to these settings. In fact, we recommend playing around with some of the settings to see how it affects the results, which might help build your intuition for seismic analyses.

Note: Please keep in mind that, while we have extensively tested a majority of our options, we are continuously adding new ones which ultimately might break something. If this happens, we encourage you to submit an issue [here](#) and thank you in advance for helping make pySYD even better!

4.1.1 CLI help

To give you a glimpse into the insanely large amount of available options, open up a terminal window and enter the help command for the main pipeline execution (run aka *pysyd.pipeline.run*), since this mode inherits all command-line parsers.

```
$ pysyd run --help

usage: pySYD run [-h] [--in str] [--infdir str] [--out str] [-s] [-o] [-v]
                [--cli] [--notebook] [--star [str [str ...]]] [--file str]
                [--info str] [--gap int] [-x] [--of int] [-k]
                [--dnu [float [float ...]]] [--le [float [float ...]]]
                [--ue [float [float ...]]] [-n] [-e] [-j] [--def str]
                [--sw float] [--bin float] [--bm str] [--step float]
                [--trials int] [-a] [--lx [float [float ...]]]
                [--ux [float [float ...]]] [-b] [--basis str] [--bf float]
                [--iw float] [--rms int] [--laws int] [-w] [--metric str]
                [--lb [float [float ...]]] [--ub [float [float ...]]] [-g]
```

(continues on next page)

(continued from previous page)

```
[--numax [float [float ...]]] [--lp [float [float ...]]]
[--up [float [float ...]]] [--ew float] [--sm float]
[--sp float] [-f] [--thresh float] [--peak int] [--mc int]
[-m] [--all] [-d] [--cm str] [--cv float] [-y] [-i]
[--nox int] [--noy str] [--npb int] [--se float]
```

optional arguments:

```
-h, --help          show this help message and exit
```

This was actually just a teaser!

If you ran it from your end, you probably noticed an output that was a factor of ~5-10 longer! It may seem like an overwhelming amount but do not fret, this is for good reason – and that’s to make your asteroseismic experience as customized as possible.

Currently pySYD has four parsers: the `parent_parser` for high-level functionality, the `data_parser` for anything related to data loading and manipulation, the `main_parser` for everything related to the core analyses, and the `plot_parser` for (yes, you guessed it!) plotting. In fact, the main parser is so large that comprises four subgroups, each related to the corresponding steps in the main pipeline execution. **BTW** see [here](#) for more information on which parsers a given pipeline mode inherits.

Sections

- *high-level functionality*
- *data analyses*
- *core asteroseismic analyses*
 - *search & estimate*
 - *background fit*
 - *global fit*

Note: as you are navigating this page, keep in mind that we also have a special [glossary](#) for all our command-line options. This includes everything from the variable type, default value and relevant units to how it’s stored within the software itself. There are glossary links at the bottom of every section for each of the parameters discussed within that subsection.

4.1.2 High-level functionality

aka the `parent_parser`

All pySYD modes inherit the `parent_parser` and therefore, mostly pertains to paths and how you choose to run the software (i.e. save files and if so, whether or not to overwrite old files with the same extension, etc.)

```
--in str, --input str, --inpdire str
                        Input directory
--infdire str          Path to relevant pySYD information
--out str, --outdir str, --output str
                        Output directory
-s, --save             Do not save output figures and results.
```

(continues on next page)

(continued from previous page)

<code>-o, --overwrite</code>	Overwrite existing files with the same name/path
<code>-v, --verbose</code>	turn on verbose output
<code>--cli</code>	Running from command line (this should not be touched)
<code>--notebook</code>	Running from a jupyter notebook (this should not be touched)

Glossary terms (alphabetical order): `-cli`, `-file`, `-in`, `-info`, `-information`, `-inpdire`, `-input`, `-list`, `-notebook`, `-o`, `-out`, `-overwrite`, `-s`, `-save`, `-outdir`, `-output`, `-todo`, `-v`, `-verbose`

4.1.3 Data analyses

aka the data_parser

The following features are primarily related to the input data and when applicable, what tools to apply to the data. All data manipulation relevant to this step happens *prior* to any pipeline analyses. **Currently this is mostly frequency-domain tools but we are working on implementing time-domain tools as well!**

<code>--star [str [str ...]]</code>	<code>--stars [str [str ...]]</code> list of stars to process
<code>--file str</code>	<code>--list str</code> , <code>--todo str</code> list of stars to process
<code>--info str</code>	<code>--information str</code> list of stellar parameters and options
<code>--gap int</code>	<code>--gaps int</code> What constitutes a time series 'gap' (i.e. n x the cadence)
<code>-x</code>	<code>--stitch</code> , <code>--stitching</code> Correct for large gaps in time series data by 'stitching' the light curve
<code>--of int</code>	<code>--over int</code> , <code>--oversample int</code> The oversampling factor (OF) of the input power spectrum
<code>-k</code>	<code>--kc</code> , <code>--kepcorr</code> Turn on the Kepler short-cadence artefact correction routine
<code>--dnu [float [float ...]]</code>	spacing to fold PS for mitigating mixed modes
<code>--le [float [float ...]]</code>	<code>--lowere [float [float ...]]</code> lower frequency limit of folded PS to whiten mixed modes
<code>--ue [float [float ...]]</code>	<code>--uppere [float [float ...]]</code> upper frequency limit of folded PS to whiten mixed modes
<code>-n</code>	<code>--notch</code> another technique to mitigate effects from mixed modes (not fully functional, creates weirds effects for higher SNR cases??)

Glossary terms (alphabetical order): `-dnu`, `-k`, `-le`, `-lowere`, `-kc`, `-kepcorr`, `-of`, `-over`, `-oversample`, `-star`, `-stars`, `-stitch`, `-stitching`, `-ue`, `-uppere`, `-x`

4.1.4 Core asteroseismic analyses

aka the `main_parser`

The main parser holds a majority of the parameters that are relevant to core functions of the software. Since it is so large, it is broken down into four different “groups” which are related to their application.

Search & estimate

The following options are relevant for the first, optional module that is designed to search for power excess due to solar-like oscillations and estimate rough starting points for its main properties.

```
-e, --est, --estimate          Turn off the optional module that estimates numax
-j, --adjust                  Adjusts default parameters based on region of
                              oscillations
--def str, --defaults str      Adjust defaults for low vs. high numax values (e.g.,
                              smoothing filters)
--sw float, --smoothwidth float
                              Box filter width (in muHz) for smoothing the PS
--bin float, --binning float   Binning interval for PS (in muHz)
--bm str, --mode str, --bmode str
                              Binning mode
--step float, --steps float
--trials int, --ntrials int
-a, --ask                     Ask which trial to use
--lx [float [float ...]], --lowerx [float [float ...]]
                              Lower frequency limit of PS
--ux [float [float ...]], --upperx [float [float ...]]
                              Upper frequency limit of PS
```

Glossary terms (alphabetical order): *-a, -ask, -bin, -binning, -bm, -bmode, -e, -est, -estimate, -lowerx, -lx, -mode, -ntrials, -step, -steps, -sw, -smoothwidth, -trials, -upperx, -ux*

Background fit

Below is a complete list of parameters relevant to the background-fitting routine:

```
-b, --bg, --background        Turn off the routine that determines the stellar
                              background contribution
--basis str                   Which basis to use for background fit (i.e. 'a_b',
                              'pgran_tau', 'tau_sigma'), *** NOT operational yet ***
--bf float, --box float, --boxfilter float
                              Box filter width [in muHz] for plotting the PS
--iw float, --indwidth float   Width of binning for PS [in muHz]
--rms int, --nrms int          Number of points to estimate the amplitude of red-
                              noise component(s)
```

(continues on next page)

(continued from previous page)

```

--laws int, --nlaws int
                        Force number of red-noise component(s)
-w, --wn, --fixwn      Fix the white noise level
--metric str           Which model metric to use, choices=['bic','aic']
--lb [float [float ...]], --lowerb [float [float ...]]
                        Lower frequency limit of PS
--ub [float [float ...]], --upperb [float [float ...]]
                        Upper frequency limit of PS

```

Glossary terms (alphabetical order): *-b, -background, -basis, -bf, -bg, -box, -boxfilter, -fixwn, -iw, -indwidth, -laws, -lb, -lowerb, -metric, -nrms, -rms, -nlaws, -ub, -upperb, -w, -wn*

Global parameters

All of the following are related to deriving global asteroseismic parameters, *numax* (ν_{\max}) and *dnu* ($\Delta\nu$).

```

-g, --globe, --global
                        Disable the main global-fitting routine
--numax [float [float ...]]
                        initial estimate for numax to bypass the forst module
--lp [float [float ...]], --lowerp [float [float ...]]
                        lower frequency limit for the envelope of oscillations
--up [float [float ...]], --upperp [float [float ...]]
                        upper frequency limit for the envelope of oscillations
--ew float, --exwidth float
                        fractional value of width to use for power excess,
                        where width is computed using a solar scaling
                        relation.
--sm float, --smpar float
                        smoothing parameter used to estimate the smoothed
                        numax (typically before 1-4 through experience --
                        **development purposes only**)
--sp float, --smoothps float
                        box filter width [in muHz] of PS for ACF
-f, --fft              Use :mod:`numpy.correlate` instead of fast fourier
                        transforms to compute the ACF
--thresh float, --threshold float
                        fractional value of FWHM to use for ACF
--peak int, --peaks int, --npeaks int
                        number of peaks to fit in the ACF

```

Glossary terms (alphabetical order): *-ew, -exwidth, -g, -global, -globe, -lp, -lowerp, -npeaks, -numax, -peak, -peaks, -sm, -smpar, -up, -upperp -dnu, -sp, -smoothps, -thresh*

Sampling & uncertainties

All CLI options relevant for the Monte-Carlo sampling in order to estimate uncertainties:

<code>--mc int, --iter int, --mciter int</code>	number of Monte-Carlo iterations to run for estimating uncertainties (typically 200 is sufficient)
<code>-m, --samples</code>	save samples from the Monte-Carlo sampling

Glossary terms (alphabetical order): *-iter, -m, -mc, -mciter, -samples*

4.1.5 Plotting

aka the plot_parser

Anything related to the plotting of results for *any* of the modules is in this parser. Its currently a little heavy on the *echelle diagram* end because this part of the plot is harder to hack, so we tried to make it as easily customizable as possible.

<code>--all, --showall</code>	plot background comparison figure
<code>-d, --show, --display</code>	show output figures
<code>--cm str, --color str</code>	Change colormap of ED, which is `binary` by default
<code>--cv float, --value float</code>	Clip value multiplier to use for echelle diagram (ED). Default is 3x the median, where <code>clip_value == `3`</code> .
<code>-y, --hey</code>	plugin for Daniel Hey's echelle package **not currently implemented**
<code>-i, --ie, --interpech</code>	turn on the interpolation of the output ED
<code>--nox int, --nacross int</code>	number of bins to use on the x-axis of the ED (currently being tested)
<code>--noy str, --ndown str, --norders str</code>	NEW!! Number of orders to plot pm how many orders to shift (if ED is not centered)
<code>--npb int</code>	NEW!! npb == "number per bin", which is option instead of nox that uses the frequency resolution and spacing to compute an appropriate bin size for the ED
<code>--se float, --smoothech float</code>	Smooth ED using a box filter [in muHz]

Glossary terms (alphabetical order): *-ce, -cm, -color, -cv, -d, -display, -hey, -i, -ie, -interpech, -nox, -nacross, -ndown, -norders, -noy, -npb, -se, -show, -smoothech, -value, -y*

On the next page, we will show applications for some of these options in command-line examples.

We also have our advanced usage page, which is specifically designed to show these in action by providing before and after references. You can also find descriptions of certain commands available in the notebook tutorials.

This page has command-line examples for different usage scenarios, including several customized single star applications as well as running an ensemble of stars using a single-line command.

We also tried to include examples demonstrating different signal-to-noise detections, including what to look for in each scenario.

4.2 Single star applications

For applications to single stars, we will start with a very easy, high signal-to-noise (SNR) example, followed by medium and low SNR examples as well as a null detection. These examples will not be as detailed as the *quickstart example* – our goal here is to provide pointers on what to look for in each case.

4.2.1 A diablo detection

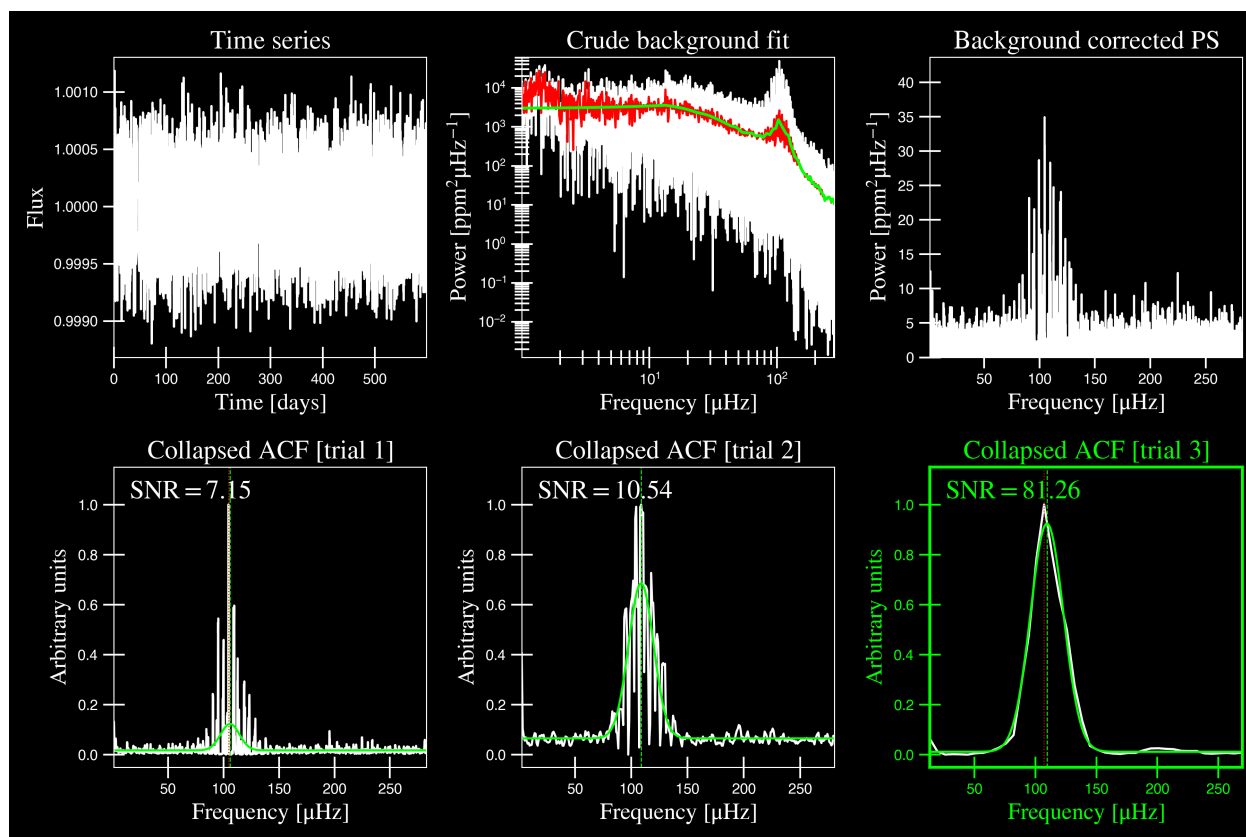
KIC 11618103 is our most evolved example, an RGB star with numax of $\sim 100\mu\text{Hz}$. We will now admit that while the default settings work for *most* stars, some of the defaults could/should (or even in some cases *need*) to be changed for more evolved stars like this example.

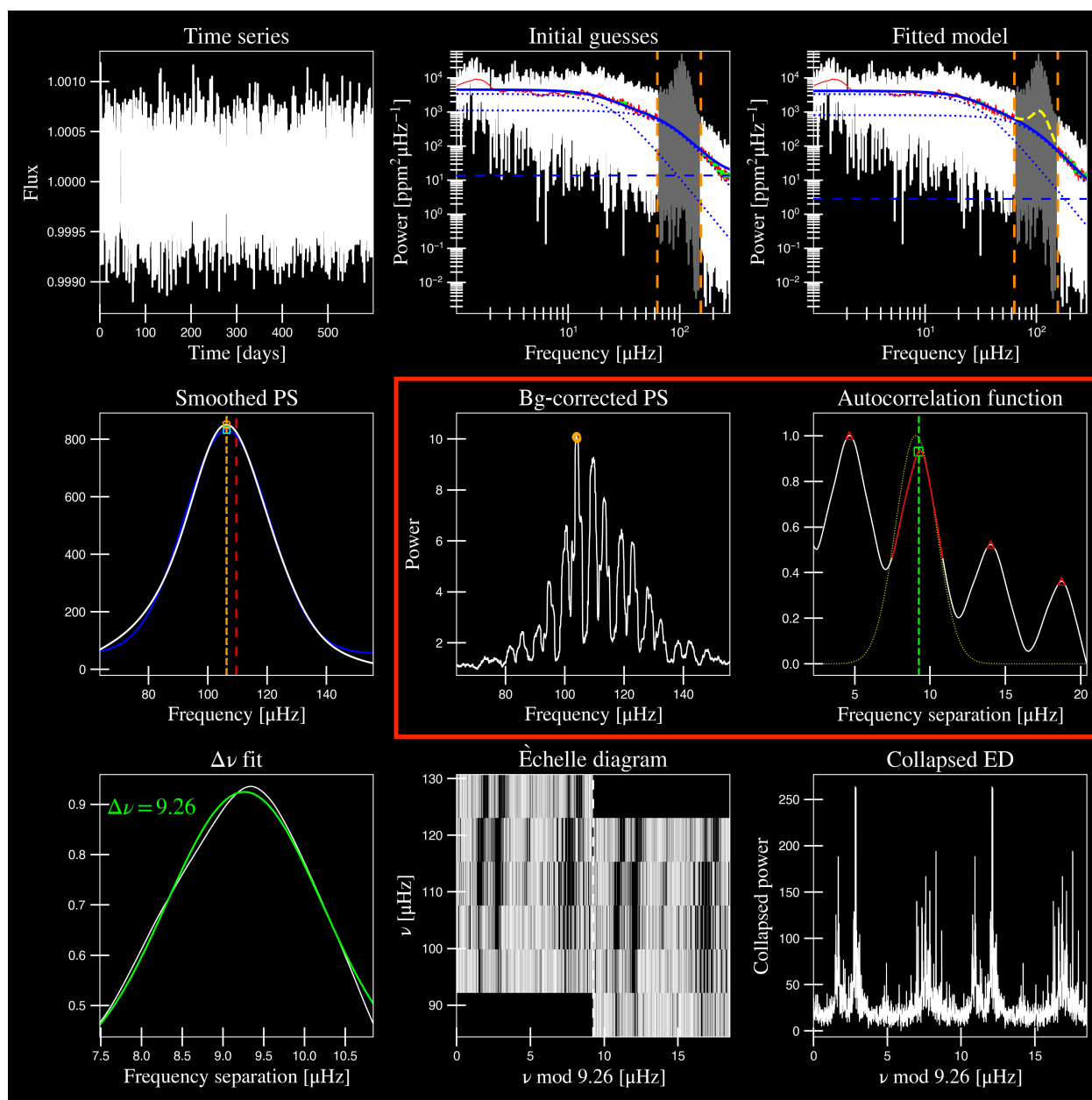
It doesn't necessarily mean that it will get the answers wrong, but we will take you through a few different runs and change some of the settings with each run.

