

---

# GaussPy Documentation

*Release 1.0*

**Robert Lindner, Carlos Vera-Ciro, Claire Murray, Elijah Bernstein-C**

March 18, 2016

## CONTENTS

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Installation</b>	<b>3</b>
2.1	Dependencies . . . . .	3
2.2	Optional Dependencies . . . . .	3
2.3	Download GaussPy . . . . .	3
2.4	Installing Dependencies on Linux . . . . .	3
2.5	Installing Dependencies on OSX . . . . .	4
2.6	Installing GaussPy . . . . .	4
<b>3</b>	<b>Simple Example Tutorial</b>	<b>5</b>
3.1	Constructing a GaussPy-Friendly Dataset . . . . .	5
3.2	Running GaussPy . . . . .	5
3.3	Plot Decomposition Results . . . . .	7
<b>4</b>	<b>Multiple Gaussians Tutorial</b>	<b>9</b>
4.1	Constructing a GaussPy-Friendly Dataset . . . . .	9
4.2	Running GaussPy . . . . .	11
4.3	Plot Decomposition Results . . . . .	11
4.4	Creating a Synthetic Training Dataset . . . . .	12
4.5	Training the Algorithm . . . . .	14
4.6	Running GaussPy using Trained $\alpha$ . . . . .	15
<b>5</b>	<b>Two-Phase Decompositon</b>	<b>17</b>
5.1	Training for Two Phases: $\alpha_1$ and $\alpha_2$ . . . . .	17
<b>6</b>	<b>Prepping a Datacube</b>	<b>18</b>
<b>7</b>	<b>Behind the Scenes</b>	<b>19</b>
7.1	Basic concepts . . . . .	19
7.2	Dealing with noise . . . . .	20
7.3	Two phases . . . . .	22
<b>8</b>	<b>Indices and tables</b>	<b>23</b>

Contents:

## INTRODUCTION

When interpreting data, it is useful to fit models made up of parametrized functions. Gaussian functions, described simply by three parameters and often physically well-motivated, provide a convenient basis set of functions for such a model. However, any set of Gaussian functions is not an orthogonal basis, and therefore any solution implementing them will not be unique. Furthermore, determining fits to spectra involving more than one Gaussian function is complex, and the crucial step of guessing the number of functions and their parameters is not straightforward. Typically, reasonable fits can be determined iteratively and by-eye. However, for large sets of data, analysis by-eye requires an unreasonable amount of processing time and is unfeasible.

This document describes the installation and use of GaussPy, a code for implementing an algorithm called Autonomous Gaussian Decomposition (AGD). AGD uses computer vision and machine learning techniques to provide optimized initial guesses for the parameters of a multi-component Gaussian model automatically and efficiently. The speed and adaptability of AGD allow it to interpret large volumes of spectral data efficiently. Although it was initially designed for applications in radio astrophysics, AGD can be used to search for one-dimensional Gaussian (or any other single-peaked spectral profile)-shaped components in any data set.

To determine how many Gaussian functions to include in a model and what their parameters are, AGD uses a technique called derivative spectroscopy. The derivatives of a spectrum can efficiently identify shapes within that spectrum corresponding to the underlying model, including gradients, curvature and edges. The details of this method are described fully in [Lindner et al. \(2015\)](#), *AJ*, 149, 138.

## INSTALLATION

### 2.1 Dependencies

You will need the following packages to run Gausspy. We list the version of each package which we know to be compatible with Gausspy.

- `python 2.7`
- `numpy (v1.6.1)`
- `scipy (v0.17.0)`
- `lmfit (v0.9.2)`
- `h5py (v2.0.1)`

If you do not already have Python 2.7, you can install the [Anaconda Scientific Python distribution](#), which comes pre-loaded with *numpy*, *scipy*, and *h5py*.

### 2.2 Optional Dependencies

- `matplotlib (> v1.1.1)`
- GNU Scientific Library (GSL)

### 2.3 Download GaussPy

Download GaussPy from...

