

# PyCFIT User Guide

Python Component Fitting System – Version 1.0

Ayris Narock  
ADNET Systems, Inc.  
NASA Goddard Space Flight Center  
January 2026

## Contents

1	Introduction.....	2
1.1	Purpose and Description .....	2
1.2	Inspiration .....	2
2	Single Spectrum Fitting.....	2
2.1	Calling the Program .....	2
2.2	Adding and Removing Components .....	4
2.3	Finding the ‘Best Fit’ .....	6
2.4	Adjusting Component Conditions.....	7
2.4.1	Using the Interactive Graph Window .....	7
2.4.2	Using the Parameter Dialog.....	8
2.4.3	Naming Your Components .....	10
2.5	Storing and Recovering the Fit .....	12
3	Raster Fitting .....	13
3.1	Motivation .....	13
3.2	Fitting a Data Cube.....	14
3.3	Inspect and Modify Raster Fit .....	15
3.3.1	Interface Description .....	15
3.3.2	Adjusting a Fit .....	18
3.4	Storing and Recovering Results .....	21
3.4.1	The `get_results` method.....	21
3.4.2	ASDF Files .....	22
4	Interactive Help System.....	22
Appendix A:	Installation .....	24
A.1	System Requirements .....	24
A.2	Python Package Install.....	24
Appendix B:	Command-line Usage .....	25
B.1	Overview .....	25
B.2	Single Spectrum Fitting Commands .....	25
B.3	Raster Fitting Commands .....	28

# 1 Introduction

## 1.1 Purpose and Description

The PyCFIT (Python Component Fitting System) software has been developed to facilitate the analysis of raster data from solar imaging spectrographs and is comprised of two main modules. The first is an interface for using simple model components to build a complex model suitable for fitting to a single spectrum. The second is an interface to extend this initial model across a data block efficiently. While PyCFIT has been developed so that these functions can be completed at the command line, its strongest value is with its graphical components, which make finding and understanding these fits easier and more intuitive.

## 1.2 Inspiration

PyCFIT was inspired by the longstanding [Component Fitting System \(CFIT\) for IDL](#), developed initially in support of the SoHO mission. As many in the community are becoming more reliant on Python and less on IDL, we thank the IDL CFIT developers for their whole-hearted support for this Python tool and their lessons-learned which informed our design.

# 2 Single Spectrum Fitting

## 2.1 Calling the Program

To initiate the graphical user interface (GUI) for the single spectrum fitting tool, execute the `cfit_gui` function from within your Python environment:

```
import pycfit
model = pycfit.cfit_gui(wavelength, intensity, uncertainty = uncertainty)
```

where `wavelength`, `intensity`, and `uncertainty` are one-dimensional arrays of the same shape. The example data used here are included in the `pycfit.data` module:

```
wavelength,intensity,uncertainty = pycfit.data.get_example_single_spectrum()
```

If the intent is to develop an Astropy model for fitting a larger data set, we recommend that an average of the larger data set be used at this stage.

This will create a graphical window similar to Figure 1, with the main plot area showing the intensity vs. wavelength of the data to be fit. If optional uncertainty data has been passed then this is also displayed, first as error bars on each of the main data points and again as error bars in the smaller residual plot window below. At this point, no residuals exist without a fit function defined.

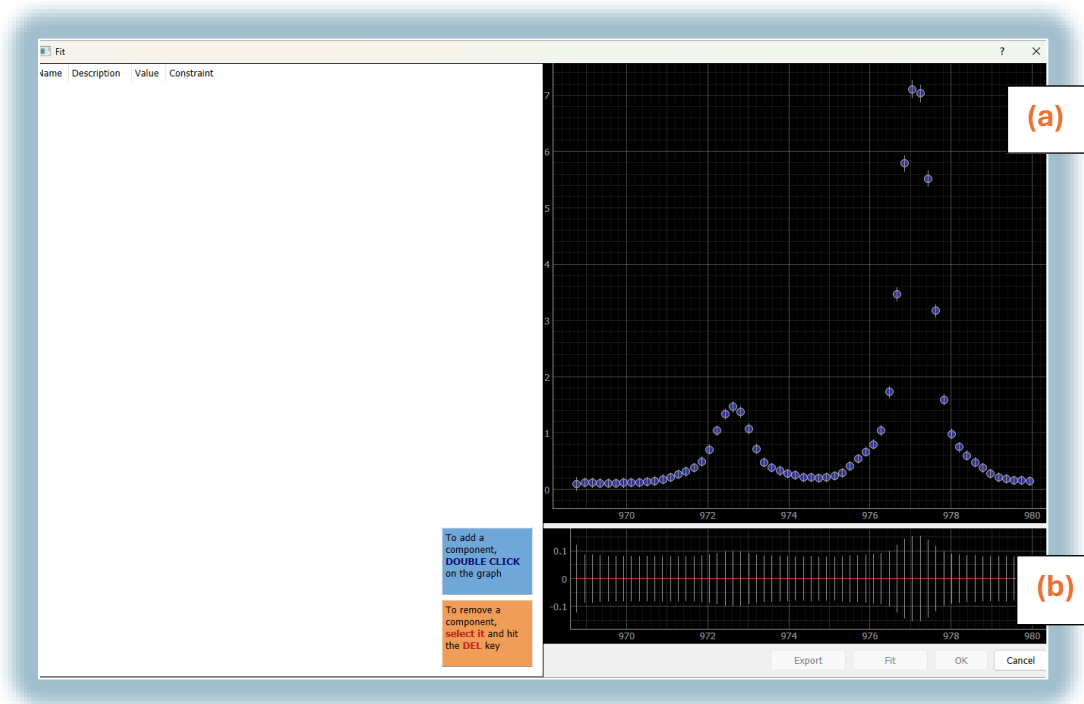


Figure 1

You may optionally pass an initial model to be used for fitting via the `function` keyword to `cfits_gui`. It accepts either an Astropy `Model` / `CompoundModel` object or a file path to an appropriately defined Python model definition file. If providing a file, it should implement a `define_model()` function. Use `Export` as described in the ‘[Storing and Recovering the Fit](#)’ section to see an example `.py` file.

```
from astropy.modeling.models import Gaussian1D

model_in = Gaussian1D(amplitude=1.5, mean=972, stddev=0.57)
model = pycfit.cfits_gui(wavelength,
                        intensity,
                        uncertainty = uncertainty
                        function = model_in)
```

This then opens a window as before, but with the initial fit model also displayed, as well as the residuals between it and the measured data.

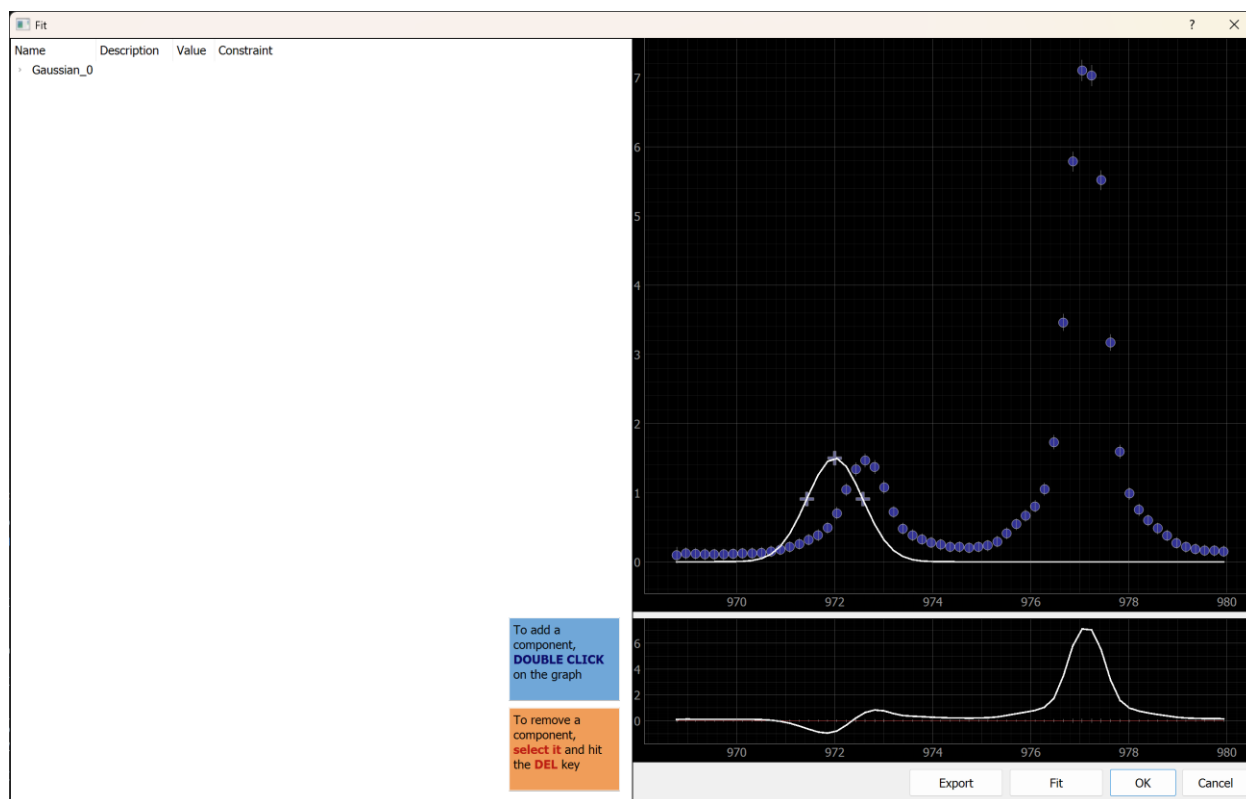


Figure 2

## 2.2 Adding and Removing Components

A User builds a model for fitting to the data by creating a combination of supported component models. To add a new Component, double click on the graphical display and select the type of component to add (See Figure 3). Default parameters for each component will be set based on where in the graph window you have clicked, the data to be fit, and in some cases, the existing residuals. Exact default behavior is dependent on the Component type.

Constant and Linear components add a line crossing through the point clicked. A Constant component will always have a slope of 0, while the Linear will default to the slope between the first and last data point.

Quadratic components, when added, will default to fitting a 2<sup>nd</sup> degree polynomial through the two endpoints of the spectrum and the clicked point.

Gaussian and Moffat components will choose defaults such that the peak will occur at the x-axis value where the User clicked and have a y-value which brings the residual of the compound model at that point to 0.