Run 1:

If we run it straight “out-of-the-box” with our usual command:

```
pysyd run --star 11618103 -dv
```





The autocorrelation function (or *ACF*) in panel 6 looks very smooth - I'd say almost a little *too* smooth. In fact if you look at the panel directly to the left under “Bg-corrected PS”, the power spectrum also looks a little strange, right?

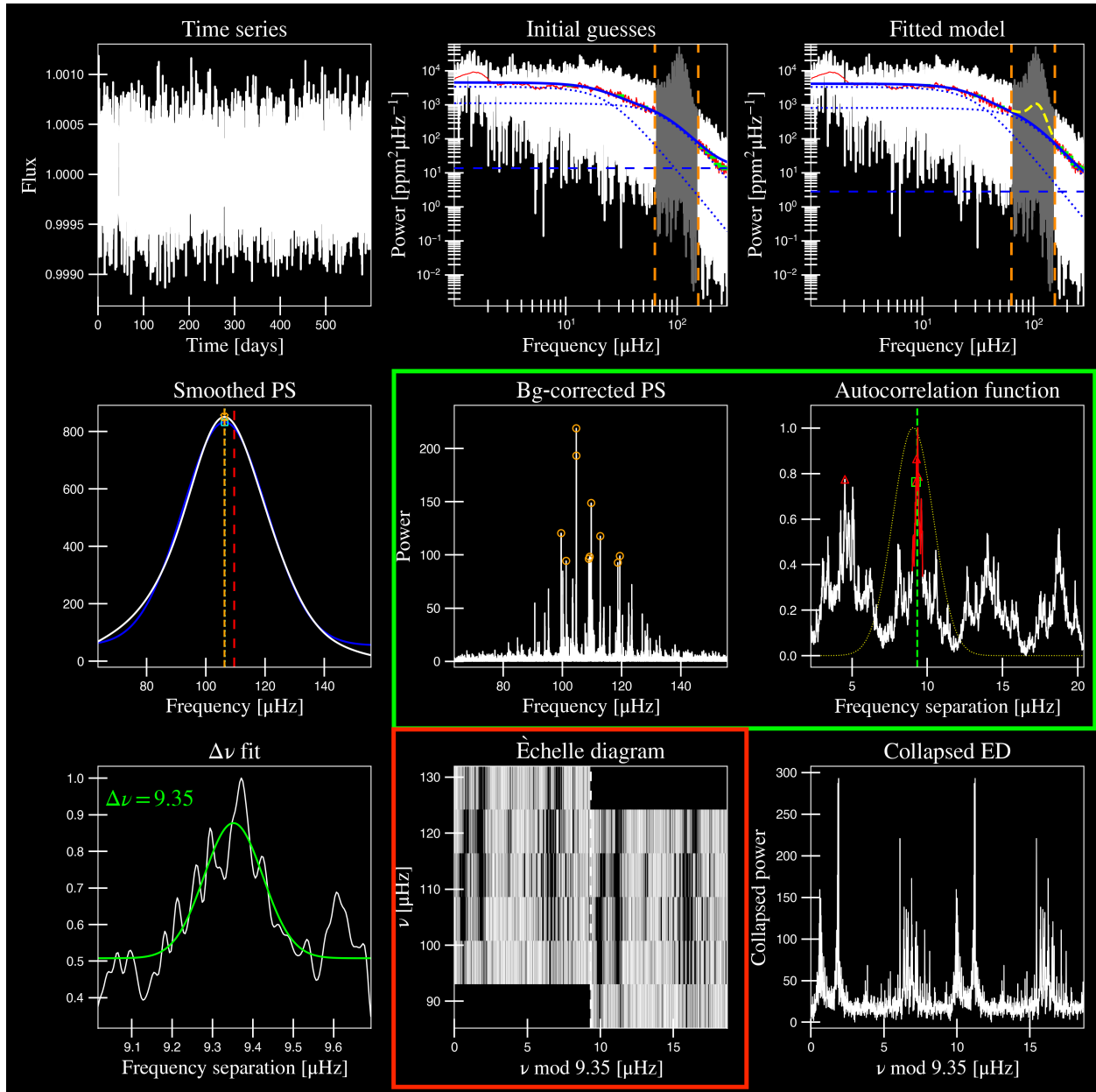
This is because our smoothing filter (or box filter) has a default value of $2.5\mu\text{Hz}$, which is quite high for this star. Typically a common value is $1.0\mu\text{Hz}$, if at all, but usually much much less than our expected numax.

Run 2:

So for our first change, we are going to tone down the “smoothing” by setting it to zero i.e. not smoothing it at all. We can see how that will affect the calculated ACF (again, panels 5+6).

```
pysyd run --star 11618103 -dv --sp 0.0
```

Since we are not changing anything from the first part, we will leave out the first plot for brevity.



As you can see above, the bg-corrected power spectrum and ACF both look more reasonable now – it didn’t change the quality of the fit or our answer but it definitely looks better.

Now if you look at the echelle diagram (panel 8), it almost looks like we aren’t capturing all oscillation modes – our ridges look cut off so let’s plot more bins on the y axis.

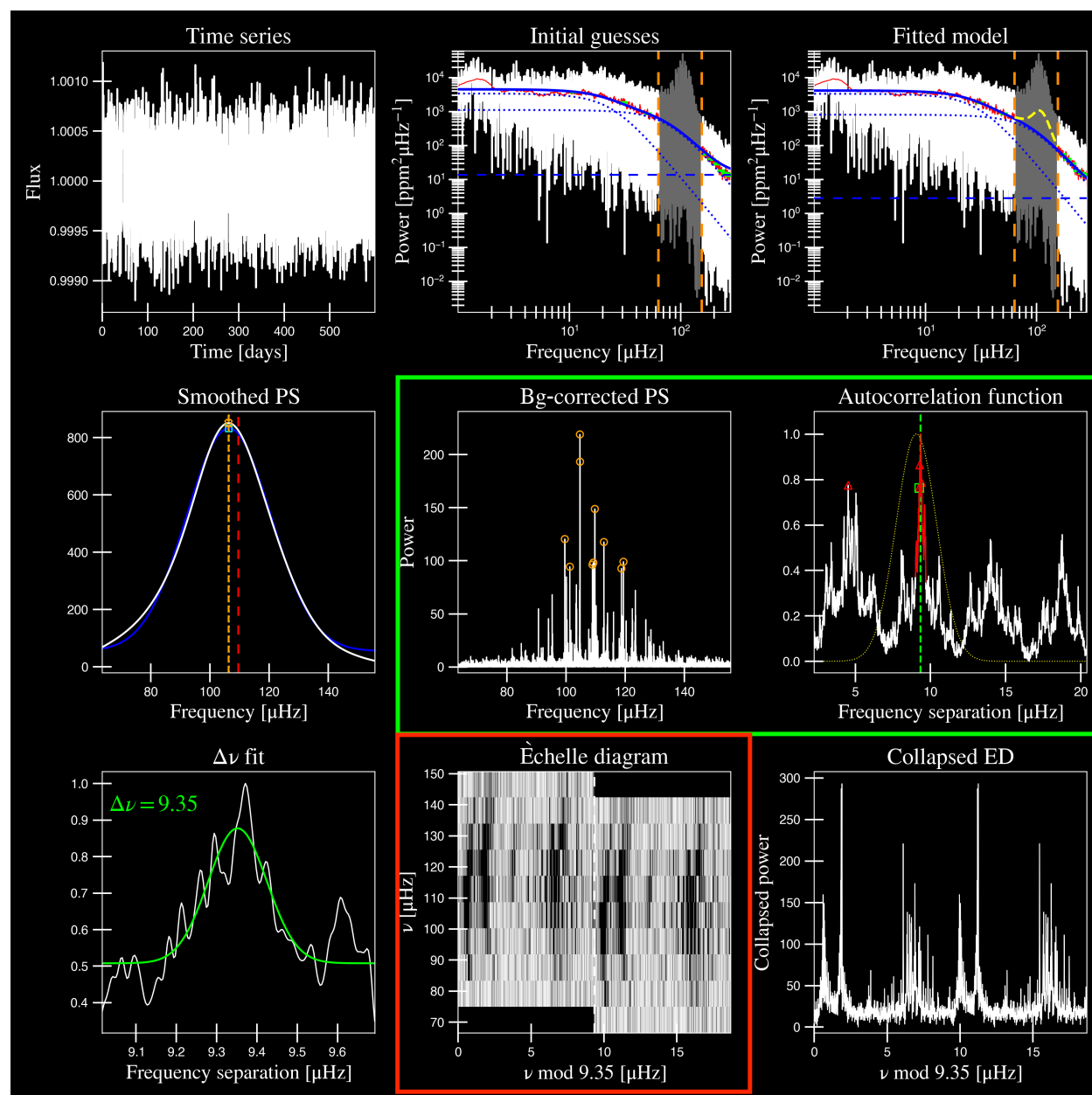
Run 3:

We've tried to make the commands as obvious as possible to make it easier to digest. For example, here we are changing the number of bins on the y axis (or *-noy*-*noy*) of the echelle diagram, which is currently equal to 5 (also corresponds to 5 radial orders).

Let's change it to something higher.

```
pySYD run --star 11618103 -dv --sp 0.0 --noy 9+0
```

You'll see that we provided a keyword argument with a length of 3. The first digit is the number of bins (or radial orders) to plot and the next two digits provide the ability to shift the entire plot up/down by n orders as well! If 0 is provided as the second part of this value, it will center it on our expected numax. FWIW: *-noy 9-0* would plot exactly the same thing.



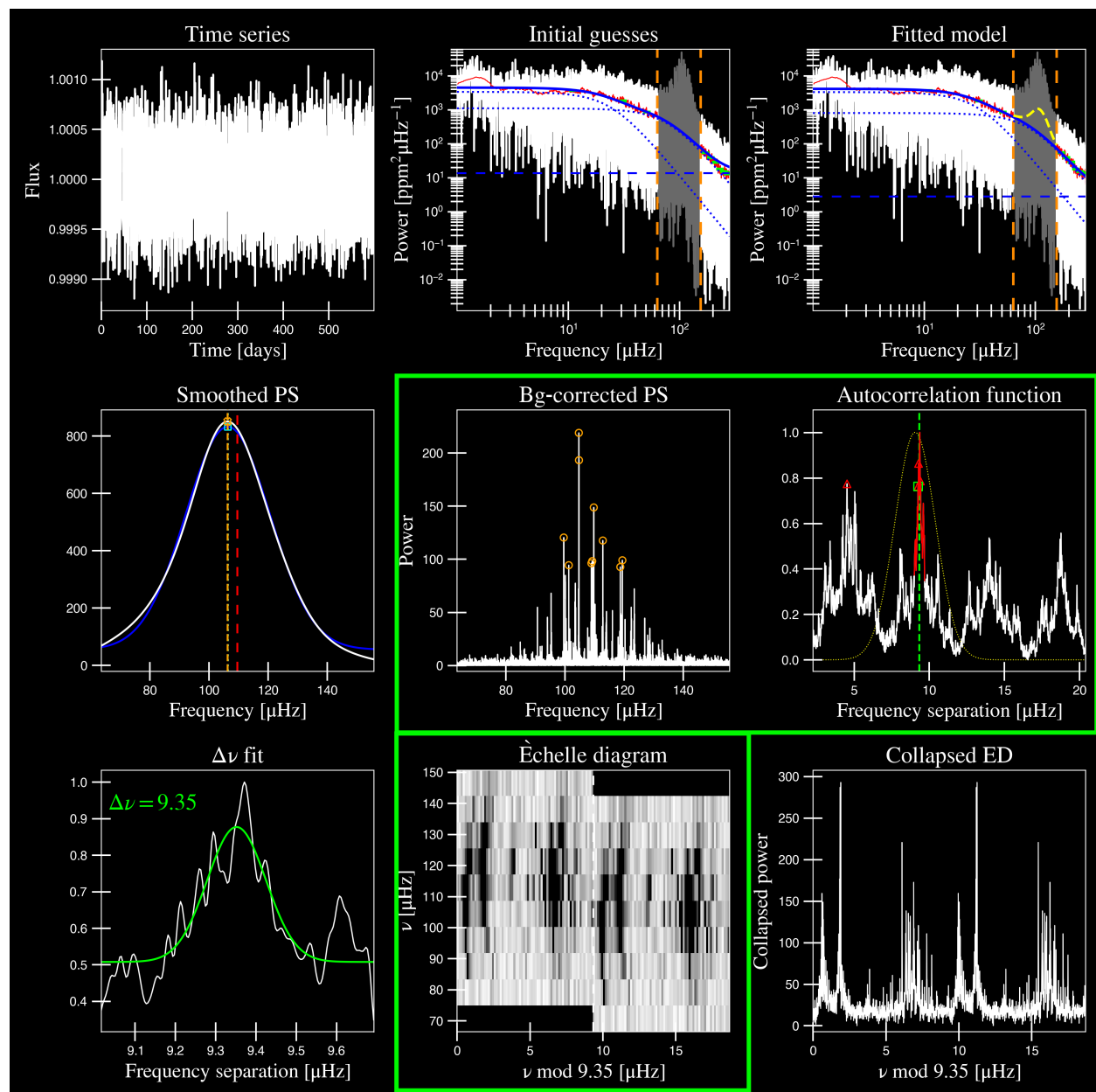
This looks a lot better and it looks like we are capturing all features in the new y-axis range. Turns out we can also change the number of bins (or bin resolution) on the x axis of the echelle diagram as well.

Run 4:

Using basic logic, you can deduce that the relevant keyword argument here is indeed `-nox`. However, the number of bins on the x axis is more arbitrary here and depends on a couple different things, primarily the spacing (or $\Delta\nu$) and the frequency resolution of the power spectrum.

Since changing the number of bins using `-nox` is somewhat arbitrary – we’ve created an additional argument that calculates the number of points per bin or `n timer` (`-npb`). Therefore this option uses information from both the spacing and the frequency resolution to estimate a more relevant number to use on the x axis.

```
pysyd run --star 11618103 -dv --sp 0.0 --noy 9+0 --npb 35
```

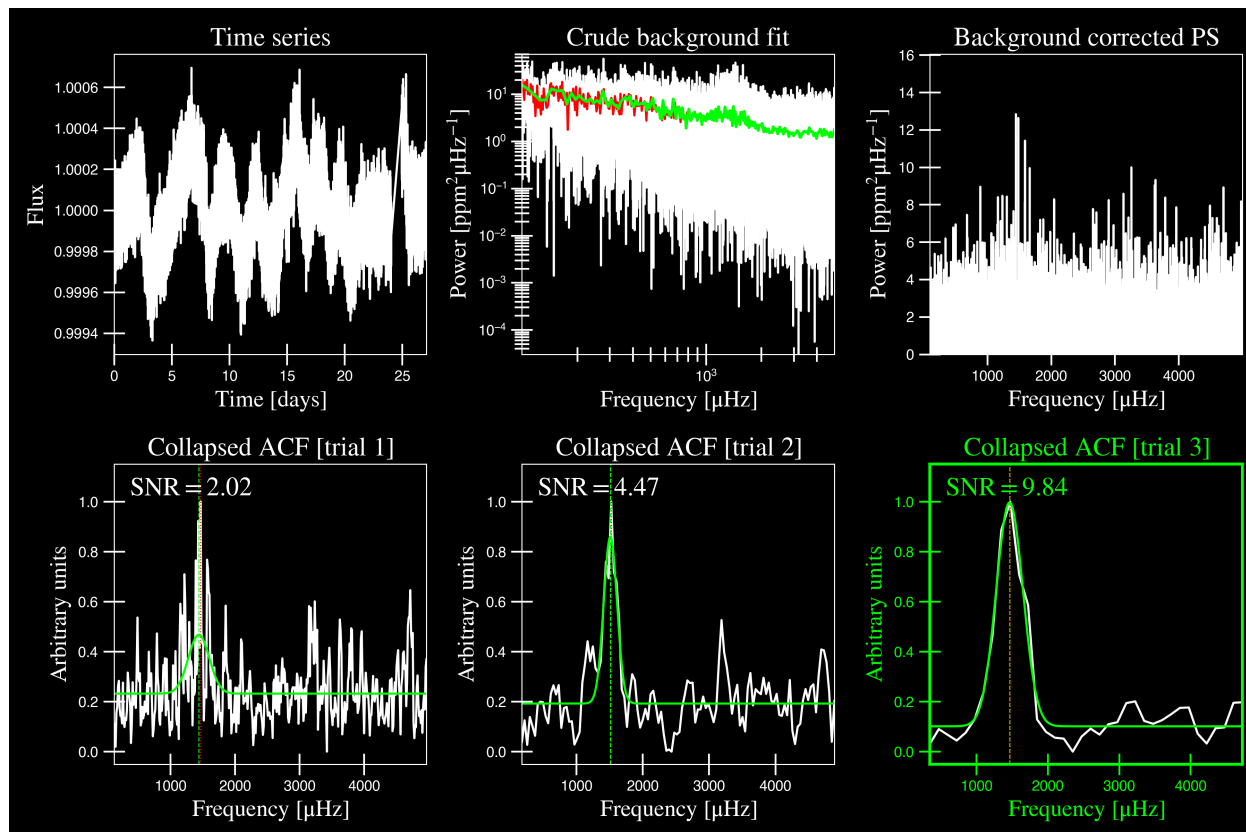


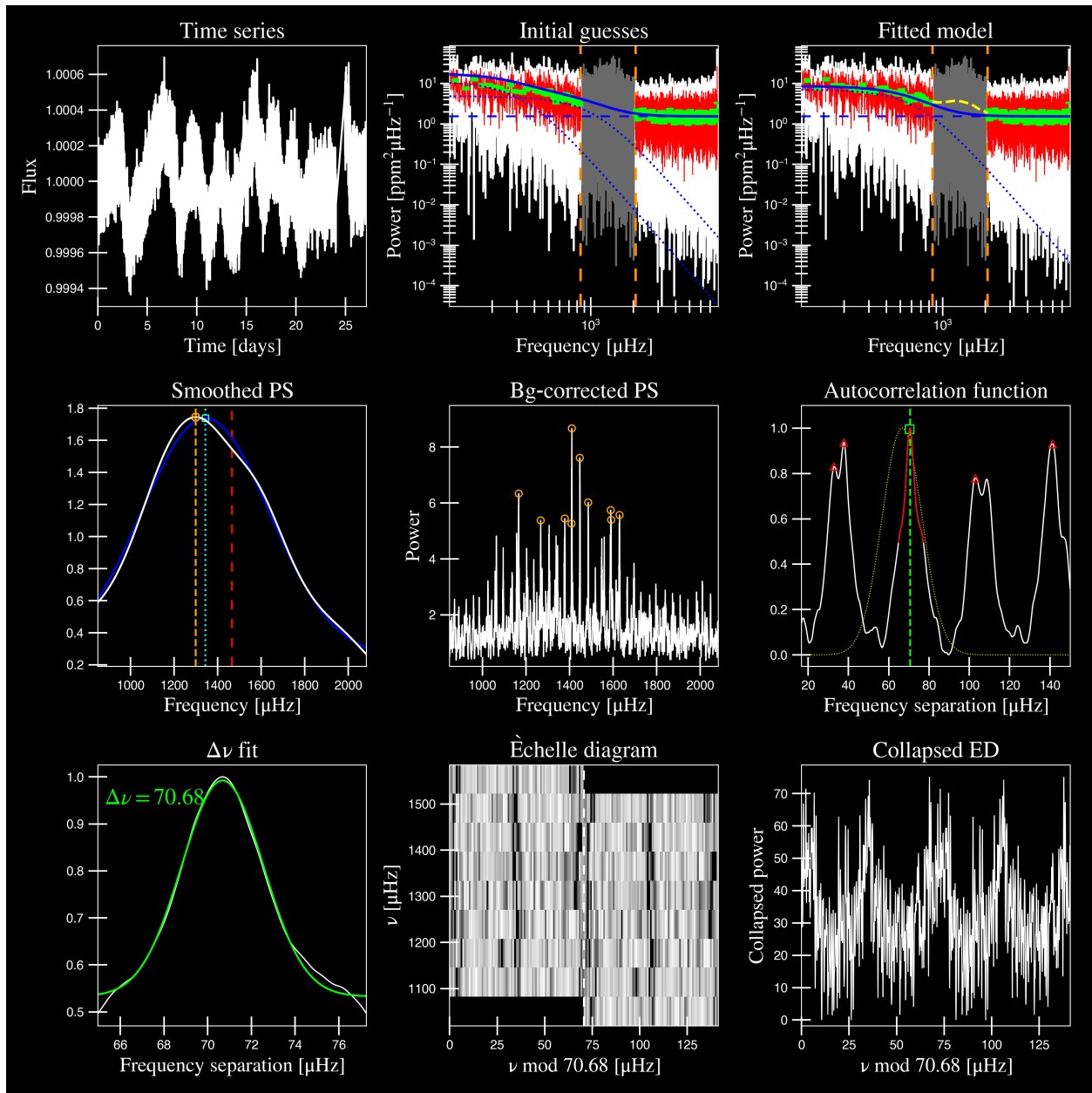
But this is just the tip of the iceberg – please see our complete [list of available options](#)!