### 2.4 Installing Dependencies on Linux

You will need several libraries which the GSL, h5py, and scipy libraries depend on. Install these required packages with:

```
sudo apt-get install libblas-dev liblapack-dev gfortran libgsl0-dev libhdf5-serial-dev  
sudo apt-get install hdf5-tools
```

Install pip for easy installation of python packages:

```
sudo apt-get install python-pip
```

Then install the required python packages:

```
sudo pip install scipy numpy h5py lmfit
```

Install the optional dependencies for plotting and optimization:

```
sudo pip install matplotlib
sudo apt-get install libgsl0-dev
```

## 2.5 Installing Dependencies on OSX

Installation on OSX can be done easily with homebrew. Install pip for easy installation of python packages:

```
sudo easy_install pip
```

Then install the required python packages:

```
sudo pip install numpy scipy h5py lmfit
```

Install the optional dependencies for plotting and optimization:

```
sudo pip install matplotlib
sudo brew install gsl
```

## 2.6 Installing GaussPy

To install make sure that all dependences are already installed and properly linked to python –python has to be able to load them–. Then cd to the local directory containing gausspy and type

```
python setup.py install
```

If you don't have root access and/or wish a local installation of gausspy then use

```
python setup.py install --user
```

change the 'requires' statement in setup.py to include scipy and lmfit

## **SIMPLE EXAMPLE TUTORIAL**

### **3.1 Constructing a GaussPy-Friendly Dataset**

Before implementing AGD, we first must put data into a format readable by GaussPy. GaussPy requires the independent and dependent spectral arrays (e.g., channels and amplitude) and an estimate of the per-channel noise in the spectrum.

To begin, we can create a simple Gaussian function of the form:

$$S(x_i) = \sum_{k=1}^{\text{NCOMPS}} \text{AMP}_k \exp \left[ -\frac{4 \ln 2 (x_i - \text{MEAN}_k)^2}{\text{FWHM}_k^2} \right] + \text{NOISE}, \quad i = 1, \dots, \text{NCHANNELS} \quad (3.1)$$

where,

1. `NCOMPS` is the number of Gaussian components in each spectrum
2. `(AMP, MEAN, FWHM)` are the parameters of each Gaussian component
3. `NCHANNELS` is the number of channels in the spectrum (sets the resolution)
4. `NOISE` is the level of noise introduced in each spectrum, described by the root mean square (RMS) noise per channel.

In the next example we will show how to implement this in python. We have made the following assumptions:

1. `NCOMPS = 1` (to begin with a simple, single Gaussian)
2. `AMP = 1.0, MEAN = 256, FWHM = 20` (fixed Gaussian parameters)
3. `NCHANNELS = 512`
4. `RMS = 0.05`

The following code describes an example of how to create spectrum with Gaussian shape and store the channels, amplitude and error arrays in a python pickle file to be read later by GaussPy.

In Fig. 3.1 we display the simple Gaussian spectrum constructed above.

### **3.2 Running GaussPy**

With our simple dataset in hand, we can use GaussPy to decompose the spectrum into Gaussian functions. To do this, we must specify the smoothing parameter  $\alpha$ . For now, we will guess a value of  $\alpha = 10$ . Later in this chapter we will learn about training AGD to select the optimal value of  $\alpha$ .

The following is an example code for running GaussPy. We will use the “one-phase” decomposition to begin with. We must specify the following parameters:

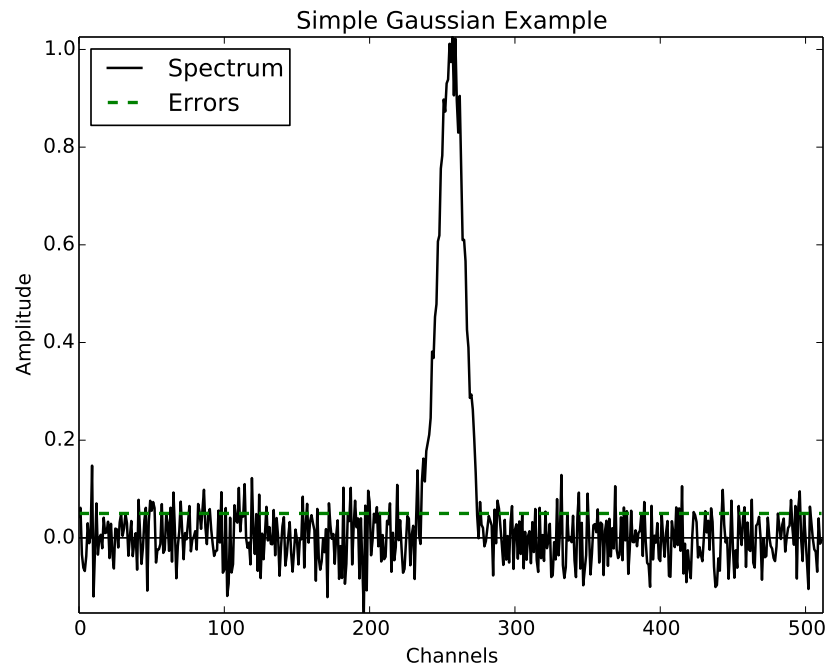


Fig. 3.1: Example spectrum containing a single Gaussian function with added spectral noise.

1. `alpha1`: our choice for the value of  $\alpha$ .
2. `snr_thresh`: the signal-to-noise ratio threshold below which amplitude GaussPy will not fit a component.
3. `DATA`: the filename containing the dataset to-be-decomposed, constructed in the previous section (or any GaussPy-friendly dataset)
4. `DATA_out`: filename to store the decomposition results from GaussPy.

```
# Decompose simple dataset using AGD
import pickle
import gausspy.gp as gp

# Specify necessary parameters
alpha1 = 10.
snr_thresh = 5.
DATA = 'simple_gaussian.pickle'
DATA_out = 'simple_gaussian_decomposed.pickle'

# Load GaussPy
g = gp.GaussianDecomposer()

# Setting AGD parameters
g.set('phase', 'one')
g.set('SNR_thresh', [snr_thresh, snr_thresh])
g.set('alpha1', alpha1)
g.set('mode', 'conv')

# Run GaussPy
decomposed_data = g.batch_decomposition(DATA)
```



```
# Save decomposition information
pickle.dump(decomposed_data, open(DATA_out, 'w'))
```

After AGD determines the Gaussian decomposition, GaussPy then performs a least squares fit of the initial AGD model to the data to produce a final fit solution. The file containing the fit results is a python pickle file. The contents of this file can be viewed by printing the keys within the saved dictionary via,

```
print decomposed_data.keys()
```

The most salient information included in this file are the values for the `amplitudes`, `fwhms` and `means` of each fitted Gaussian component. These include,

1. `amplitudes_initial`, `fwhms_initial`, `means_initial` : the parameters of each Gaussian component determined by AGD (each array has length equal to the number of fitted components).
2. `amplitudes_fit`, `fwhms_fit`, `means_fit` : the parameters of each Gaussian component following a least-squares fit of the initial AGD model to the data.
3. `amplitudes_fit_err`, `fwhms_fit_err`, `means_fit_err` : uncertainties in the fitted Gaussian parameters, determined from the least-squares fit.

GaussPy also stores the reduced  $\chi^2$  value from the least-squares fit (`rchi2`), but this is currently under construction. This value can be computed outside of GaussPy easily.

### 3.3 Plot Decomposition Results

The following is an example python script for plotting the original spectrum and GaussPy decomposition results. We must specify the following parameters:

1. `DATA`: the filename containing the dataset to-be-decomposed.
2. `DATA_decomposed`: the filename containing the GaussPy decomposition results.

```
# Plot GaussPy results
import numpy as np
import matplotlib.pyplot as plt
import pickle

def gaussian(amp, fwhm, mean):
    return lambda x: amp * np.exp(-(x-mean)**2/2./ (fwhm/2.355)**2)

def unravel(list):
    return np.array([i for array in list for i in array])

DATA = 'simple_gaussian.pickle'
DATA_decomposed = 'simple_gaussian_decomposed.pickle'

data = pickle.load(open(DATA))
spectrum = unravel(data['data_list'])
chan = unravel(data['x_values'])
errors = unravel(data['errors'])

decomposed_data = pickle.load(open(DATA_decomposed))
means_fit = unravel(decomposed_data['means_fit'])
amps_fit = unravel(decomposed_data['amplitudes_fit'])
fwhms_fit = unravel(decomposed_data['fwhms_fit'])

fig = plt.figure()
```

```

ax = fig.add_subplot(111)

model = np.zeros(len(chan))

for j in range(len(means_fit)):
    component = gaussian(amps_fit[j], fwhms_fit[j], means_fit[j])(chan)
    model += component
    ax.plot(chan, component, color='red', lw=1.5)

ax.plot(chan, spectrum, label='Data', color='black', linewidth=1.5)
ax.plot(chan, model, label = r'$\alpha=10.$', color='purple', linewidth=2.)
ax.plot(chan, errors, label = 'Errors', color='green', linestyle='dashed', linewidth=2.)

ax.set_xlabel('Channels')
ax.set_ylabel('Amplitude')

ax.set_xlim(0, len(chan))
ax.set_ylim(np.min(spectrum), np.max(spectrum))
ax.legend(loc=2)

plt.show()

```

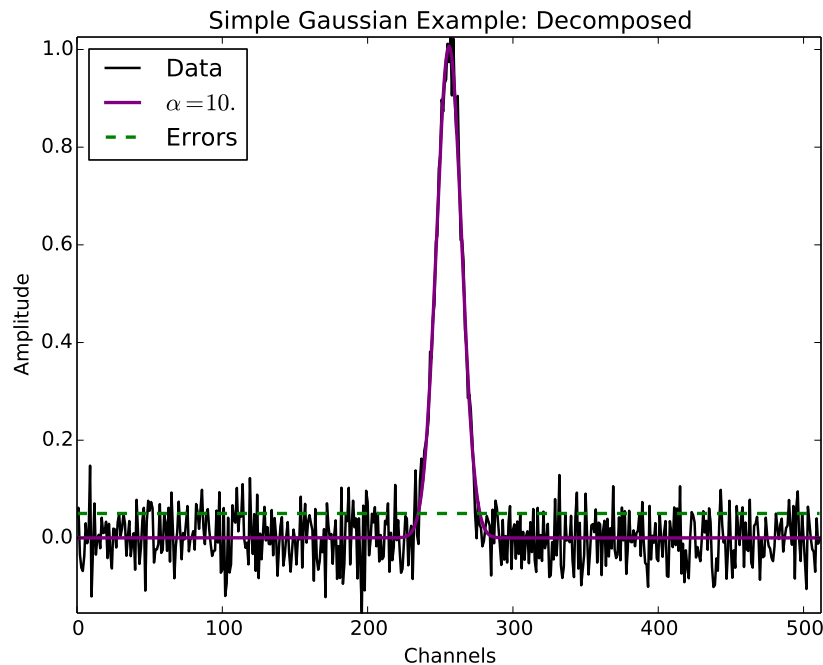


Fig. 3.2: Example spectrum containing a single Gaussian function with added spectral noise, decomposed using GaussPy.

Fig. 3.2 displays the results of the decomposition using the above example python code. Clearly the fit to the simple Gaussian spectrum is good. If we were to vary the value of  $\alpha$ , the fit would not change significantly as the fit to a spectrum containing a single Gaussian function does not depend sensitively on the initial guesses, especially because GaussPy performs a least-squares fit after determining initial guesses for the fitted Gaussian parameters with AGD.

We can now move on from the simple example above to vary the complexity of the spectra to be decomposed, as well as the effect of different values of  $\alpha$  on the decomposition.

## MULTIPLE GAUSSIANS TUTORIAL

### 4.1 Constructing a GaussPy-Friendly Dataset

As discussed in the *Simple Example Tutorial*, before running GaussPy we must ensure that our data is in a format readable by GaussPy. In particular, for each spectrum, we need to provide the independent and dependent spectral arrays (i.e. channels and amplitudes) and an estimate of the uncertainty per channel. In the following example we will construct a spectrum containing multiple overlapping Gaussian components with added spectral noise, using Equation (3.1), and plot the results.