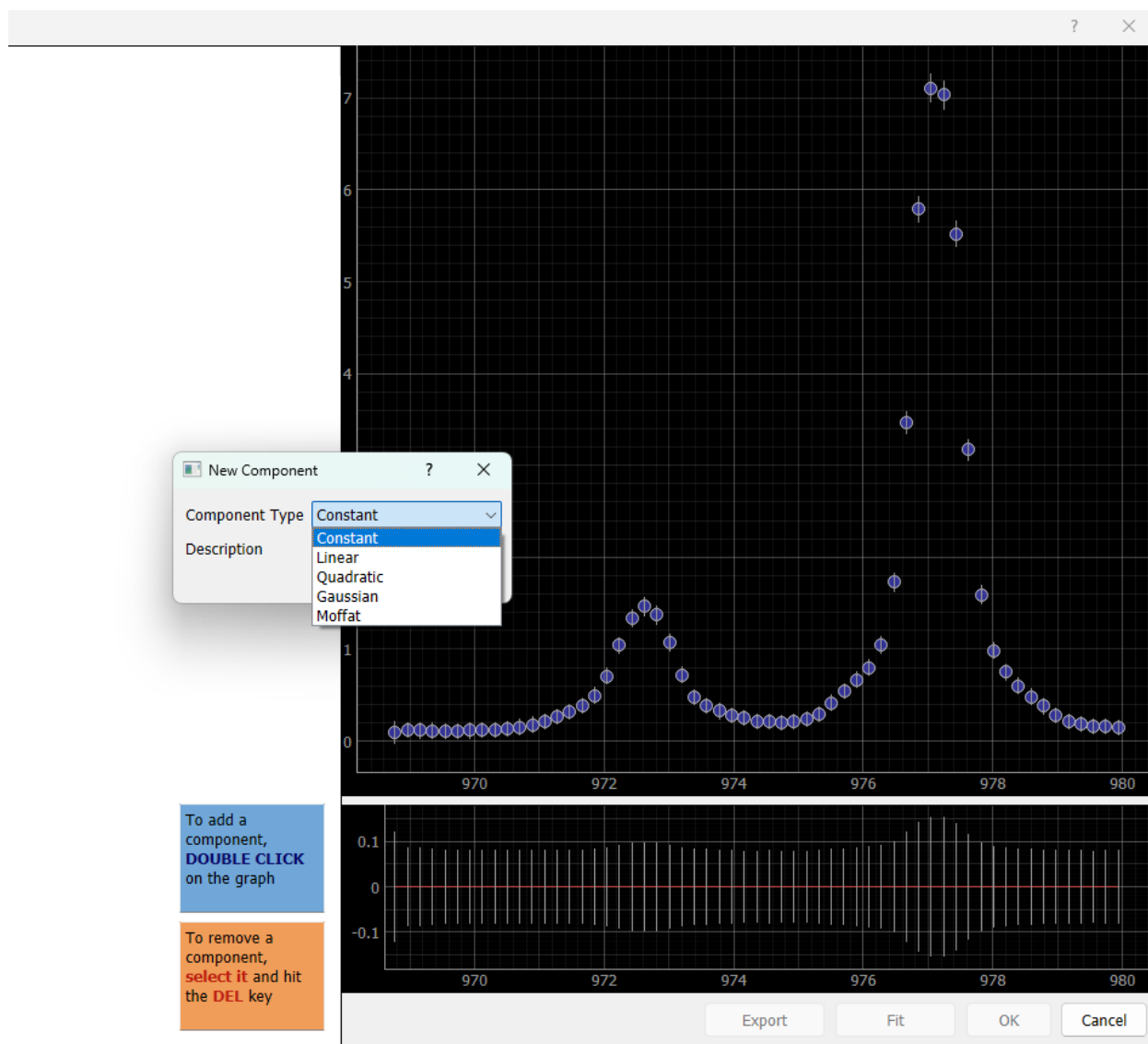


Figure 3

After components have been added, they will each be reflected on the plot window as a thin, white line and their name and parameter information displayed in the Component Tree display on the left side. A thicker white line displays the Compound Model and the residuals between the data and model, respectively. See Figure 4.

Selecting one of the elements from the Component Tree display with the mouse will highlight that Component graph in red within the plot window. While selected, you can use the **DEL** key to remove that Component from the Compound Model.

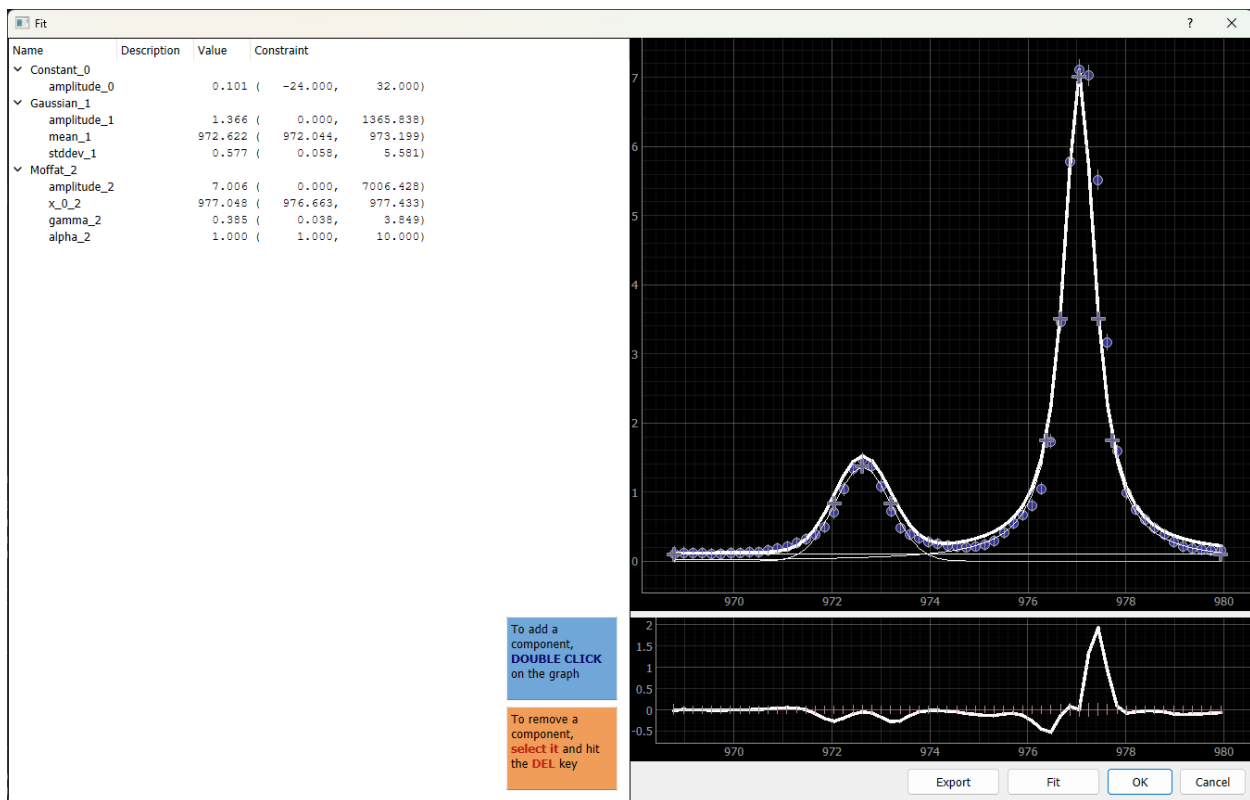


Figure 4

## 2.3 Finding the 'Best Fit'

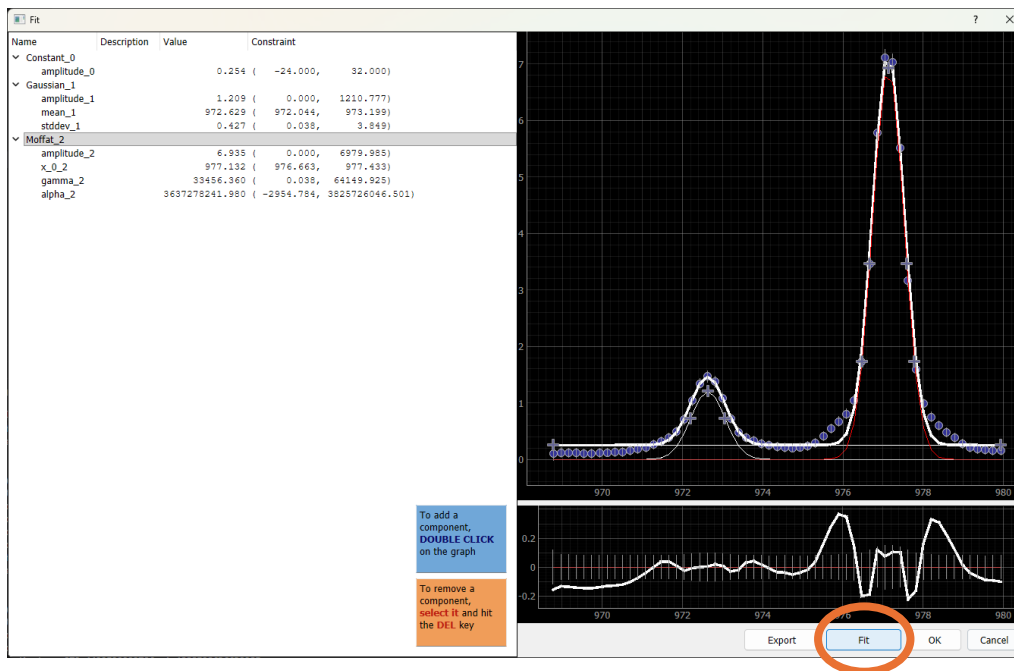


Figure 5

Selecting the **Fit** button (See Fig. 5) will use Astropy's Trust Region Reflective algorithm and least squares statistic (**TRFLSQFitter()**) to fit the currently active Compound Model to the data. Bounding conditions for each Parameter of the model can be set via the **Parameter Dialog** in the Component Tree.

## 2.4 Adjusting Component Conditions

We attempt to create reasonable defaults when adding a Component to the system, however, you may wish to alter these to better fit your use case. This can be done either via the interactive graph window or through adjustments to individual parameters in the Component Tree display.

### 2.4.1 Using the Interactive Graph Window

Each Component element visualized in the plot window has two or more plus shaped ('+') grab points for interaction. Use a click-hold and drag with the mouse to reshape the component to best suit the target fit (See Fig. 6 [right side]). As the Component is adjusted graphically, the associated initial conditions for the parameters of that Component will be adjusted to match and can be inspected within the Component Tree display.