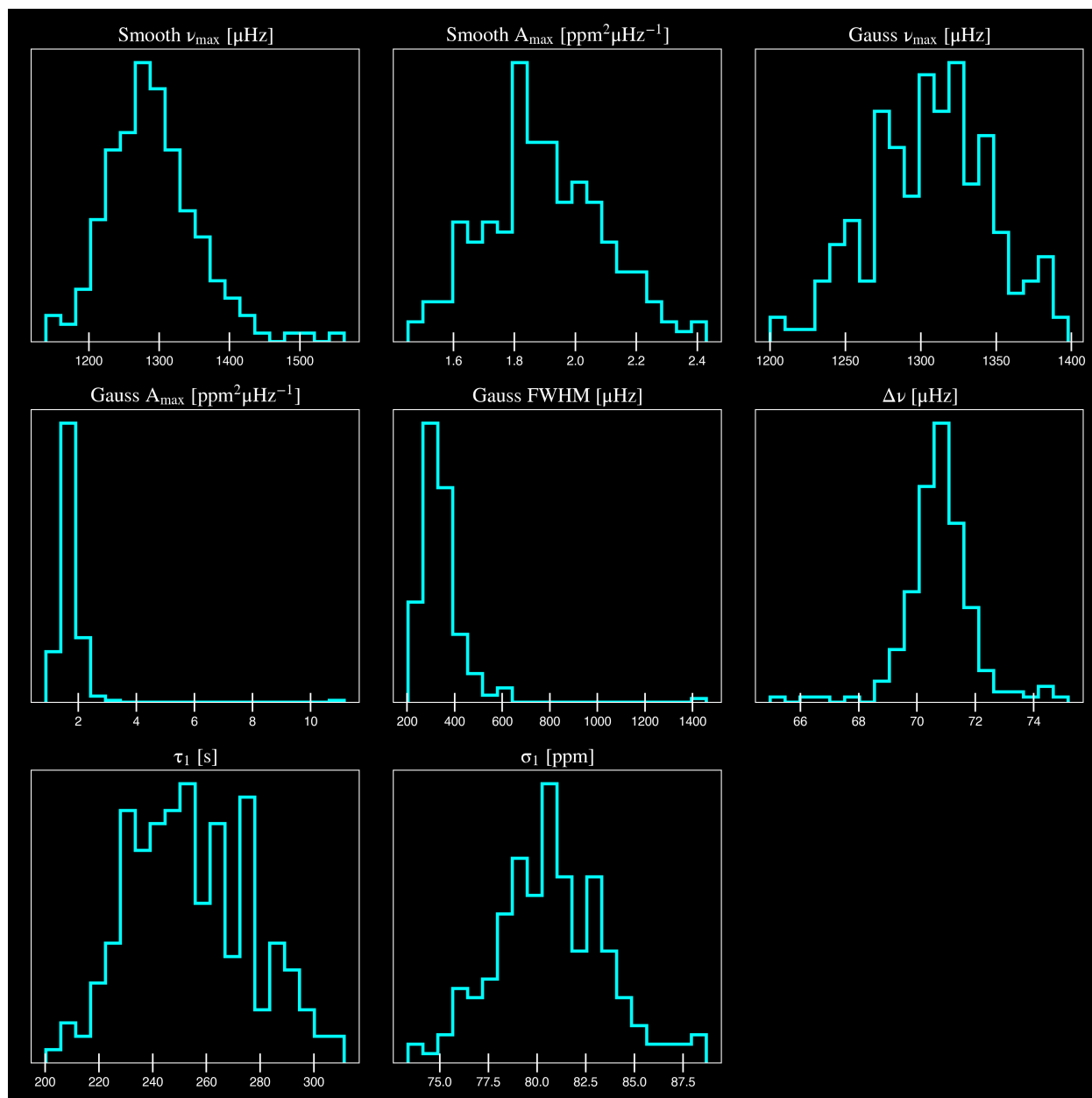
4.2.2 A hot-to-fire detection

(yes, we are using taco bell sauces to quantify the signal-to-noise of these cases) ``1`

We used this example for new users just getting started and therefore we will only show the output and figures. Feel free to visit our crash course in asteroseismology, or [crashteroseismology](#) page, which breaks down every step in great detail.



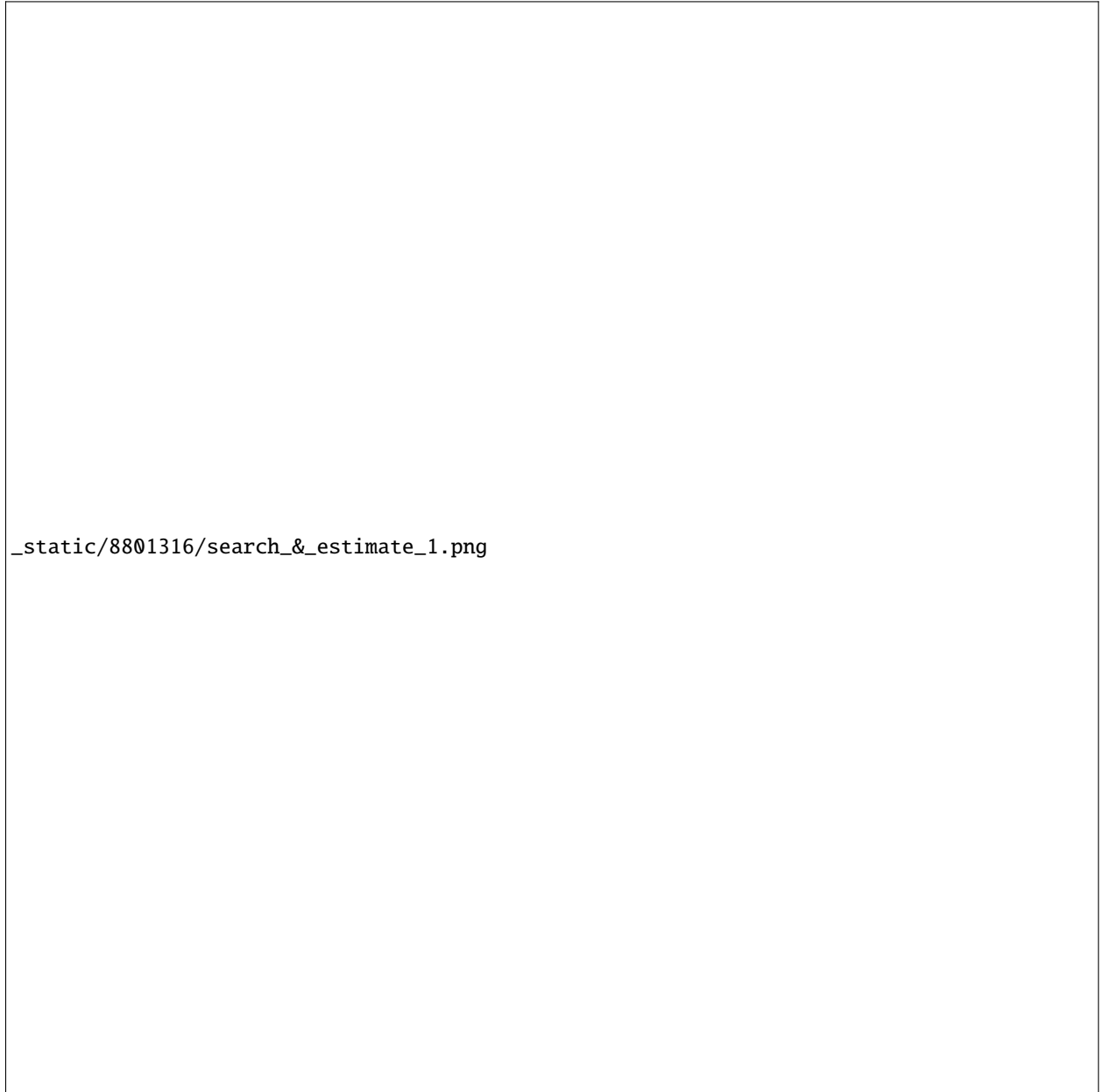




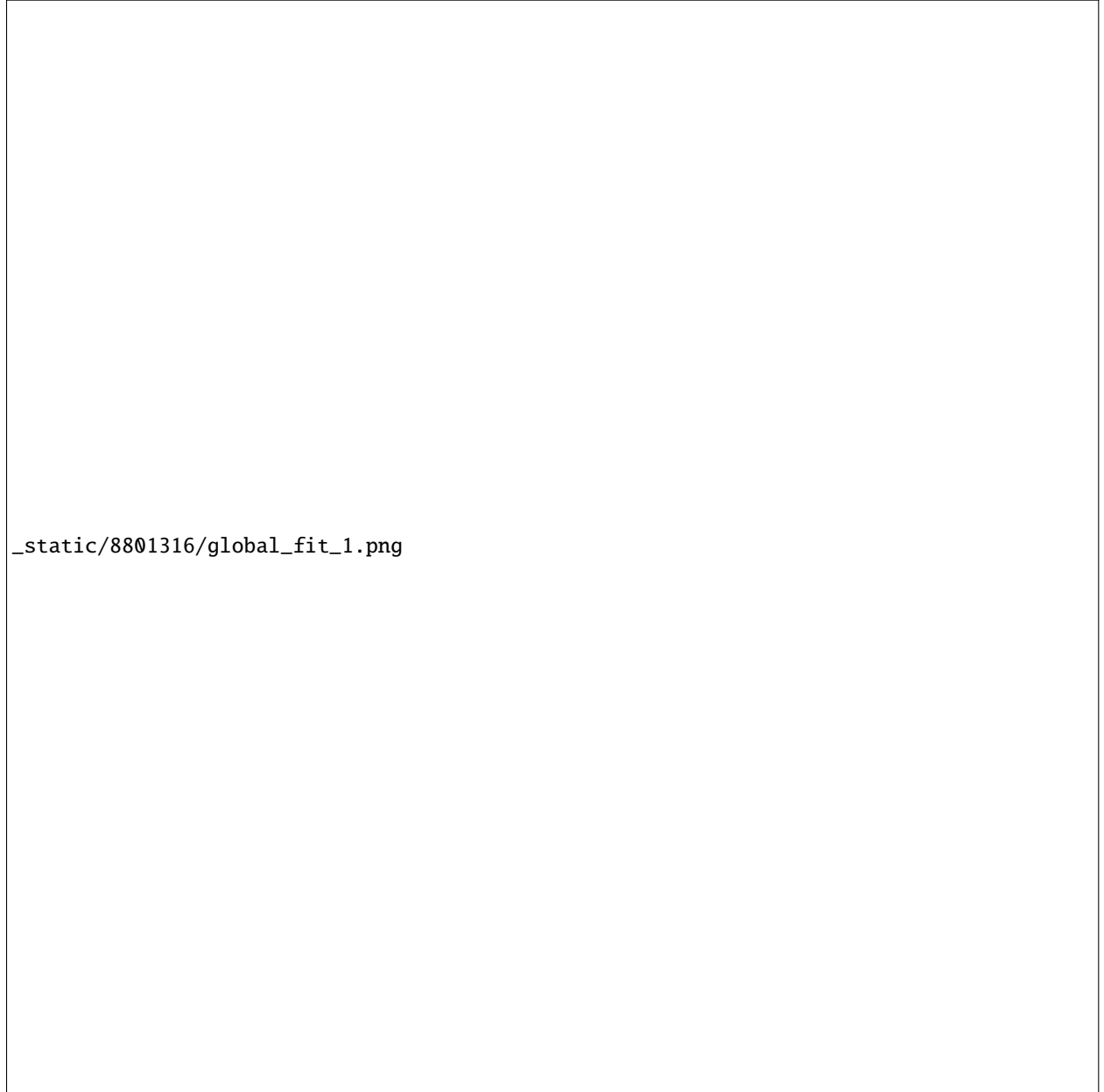
4.2.3 A mild detection

As if asteroseismology wasn't hard enough, let's make it even more difficult for you!

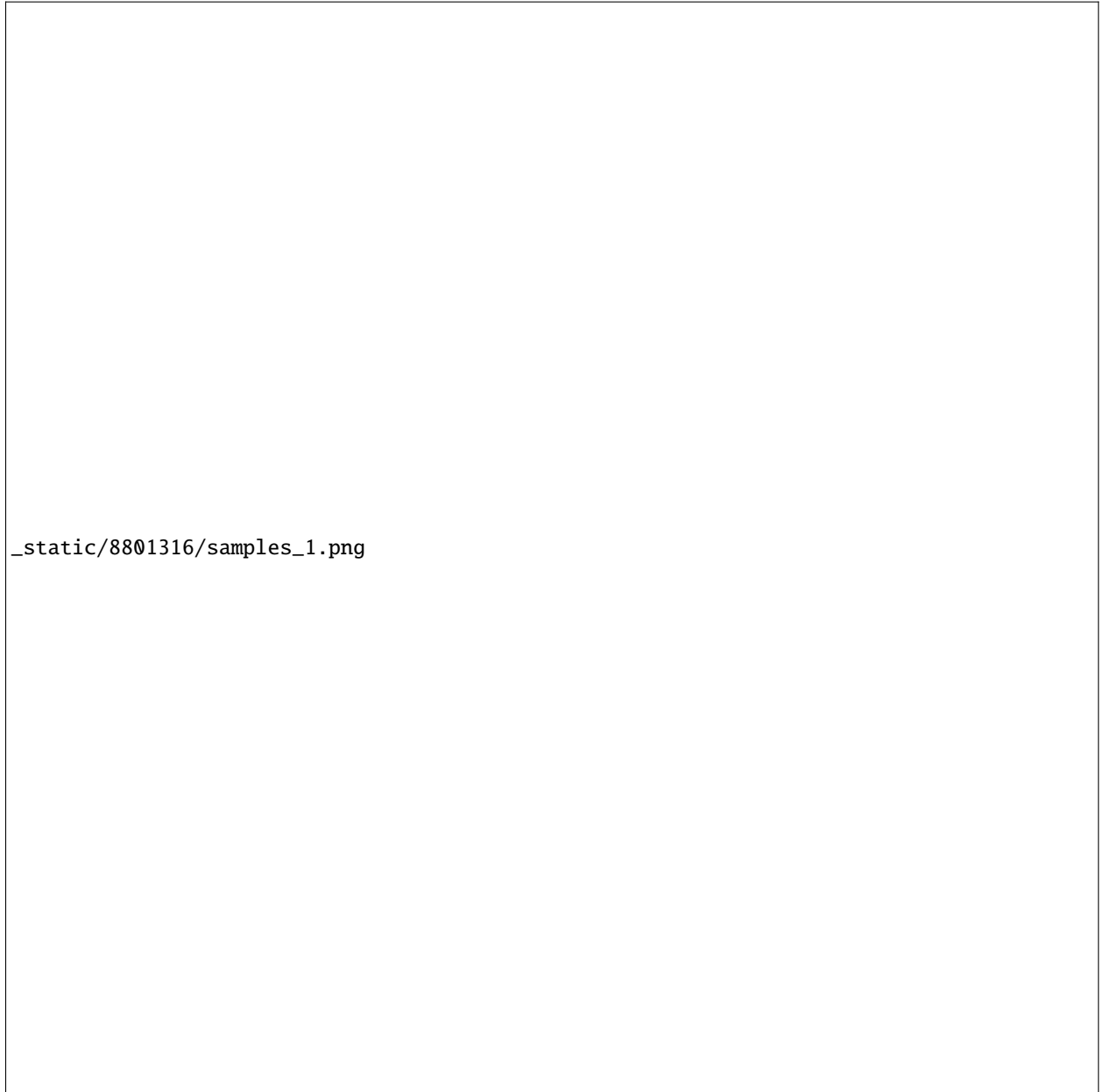
KIC 8801316 is a subgiant with a $\nu_{\max} \sim 1100$ μHz , shown in the figures below.