We will make the following choices for parameters in this example:

1. NCOMPS = 3 : to include 3 Gaussian functions in the spectrum
2. AMPS = [3, 2, 1] : amplitudes of the included Gaussian functions
3. FWHMS = [10, 20, 30] : FWHM (in channels) of the included Gaussian functions
4. MEANS = [10, 20, 30] : mean positions (in channels) of the included Gaussian functions
5. NCHANNELS = 512 : number of channels in the spectrum
6. RMS = 0.05 : RMS noise per channel
7. filename : name of file to write output data to

The following code provides an example of how to construct a Gaussian function with the above parameters and store it in GaussPy-friendly format.

```
# Create profile with multiple, blended Gaussians and added noise
# Store in format required for GaussPy

import numpy as np
import pickle

def gaussian(amp, fwhm, mean):
    return lambda x: amp * np.exp(-(x-mean)**2/4./ (fwhm/2.355)**2)

# Specify filename of output data
FILENAME = 'multiple_gaussians.pickle'

# Number of Gaussian functions per spectrum
NCOMPS = 3

# Component properties
AMPS = [3, 2, 1]
FWHMS = [15, 50, 30] # channels
MEANS = [210, 250, 310] # channels
```

```

# Data properties
RMS = 0.05
NCHANNELS = 512

# Initialize
gausspy_data = {}
chan = np.arange(NCHANNELS)
errors = np.ones(NCHANNELS) * RMS

spectrum = np.random.randn(NCHANNELS) * RMS

# Create spectrum
for a, w, m in zip(AMPS, FWHMS, MEANS):
    spectrum += gaussian(a, w, m)(chan)

# Enter results into AGD dataset
gausspy_data['data_list'] = gausspy_data.get('data_list', []) + [spectrum]
gausspy_data['x_values'] = gausspy_data.get('x_values', []) + [chan]
gausspy_data['errors'] = gausspy_data.get('errors', []) + [errors]

pickle.dump(gausspy_data, open(FILENAME, 'w'))

```

A plot of the spectrum constructed above is included in Fig. 4.1.

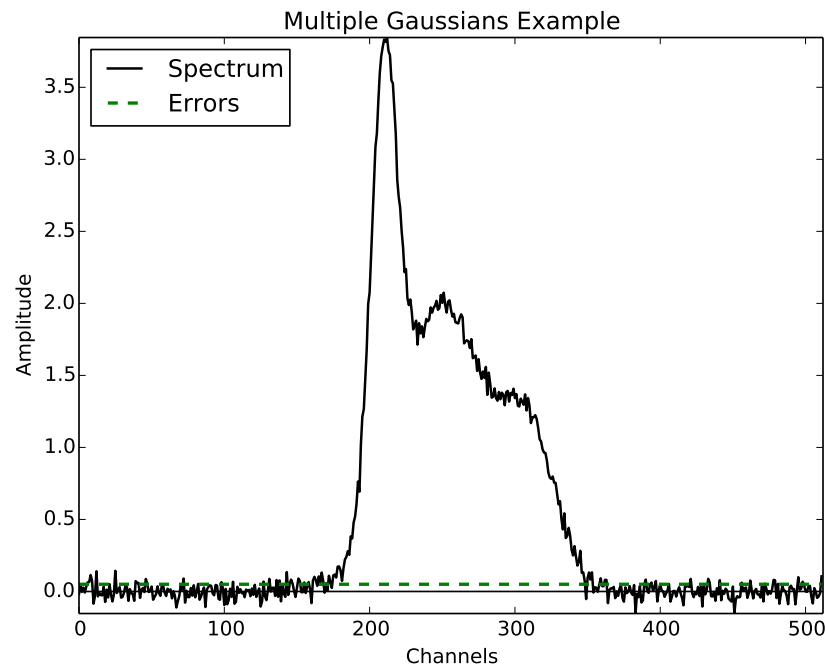


Fig. 4.1: Example spectrum containing multiple Gaussian functions with added spectral noise.