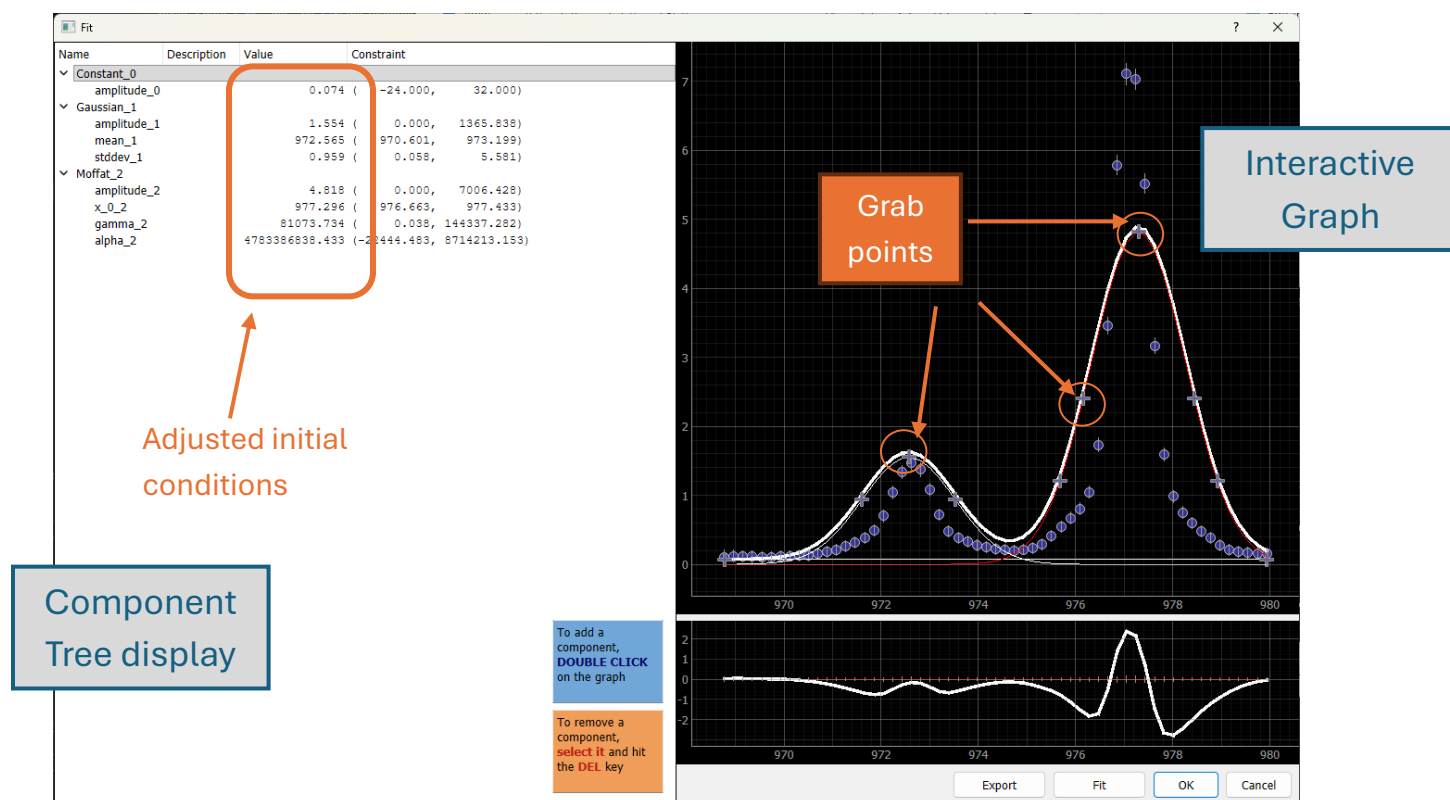


Figure 6



## 2.4.2 Using the Parameter Dialog

Navigate to the Parameter Dialog in the Component Tree display to fine-tune the initial conditions of the model Components and to adjust the fitting constraints. Within the Tree Display (Fig. 6 [left side]), expand the Component of interest so that you can see the component model parameters. In Figure 7, see that the Gaussian model has three parameters: amplitude, mean, and standard deviation. Double-clicking on a parameter will open the Parameter Dialog as shown.

Within this dialog you can change the value for the model parameter. Modifying this will adjust the displayed component and the compound model in the plot panel. You also have the option to designate this parameter as free, fixed, or tied to another parameter during fitting. When the parameter is allowed to be 'free', you can optionally select upper and lower bounds between which it can range. A 'fixed' parameter will be held at the value you have set at the time of fitting.

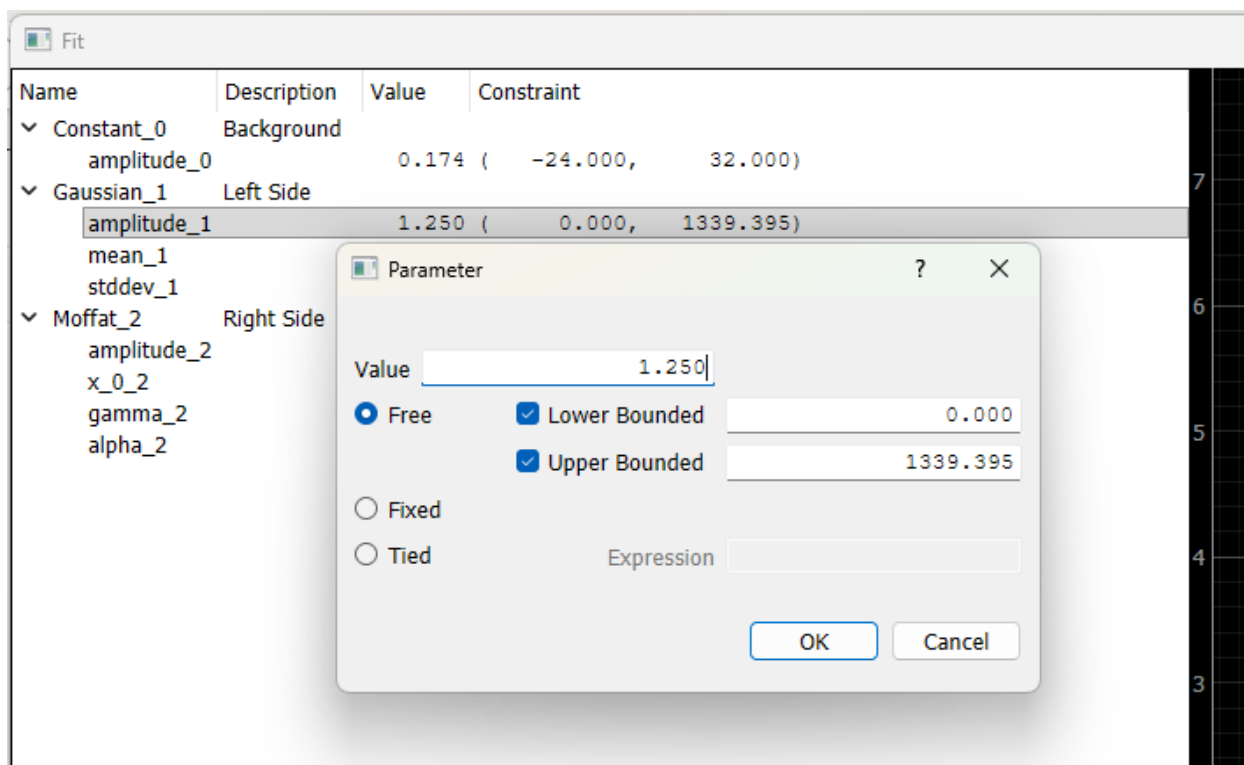


Figure 7

Finally, you could choose to tie the fitting of this parameter to the other parameters in the Compound Model. After selecting 'Tied' you will need to supply an arithmetic expression defining the relationship you want to maintain. The identifiers for each Parameter are unique, with a trailing numerical tag that corresponds to their Component name. In the example shown in Figure 8, we are tying the amplitude of the Gaussian Component with the amplitude of the Moffat Component. From the Component Tree display we see that the name for the amplitude in the Gaussian is 'amplitude\_1' and for the Moffat function it is

'amplitude\_2'. Setting our Tied function to " $0.3 * \text{amplitude\_2}$ " constrains the fit so that the final result will always satisfy the equation  $\text{amplitude\_1} = 0.3 * \text{amplitude\_2}$ .

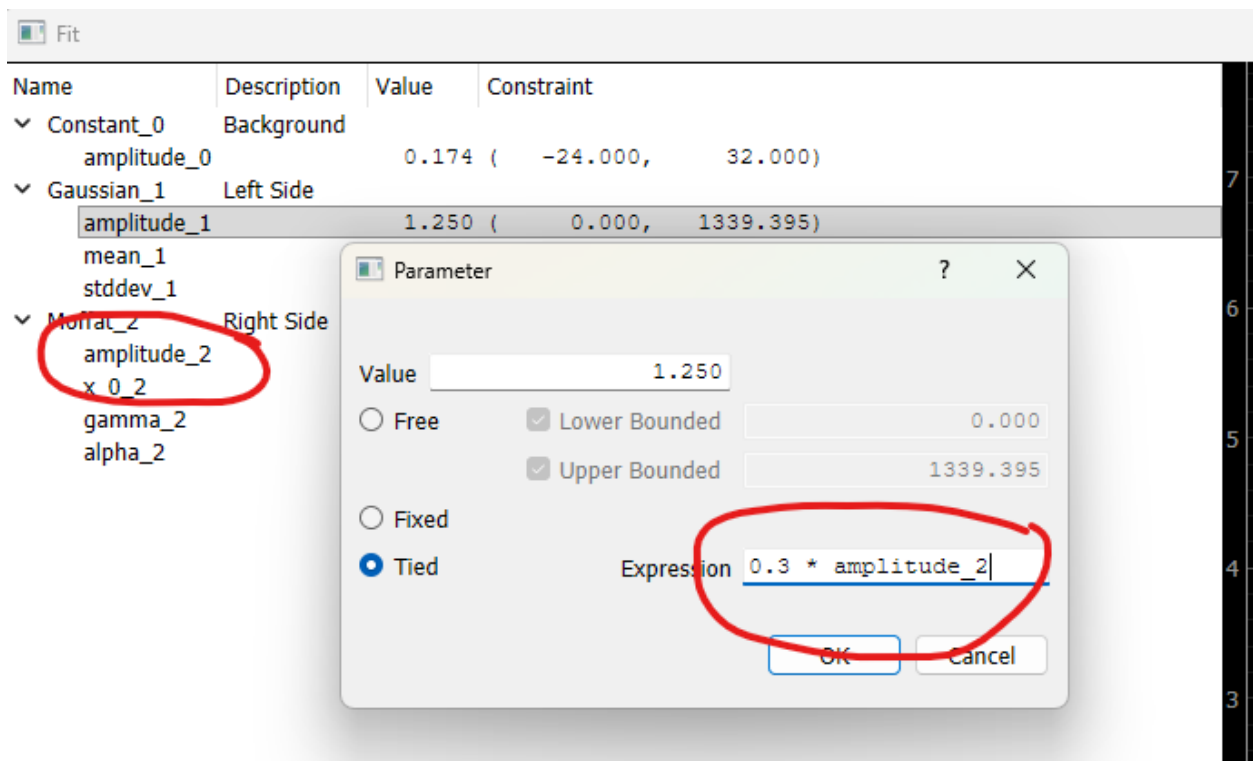


Figure 8

With the amplitude parameters Tied as shown above, the resulting best fit is shown in Figure 9. The Gaussian amplitude is 2.228 and the tied Moffat amplitude is 7.419.