`_static/8801316/search_&_estimate_1.png`



_static/8801316/global_fit_1.png



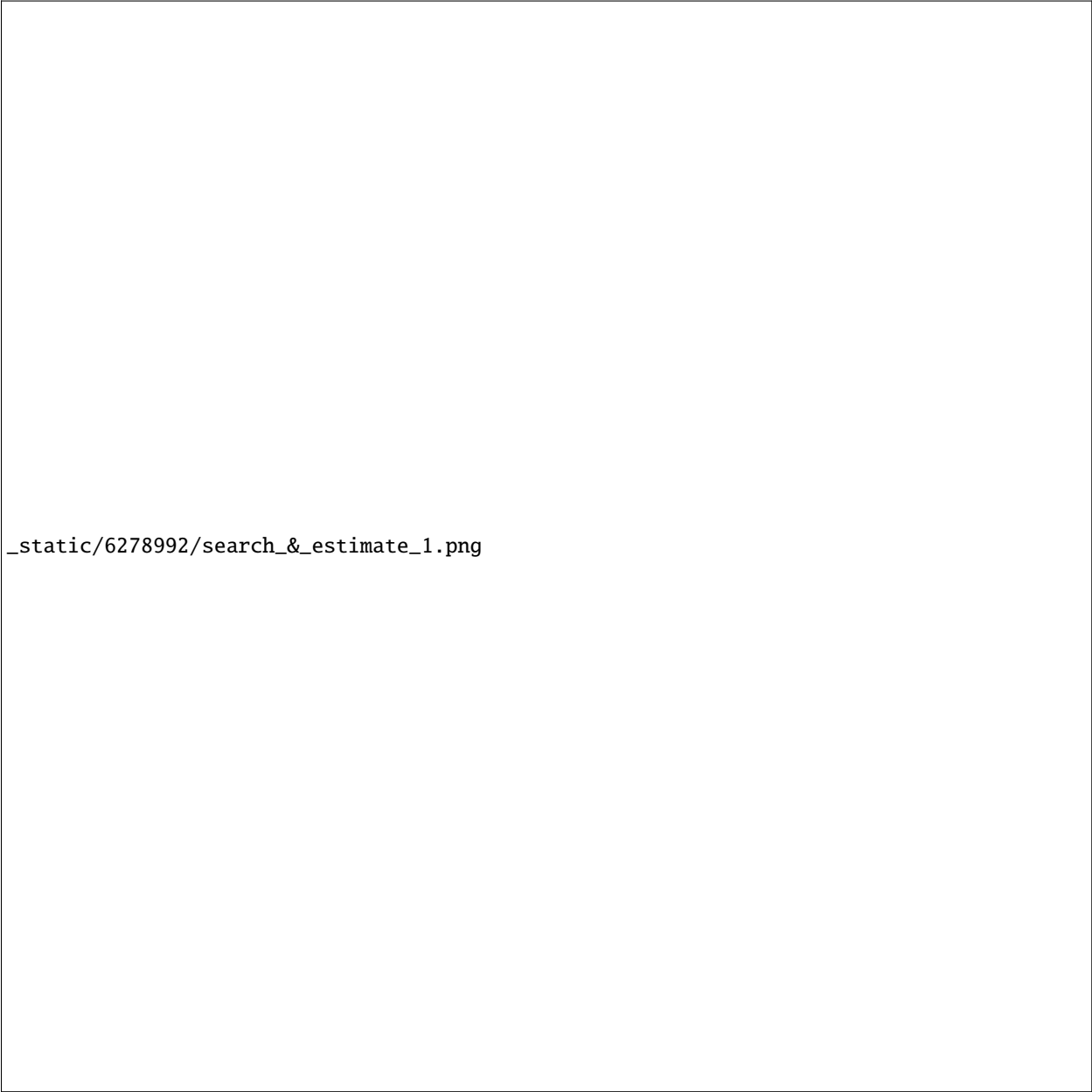
`_static/8801316/samples_1.png`

This would be classified as a detection despite the low SNR due to the following reasons:

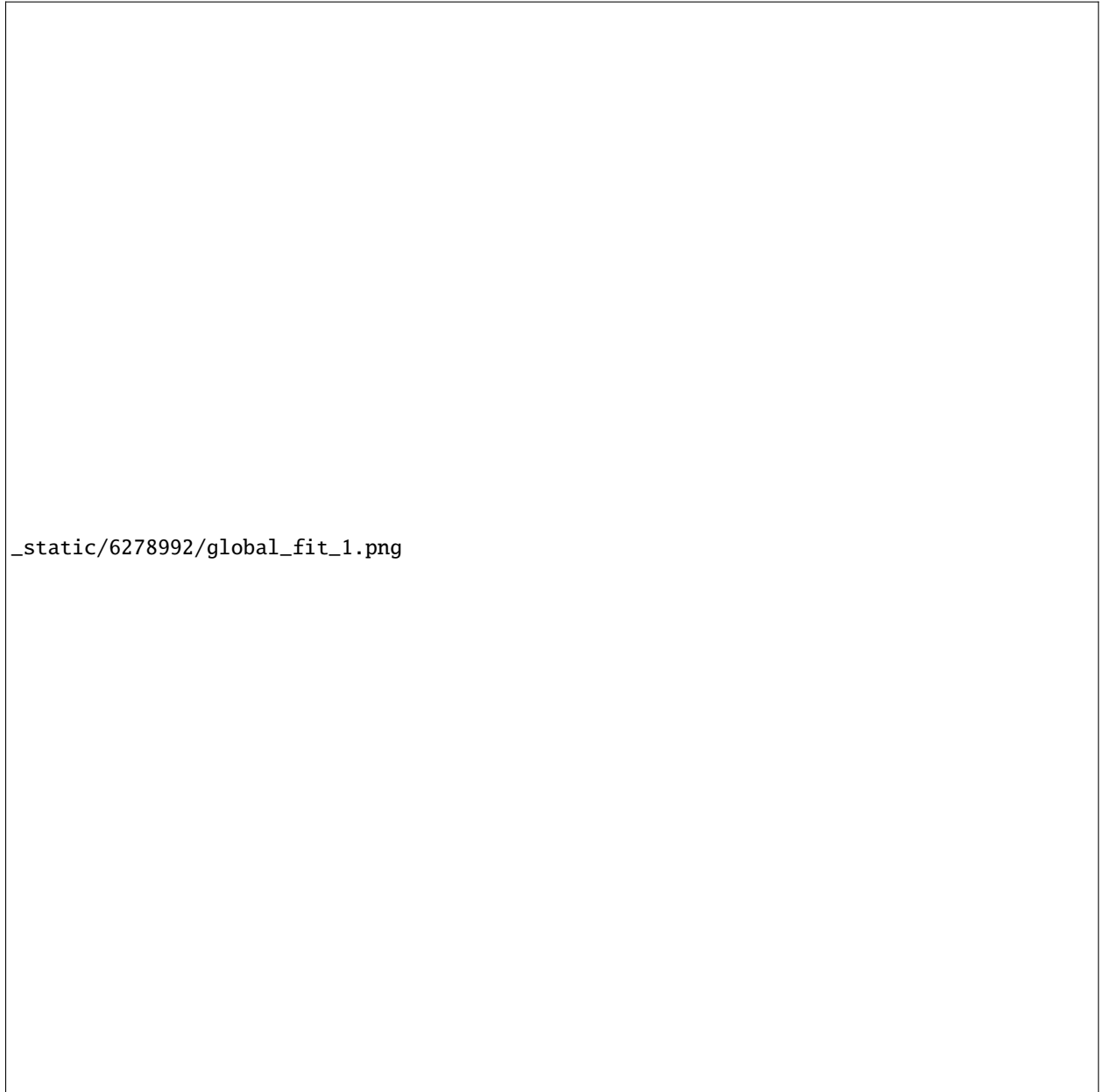
- there is a clear power excess as seen in panel 3
- the power excess has a Gaussian shape as seen in panel 5 corresponding to the solar-like oscillations
- the autocorrelation function (ACF) in panel 6 show periodic peaks
- the echelle diagram in panel 8 shows the ridges, albeit faintly

4.2.4 No SNR: KIC 6278992

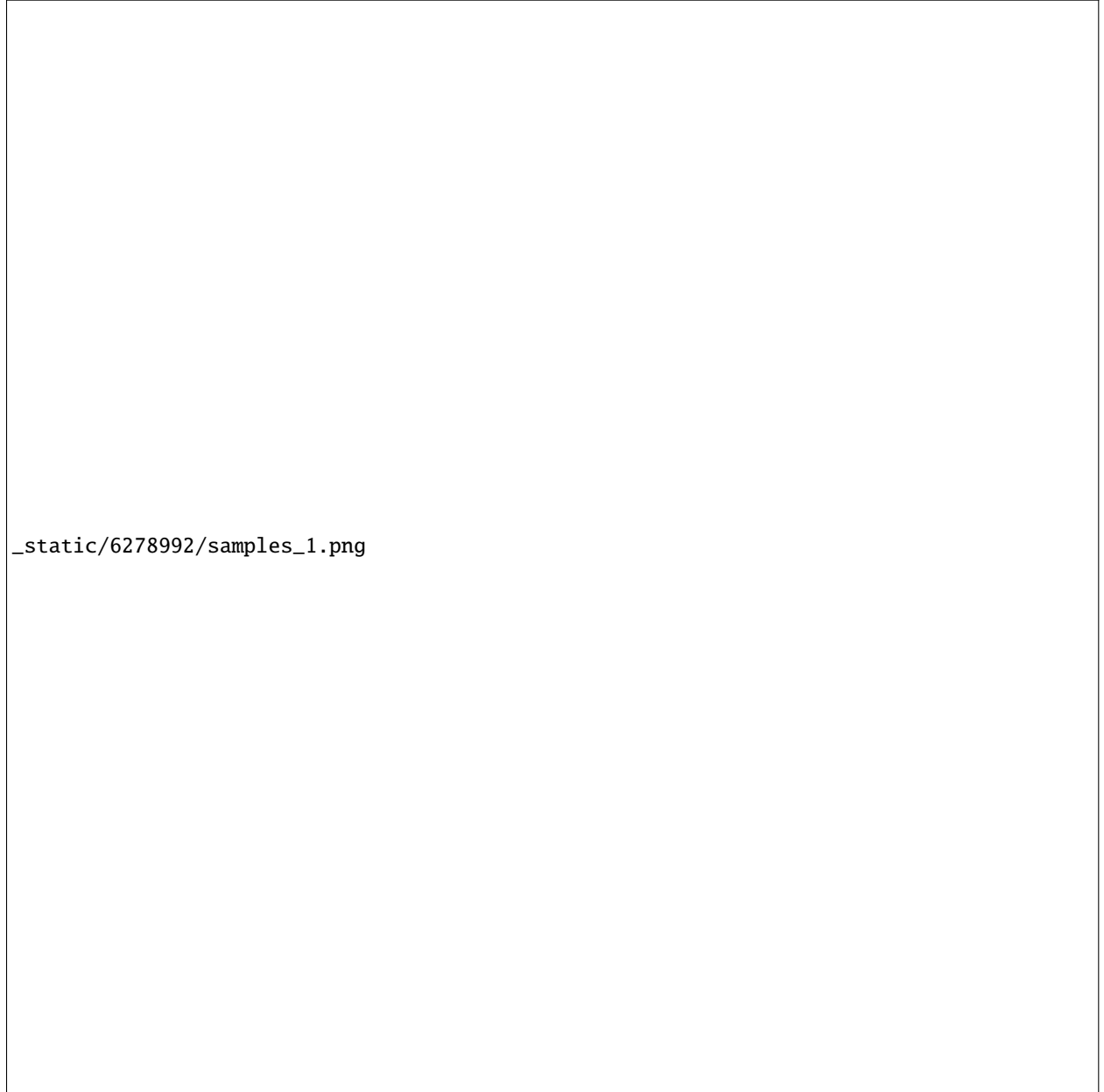
KIC 6278992 is a main-sequence star with no solar-like oscillations.



`_static/6278992/search_&_estimate_1.png`



`_static/6278992/global_fit_1.png`



`_static/6278992/samples_1.png`

4.3 Star sample

Depending on how large your sample is, you may choose to do it one of two ways.

4.3.1 Regular mode

Since this is optimized for running many stars via command line, the star names will be read in and processed from 'info/todo.txt' if nothing else is provided:

```
$ pysyd run
```

4.3.2 Parallel mode

There is a parallel processing option included in the software, which is helpful for running many stars. This can be accessed through the following command:

```
$ pysyd parallel
```

For parallel processing, pySYD will divide and group the list of stars based on the available number of threads. By default, this value is 0 but can be specified via the command line. If it is *not* specified and you are running in parallel mode, pySYD will use multiprocessing package to determine the number of CPUs available on the current operating system and then set the number of threads to this value (minus 1).

If you'd like to take up less memory, you can easily specify the number of threads with the *-nthreads* command:

```
$ pysyd parallel --nthreads 10 --list path_to_star_list.txt
```



4.4 Advanced options

Below are examples of different commands, including their before and after plots to demonstrate the desired effects.

4.4.1 Changing the fractional width of the power excess

via *-ew* & *-exwidth*

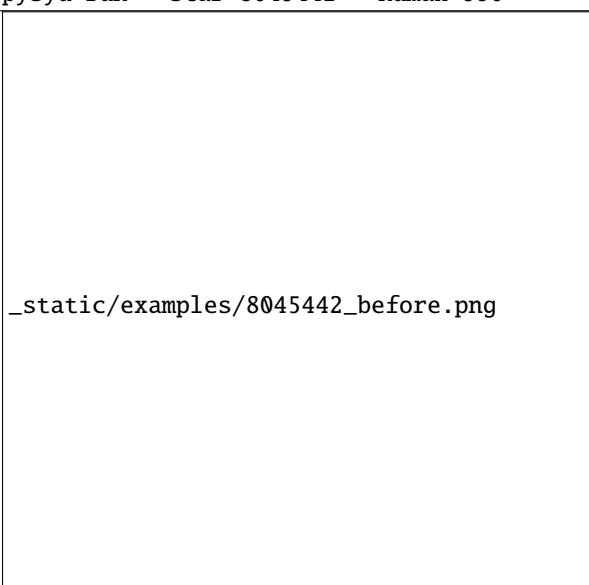

Fractional amount to scale the width of the oscillations envelope by – which is normally calculated w.r.t. solar values.

Before	After
<code>pysyd run --star 9542776 --numax 900</code>	<code>pysyd run --star 9542776 --numax 900 --ew 1.5</code>
	

4.4.2 Mitigating known *Kepler* artefacts

via `-k`, `-kc` & `-kepcorr`

Remove the well-known *Kepler* short-cadence artefact that occurs at/near the long-cadence *nyquist frequency* ($\sim 270\mu\text{Hz}$) by simulating white noise



Before	After
<code>pysyd run --star 8045442 --numax 550</code>	<code>pysyd run --star 8045442 --numax 550 --kc</code>
	

4.4.3 Hard-wiring the lower/upper limits of the power excess

via `-lp` & `-lowerp`

Manually set the lower frequency bound (or limit) of the power excess, which is helpful in the following scenarios:



- 1. the width of the power excess is wildly different from that estimated by the solar scaling relation
- 2. artefact or strange (typically not astrophysical) feature is close to the power excess and cannot be removed otherwise
- 3. power excess is near the *nyquist frequency*

Before	After
<code>pysyd run --star 10731424 --numax 750</code>	<code>pysyd run --star 10731424 --numax 750 --lp 490</code>
	

4.4.4 I'm not sure how I feel about this one

via `-npeaks` & `-peaks`

Change the number of peaks chosen in the autocorrelation function (*ACF*) - this is especially helpful for low S/N cases, where the spectrum is noisy and the ACF has many peaks close the expected spacing (**FIX THIS**)



Before	After
<code>pysyd run --star 9455860</code>	<code>pysyd run --star 9455860 --npeaks 10</code>
 <p>_static/examples/9455860_before.png</p>	 <p>_static/examples/9455860_after.png</p>

4.4.5 Provide estimate for numax and save some time

via `--numax`

Turns out that a majority of the scaling relations used in this software can be written in terms of `numax` and therefore with the single estimate, we can guess the rest of the parameters (and fairly well, at that!)



If the value of ν_{\max} is known, this can be provided to bypass the first module and save some time. There are also other ways to go about doing this, please see our notebook tutorial that goes through these different ways.

Before	After
<code>pysyd run --star 5791521</code>	<code>pysyd run --star 5791521 --numax 670</code>
 _static/examples/5791521_before.png	 _static/examples/5791521_after.png

4.4.6 Setting different frequency limits for the

via `-ux` & `-upperx`

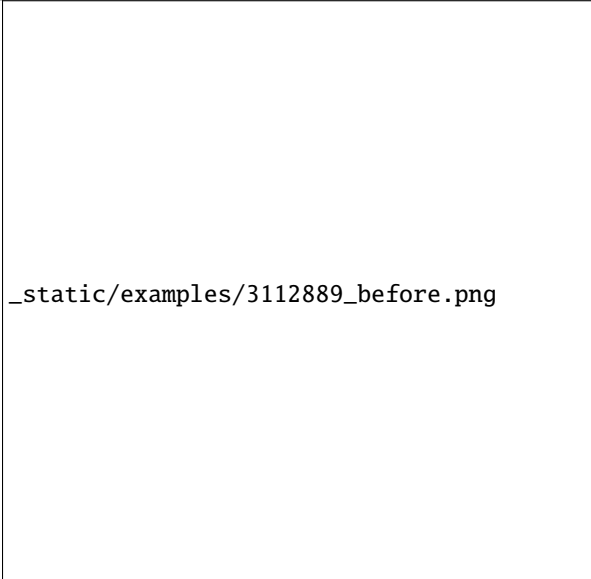

Set the upper frequency limit in the power spectrum when estimating ν_{\max} before the main fitting routine. This is helpful if there are high frequency artefacts that the software latches on to.

Before	After
<code>pysyd run --star 11769801</code>	<code>pysyd run --star 11769801 --ux 3500</code>
 _static/examples/11769801_before.png	 _static/examples/11769801_after.png

4.4.7 Smooth the echelle diagram by using matplotlib's built-in interpolator

via `-i`, `-ie` & `-interpech`

Smooth the echelle diagram output by turning on the (bilinear) interpolation, which is helpful for identifying ridges in low S/N cases

Before	After
<code>pysyd run 3112889 --numax 871.52</code>	<code>pysyd run --star 3112889 --numax 871.52 --ie</code>
 _static/examples/3112889_before.png	 _static/examples/3112889_after.png

4.5 Interactive usage

If there's something you would like to learn more about, you can request a new topic or tutorial by submitting a pull request!

4.5.1 Single star

Jump down to condensed example

This notebook is a basic runthrough for a single star from beginning to end

```
[1]: from pysyd import plots
      from pysyd.target import Target
      from pysyd.utils import Parameters
```

The Parameters object is a container class for default pySYD parameters. Since the software is customizable down to the individual star level - we create one large, default dictionary, check for star-specific information and then copy that to the individual star's dictionary. So for n stars, you will have at least n keys in the main parameter dictionary.

KIC 2309595

Step 1. Load pySYD default parameters

```
[2]: params = Parameters()
      print(params)

<Parameters>
```

Step 2. Add a target (or any number of targets)

```
[3]: name = '2309595'
      params.add_targets(stars=name)

# Both verbose output and displaying of figures are disabled since the software is
# optimized for running many stars, so let's change those!

params.params[name]['show'], params.params[name]['verbose'] = True, True
```

Now that we have the relevant information we want, let's create a pipeline Target object (or star).