## 4.2 Running GaussPy

With our GaussPy-friendly dataset, we can now run GaussPy. As in the *Simple Example Tutorial*, we begin by selecting a value of  $\alpha$  to use in the decomposition. In this case we will select  $\alpha = 20$  to begin with. As before, the important parameters to specify are:

1. `alpha1`: our choice for the value of  $\alpha$ .
2. `snr_thresh`: the signal-to-noise ratio threshold below which amplitude GaussPy will not fit a component.
3. `DATA`: the filename containing the dataset to-be-decomposed, constructed above (or any GaussPy-friendly dataset)
4. `DATA_out`: filename to store the decomposition results from GaussPy.

```
# Decompose multiple Gaussian dataset using AGD
import pickle
import gausspy.gp as gp

# Specify necessary parameters
alpha1 = 20.
snr_thresh = 5.
DATA = 'multiple_gaussians.pickle'
DATA_out = 'multiple_gaussians_decomposed.pickle'

# Load GaussPy
g = gp.GaussianComposer()

# Setting AGD parameters
g.set('phase', 'one')
g.set('SNR_thresh', [snr_thresh, snr_thresh])
g.set('alpha1', alpha1)
g.set('mode', 'conv')

# Run GaussPy
decomposed_data = g.batch_decomposition(DATA)

# Save decomposition information
pickle.dump(decomposed_data, open(DATA_out, 'w'))
```

## 4.3 Plot Decomposition Results

Following the decomposition by GaussPy, we can explore the effect of the choice of  $\alpha$  on the decomposition. In Fig. 4.2, we have run GaussPy on the multiple-Gaussian dataset constructed above for three values of  $\alpha$ , including  $\alpha = 20$ ,  $\alpha = 2$  and  $\alpha = 10$  and plotted the results.

These results demonstrate that our choice of  $\alpha$  has a significant effect on the success of the GaussPy model. In order to select the right value of  $\alpha$  for a given dataset, we need to train the AGD algorithm using a training set. This process is described in the following section.