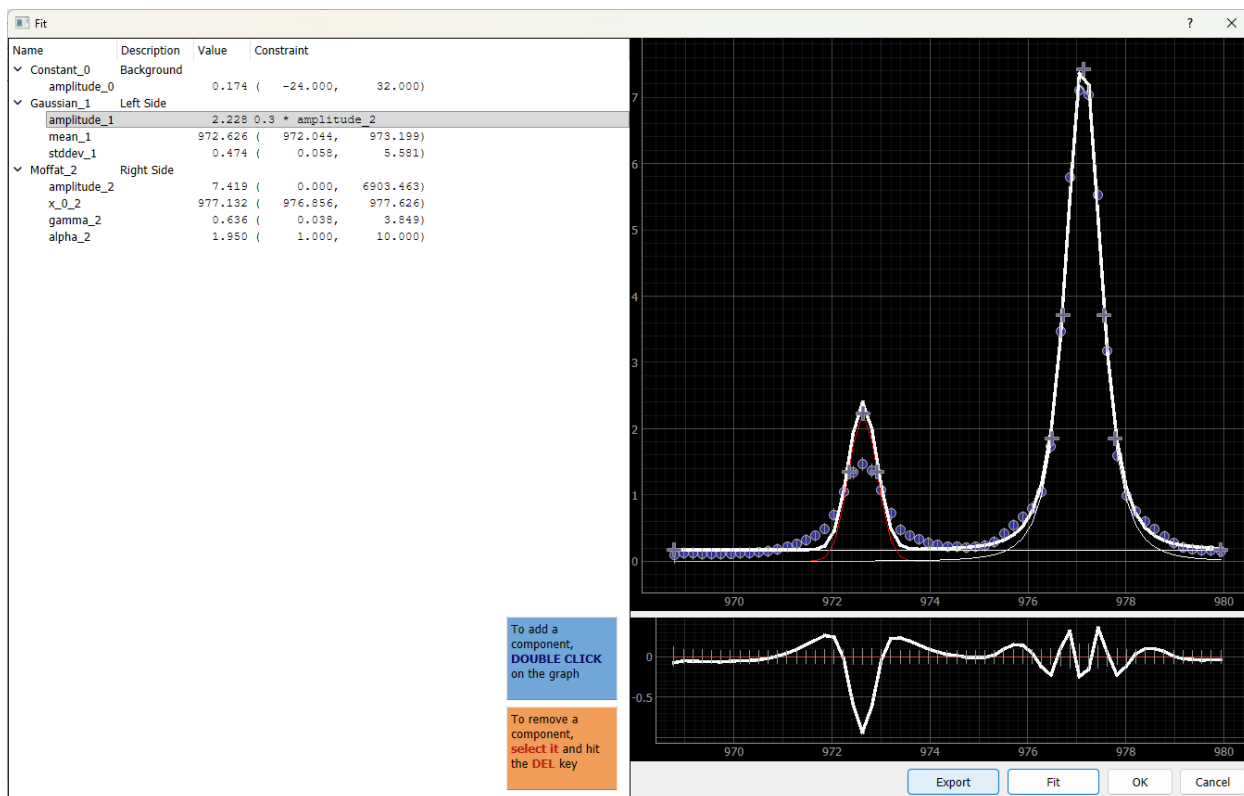


Figure 9

### 2.4.3 Naming Your Components

Each Component in the Compound Model can have a descriptive name associated with it, in addition to the identifier assigned by the program. By default, the components are unnamed. Set a descriptive name by double-clicking on the Component name in the Component Tree display. (Figure 10)

After adding your descriptive text and choosing **OK**, your description will be associated with that Component and visible in the Component Tree display in the “Description” column. It will also be stored with the model returned from the program and stored when using the **Export** feature. (Figure 11)

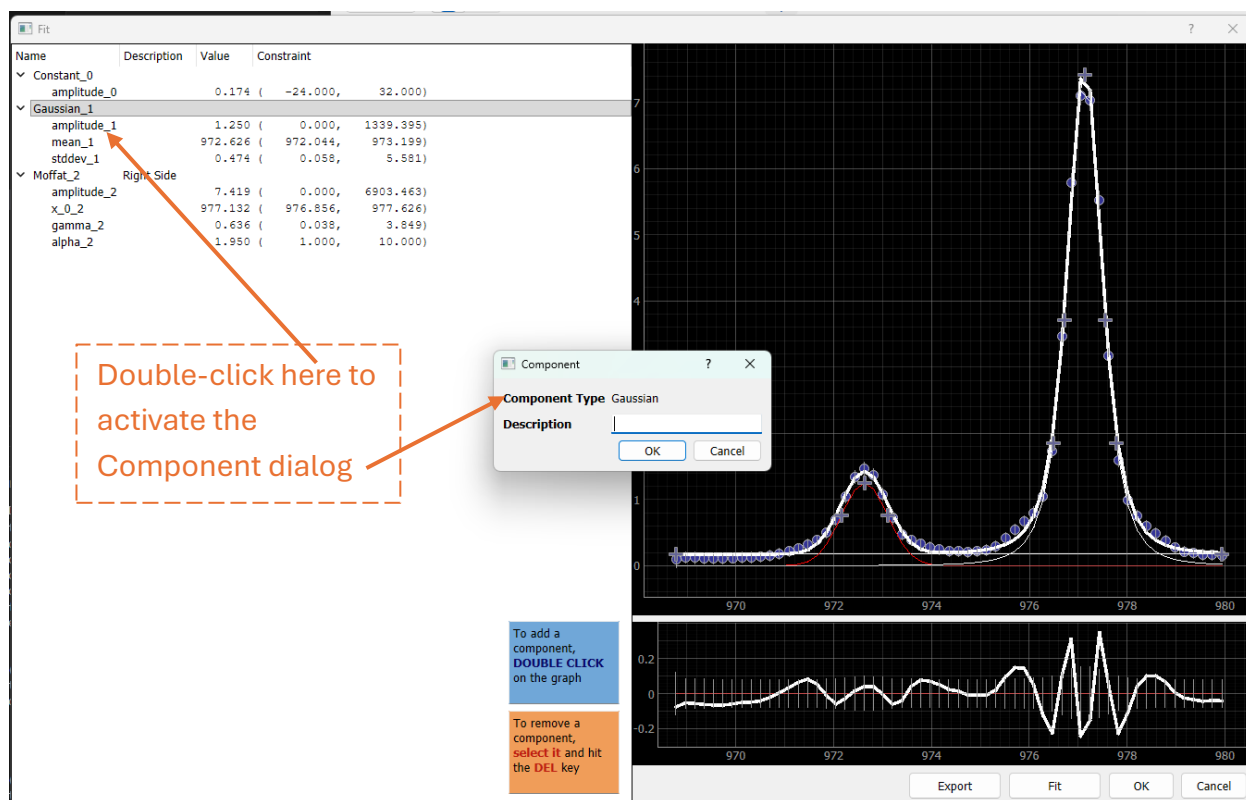


Figure 10

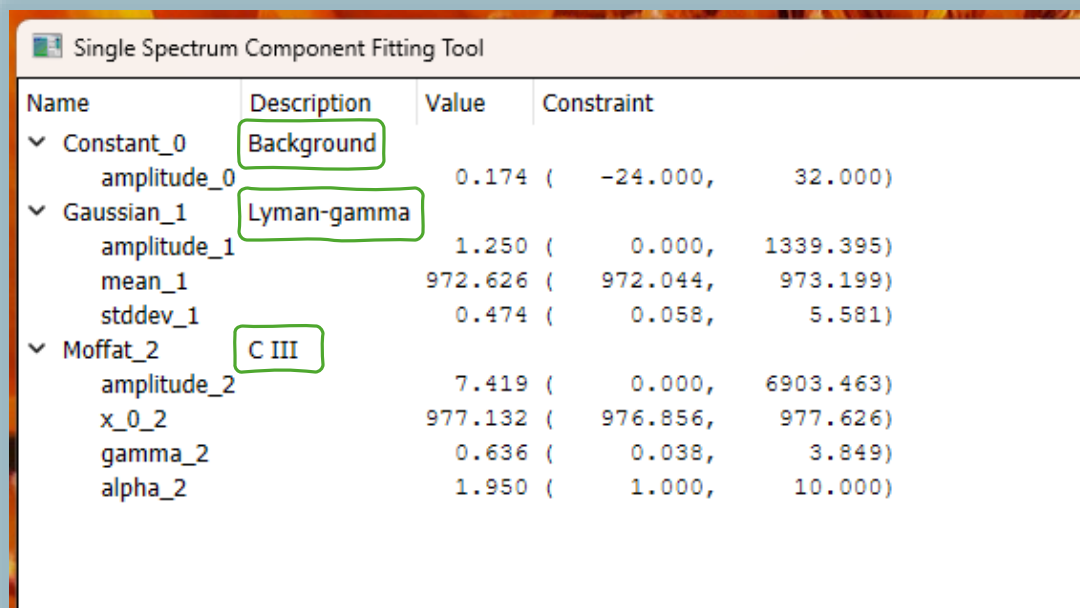


Figure 11: A view of the Component Tree display after some descriptive names have been added

## 2.5 Storing and Recovering the Fit

The fit found in this single-spectrum fitting phase is often used as an initial condition for a grid-based fitting. The User can get this model either interactively at the time of creation or via export to a file and a later re-instantiation of the model.

Recall, we initiated the interface with:

```
model = pycfit.cfit_gui(wavelength, intensity, uncertainty = uncertainty)
```

When the User selects the **OK** or **Cancel** button, this single spectrum GUI program is closed and returns to the Python command line with a return value of either an Astropy Model or **None**. When the **OK** button is selected the compound model currently displayed in the GUI is returned. If **Cancel** is selected and the program was called with the optional **function=** keyword, then the initial model is returned as an Astropy Model object, otherwise **None** is returned.

To store the model for later use, select **Export** from the button bank in the main window.

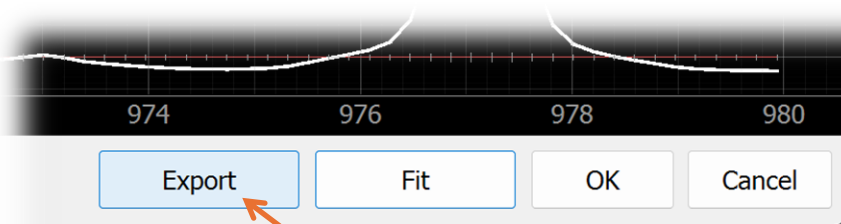


Figure 12

To store the model as a Python module or a Pickle file

You can select to pickle the Astropy model or store it in a readable Python file. If the Python file is chosen, all aspects of the model and the fitting constraints will be preserved in the generated Python module.

```

1
2 from astropy.modeling.models import Const1D
3 from astropy.modeling.models import Linear1D
4 from astropy.modeling.models import Polynomial1D
5 from astropy.modeling.models import Gaussian1D
6 from astropy.modeling.models import Moffat1D
7
8
9 def define_model():
10     model = Const1D() + Gaussian1D() + Moffat1D()
11
12     # Constant_0
13     model.amplitude_0.value = 0.174
14     model.amplitude_0.fixed = False
15     model.amplitude_0.bounds = (-24.0, 32.0)
16     model.amplitude_0.tied = False
17
18     # Gaussian_1
19     model.amplitude_1.value = 1.250
20     model.amplitude_1.fixed = False
21     model.amplitude_1.bounds = (0.0, 1339.395)

```

Figure 13

The file includes an importable function [`define_model()`] used to create the model (See Fig. 13). For instance, if you have exported to the file ‘mymodel.py’, you can access this model in the future this way:

```

import mymodel as my_model
model = my_model.define_model()

```

Alternately, you can initiate a new GUI session by passing this filename via the optional `function=` keyword argument to `pycfits.cfit_gui()`:

```

model = pycfits.cfit_gui(wavelength, intensity, function='mymodel.py')

```

## 3 Raster Fitting

### 3.1 Motivation

After analyzing a single spectrum, it is often desirable to apply a similar fit to a whole data block. PyCFIT provides this functionality through its grid interface.