Step 3. Create pipeline Target

```
[4]: star = Target(name, params)
      print(star)

<Star 2309595>
```

The individual star's dictionary is copied to the main params class for this object, so now you only have the single dictionary (you can think of it as a pop of the main dictionary, but it makes copies instead of removing). This means we can directly access the defaults without using the star's name as a keyword – so now we can change whatever we want directly!

```
[5]: print(star.params)

{'path': '/Users/ashleychontos/Research/Code/special/pySYD/dev/results/2309595', 'star':
↳ '2309595', 'lower_ex': 100.0, 'smooth_width': 20.0, 'lower_bg': 100.0, 'seed': None,
↳ 'show': True, 'save': True, 'test': False, 'verbose': True, 'overwrite': False,
↳ 'warnings': False, 'stitch': False, 'gap': 20, 'kep_corr': False, 'oversampling_factor
↳ ': None, 'estimate': True, 'numax': None, 'force': False, 'dnu': None, 'binning': 0.
↳ 005, 'bin_mode': 'mean', 'upper_ex': None, 'step': 0.25, 'n_trials': 3, 'ask': False,
↳ 'background': True, 'basis': 'tau_sigma', 'box_filter': 1.0, 'fix_wn': False, 'n_laws':
↳ None, 'ind_width': 20.0, 'upper_bg': None, 'metric': 'bic', 'n_rms': 20, 'globe':
88True, 'ex_width': 1.0, 'lower_ps': None, 'upper_ps': None, 'sm_par'
↳ 5, 'smooth_ps': 2.5, 'fft': True, 'threshold': 1.0, 'hey': False, 'cmap': 'binary',
↳ 'clip_value': 3.0, 'interp_ech': False, 'notching': False, 'lower_ech': None, 'upper_
↳ ech': None, 'npb': 10, 'nox': None, 'noy': '0+0', 'ridges': False, 'smooth_ech': None,
↳ 'iterations': 1, 'deconvolve': False, 'deconvolve_n': 1, 'individuals': 1, 'Users/ashleychontos/

Chapter 4: User's guide
```

(continued from previous page)

Now we will attempt to load in the target data which will return a boolean that says if it's ok to proceed.

```
[6]: print(star.load_data())

-----
Target: 2309595
-----
# LIGHT CURVE: 41949 lines of data read
# Time series cadence: 59 seconds
# POWER SPECTRUM: 106123 lines of data read
# PS oversampled by a factor of 5
# PS resolution: 0.400298 muHz
True
```

Looks like we've been cleared!

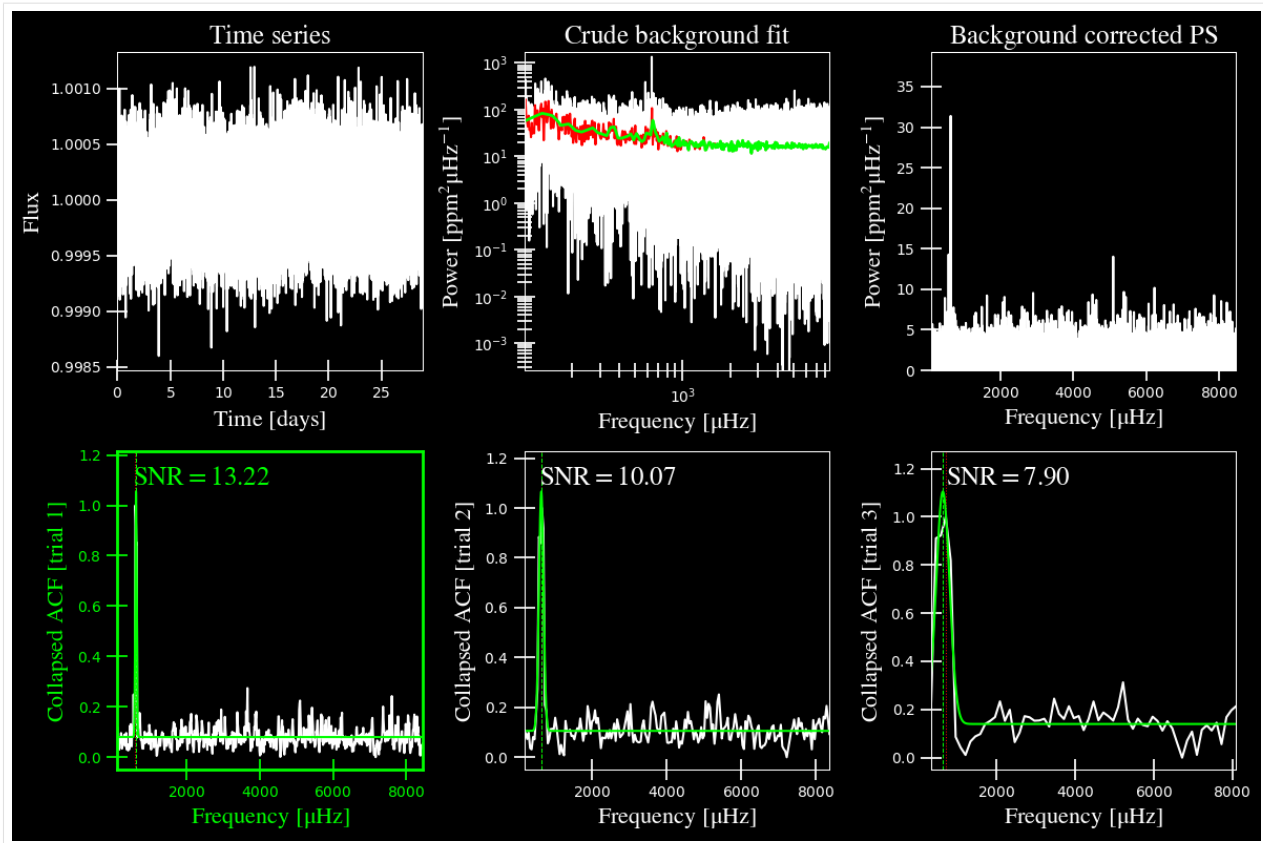
Let's estimate some starting points for the main module.

Step 3. Estimate parameters

```
[7]: star.estimate_parameters()
plots.plot_estimates(star)

-----
PS binned to 219 datapoints

Numax estimates
-----
Estimate 1: 631.20 +/- 9.69
S/N: 13.22
Estimate 2: 635.72 +/- 30.78
S/N: 10.07
Estimate 3: 650.06 +/- 85.26
S/N: 7.90
Selecting model 1
```



All the trials give consistent answers for ν_{max} that I also agree with by eye, so I think we can move on to the full fit.

Step 4. Derive parameters

```
[8]: star.derive_parameters()
plots.plot_parameters(star)
```

```
-----
GLOBAL FIT
-----
```

```
PS binned to 392 data points
```

```
Background model
-----
```

```
Comparing 4 different models:
```

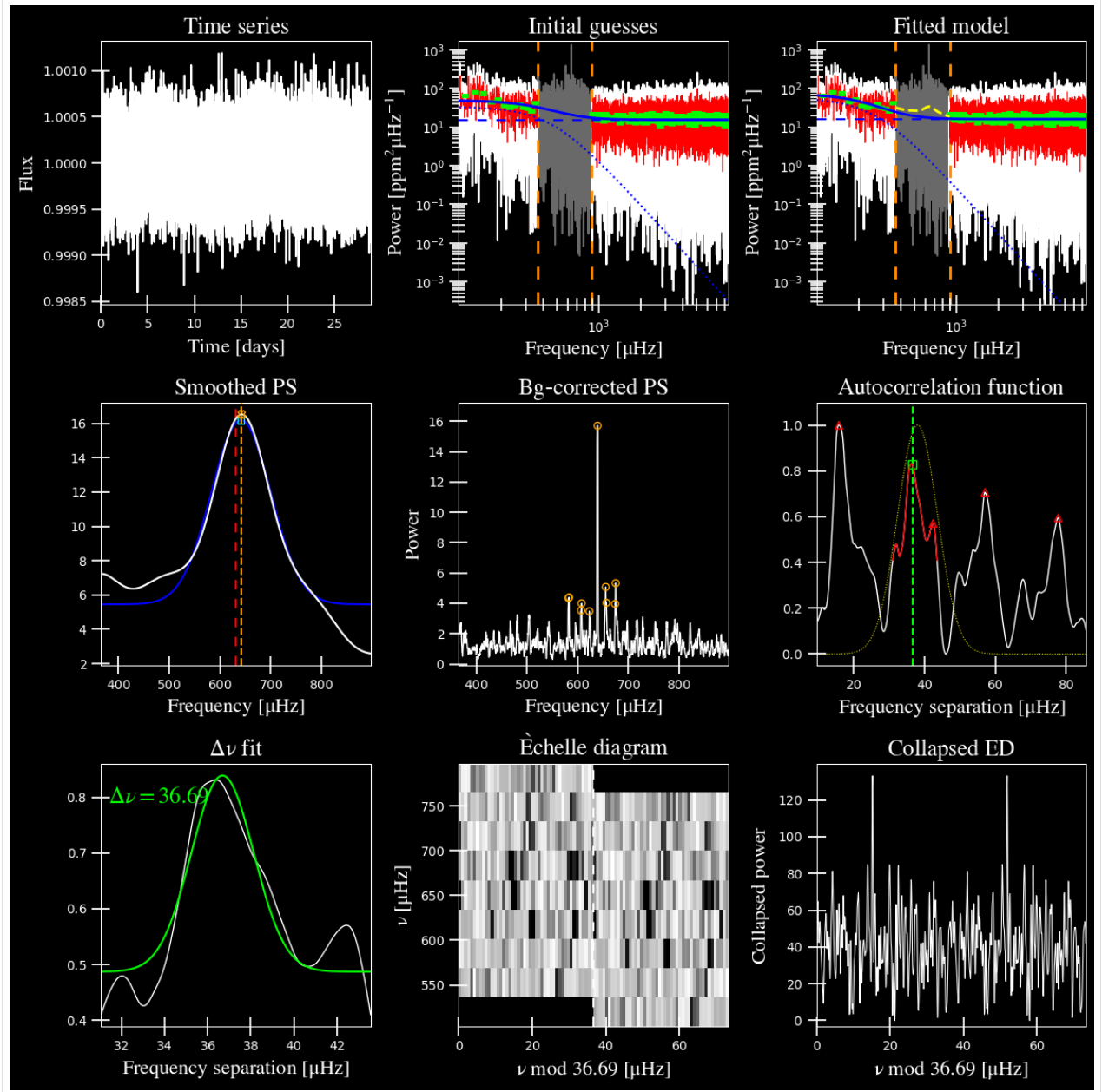
```
Model 0: 0 Harvey-like component(s) + white noise fixed
BIC = 18706.87 | AIC = 47.72
```

```
Model 1: 0 Harvey-like component(s) + white noise term
BIC = 17662.98 | AIC = 45.05
```

```
Model 2: 1 Harvey-like component(s) + white noise fixed
BIC = 4116.35 | AIC = 10.48
```

```
Model 3: 1 Harvey-like component(s) + white noise term
BIC = 3631.91 | AIC = 9.23
```

```
Based on BIC statistic: model 3
```



In the verbose output, the ‘output parameters’ have no uncertainties on the derived values. This is because the number of iterations is 1 by default, for a single iteration. You also might’ve noticed that there are two different estimates for ν_{\max} . **For posterity, the “SYD” pipeline also estimated both of these values but traditionally used $\nu_{\max, \text{smooth}}$ within the literature. *We recommend that you do the same.***

To estimate uncertainties for these parameters, we’ll need to set the number of iterations to something much higher (typically on the order of a hundred or so).

Step 5. Derive uncertainties

```
[9]: star.params['show'], star.params['mc_iter'] = False, 200
star.process_star()
```

```
-----
PS binned to 219 datapoints
```

```
Numax estimates
```

```
-----
Estimate 1: 631.20 +/- 9.69
S/N: 13.22
Estimate 2: 635.72 +/- 30.78
S/N: 10.07
Estimate 3: 650.06 +/- 85.26
S/N: 7.90
Selecting model 1
```

```
-----
GLOBAL FIT
```

```
-----
PS binned to 392 data points
```

```
Background model
```

```
-----
Comparing 4 different models:
Model 0: 0 Harvey-like component(s) + white noise fixed
  BIC = 18706.87 | AIC = 47.72
Model 1: 0 Harvey-like component(s) + white noise term
  BIC = 17662.98 | AIC = 45.05
Model 2: 1 Harvey-like component(s) + white noise fixed
  BIC = 4116.35 | AIC = 10.48
Model 3: 1 Harvey-like component(s) + white noise term
  BIC = 3631.91 | AIC = 9.23
Based on BIC statistic: model 3
```

```
-----
Sampling routine (using seed=2904822):
```

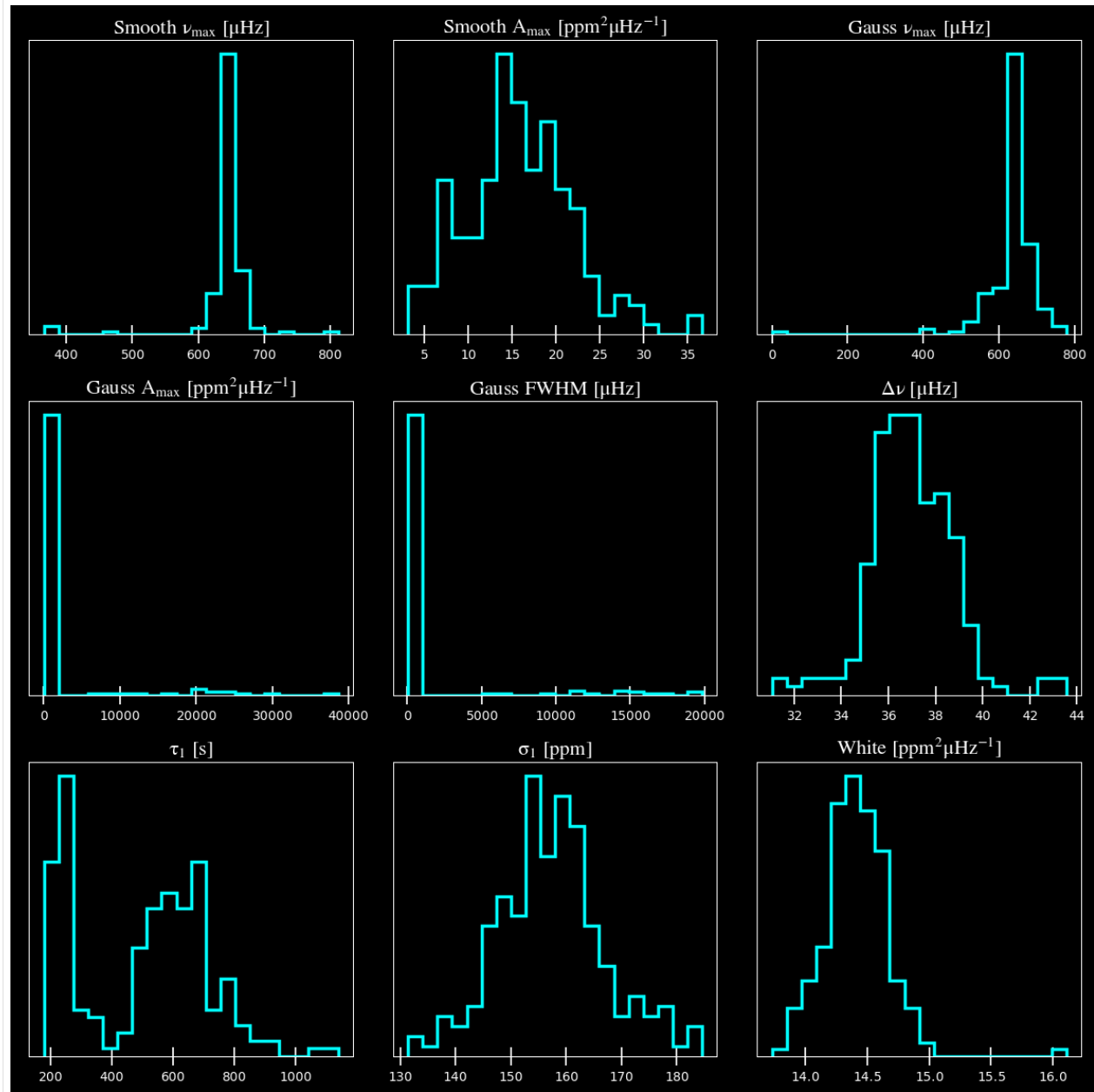
```
100%| 200/200 [00:17<00:00, 11.15it/s]
```

```
-----
Output parameters
```

```
-----
numax_smooth: 642.64 +/- 11.57 muHz
A_smooth: 16.53 +/- 5.34 ppm^2/muHz
numax_gauss: 641.23 +/- 23.20 muHz
A_gauss: 10.73 +/- 4.62 ppm^2/muHz
FWHM: 58.58 +/- 21.93 muHz
dnu: 36.69 +/- 1.58 muHz
tau_1: 605.73 +/- 226.09 s
sigma_1: 155.44 +/- 8.63 ppm
white: 16.11 +/- 0.23 ppm^2/muHz
-----
```


Step 6. Peep results

```
[10]: star.params['show'] = True
      plots.plot_samples(star)
```



As you can see, it still liked the same model (good sanity check) and the derived value for ν_{\max} was robust to this and did not change.

So now we have both parameters and uncertainties!

4.5.2 Condensed version

[Putting it all together with `star.process_star()`]

```
[13]: name='2309595'
      params = Parameters()
      params.add_targets(stars=name)
      params.params[name]['verbose'], params.params[name]['mc_iter'] = True, 200
      star = Target(name, params)
      if star.load_data():
          star.process_star()
```

```
-----
Target: 2309595
-----
```

```
# LIGHT CURVE: 41949 lines of data read
# Time series cadence: 59 seconds
# POWER SPECTRUM: 106123 lines of data read
# PS oversampled by a factor of 5
# PS resolution: 0.400298 muHz
-----
```

```
PS binned to 219 datapoints
```

```
Numax estimates
-----
```

```
Estimate 1: 631.20 +/- 9.69
S/N: 13.22
Estimate 2: 635.72 +/- 30.78
S/N: 10.07
Estimate 3: 650.06 +/- 85.26
S/N: 7.90
Selecting model 1
-----
```

```
GLOBAL FIT
-----
```

```
PS binned to 392 data points
```

```
Background model
-----
```

```
Comparing 4 different models:
```

```
Model 0: 0 Harvey-like component(s) + white noise fixed
      BIC = 18706.87 | AIC = 47.72
```

```
Model 1: 0 Harvey-like component(s) + white noise term
      BIC = 17662.98 | AIC = 45.05
```

```
Model 2: 1 Harvey-like component(s) + white noise fixed
      BIC = 4116.35 | AIC = 10.48
```

```
Model 3: 1 Harvey-like component(s) + white noise term
      BIC = 3631.91 | AIC = 9.23
```

```
Based on BIC statistic: model 3
-----
```

```
Sampling routine (using seed=2904822):
```

```
100%| 200/200 [00:17<00:00, 11.52it/s]
```

Output parameters

```

numax_smooth: 642.64 +/- 11.57 muHz
A_smooth: 16.53 +/- 5.34 ppm^2/muHz
numax_gauss: 641.23 +/- 23.20 muHz
A_gauss: 10.73 +/- 4.62 ppm^2/muHz
FWHM: 58.58 +/- 21.93 muHz
dnu: 36.69 +/- 1.58 muHz
tau_1: 605.73 +/- 226.09 s
sigma_1: 155.44 +/- 8.63 ppm
white: 16.11 +/- 0.23 ppm^2/muHz

```

[]:

4.5.3 Estimating ν_{\max} hacks

For first-time users, we'll assume you do not know what ν_{\max} is for a given star [and that's totally ok]

In these scenarios, we have a convenient built-in method that uses the available information and/or data on a target to estimate a value for numax (ν_{\max}). This is really helpful (and ***strongly encouraged***) for our non-expert users. The primary reason for this is that the main module (i.e. the global fit) derives all parameters *but* models the stellar background and solar-like oscillations separately in two steps.