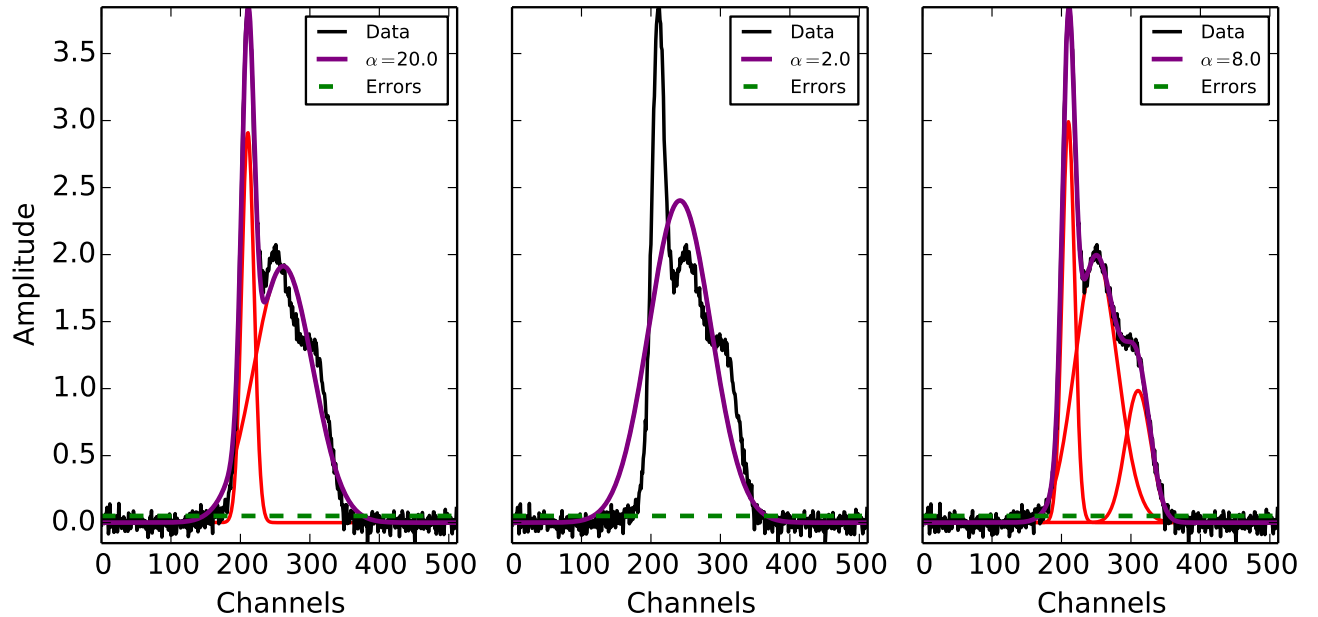


Fig. 4.2: Example spectrum containing multiple Gaussian functions with added spectral noise, decomposed using GaussPy for three values of the smoothing parameter  $\alpha$ .

#### 4.3.1 <<<<<< HEAD

#### 4.3.2 Training AGD to select Alpha

#### 4.3.3 Training AGD to select $\alpha$

```
>>>>>>> e9b9f3ecc1505f858a999f72e6a46e85f56fe223
```

### 4.4 Creating a Synthetic Training Dataset

To select the optimal value of the smoothing parameter  $\alpha$ , you must train the AGD algorithm using a training dataset with known underlying Gaussian decomposition. In other words, you need to have a dataset for which you know (or have an estimate of) the true Gaussian model. This training dataset can be composed of real (i.e. previously analyzed) or synthetically-constructed data, for which you have prior information about the underlying decomposition. This prior information is used to maximize the model accuracy by calibrating the  $\alpha$  parameter used by AGD.

Training datasets can be constructed by adding Gaussian functions with parameters drawn from known distributions with known uncertainties. For example, we can create a mock dataset with NSPECTRA-realizations of Equation (3.1).

In the next example we will show how to implement this in python. For this example we will construct a synthetic training dataset with parameters similar to those found in the [Multiple Gaussians Tutorial](#) example. We must set the following parameters:

1. NOISE  $\sim N(0, \text{RMS}) + f \times \text{RMS}$  with  $\text{RMS}=0.05$  and  $f=0$
2. NCOMPS = 3
3. NCHANNELS = 512 This number sets the resolution of each spectrum. **Does this number need to be the same for all spectra in AGD?**

4.  $\text{AMP} \sim \mu(0.5, 4)$ , this way we ensure that every spectral feature is above the noise level. Spectra with a more dominant contribution from the noise can also be generated and used as training sets for AGD
5.  $\text{FWHM} \sim \mu(10, 80)$  and  $\text{MEAN} \sim \mu(0.25, 0.75) \times \text{NCHANNELS}$ , note that for our choice of the number of channels, this selection of FWHM ensures that even the wider component can be fit within the spectrum.
6. `TRAINING_SET : True`, determines whether the decomposition “true answers” are stored along with the synthetic spectra for accuracy verification in training.
7. `FILENAME` : filename for storing the synthetically-constructed data

```
# Create training dataset with Gaussian profiles

import numpy as np
import pickle

# Specify the number of spectral channels (NCHANNELS)
NCHANNELS = 512

# Specify the number of spectra (NSPECTRA)
NSPECTRA = 200

# Estimate of the root-mean-square uncertainty per channel (RMS)
RMS = 0.05

# Estimate the number of components
NCOMPS = 3

# Specify the min-max range of possible properties of the Gaussian function parameters:
AMP_lims = [0.5, 4]
FWHM_lims = [10, 80] # channels
MEAN_lims = [0.25*NCHANNELS, 0.75*NCHANNELS] # channels

# Indicate whether the data created here will be used as a training set
# (a.k.a. decide to store the "true" answers or not at the end)
TRAINING_SET = True

# Specify the pickle file to store the results in
FILENAME = 'training_data.pickle'
```