## 3.2 Fitting a Data Cube

In PyCFIT, extending a fit across a 2-dimensional raster of data is accomplished via the `Grid` object. It can be instantiated with either a call to `pycfit.cfit_grid()` or `pycfit.cfit_grid_gui()`. In both cases it takes three required arguments:

- **model:**
  - An `astropy.Model` type describing the underlying model to be used for fitting across all unmasked points. Initial parameters for fitting at each of the grid points will be set to match this model. This can be set up using `pycfit.cfit_gui()` or independently.
- **wavelength:**
  - A length `N`, 1-dimensional `numpy.array` type representing the independent variable at each of the grid points. `(N,)`
- **intensity:**
  - A 3-dimensional `numpy.array` type representing the dependent variable. For a Grid of shape `(X, Y)`, this array has shape `(N, Nx, Ny)`

Optional uncertainty and mask parameters can also be passed:

- **uncertainty:**
  - (`numpy.array`) Uncertainty of the dependent variable. [Default is equal weighting to all] `(N, Nx, Ny)`
- **mask:**
  - (`boolean numpy.array`) Indicates data to ignore during fitting. Use is driven by bad data or other data filters. [Default is no mask] `(N, Nx, Ny)`
- **user\_mask:**
  - (`boolean numpy.array`) Location of `Grid` points to ignore and not fit. Use is driven by User not wanting to fit the spectrum at a given location. [Default is no mask] `(Nx, Ny)`

Example grid data are also included in the `pycfit.data` module:

```
( wavelength,
  intensity,
  uncertainty,
  mask) = pycfit.data.get_example_grid_spectra(patch=True)
```

The `patch=True` argument returns only a small subset of the full raster, for quicker interactions and fitting when learning the pyCFIT interface.

Calling the non-gui version:

```
GM = pycfit.cfit_grid(model, wavelength, intensity, uncertainty)
```

will create a `Grid` object without fitting across the grid by default, allowing the User to modify initial conditions or mask points at the command line before an initial fit is done. See the instructions in [Appendix B: Command-line Usage](#) for details.

Calling the GUI version,

```
GM = pycfit.cfit_grid_gui(model,
                           wavelength,
                           intensity,
                           uncertainty,
                           mask=mask)
```

will create the `Grid` object, perform an initial fitting across the grid, and open the graphical interface for the inspection and modification of the block fitting.

### 3.3 Inspect and Modify Raster Fit

The raster fit can be inspected with the grid fit GUI. It can be opened as described above, with base-model and data arrays, or by passing in an already instantiated `Grid` object.

```
GM = pycfit.cfit_grid_gui(GM)
```

This call will fit across the grid if no previous fit has been completed.

#### 3.3.1 Interface Description

The Grid interface has 7 panel views to examine the fitting at a selected point. The active point can be changed by double clicking within any panel in the `(b)`, `(c)`, or `(d)` columns, or by adjusting the text in the `"λ index"`, `"x index"`, and `"y index"` input boxes and hitting the `"Set Point"` button.



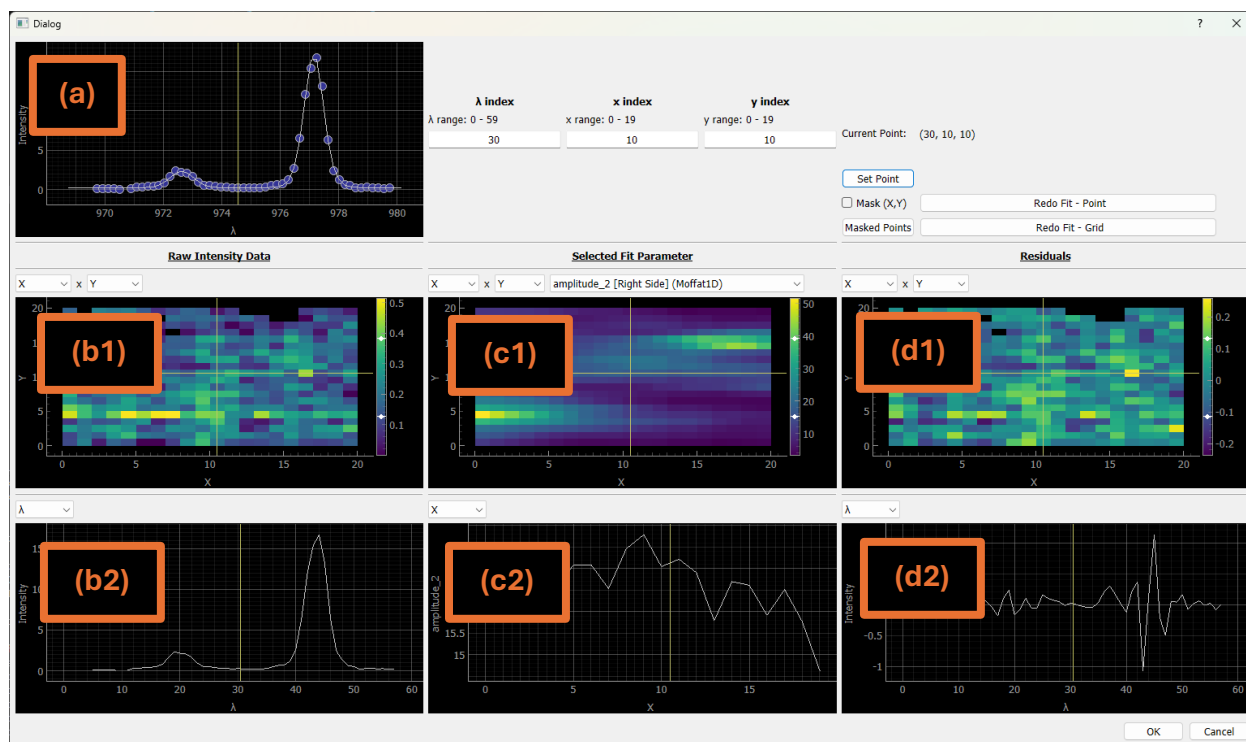


Figure 14

The **(a)** panel is an overview of the wavelength vs. intensity at the given X,Y in the grid, showing both the measured data and the fitted model.

The **(b)** column allows exploration of the intensity values across the data cube. The axes can be changed with the drop-down menus. By default, **(b1)** shows the intensity at the selected  $\lambda$  across each X,Y point and **(b2)** the measured intensity for all  $\lambda$  at the selected X,Y.

The **(c)** column is for exploring the fitted model parameter values across the space. In addition to the axes, at the top of this column there is a drop-down menu to select the parameter of interest (Figure 15). Each entry is described by the specific parameter name from the base model, any User defined descriptive names [in square brackets], and the Astropy component type (Gaussian1D, Constant, etc.).

Finally, use the **(d)** column to examine the intensity residuals across X, Y, and  $\lambda$ .

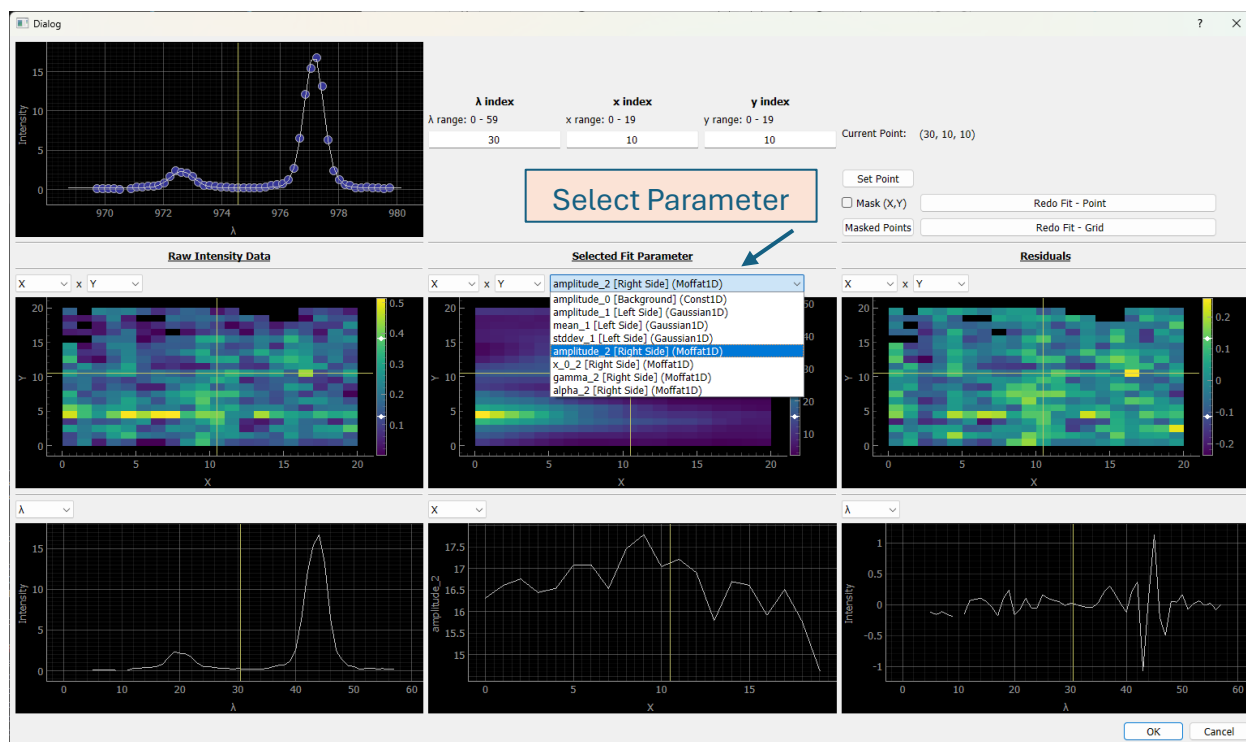


Figure 15

Each of the spectrogram plots (panels (b1), (c1), and (d1)) has a color table and range adjustable through the interactive color bar. To change the color table, hover over the color bar and activate the context menu (right-click on Windows) and navigate to the table of your choice. Selecting **None** will give grayscale.