Basically we use the estimated ν_{\max} to mask out the region in the power spectrum believed to be exhibiting the solar-like oscillations so that it will not influence the stellar background estimates. The reverse is also true, hence why we do this in two steps. Consequently, we correct for the background contribution in the power spectrum before measuring our global properties ν_{\max} and $\Delta\nu$.

ν_{\max} hack summary

There are three main ways to brute force ν_{\max} while still running the `pysyd.target.Target.estimate_parameters` method:

- ****Option 1:**** enter the desired trial number by using the `--ask` flag
- ****Option 2:**** provide your own value for ν_{\max} by using the same flag
- ****Option 3:**** use an upper frequency limit to cut out sharp (likely not astrophysical) features in the power spectrum

```

[1]: from pysyd.utils import Parameters
      from pysyd import plots
      from pysyd.target import Target

```

Load in default settings for KIC 1435467

Since pySYD is optimized for command-line use as well as processing multiple stars, a lot of the options such as showing figures and printing verbose output are disabled. However for demonstration purposes, we'll change two of the defaults.

```
[2]: params = Parameters()
      params.add_targets(stars=['1435467'])
      star = Target('1435467', params)
      star.params['verbose'], star.params['upper_ex'] = True, None
      if star.load_data():
          star.estimate_parameters()
```

```
-----
Target: 1435467
-----
```

```
# LIGHT CURVE: 37919 lines of data read
# Time series cadence: 59 seconds
# POWER SPECTRUM: 99518 lines of data read
# PS oversampled by a factor of 5
# PS resolution: 0.426868 muHz
-----
```

```
PS binned to 219 datapoints
```

```
Numax estimates
-----
```

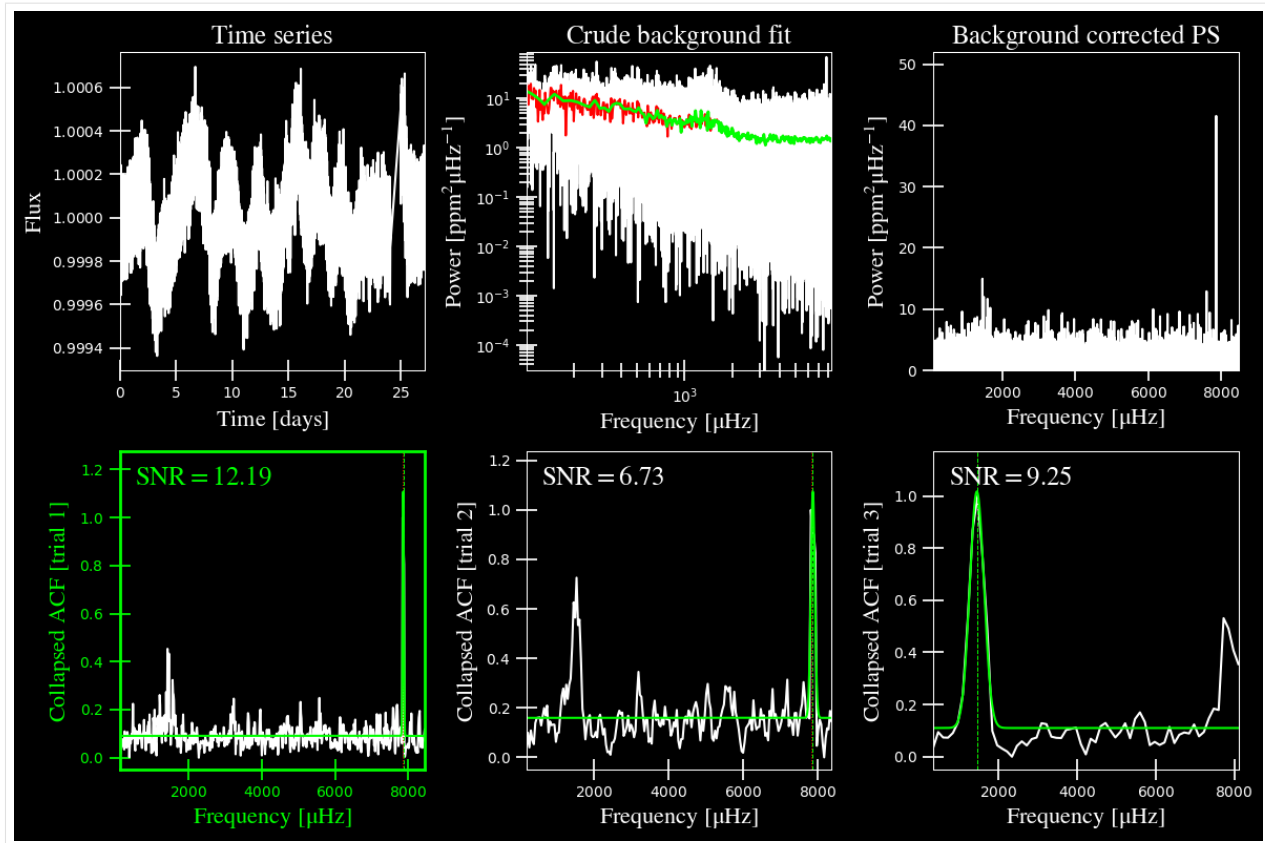
```
Estimate 1: 7859.47 +/- 8.34
S/N: 12.19
Estimate 2: 7876.29 +/- 27.05
S/N: 6.73
Estimate 3: 1457.85 +/- 90.46
S/N: 9.25
Selecting model 1
```

By default, it selects the trial that has the highest signal-to-noise (S/N) detection – which here was the first estimate with S/N ~12. However, the value for numax is really high ($\nu_{\text{max}} \sim 7860\mu\text{Hz}$) and is likely latching on to an artefact or something else that is not astrophysical.

To be sure though, let's take a look at the plots and results to see what's going on.

Plot estimates

```
[3]: star.params['show'] = True
      plots.plot_estimates(star)
```



As suspected, it selected and highlighted the first “trial” due to some high frequency artefact. It was still behaving how it was expected to, but that’s not the type of feature we are looking for. In the upper righthand corner, we can see power excess in the ~1000-2000 μHz frequency region.

We have a few options of how we can go about this. The first is to provide an upper frequency limit that will restrict the power spectrum to below that large spike. Although, if you take a look at the lower righthand corner, the third trial does end up estimating a good value for numax. Therefore we can brute force this trial with the `--ask` option, which is `False` by default.

Option 1: enter the trial number

We will re-load the star and this time change the `--ask` command to `True`. This option will literally *ask* you which trial you prefer. This will also display the plot regardless of your preset options, that way you can make the most informed decision for which to select. As a result, we do *not* recommend using this for many stars.

```
[4]: star.params['ask'] = True
      star.estimate_parameters()
```

```
-----
PS binned to 219 datapoints
```

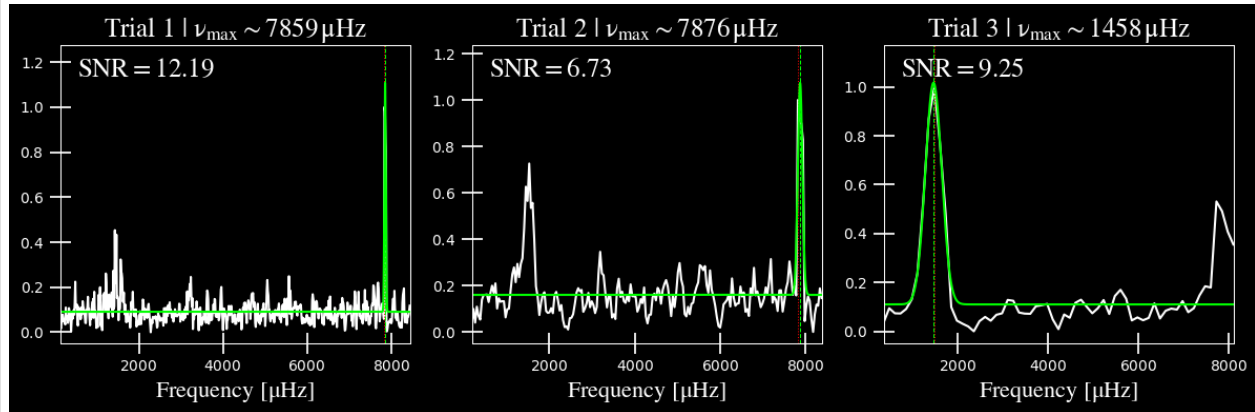
```
Numax estimates
```

```
-----
Estimate 1: 7859.47 +/- 8.34
```

(continues on next page)

(continued from previous page)

S/N: 12.19
 Estimate 2: 7876.29 +/- 27.05
 S/N: 6.73
 Estimate 3: 1457.85 +/- 90.46
 S/N: 9.25



Which estimate would you like to use? 3
 Selecting model 3

Now we can feel more confident about the value that we provide to the main method.

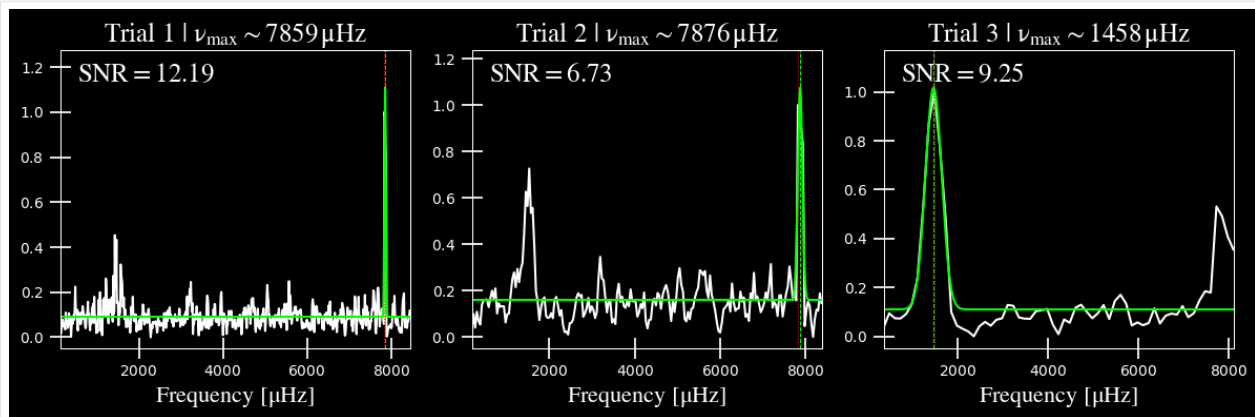
Before we go to our other main alternative (masking the higher frequencies), there's one other thing I'd like to point out. Let's assume that all three of these trials had bogus values but we are pretty confident that the numax was around ~1450. Turns out that we can use the *same flag* but instead, provide our own value.

Option 2: provide your own estimate

```
[5]: star.params['ask'] = True
star.estimate_parameters()

-----
PS binned to 219 datapoints

Numax estimates
-----
Estimate 1: 7859.47 +/- 8.34
S/N: 12.19
Estimate 2: 7876.29 +/- 27.05
S/N: 6.73
Estimate 3: 1457.85 +/- 90.46
S/N: 9.25
```



```
Which estimate would you like to use? 5
ERROR: please select an integer between 1 and 3
      (or 0 to provide your own value for numax)
```

```
Which estimate would you like to use? 0
```

```
What is your value for numax? 1400.00
Using numax of 1400.00 muHz as an initial guess
```

So we intentionally provided an integer value for a trial that does not exist so you could see the alternate option(s) for input data. By entering 0, we are able to provide a float number that we will set numax to.

The plot will still show the three trials but you might've noticed that it didn't highlight any trials since we are using our own value.

Finally, we can catch this problem earlier on by providing an upper frequency limit for the power spectrum that is used in this routine.

Option 3: Provide upper limit

Another option is to set the upper frequency limit (`--upper_ex`) to something below the high-frequency artefacts, say $6000\mu\text{Hz}$.

Note: you'll notice there are a lot of lower/upper bounds but their naming will start to make sense as you use the software more. For example, this first routine that estimates numax used to be called `find_excess()` (since that's technically what it does!) and hence, the "ex" for the flag. The same is true for the **BackGround**-fitting routine (`--lower_bg/--upper_bg`)

```
[6]: star.params['ask'], star.params['upper_ex'] = False, 6000.0
star.estimate_parameters()
```

```
-----
PS binned to 189 datapoints
```

```
Numax estimates
```

(continues on next page)

(continued from previous page)

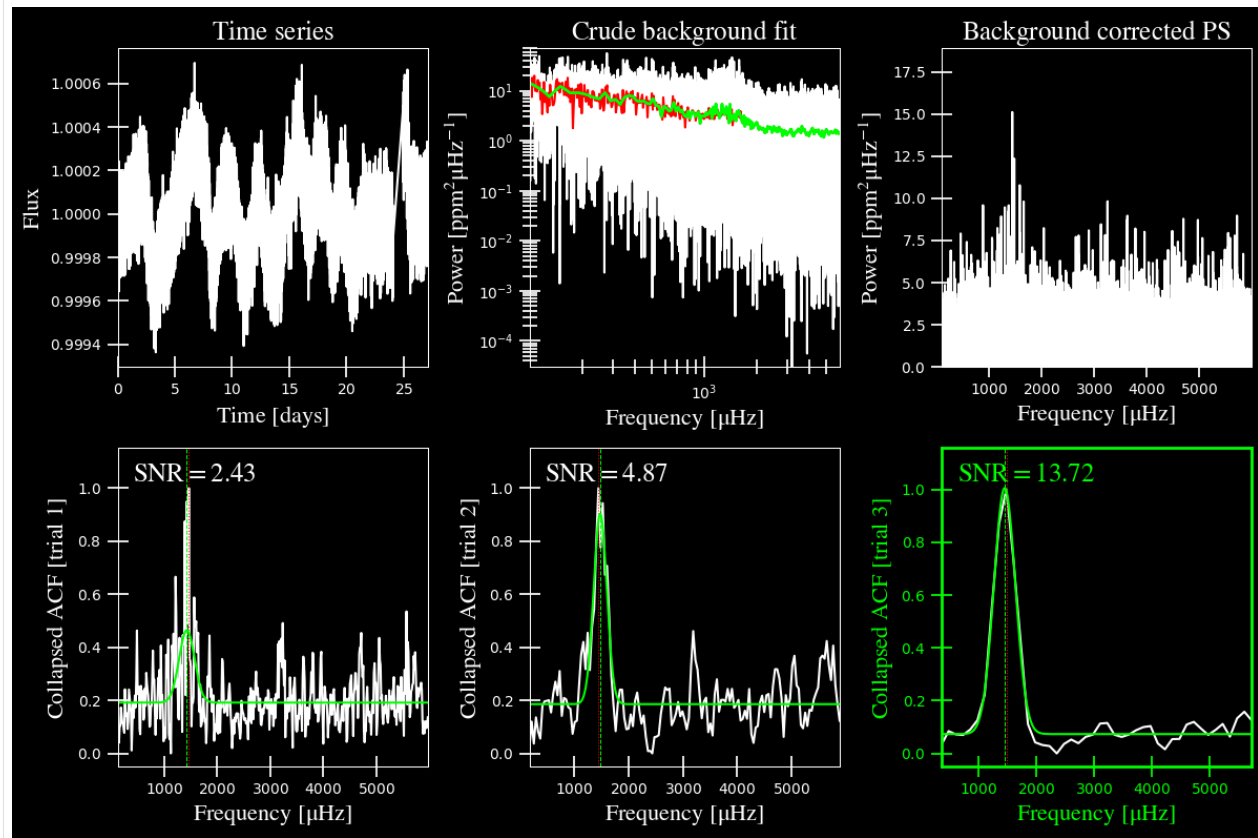
```

-----
Estimate 1: 1430.02 +/- 72.61
S/N: 2.43
Estimate 2: 1479.46 +/- 60.64
S/N: 4.87
Estimate 3: 1447.42 +/- 93.31
S/N: 13.72
Selecting model 3

```

Remember that the figure was only displaying because we enacted the `--ask` option, so let's double check the figure to be sure.

```
[7]: plots.plot_estimates(star)
```



If you look in the upper righthand corner of the figure, you'll notice that the power spectrum only goes up to $6000 \mu\text{Hz}$ this time. That enabled three nice, equally-usuable trial runs *but* you can see that the values for ν_{max} are all consistent with one another to within $\pm 1\sigma$. Therefore any of these guesses would suffice!

4.6 pySYD option glossary

Below is a complete list of pySYD parameters in alphabetical order.

-a, --ask

the option to select which trial (or estimate) of numax to use from the first module

- `dest = args.ask`
- `type = bool`
- `default = False`
- `action = store_true`
- **see also:** *-trials, -ntrials*

--all, --showall creates an additional figure that shows all the iterated background models, which will highlight the selected model

- `dest = args.showall`
- `type = bool`
- `default = False`
- `action = store_true`

-b, --bg, --background controls the background-fitting procedure – BUT this should never be touched since a majority of the work done in the software happens here and it should not need to be turned off

- `dest = args.background`
- `type = bool`
- `default = True`
- `action = store_false`

--basis which basis to use for the background fitting (i.e. 'a_b', 'pgran_tau', 'tau_sigma'), **NOT OPERATIONAL YET**

- `dest = args.basis`
- `type = str`
- `default = 'tau_sigma'`

--bf, --box, --boxfilter box filter width for plotting the power spectrum **TODO:** make sure this does not affect any actual measurements and this is just an aesthetic

- `dest = args.box_filter`
- `type = float`
- `default = 1.0`
- `unit = μHz`

--bin, --binning interval for the binning of spectrum in $\log(\mu\text{Hz})$ *this bins equally in logspace*

- `dest = args.binning`
- `type = float`
- `default = 0.005`
- `unit = $\log(\mu\text{Hz})$`

--bm, --mode, --bmode which mode to choose when binning. Choices are ~["mean", "median", "gaussian"]

- `dest = args.mode`
- `type = str`
- `default = "mean"`

--ce, --cm, --color change the colormap used in the echelle diagram, which is 'binary' by default

- `dest = args.cmap`
- `type = str`
- `default = 'binary'`

--cv, --value the clip value to use for the output echelle diagram if and only if `args.clip_ech` is True. If none is provided, it will use a value that is 3x the median value of the folded power spectrum

- `dest = args.clip_value`
- `type = float`
- `default = 3.0`
- `unit = fractional psd`

--cli

this should never be touched - for internal workings on how to retrieve and save parameters

- `dest = args.cli`
- `type = bool`
- `default = True`
- `action = store_true`

-d, --show, --display show output figures, which is not recommended if running many stars

- `dest = args.show`
- `type = bool`
- `default = False`
- `action = store_true`

--dnu option to provide the spacing to fold the power spectrum and “whiten” effects due to mixed modes ([*pysyd.target.Target.whiten_mixed*](#)), which also requires a lower and upper folded frequency (i.e. \leq dnu) via *-le* and *-ue*

- `dest = args.dnu`
- `type = float`
- `nargs = '*'`
- `default = None`
- ****REQUIRES:**** *-le*—*lowere* and *-ue*—*uppere*

-e, --est, --estimate turn off the first module that searches and identifies power excess due to solar-like oscillations, which will automatically happen if *numax* is provided

- `dest = args.estimate`
- `type = bool`

- default = `True`
- action = `store_false`

--ew, --ewidth the fractional value of the width to use surrounding the power excess, which is computed using a solar scaling relation (and then centered on the estimated ν_{\max})

- dest = `args.width`
- type = `float`
- default = `1.0`
- unit = fractional μHz
- see also: *-lp, -lowerp, -up, -upperp*

-f, --fft

use the `numpy.correlate` module instead of *FFTs* to compute the ACF

- dest = `args.fft`
- type = `bool`
- default = `True`
- action = `store_false`

--file, --list, --todo the path to the text file that contains the list of stars to process, which is convenient for running many stars

- dest = `args.file`
- type = `str`
- default = `TODODIR`
- see also: *-star, -stars*

-g, --globe, --global do not estimate the global asteroseismic parameter `numax` and `dnu`. This is helpful for the application to cool dwarfs, where detecting solar-like oscillations is quite difficult but you'd still like to characterize the granulation components.