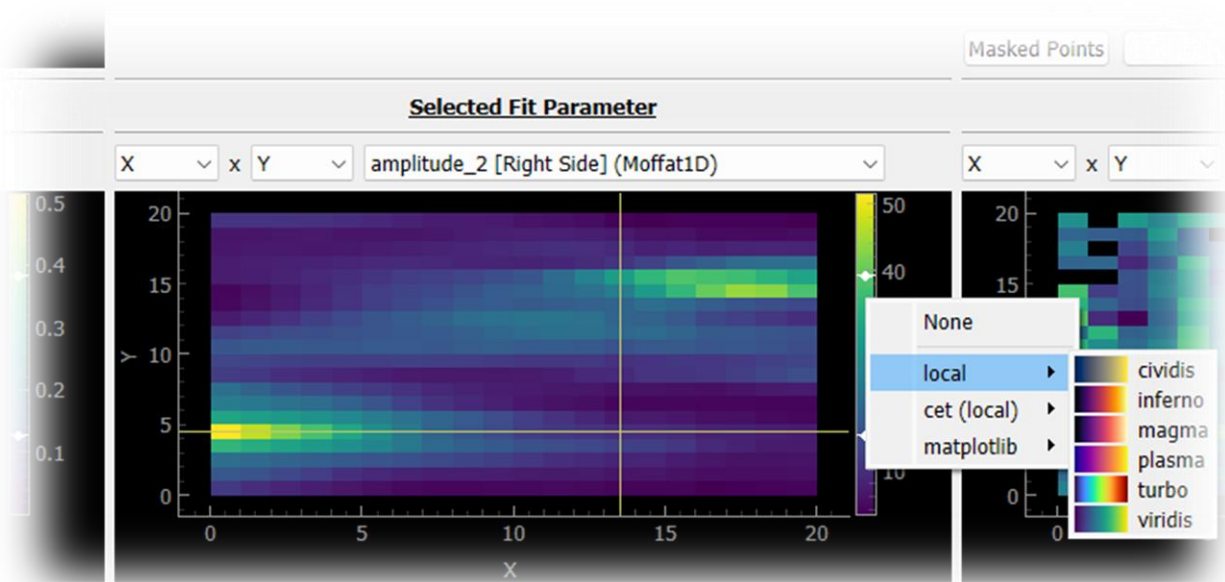


Figure 16

Doing a click-drag with either of the white drag-points within the color bar will allow you to adjust the range over which the color table is applied.

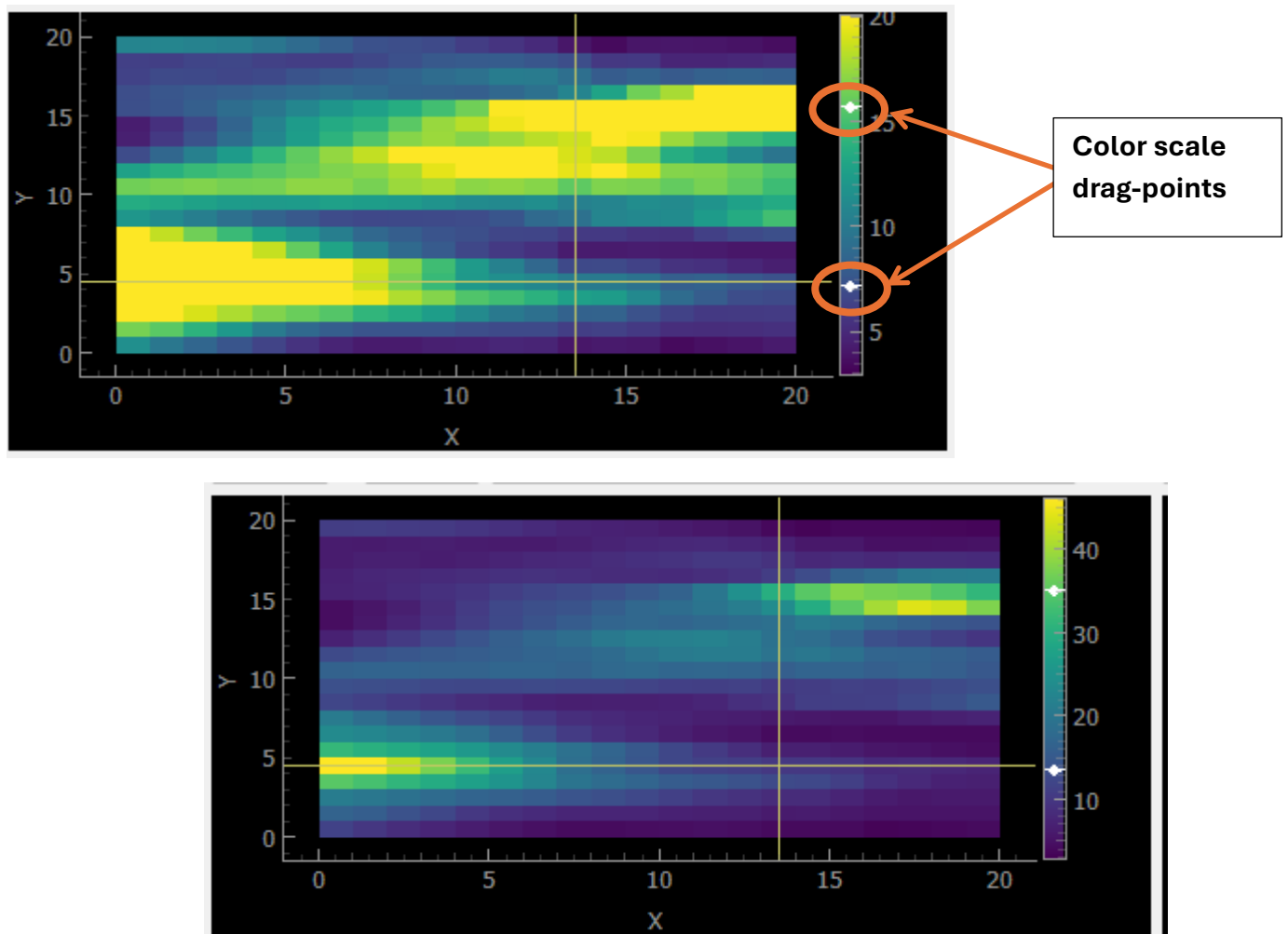


Figure 17

### 3.3.2 Adjusting a Fit

At any given X,Y point in the grid the fit can have its parameters adjusted and be refit by selecting the **"Redo Fit - Point"** button (Figure 18) in the main Grid inspection interface:

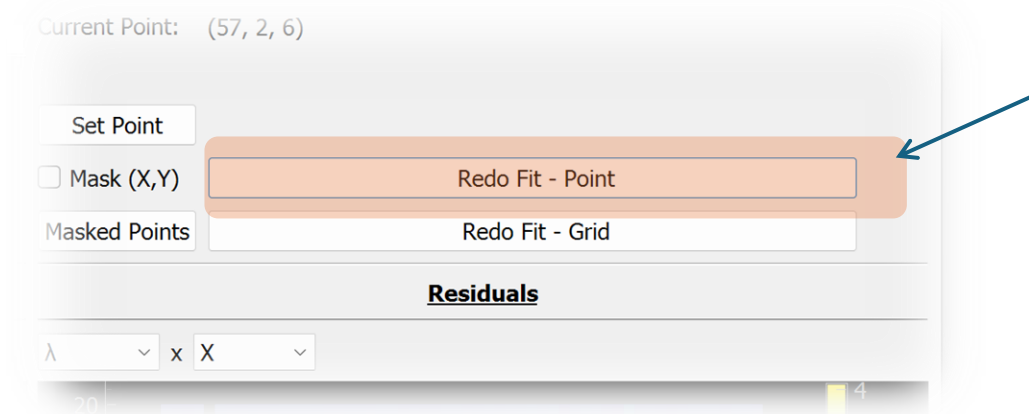


Figure 18

This will open a single spectrum fit window similar to the one from the call `pycfit.cfit_gui()` (Figures 19-20). Initial conditions, boundaries, and tied functions can all be set for each Parameter. Adding and removing Components are disabled when working within the Grid redo-fit so that each point is fit from the same base model. However, a Component can be effectively removed by setting it to a fixed value of 0.

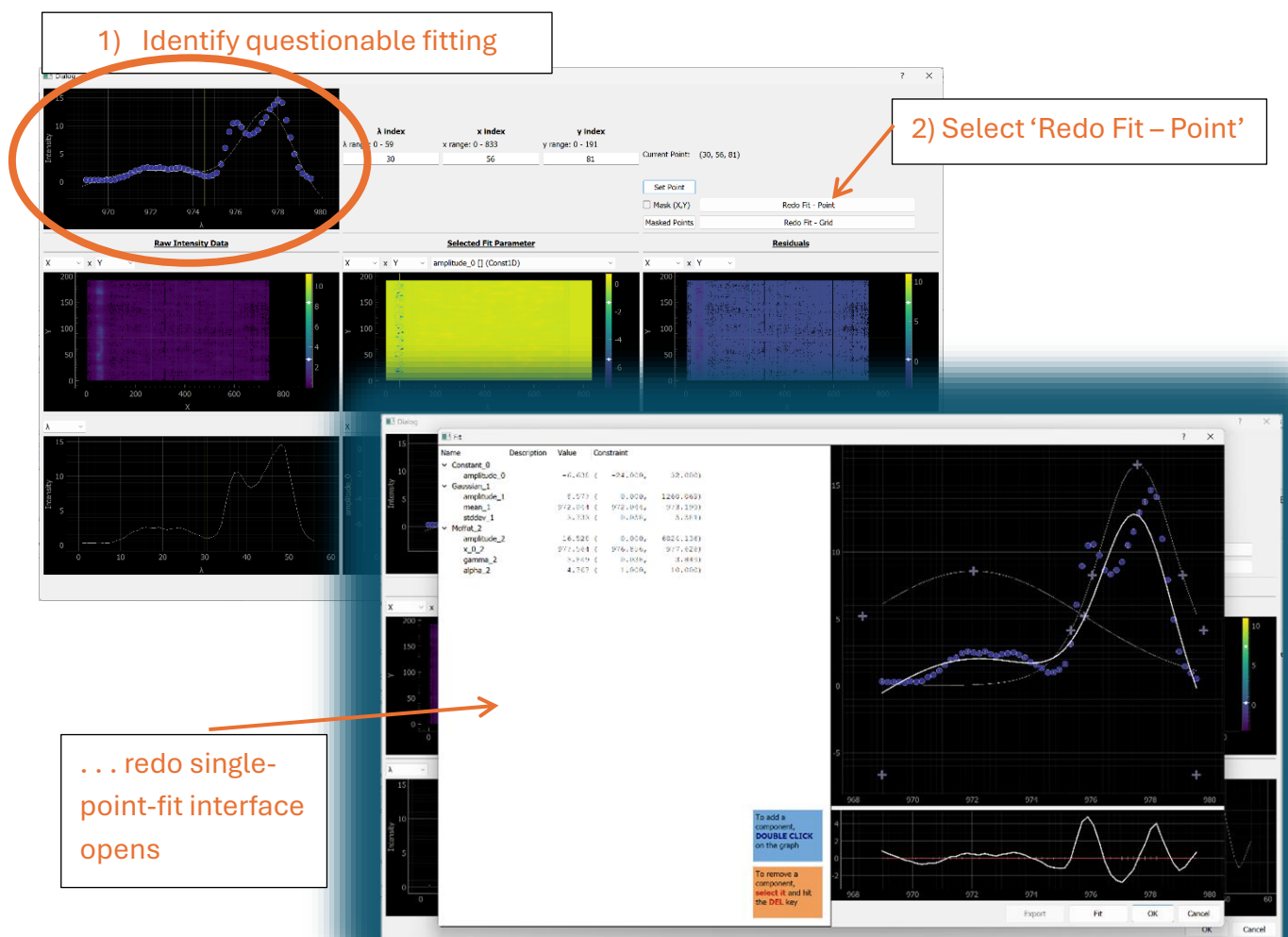


Figure 19

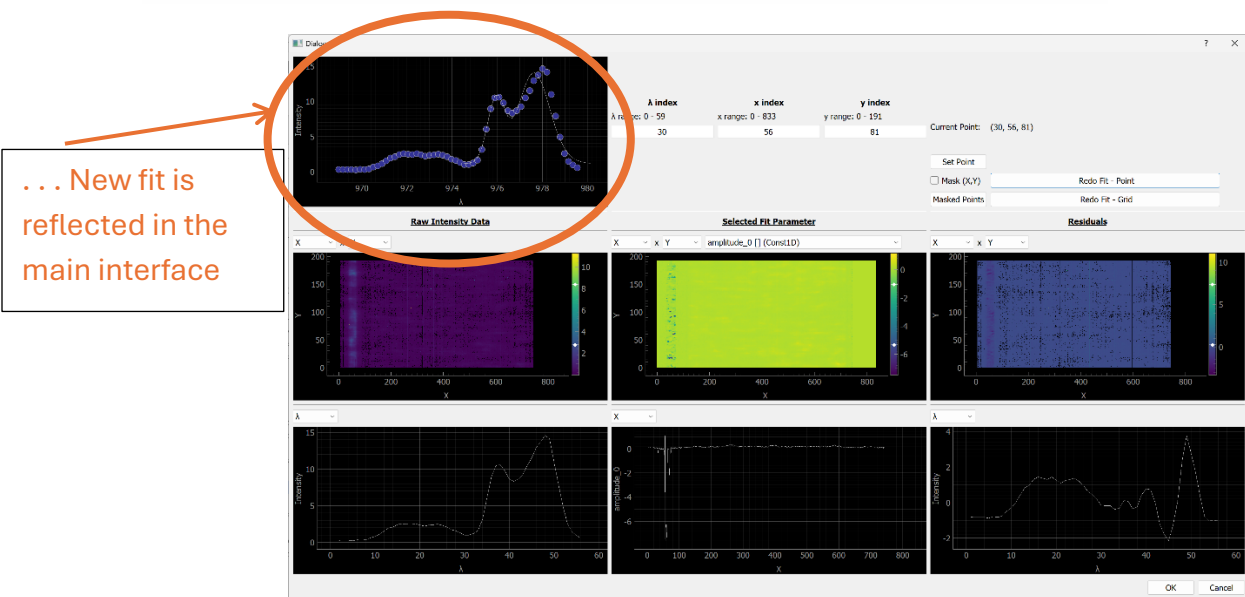
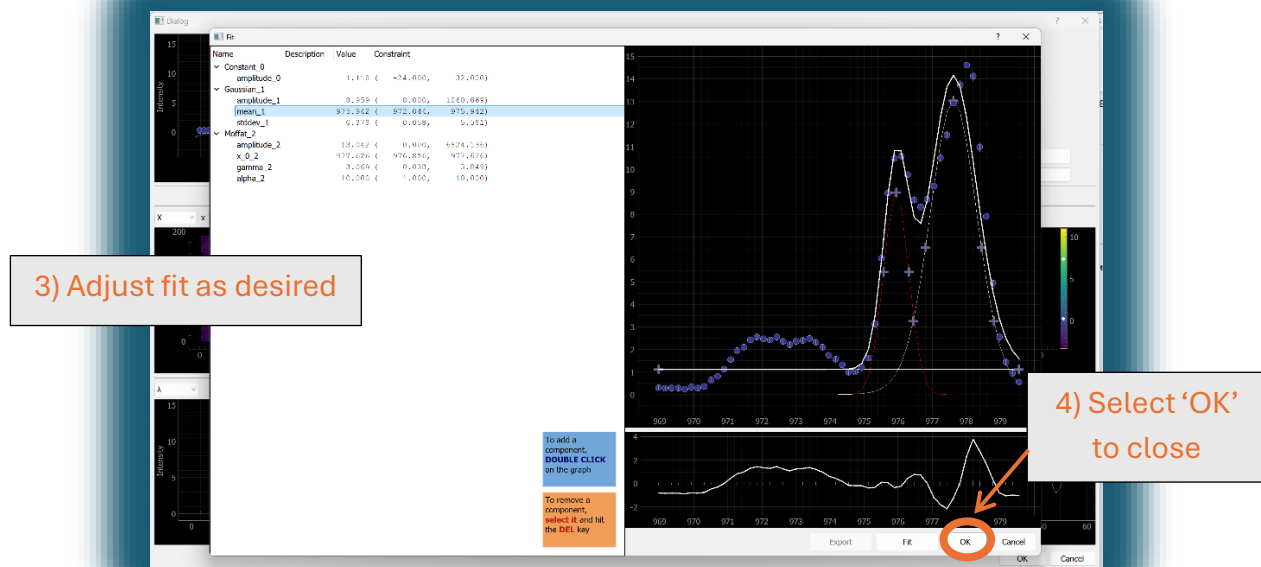


Figure 20

If a suitable fit cannot be found, selecting the **"Mask (X,Y)"** checkbox will remove this point from the returned results.

## 3.4 Storing and Recovering Results

### 3.4.1 The `get\_results` method

Use the `Grid` method `get_results` to get a dictionary of fit information and parameter values from the Object:

```
In []: res = GM.get_results()
In []: print(res.keys())

Out[]: dict_keys(['base_model', 'mask', 'fit_info', 'std_err', 'p_values',
'amplitude_0', 'amplitude_1', 'mean_1', 'stddev_1', 'amplitude_2', 'x_0_2',
'gamma_2', 'alpha_2'])
```

<code>base_model:</code>	The Astropy model used in the initial fit of all grid points
<code>mask:</code>	A Boolean array of shape <code>(N<sub>x</sub>, N<sub>y</sub>)</code> indicating <code>True</code> where a User-defined point has been removed from fitting
<code>fit_info:</code>	An Astropy <code>FitInfoArrayContainer</code> of shape <code>(N<sub>x</sub>, N<sub>y</sub>)</code> with covariance and other fit information generated by Astropy during the function fitting at each point.
<code>std_err:</code>	<p>A dictionary containing np.arrays of shape <code>(N<sub>x</sub>, N<sub>y</sub>)</code> for each model parameter indicating the standard error for the fitted value of the given parameter.</p> <p>E.G.</p> <pre>res['std_error']['mean_1'].shape = (N<sub>x</sub>, N<sub>y</sub>) res['std_error']['x_0_2'].shape = (N<sub>x</sub>, N<sub>y</sub>)</pre>
<code>p_values:</code>	An array of shape <code>(N<sub>x</sub>, N<sub>y</sub>)</code> indicating the goodness-of-fit. The p-values represent the probability of obtaining the observed chi-squared statistic (or larger) under the assumption that the fitted model adequately describes the data. Low p-values suggest poor model fit, while high p-values indicate the model is consistent with the observations within measurement uncertainties.
<code>&lt;parameter_name&gt;:</code> E.G. 'amplitude_2', 'stddev_1', etc.	For each parameter in the base model, an array with the parameter value after fitting. Shape <code>(N<sub>x</sub>, N<sub>y</sub>)</code> .

### 3.4.2 ASDF Files

The PyCFIT Grid Object can be stored in and restored from a file in Advanced Scientific Data Format ([ASDF](#)) via the “Import” and “Export” buttons in the Raster Fitting interface.

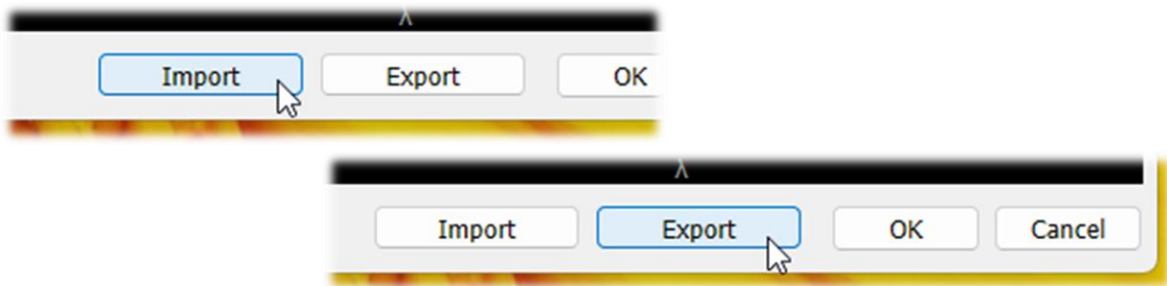


Figure 21

If you do not already have the graphical interface open, you can start one from a PyCFIT ASDF file by first loading the file at the command line and then starting the GUI from the restored Grid object. Use the utility function `load_asdf()` for this:

```
GM = pycfit.load_asdf('my_stored_raster_fit.asdf')
GM = pycfit.cfit_grid_gui(GM)
```

## 4 Interactive Help System

Both the Single-Spectrum-Fit GUI and the Grid-Spectra-Fit GUI provide an interactive help system to clarify the meaning and use for certain application elements. Activate this system the “?” button in the top-right of the interface window:

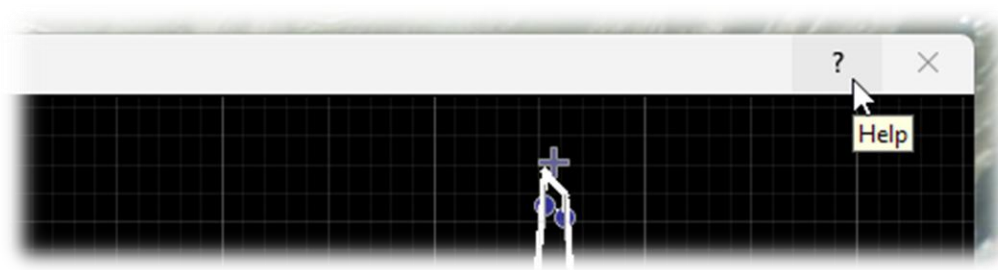


Figure 22

This will alter your mouse cursor to indicate whether a help message is available (Figures 23 - 24) for a given interface element or not.