- dest = `args.globe`
- type = `bool`
- default = `True`
- action = `store_false`

--gap, --gaps

what constitutes a time series gap (i.e. how many cadences)

- dest = `args.gap`
- type = `int`
- default = `20`
- see also: *-x, -stitch, -stitching*

-i, --ie, --interpech turn on the bilinear interpolation of the plotted echelle diagram

- dest = `args.interp_ech`
- type = `bool`

- default = `False`
- action = `store_true`
- see also: `-se`, `-smoothech`

--in, --input, --inpdire

path to the input data

- dest = `args.inpdire`
- type = `str`
- default = `INPDIRE`

--infdire

path to relevant pySYD information (defined in init file)

- dest = `args.infdire`
- type = `str`
- default = `INFDIRE`
- see also: `-file`, `-info`, `-information`, `-list`, `-todo`

--info, --information path to the csv containing all the stellar information (although *not* required)

- dest = `args.info`
- type = `str`
- default = `star_info.csv`

--iw, --indwidth width of binning for the power spectrum used in the first module **TODO: CHECK THIS**

- dest = `args.ind_width`
- type = `float`
- default = `20.0`
- unit = μHz

-k, --kc, --kepcorr turn on the *Kepler* short-cadence artefact correction module. if you don't know what a *Kepler* short-cadence artefact is, chances are you shouldn't mess around with this option yet

- dest = `args.kepcorr`
- type = `bool`
- default = `False`
- action = `store_true`

--laws, --nlaws force the number of red-noise component(s). **fun fact:** the older IDL version of SYD fixed this number to 2 for the *Kepler* legacy sample – now we have made it customizable all the way down to an individual star!

- dest = `args.n_laws`
- type = `int`
- default = `None`
- see also: `-w`, `-wn`, `-fixwn`

--lb, --lowerb the lower frequency limit of the power spectrum to use in the background-fitting routine. **Please note:** unless ν_{\max} is known, it is highly recommended that you do *not* fix this beforehand

- `dest = args.lower_bg`
- `type = float`
- `nargs = '*'`
- `default = 1.0`
- `unit = μ Hz`
- **see also:** *-ub, -upperb*

--le, --lowere the lower frequency limit of the folded power spectrum to “whiten” mixed modes before estimating the final value for `dnu`

- `dest = args.lower_ech`
- `type = float`
- `nargs = '*'`
- `default = None`
- `unit = μ Hz`
- **REQUIRES:** *-uel-upperp* and *-dnu*

--lp, --lowerp to change the lower frequency limit of the zoomed in power spectrum (i.e. the region with the supposed power excess due to oscillations). Similar to `-ew` but instead of a fractional value w.r.t. the scaled solar value, you can provide hard boundaries in this case **TODO** check if it requires an upper bound – pretty sure it doesn’t but should check

- `dest = args.lower_ps`
- `type = float`
- `nargs = '*'`
- `default = None`
- `unit = μ Hz`
- **see also:** *-up, -upperp*

--lx, --lowerx the lower limit of the power spectrum to use in the first module (to estimate `numax`)

- `dest = args.lower_ex`
- `type = float`
- `default = 1.0`
- `unit = μ Hz`
- **see also:** *-ux, -upperx*

-m, --samples option to save the samples from the Monte-Carlo sampling (i.e. parameter posteriors) in case you’d like to reproduce your own plots, etc.

- `dest = args.samples`
- `type = bool`
- `default = False`
- `action = store_true`

--mc, --iter, --mciter number of Monte-Carlo-like iterations. This is 1 by default, since you should always check the data and output figures before running the sampling algorithm. But for purposes of generating uncertainties, `n=200` is typically sufficient.

- `dest = args.mc_iter`
- `type = int`
- `default = 1`

--metric which model metric to use for the best-fit background model, current choices are `~['bic', 'aic']` but **still being developed and tested**

- `dest = args.metric`
- `type = str`
- `default = 'bic'`

-n, --notch use notching technique to reduce effects from mixes modes (pretty sure this is not full functional yet, creates weird effects for higher SNR cases)

- `dest = args.notching`
- `type = bool`
- `default = False`
- `action = store_true`

--notebook similar to `-cli`, this should not need to be touched and is primarily for internal workings and how to retrieve parameters

- `dest = args.notebook`
- `type = bool`
- `default = False`
- `action = store_true`

--nox, --nacross specifies the number of bins (i.e. the resolution) to use for the x-axis of the echelle diagram – fixing this number if complicated because it depends on both the resolution of the power spectrum as well as the characteristic frequency separation. This is another example where, if you don't know what this means, you probably should not change it.

- `dest = args.nox`
- `type = int`
- `default = None`
- **see also:** `-noy`, `-ndown`, `-norders`, `-npb`

--noy, --ndown, --norders specifies the number of bins (or radial orders) to use on the y-axis of the echelle diagram **NEW:** option to shift the entire figure by `n` orders - the first part of the string is the number of orders to plot and the `+/- n` is the number orders to shift the ED by

- `dest = args.noy`
- `type = str`
- `default = 0+0`
- **see also:** `-nox`, `-nacross`, `-npb`

--npb option for echelle diagram to use information from the spacing and frequency resolution to calculate a better grid resolution (`npb == number per bin`)

- `dest = args.npb`
 - `type = int`
 - `default = 10`
 - **see also:** `-nox`, `-nacross`, `-noy`, `-ndown`, `-norders`
- nt, --nthread, --nthreads** the number of processes to run in parallel. If nothing is provided when you run in `pysyd.parallel` mode, the software will use the `multiprocessing` package to determine the number of CPUs on the operating system and then adjust accordingly. **In short:** this probably does not need to be changed
- `dest = args.n_threads`
 - `type = int`
 - `default = 0`
- numax** brute force method to bypass the first module and provide an initial starting value for ν_{\max} Asserts `len(args.numax) == len(args.targets)` * `dest = args.numax` * `type = float` * `nargs = '*'` * `default = None` * `unit = μ Hz`
- o, --overwrite** newer option to overwrite existing files with the same name/path since it will now add extensions with numbers to avoid overwriting these files
- `dest = args.overwrite`
 - `type = bool`
 - `default = False`
 - `action = store_true`
- of, --over, --oversample** the oversampling factor of the provided power spectrum. Default is 0, which means it is calculated from the time series data. **Note:** this needs to be provided if there is no time series data!
- `dest = args.oversampling_factor`
 - `type = int`
 - `default = None`
- out, --output, --outdir** path to save results to
- `dest = args.outdir`
 - `type = str`
 - `default = 'OUTDIR'`
- peak, --peaks, --npeaks** the number of peaks to identify in the autocorrelation function
- `dest = args.n_peaks`
 - `type = int`
 - `default = 5`
- rms, --nrms** the number of points used to estimate the amplitudes of individual background (red-noise) components *Note: this should only rarely need to be touched*
- `dest = args.n_rms`
 - `type = int`
 - `default = 20`
- s, --save** turn off the automatic saving of output figures and files

- `dest = args.save`
- `type = bool`
- `default = True`
- `action = store_false`

--se, --smoothech option to smooth the echelle diagram output using a box filter of this width

- `dest = args.smooth_ech`
- `type = float`
- `default = None`
- `unit = μHz`
- **see also:** `-e, -ie, -interpech`

--sm, --smpar the value of the smoothing parameter to estimate the smoothed numax (that is really confusing) **note:** typical values range from 1-4 but this is fixed based on years of trial & error

- `dest = args.sm_par`
- `type = float`
- `default = None`
- `unit = fractional μHz`

--sp, --smoothps the box filter width used for smoothing of the power spectrum. The default is 2.5 but will switch to 0.5 for more evolved stars (if $\nu_{\text{max}} < 500 \mu\text{Hz}$)

- `dest = args.smooth_ps`
- `type = float`
- `default = 2.5`
- `unit = μHz`

--star, --stars list of stars to process. Default is `None`, which will read in the star list from `args.file` instead

- `dest = args.star`
- `type = str`
- `nargs = '*'`
- `default = None`
- **see also:** `-file, -list, -todo`

--step, --steps the step width for the collapsed autocorrelation function w.r.t. the fraction of the boxsize. **Please note:** this should not be adjusted

- `dest = args.step`
- `type = float`
- `default = 0.25`
- `unit = fractional μHz`

--sw, --smoothwidth the width of the box filter that is used to smooth the power spectrum

- `dest = args.smooth_width`
- `type = float`

- default = 20.0
- unit = μHz
- **see also:** *-sp*, *-smoothps*

Warning: All parameters are optimized for most star types but some may need adjusting. An example is the smoothing width (*--sw*), which is 20 μHz by default, but may need to be adjusted based on the nyquist frequency and frequency resolution of the input power spectrum.

--thresh, --threshold the fractional value of the autocorrelation function's full width at half maximum (which is important in this scenario because it is used to determine $\Delta\nu$)

- dest = args.threshold
- type = float
- default = 1.0
- unit = fractional μHz

--trials, --ntrials the number of trials used to estimate numax in the first module – can be bypassed if *-numax* is provided.

- dest = args.n_trials
- type = int
- default = 3

--ub, --upperb the upper limit of the power spectrum used in the background-fitting module **Please note:** unless ν_{max} is known, it is highly recommended that you do *not* fix this beforehand

- dest = args.upper_bg
- type = float
- nargs = '*'
- default = 6000.0
- unit = μHz
- **see also:** *-lb*, *-lowerb*

--ue, --uppere the upper frequency limit of the folded power spectrum used to “whiten” mixed modes before determining the correct $\Delta\nu$

- dest = args.upper_ech
- type = float
- nargs = '*'
- default = None
- unit = μHz
- **REQUIRES:** *-lel-lowerb* and *-dnu*

--up, --upperp the upper frequency limit used for the zoomed in power spectrum. In other words, this is an option to use a different upper bound than the one determined automatically

- dest = args.upper_ps
- type = float

- nargs = '*'
- default = `None`
- unit = μHz
- see also: `-lp`, `-lowerp`

--ux, --upperx the upper frequency limit of the power spectrum to use in the first module

- dest = args.upper_ex
- type = `float`
- default = `6000.0`
- unit = μHz
- see also: `-lx`, `-lowerx`

-v, --verbose turn on the verbose output (also not recommended when running many stars, and definitely *not* when in parallel mode) **Check** this but I think it will be disabled automatically if the parallel mode is `True`

- dest = args.verbose
- type = `bool`
- default = `False`
- action = store_true

-w, --wn, --fixwn

fix the white noise level in the background fitting TODO: this still needs to be tested

- dest = args.fix
- type = `bool`
- default = `False`
- action = store_true
- see also: `-laws`, `-nlaws`

-x, --stitch, --stitching

correct for large gaps in time series data by ‘stitching’ the light curve

- dest = args.stitch
- type = `bool`
- default = `False`
- action = store_true
- see also: `-gap`, `-gaps`

-y, --hey plugin for Daniel Hey’s interactive echelle package **but is not currently implemented TODO**

- dest = args.hey
- type = `bool`
- default = `False`
- action = store_true

4.7 et al.

Please share how we can make your experience even better!

We love hearing new ideas – if you feel like there’s something missing or literally anything you’d like to learn more about, you can request a new topic/tutorial by submitting a pull request.

GLOSSARY OF DOCUMENTATION TERMS

AIC

Akaike Information Criterion a common metric for model selection that prevents overfitting of data by penalizing models with higher numbers of parameters (k)

- **definition:**

$$\text{AIC} = 2k - 2\ln(\hat{L})$$

asteroseismology the study of oscillations in stars

ACF

autocorrelation function in this context it is a small range of frequencies in the power spectrum surrounding the solar-like oscillations, then the power array is correlated (or convolved) with a copy of the power array. This is a helpful diagnostic tool for quantitatively confirming the p-mode oscillations, since they have regular spacings in the frequency domain and therefore should create strong peaks at integer and half integer harmonics of $\Delta\nu$

background this basically means any other noise structures present in the power spectrum that are *not* due to solar-like oscillations. This is traditionally parametrized as:

$$B(\nu) = W + \sum_{i=0}^n \frac{4\sigma_i^2\tau_i}{1 + (2\pi\nu\tau_i)^2 + (2\pi\nu\tau_i)^4}$$

BCPS

background-corrected power spectrum the power spectrum after removing the best-fit stellar background model. In general, this step removes any slopes in power spectra due to correlated red-noise properties

Note: A *background-corrected power spectrum (BCPS)* is an umbrella term that has the same meanings as a *background-divided power spectrum (BDPS)* and a *background-subtracted power spectrum (BSPS)*. Thus it is best ***to avoid*** this phrase if at all possible since it does not specify how the power spectrum has been modified.

BDPS

background-divided power spectrum the power spectrum divided by the best-fit stellar background model. Using this method for data analysis is great for first detecting and identifying any solar-like oscillations since it will make the power excess due to stellar oscillations appear higher signal-to-noise

BSPS

background-subtracted power spectrum the best-fit stellar background model is subtracted from the power spectrum. While this method appears to give a lower signal-to-noise detection, the amplitudes measured through this analysis are physically-motivated and correct (i.e. can be compared with other literature values)

BIC

Bayesian Information Criterion a common metric for model selection

cadence

the median absolute difference between consecutive time series observations

- **variable:** Δt
- **units:** s
- **definition:**

critically-sampled power spectrum when the frequency resolution of the power spectrum is exactly equal to the inverse of the total duration of the time series data it was calculated from

ED

echelle diagram a diagnostic tool to confirm that *dnu* is correct. This is done by folding the power spectrum (*FPS*) using *dnu* (you can think of it as the PS modulo the spacing) – which if the *large frequency separation* is correct – the different oscillation modes will form straight ridges. **Fun fact:** the word ‘echelle’ is actually French for ladder

FFT

fast fourier transform a method used in signal analysis to determine the most dominant periodicities present in a *light curve*

FPS

folded power spectrum the power spectrum folded (or stacked) at some frequency, which is typically done with the *large frequency separation* to construct an *echelle diagram*

numax

frequency of maximum power the frequency corresponding to maximum power, which is roughly the center of the Gaussian-like envelope of oscillations

- **variable:** ν_{\max}
- **units:** μHz

scales with evolutionary state, $\log g$, acoustic cutoff

frequency resolution the resolution of a *power spectrum* is set by the total length of the time series (ΔT^{-1})

FWHM

full-width half maximum for a Gaussian-like distribution, the full-width at half maximum (or full-width half max) is approximately equal to $\pm 1\sigma$

global properties in asteroseismology, the global asteroseismic parameters or properties refer to ν_{\max} (*numax*) and $\Delta\nu$ (*dnu*)

granulation the smallest (i.e. quickest) scale of convective processes

Harvey-like component

Harvey-like model named after the person who first person who discovered the relation – and found it did a good job characterizing granulation amplitudes and time scales in the Sun

Kepler artefact *Kepler* short-cadence artefact in the power spectrum from a short-cadence light curve occurring at the nyquist frequency for long-cadence (i.e. $\sim 270\mu\text{Hz}$)

Kepler legacy sample a sample of well-studied *Kepler* stars exhibiting solar-like oscillations (cite Lund+2014)

dnu

large frequency separation the so-called large frequency separation is the inverse of twice the sound travel time between the center of the star and the surface. Even more generally, this is the comb pattern or regular spacing observed for solar-like oscillations. It is exactly equal to the frequency spacing between modes with the same spherical degree and consecutive radial order's.

- **variable:** $\Delta\nu$
- **units:** μHz
- **definition:**

$$\Delta\nu = \left[2 \int_0^R \frac{dr}{c} \right]^{-1} \propto \bar{\rho}$$

light curve the measure of an object's brightness with time

mesogranulation the intermediate scale of convection

mixed modes in special circumstances, pressure (or p-) modes couple with gravity (or g-) modes and make the spectrum of a solar-like oscillator much more difficult to interpret – in particular, for measuring the *large frequency separation*

notching a process used to mitigate features in the frequency domain (e.g., mixed modes) by setting specific values to the minimum power in the array

nyquist frequency the highest frequency that can be sampled, which is set by the *cadence* of observations (Δt)

- **variable:** ν_{nyq}
- **units:** μHz
- **definition:**

$$\nu_{\text{nyq}} = \frac{1}{2\Delta t}$$

Note: *Kepler* example

Kepler short-cadence data has a cadence, $\Delta t \sim 60\text{s}$. Therefore, the nyquist frequency for short-cadence *Kepler* data is:

$$\nu_{\text{nyq}} = \frac{1}{2 \cdot 60\text{s}} \times \frac{10^6 \mu\text{Hz}}{1 \text{ Hz}} \approx 8333 \mu\text{Hz}$$

oversampled power spectrum if the resolution of the power spectrum is greater than $1/T$

p-mode oscillations

solar-like oscillations implied in the name, these oscillations are driven by the same mechanism as that observed in the Sun, which is due to turbulent, near-surface convection. They are also sometimes referred to as **p-mode oscillations**, after the pressure-driven (or acoustic sound) waves that are resonating in the stellar cavity.

power excess the region in the power spectrum believed to show solar-like oscillations is typically characterized by a Gaussian-like envelope of oscillations, $G(\nu)$

$$G(\nu) = A_{\text{osc}} \exp \left[- \frac{(\nu - \nu_{\text{max}})^2}{2\sigma_{\text{osc}}^2} \right]$$

PSD

power spectral density when the power of a frequency spectrum is normalized s.t. it satisfies Parseval's theorem (which is just a fancy way of saying that the fourier transform is unitary)

- **unit:** $\text{ppm}^2 \mu\text{Hz}^{-1}$
-

PS

power spectrum any object that varies in time also has a corresponding frequency (or power) spectrum, which is computed by taking the *fast fourier transform* of the *light curve*. A general model to describe characteristics of a power spectrum is generalized by the equation below, where W is a constant (frequency-independent) noise term, primarily due to photon noise. B and G correspond to the background and Gaussian-like power excess components, respectively. Finally, R corresponds to the response function, or the attenuation of signals due to time-averaged observations.

$$P(\nu) = W + R(\nu)[B(\nu) + G(\nu)]$$

scaling relations empirical relations for fundamental stellar properties that are scaled with respect to the Sun, since it is the star we know best. In asteroseismology, the most common relations combine *global asteroseismic parameters* with spectroscopic effective temperatures to derive stellar masses and radii:

$$\frac{R_{\star}}{R_{\odot}} = \left(\frac{\nu_{\max}}{\nu_{\max,\odot}} \right) \left(\frac{\Delta\nu}{\Delta\nu_{\odot}} \right)^{-2} \left(\frac{T_{\text{eff}}}{T_{\text{eff},\odot}} \right)^{1/2}$$
$$\frac{M_{\star}}{M_{\odot}} = \left(\frac{\nu_{\max}}{\nu_{\max,\odot}} \right)^3 \left(\frac{\Delta\nu}{\Delta\nu_{\odot}} \right)^{-4} \left(\frac{T_{\text{eff}}}{T_{\text{eff},\odot}} \right)^{3/2}$$

whiten

whitening a process to remove undesired artefacts or effects present in a frequency spectrum by taking that frequency region and replacing it with simulated white noise. This is typically done for subinants with *mixed modes* in order to better estimate *dnu*. This can also help mitigate the short-cadence *Kepler artefact*.