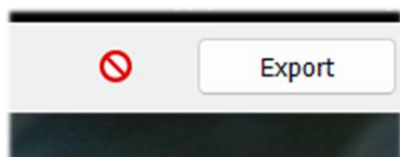


Figure 23: Not available

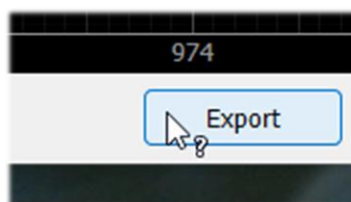


Figure 24: Available help

Clicking when the “available help” icon is showing will present a tooltip for the given element:

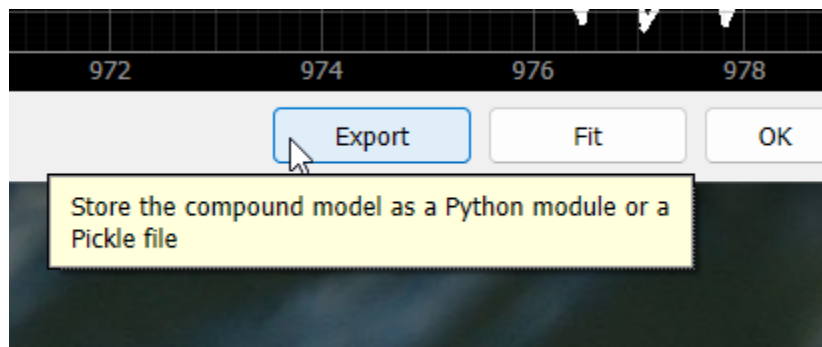


Figure 25

Alternately, the help message for most of the elements can also be accessed via their context menu under the “What’s This?” heading (Figure 26). On Windows, this is generally accessed via a mouse right-click.

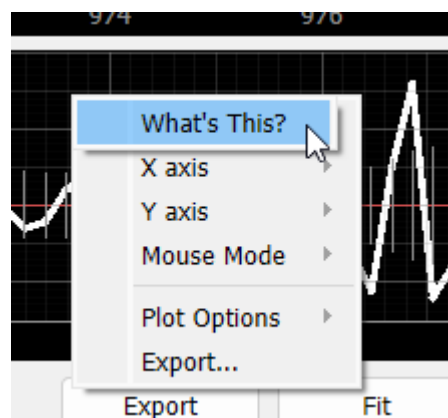


Figure 26



# Appendix A: Installation

## A.1 System Requirements

Windows, MacOS, or Linux system with a base Python version 3.11 or higher and `pip` package installer.

## A.2 Python Package Install

```
pip install pycfit
```

This will install all required dependencies, including Astropy  $\geq 7.1$ , dask, and PyQt5.

## Appendix B: Command-line Usage

### B.1 Overview

PyCFIT follows a Model-View design, keeping the data and actions available on the data encapsulated in objects entirely independent of the graphical interfaces to them. The Single-Spectrum-Fit GUI is a View on the internally defined `pycfit._function.FunctionFitter` object type (the *Model*) and directly invokes its methods. Similarly, the Grid-Spectra-Fit GUI is a View on the `pycfit._grid.Grid` object type. With this design paradigm, most of the functionality that can be performed via the graphical interfaces can also be accomplished entirely at the command line.

This section summarizes actions that can be performed with the data objects themselves, independent of the graphical interfaces.

### B.2 Single Spectrum Fitting Commands

To work at the command line for a single spectrum fitting, create the `FunctionFitter` object by invoking `cfit()`:

```
FF = pycfit.cfit(wavelength, intensity, uncertainty = uncert, function=model)
```

Or, create it without a model defined and add one later with the `update_model()` method:

```
FF = pycfit.cfit(wavelength, intensity, uncertainty = uncert)
FF.update_model(model)
```

Or, create the `FunctionFitter` then sequentially add Component models with the `add_component()` method to create a compound model:

```
FF = pycfit.cfit(wavelength, intensity, uncertainty = uncert)
FF.add_component('Constant')
FF.add_component('Gaussian')
FF.add_component('Moffat')
```

Using the `get_model()` method, retrieve and modify the parameters:

```
In []: M = FF.get_model()
In []: M
Out[]: <CompoundModel(amplitude_0=1., amplitude_1=1., mean_1=0., stddev_1=1.,
amplitude_2=1., x_0_2=0., gamma_2=1., alpha_2=1.)>

In []: M.mean_1 = 976
In []: M.x_0_2 = 978
In []: FF.update_model(M)
In []: FF.get_model()
Out[]: <CompoundModel(amplitude_0=1., amplitude_1=1., mean_1=976.,
stddev_1=1., amplitude_2=1., x_0_2=978., gamma_2=1., alpha_2=1.)>
```

Notice that each Parameter of the compound model has a trailing tag indicating the **ID** of its parent Component (`_0`, `_1`, etc.). So, for our example:

**amplitude\_0** is the single Parameter for the Constant Component,

**amplitude\_1** are the three Parameters associated with the Gaussian Component,  
**mean\_1**  
**stddev\_1**

and

**amplitude\_2** are the Parameters for the Moffat Component.  
**x\_0\_2**  
**gamma\_2**  
**alpha\_2**

These **IDs** correspond to the order in which each Component was added to the **FunctionFitter** object. In our example, the 'Constant' component has **ID=0**, the 'Gaussian' **ID=1**, and the 'Moffat' **ID=2**. Delete a Component by referencing its **ID**. To delete the Gaussian in our example:

```
In []: FF.delete_component(1)
In []: FF.get_model()

Out[]: <CompoundModel(amplitude_0=1., amplitude_1=1., x_0_1=978., gamma_1=1.,
alpha_1=1.)>
```

You'll notice that after a Component has been deleted, the ID numbers are reset to be contiguous. Now the 'Moffat' Component, formerly **ID = 2**, has **ID = 1**.

With the desired model initial conditions in place, use the `fit()` method to find the best-fit:

```
In []: FF.fit()
In []: FF.get_model()

Out[]: <CompoundModel(amplitude_0=0.34665268, amplitude_1=7.1515104,
x_0_1=977.13178844, gamma_1=0.92357454, alpha_1=3.68752081)>
```

Use the `get_chi_sq()` and `get_fit()` methods to retrieve information about the completed fit:

```
In []: FF.get_chi_sq()
Out[]: np.float64(167.42467387213438)
```

```
In []: FF.get_fit_info()
Out[]:
message: `ftol` termination condition is satisfied.
success: True
status: 2
  fun: [ 1.041e+00  1.327e+00 ...  1.181e+00  1.220e+00]
    x: [ 3.467e-01  7.152e+00  9.771e+02  9.236e-01  3.688e+00]
cost: 83.71233693606719
  jac: [[ 4.155e+00  3.496e-07 ...  1.972e-05 -1.104e-05]
        [ 5.860e+00  5.842e-07 ...  3.294e-05 -1.826e-05]
        ...
        [ 6.267e+00  1.904e-03 ...  9.662e-02 -2.990e-02]
        [ 6.237e+00  1.184e-03 ...  6.101e-02 -1.968e-02]]
  grad: [ 2.403e-05  3.098e-06  7.272e-05 -3.064e-03  2.766e-04]
optimality: 0.003064287750508654
active_mask: [0 0 0 0 0]
  nfev: 15
  njev: 14
param_cov: [[ 6.849e-04 -1.203e-03 ...  1.881e-03  1.320e-02]
            [-1.203e-03  5.250e-02 ... -2.659e-02 -1.502e-01]
            ...
            [ 1.881e-03 -2.659e-02 ...  4.678e-02  2.955e-01]
            [ 1.320e-02 -1.502e-01 ...  2.955e-01  1.894e+00]]
```

See the Astropy documentation for more information about the [TRFLSQFitter](#) and the SciPy documentation for further description of the [fit\\_info](#) returned.

## B.3 Raster Fitting Commands

Call the `cfits_grid()` function to create a `Grid` object without opening the GUI or fitting across the grid by default:

```
In []: wavelength.shape
Out[]: (60,)

In []: intensity.shape
Out[]: (60, 20, 20)

GM = pycfit.cfits_grid(model, wavelength, intensity, uncertainty)
```

The `Grid.shape` attribute describes the `(Nx, Ny)` dimensions of the `Grid`:

```
In []: GM.shape
Out[]: (20, 20)
```

Fit the entire grid using the currently set initial conditions using `.fit()` or remove all the fittings with `.clear_fit()`. Test if the `Grid` is in a fitted state at any time with the `is_fitted()` method:

```
In []: GM.fit()
Fitting across the grid. This may take a few minutes . . .

In []: GM.is_fitted()
Out[]: True

In []: GM.clear_fit()

In []: GM.is_fitted()
Out[]: False

In []: GM.fit()
Fitting across the grid. This may take a few minutes . . .

In []: GM.is_fitted()
Out[]: True
```

Use a 2-tuple `key` to view and modify initial conditions at any X,Y point in the `Grid`.

View:

```
In []: key = (4, 2)
In []: GM[key]
Out[]: <CompoundModel(amplitude_0=0.22542652, amplitude_1=2.30290582,
mean_1=972.67660905, stddev_1=0.36431365, amplitude_2=16.7709521,
x_0_2=977.18135174, gamma_2=1.51856316, alpha_2=10.)>
```

**Modify** the initial condition by accessing the value in the Parameter attribute. See the available parameter names via the `param_names` attribute:

```
In []: GM.param_names
Out[]:
['amplitude_0',
 'amplitude_1',
 'mean_1',
 'stddev_1',
 'amplitude_2',
 'x_0_2',
 'gamma_2',
 'alpha_2']

In []: GM.amplitude_0.value[key]
Out[]: np.float64(0.22542652393698281)

In []: GM.amplitude_0.value[key] = 0.5

In []: GM[key]
Out[]: <CompoundModel(amplitude_0=0.5, amplitude_1=2.30290582,
mean_1=972.67660905, stddev_1=0.36431365, amplitude_2=16.7709521,
x_0_2=977.18135174, gamma_2=1.51856316, alpha_2=10.)>
```

Re-fit any point in the `Grid` by passing the `key` to the `fit` method:

```
GM.fit(key)
```

The `get_results()` method, as described in [Retrieving Results](#) section, returns a dictionary with information from the fitted `Grid`:

```
In []: res = GM.get_results()

In []: res.keys()
Out[]: dict_keys(['base_model', 'mask', 'fit_info', 'std_err', 'p_values',
'amplitude_0', 'amplitude_1', 'mean_1', 'stddev_1', 'amplitude_2', 'x_0_2',
'gamma_2', 'alpha_2'])
```

Use the `save_asdf()` method to store the Grid object in an ASDF file for later retrieval and review. Restore the object with the `load_asdf()` function:

```
GM.save_asdf("my_stored_raster_fit.asdf")
GM = pycfit.load_asdf("my_stored_raster_fit.asdf")
GM = pycfit.cfit_grid_gui(GM)
```