VISION OF THE PYSYD PROJECT

The NASA space telescopes *Kepler*, K2 and TESS have recently provided very large databases of high-precision light curves of stars. By detecting brightness variations due to stellar oscillations, these light curves allow the application of asteroseismology to large numbers of stars, which requires automated software tools to efficiently extract observables.

Several tools have been developed for asteroseismic analyses, but many of them are closed-source and therefore inaccessible to the general astronomy community. Some open-source tools exist, but they are either optimized for smaller samples of stars or have not yet been extensively tested against closed-source tools.

Note: We've attempted to collect these tools in a *single place* for easy comparisons. Please let us know if we've somehow missed yours – we would be happy to add it!

6.1 Goals

The initial vision of this project was intended to be a direct translation of the IDL-based SYD pipeline

ATTRIBUTION

7.1 Citations

7.1.1 Citing pySYD

If you make use of pySYD in your work, please cite our [JOSS paper](#):

```
@article{2021arXiv210800582C,  
  author = {{Chontos}, Ashley and {Huber}, Daniel and {Sayeed}, Maryum and {Yamsiri}  
↪, Pavadol},  
  title = "{$\texttt{pySYD}$: Automated measurements of global asteroseismic_  
↪parameters}",  
  journal = {arXiv e-prints},  
  keywords = {Astrophysics - Solar and Stellar Astrophysics, Astrophysics -  
↪Instrumentation and Methods for Astrophysics},  
  year = 2021,  
  month = aug,  
  eid = {arXiv:2108.00582},  
  pages = {arXiv:2108.00582},  
archivePrefix = {arXiv},  
  eprint = {2108.00582},  
  primaryClass = {astro-ph.SR},  
  adsurl = {https://ui.adsabs.harvard.edu/abs/2021arXiv210800582C},  
  adsnote = {Provided by the SAO/NASA Astrophysics Data System}  
}
```

If applicable, please also use our [ASCL listing](#) as a software citation:

```
@misc{2021ascl.soft11017C,  
  author = {{Chontos}, Ashley and {Huber}, Daniel and {Sayeed}, Maryum and  
↪{Yamsiri}, Pavadol},  
  title = "{pySYD: Measuring global asteroseismic parameters}",  
  keywords = {Software},  
  year = 2021,  
  month = nov,  
  eid = {ascl:2111.017},  
  pages = {ascl:2111.017},  
archivePrefix = {ascl},  
  eprint = {2111.017},  
  adsurl = {https://ui.adsabs.harvard.edu/abs/2021ascl.soft11017C},
```

(continues on next page)

(continued from previous page)

```
adsnote = {Provided by the SAO/NASA Astrophysics Data System}
}
```

[Click here](#) to see projects that have already used pySYD!

7.1.2 Citing SYD

pySYD is a python-based implementation of the IDL-based SYD pipeline, which was extensively used to measure asteroseismic parameters for *Kepler* stars. Since pySYD adapted the well-tested framework from SYD, we ask that you please cite [the original paper](#) that discusses the asteroseismic analysis and methodology.

Important: This work was only possible thanks to many powerful Python libraries – we *strongly* encourage you to also consider citing its dependencies.

7.2 Projects w/ pySYD

If you, someone you know, or a project you know about has made use of pySYD, please consider visiting and contributing to the public [Projects w/ pySYD](#) thread in our GitHub repo.

Feel free to add anything and everything from early results, figure(s), student projects to published manuscripts, posters, or any other pySYD-related shoutouts. **We would love to see what you are using pySYD for firsthand!**

CONTRIBUTING

Jump to our community guidelines

8.1 The pySYD team

Our community continues to grow! See below to find out how you can help

8.1.1 Contributors

Main author: Ashley Chontos ([email](#) | [website](#))

All contributors (listed alphabetically):

- Ashley Chontos ([@ashleychontos](#))
- Sam Grayson ([@charmoniumQ](#))
- Daniel Huber ([@danxhuber](#))
- Maryum Sayeed ([@MaryumSayeed](#))
- Pavadol Yamsiri ([@pavyamsiri](#))

Important: pySYD was initially the Python translation of the IDL-based asteroseismology pipeline SYD, which was written by my PhD advisor, Dan Huber, during his PhD in Sydney (hence the name). Therefore none of this would have been possible without his i) years of hard work during his PhD as well as ii) years of patience during my PhD

~A very special shoutout to Dan~

8.1.2 Collaborators

We have many amazing collaborators that have helped with the development of the software, especially with the improvements that have been implemented – which have ultimately made pySYD more user-friendly. Many thanks to our collaborators!

pySYD collaborators:

- Tim Bedding
- Marc Hon
- Dennis Stello

8.2 Community guidelines

For most (if not all) questions/concerns, checking our [discussions](#) forum is a great place to start in case things have already been brought up and/or addressed.

If you would like to contribute, here are the guidelines we ask you to follow:

- *Question or problem*
 - *Issues & bugs*
 - *New features*
 - Contributing code
 - *Style guide*
 - *Testing*
-

8.2.1 Question or problem

Do you have a general question that is not directly related to software functionality?

Please visit our relevant [thread](#) first to see if your question has already been asked. You can also help us keep this space up-to-date, linking topics/issues to relevant threads and adding appropriate tags whenever/wherever possible. This is not only helpful to us but also helpful for the community! Once we have enough data points, we will establish a forum for frequently asked questions (FAQ).

Warning: Please do not open issues for general support questions as we want to preserve them for bug reports and new feature requests **ONLY**. Therefore to save everyone time, we will be systematically closing all issues that do not follow these guidelines.

If this still does not work for you and you would like to chat with someone in real-time, please contact [Ashley](#) to set up a chat or zoom meeting.

8.2.2 Issues & bugs

Are you reporting a bug?

If the code crashes or you find a bug, please search the issue tracker first to make sure the problem (i.e. issue) does not already exist. If and only if you do this but still don't find anything, feel free to submit an issue. And, if you're *really* brave, you can submit an issue along with a pull request fix.

Ideally we would love to resolve all issues immediately but before fixing a bug, we first need to reproduce and confirm it. There is a template of the required information when you submit an issue, but generally we ask that you:

- clearly and concisely explain the issue or bug
- provide any relevant data so that we can reproduce the error

- information on the software and operating system

You can file new issues by filling out our [bug report](#) template.

8.2.3 New features

Have an idea for a new feature or functionality?

Request

If you come up with an idea for a new feature that you'd like to see implemented in pySYD but do not plan to do this yourself, you can submit an issue with our [feature request](#) template.

We welcome any and all ideas!

Direct implementation

However, if you come up with a brilliant idea that you'd like to take a stab at – Please first consider what kind of change it is:

- For a **Major Feature**, first open an issue and outline your proposal so that it can be discussed. This will also allow us to better coordinate our efforts, prevent duplication of work, and help you to craft the change so that it is successfully accepted into the project.
 - Any smaller or **Minor Features** can be crafted and directly submitted as a pull request. However, before you submit a pull request, please see our [style guide](#) to facilitate and expedite the merge process.
-

8.2.4 Contributing code

Do you want to contribute code?

We would love for you to contribute to pySYD and make it even better than it is today!

Style guide

** A good rule of thumb is to try to make your code blend in with the surrounding code.

Code

- 4 spaces for indentation (i.e. no tabs please)
- 80 character line length
- commas last
- declare variables in the outermost scope that they are used
- camelCase for variables in JavaScript and for classes/objects in Python
- snake_case for variables in Python

Docstrings

Coding Rules

To ensure consistency throughout the source code, keep these rules in mind as you are working:

- All features or bug fixes **must be tested** by one or more specs (unit-tests).
 - We follow [Google's JavaScript Style Guide][js-style-guide].
-

8.2.5 Testing

[Click here to immediately get started!](#)

BIBLIOGRAPHY

- [C2014] Chaplin et al., 2014
- [H2011] Huber et al., 2011
- [L2017] Lund et al., 2017
- [S2017a] Serenelli et al., 2017
- [S2017b] Silva Aguirre et al., 2017
- [Y2018] Yu et al., 2018

PYTHON MODULE INDEX

p

- `pysyd`, [18](#)
- `pysyd.models`, [44](#)
- `pysyd.pipeline`, [29](#)
- `pysyd.plots`, [58](#)
- `pysyd.target`, [31](#)
- `pysyd.utils`, [47](#)

Symbols

--all, --showall, [101](#)
 --basis, [101](#)
 --bf, --box, --boxfilter, [101](#)
 --bin, --binning, [101](#)
 --bm, --mode, --bmode, [102](#)
 --ce, --cm, --color, [102](#)
 --cli, [102](#)
 --cv, --value, [102](#)
 --dnu, [102](#)
 --ew, --exwidth, [103](#)
 --file, --list, --todo, [103](#)
 --gap, --gaps, [103](#)
 --in, --input, --inpdire, [104](#)
 --infdire, [104](#)
 --info, --information, [104](#)
 --iw, --indwidth, [104](#)
 --laws, --nlaws, [104](#)
 --lb, --lowerb, [105](#)
 --le, --lowere, [105](#)
 --lp, --lowerp, [105](#)
 --lx, --lowerx, [105](#)
 --mc, --iter, --mciter, [106](#)
 --metric, [106](#)
 --notebook, [106](#)
 --nox, --nacross, [106](#)
 --noy, --ndown, --norders, [106](#)
 --npb, [106](#)
 --nt, --nthread, --nthreads, [107](#)
 --numax, [107](#)
 --of, --over, --oversample, [107](#)
 --out, --output, --outdir, [107](#)
 --peak, --peaks, --npeaks, [107](#)
 --rms, --nrms, [107](#)
 --se, --smoothech, [108](#)
 --sm, --smpar, [108](#)
 --sp, --smoothps, [108](#)
 --star, --stars, [108](#)
 --step, --steps, [108](#)
 --sw, --smoothwidth, [108](#)
 --thresh, --threshold, [109](#)
 --trials, --ntrials, [109](#)

--ub, --upperb, [109](#)
 --ue, --uppere, [109](#)
 --up, --upperp, [109](#)
 --ux, --upperx, [110](#)
 -a, --ask, [101](#)
 -b, --bg, --background, [101](#)
 -d, --show, --display, [102](#)
 -e, --est, --estimate, [102](#)
 -f, --fft, [103](#)
 -g, --globe, --global, [103](#)
 -i, --ie, --interpech, [103](#)
 -k, --kc, --kepcorr, [104](#)
 -m, --samples, [105](#)
 -n, --notch, [106](#)
 -o, --overwrite, [107](#)
 -s, --save, [107](#)
 -v, --verbose, [110](#)
 -w, --wn, --fixwn, [110](#)
 -x, --stitch, --stitching, [110](#)
 -y, --hey, [110](#)

A

ACF, [113](#)
 add_cli() (*pysyd.utils.Parameters method*), [47](#), [51](#)
 add_targets() (*pysyd.utils.Parameters method*), [47](#), [52](#)
 AIC, [113](#)
 Akaike Information Criterion, [113](#)
 asteroeismology, [113](#)
 autocorrelation function, [113](#)

B

background, [113](#)
 background() (*in module pysyd.models*), [44](#)
 background-corrected power spectrum, [113](#)
 background-divided power spectrum, [113](#)
 background-subtracted power spectrum, [113](#)
 Bayesian Information Criterion, [114](#)
 BCPS, [113](#)
 BDPS, [113](#)
 BIC, [113](#)
 BSPS, [113](#)

C

cadence, [114](#)
check() (in module `pysyd.pipeline`), [29](#)
check_cli() (`pysyd.utils.Parameters` method), [47](#), [52](#)
check_data() (in module `pysyd.plots`), [58](#)
check_numax() (`pysyd.target.Target` method), [31](#)
collapse_ed() (`pysyd.target.Target` method), [32](#)
compute_acf() (`pysyd.target.Target` method), [32](#)
compute_spectrum() (`pysyd.target.Target` method), [32](#)
Constants (class in `pysyd.utils`), [47](#)
correct_background() (`pysyd.target.Target` method), [33](#)
create_benchmark_plot() (in module `pysyd.plots`), [58](#)
critically-sampled power spectrum, [114](#)

D

delta_nu() (in module `pysyd.utils`), [49](#)
derive_parameters() (`pysyd.target.Target` method), [33](#)
dnu, [114](#)

E

echelle diagram, [114](#)
echelle_diagram() (`pysyd.target.Target` method), [33](#)
ED, [114](#)
estimate_background() (`pysyd.target.Target` method), [34](#)
estimate_numax() (`pysyd.target.Target` method), [34](#)
estimate_parameters() (`pysyd.target.Target` method), [35](#)

F

fast fourier transform, [114](#)
FFT, [114](#)
first_step() (`pysyd.target.Target` method), [35](#)
fix_data() (`pysyd.target.Target` method), [35](#)
folded power spectrum, [114](#)
FPS, [114](#)
frequency of maximum power, [114](#)
frequency resolution, [114](#)
frequency_spacing() (`pysyd.target.Target` method), [36](#)
full-width half maximum, [114](#)
fun() (in module `pysyd.pipeline`), [29](#)
FWHM, [114](#)

G

gaussian() (in module `pysyd.models`), [45](#)
get_background() (`pysyd.target.Target` method), [36](#)
get_background() (`pysyd.utils.Parameters` method), [48](#), [52](#)
get_data() (`pysyd.utils.Parameters` method), [48](#), [52](#)
get_defaults() (`pysyd.utils.Parameters` method), [48](#), [52](#)
get_dict() (in module `pysyd.utils`), [50](#)

get_epsilon() (`pysyd.target.Target` method), [36](#)
get_estimate() (`pysyd.utils.Parameters` method), [48](#), [53](#)
get_global() (`pysyd.utils.Parameters` method), [48](#), [53](#)
get_infdir() (in module `pysyd.utils`), [50](#)
get_inpdir() (in module `pysyd.utils`), [50](#)
get_main() (`pysyd.utils.Parameters` method), [49](#), [53](#)
get_outdir() (in module `pysyd.utils`), [50](#)
get_output() (in module `pysyd.utils`), [51](#)
get_parent() (`pysyd.utils.Parameters` method), [49](#), [53](#)
get_plot() (`pysyd.utils.Parameters` method), [49](#), [53](#)
get_samples() (`pysyd.target.Target` method), [37](#)
get_sampling() (`pysyd.utils.Parameters` method), [49](#), [53](#)
global properties, [114](#)
global_fit() (`pysyd.target.Target` method), [37](#)
granulation, [114](#)

H

harvey_fit() (in module `pysyd.models`), [45](#)
harvey_fourth() (in module `pysyd.models`), [45](#)
harvey_none() (in module `pysyd.models`), [45](#)
harvey_one() (in module `pysyd.models`), [45](#)
harvey_regular() (in module `pysyd.models`), [46](#)
harvey_second() (in module `pysyd.models`), [46](#)
harvey_three() (in module `pysyd.models`), [46](#)
harvey_two() (in module `pysyd.models`), [46](#)
Harvey-like component, [114](#)
Harvey-like model, [114](#)

I

initial_estimates() (`pysyd.target.Target` method), [37](#)
initial_parameters() (`pysyd.target.Target` method), [38](#)
InputError, [47](#)
InputWarning, [47](#)

K

Kepler artefact, [114](#)
Kepler legacy sample, [114](#)

L

large frequency separation, [115](#)
light curve, [115](#)
load() (in module `pysyd.pipeline`), [30](#)
load_file() (`pysyd.target.Target` method), [38](#)
load_power_spectrum() (`pysyd.target.Target` method), [38](#)
load_time_series() (`pysyd.target.Target` method), [39](#)

M

make_plots() (in module `pysyd.plots`), [58](#)
mesogranulation, [115](#)

mixed modes, [115](#)

model_background() (*pysyd.target.Target method*), [39](#)

module

 pysyd, [18](#)

 pysyd.models, [44](#)

 pysyd.pipeline, [29](#)

 pysyd.plots, [58](#)

 pysyd.target, [31](#)

 pysyd.utils, [47](#)

N

notching, [115](#)

numax, [114](#)

numax_gaussian() (*pysyd.target.Target method*), [40](#)

numax_smooth() (*pysyd.target.Target method*), [40](#)

nyquist frequency, [115](#)

O

optimize_ridges() (*pysyd.target.Target method*), [40](#)

oversampled power spectrum, [115](#)

P

p-mode oscillations, [115](#)

parallel() (*in module pysyd.pipeline*), [30](#)

Parameters (*class in pysyd.utils*), [47](#), [51](#)

plot() (*in module pysyd.pipeline*), [30](#)

plot_1d_ed() (*in module pysyd.plots*), [58](#)

plot_bgfits() (*in module pysyd.plots*), [58](#)

plot_estimates() (*in module pysyd.plots*), [58](#)

plot_light_curve() (*in module pysyd.plots*), [58](#)

plot_parameters() (*in module pysyd.plots*), [59](#)

plot_power_spectrum() (*in module pysyd.plots*), [59](#)

plot_samples() (*in module pysyd.plots*), [59](#)

power excess, [115](#)

power spectral density, [115](#)

power spectrum, [116](#)

power() (*in module pysyd.models*), [46](#)

process_star() (*pysyd.target.Target method*), [41](#)

ProcessingError, [49](#)

ProcessingWarning, [49](#)

PS, [116](#)

PSD, [115](#)

pysyd

 module, [18](#)

pysyd.models

 module, [44](#)

pysyd.pipeline

 module, [29](#)

pysyd.plots

 module, [58](#)

pysyd.target

 module, [31](#)

pysyd.utils

 module, [47](#)

R

red_noise() (*pysyd.target.Target method*), [41](#)

remove_artefact() (*pysyd.target.Target method*), [41](#)

run() (*in module pysyd.pipeline*), [30](#)

S

scaling relations, [116](#)

select_trial() (*in module pysyd.plots*), [59](#)

setup() (*in module pysyd.pipeline*), [31](#)

setup_dirs() (*in module pysyd.utils*), [51](#)

show_results() (*pysyd.target.Target method*), [42](#)

single_step() (*pysyd.target.Target method*), [42](#)

solar_scaling() (*pysyd.target.Target method*), [42](#)

solar-like oscillations, [115](#)

stitch_data() (*pysyd.target.Target method*), [43](#)

T

Target (*class in pysyd.target*), [31](#)

W

white() (*in module pysyd.models*), [47](#)

white_noise() (*pysyd.target.Target method*), [43](#)

whiten, [116](#)

whiten_mixed() (*pysyd.target.Target method*), [43](#)

whitening, [116](#)