With the above parameters specified, we can proceed with constructing a set of synthetic training data composed of Gaussian functions with known parameters (i.e., for which we know the “true” decomposition), sampled randomly from the parameter ranges specified above. The resulting data, including the channel values, spectral values and error estimates, are stored in the pickle file specified above with `FILENAME`. Because we want this to be a training set (`TRAINING_SET = True`), the true decomposition answers (in the form of amplitudes, FWHM and means for all components) are also stored in the output file. For example, to construct a synthetic dataset:

```
# Create training dataset with Gaussian profiles -cont-

# Initialize
gausspy_data = {}
chan = np.arange(NCHANNELS)
errors = np.ones(NCHANNELS) * RMS

# Begin populating data
for i in range(NSPECTRA):
    spectrum_i = np.random.randn(NCHANNELS) * RMS

    amps = []
    fwhms = []
```

```

means = []

for comp in range(ncomps):
    # Select random values for components within specified ranges
    a = np.random.uniform(AMP_lims[0], AMP_lims[1])
    w = np.random.uniform(FWHM_lims[0], FWHM_lims[1])
    m = np.random.uniform(MEAN_lims[0], MEAN_lims[1])

    # Add Gaussian profile with the above random parameters to the spectrum
    spectrum_i += gaussian(a, w, m)(chan)

    # Append the parameters to initialized lists for storing
    amps.append(a)
    fwms.append(w)
    means.append(m)

# Enter results into AGD dataset
gausspy_data['data_list'] = gausspy_data.get('data_list', []) + [spectrum_i]
gausspy_data['x_values'] = gausspy_data.get('x_values', []) + [chan]
gausspy_data['errors'] = gausspy_data.get('errors', []) + [errors]

# If training data, keep answers
if TRAINING_SET:
    gausspy_data['amplitudes'] = gausspy_data.get('amplitudes', []) + [amps]
    gausspy_data['fwhms'] = gausspy_data.get('fwhms', []) + [fwms]
    gausspy_data['means'] = gausspy_data.get('means', []) + [means]

# Dump synthetic data into specified filename
pickle.dump(gausspy_data, open(FILENAME, 'w'))

```

## 4.5 Training the Algorithm

Next, we will apply GaussPy to the real or synthetic training dataset and compare the results with the known underlying decomposition to determine the optimal value for the smoothing parameter  $\alpha$ . We must set the following parameters

1. `FILENAME`: the filename of the training dataset in GaussPy-friendly format.
2. `snr_thresh`: the signal-to-noise threshold below which amplitude GaussPy will not fit components.
3. `alpha_i`: initial choice for  $\alpha$

```

# Select the optimal value of alpha by training the AGD algorithm

import gausspy.gp as gp

# Set necessary parameters
FILENAME = 'training_data.pickle'
snr_thresh = 5.
alpha_i = 20.

g = gp.GaussianDecomposer()

# Next, load the training dataset for analysis:
g.load_training_data(FILENAME)

# Set GaussPy parameters
g.set('phase', 'one')

```



```
g.set('SNR_thresh', [snr_thresh, snr_thresh])
g.set('mode', 'conv')

# Train AGD starting with initial guess for alpha
g.train(alpha_initial = alpha_i, plot=False,
        verbose = False, mode = 'conv',
        learning_rate = 1.0, eps = 1.0, MAD = 0.1)
```

GaussPy will decompose the training dataset with the initial choice of  $\alpha_i$  and compare the results with the known underlying decomposition to compute the accuracy of the decomposition. The training process will then iteratively change the value of  $\alpha_i$  and recompute the decomposition until the process converges. Convergence is achieved when the reduced  $\chi^2$  is less than 0.03 for at least 10 iterations. The accuracy of the decomposition associated with the converged value of  $\alpha$  is a description of how well GaussPy can recover the true underlying decomposition.

The above training dataset parameters were selected with the “Multiple Gaussians” example in mind. As we saw in Chapter 4, the choice of  $\alpha$  has a significant effect on the GaussPy decomposition. In the training example above, when we choose an initial value of  $\alpha_i = 20$  the training process converges to  $\alpha = 6.8$  with an accuracy of 73.1%, and required 355 iterations.

To ensure that the training converges on the optimal value of  $\alpha$  and not a local maximum, it is useful to re-run the training process for several initial choices of  $\alpha$ . When we run the above example with an initial choice of  $\alpha_i = 2$ , AGD converges to a value of  $\alpha = 6.84$  with an accuracy of 73.4% and required 97 iterations. For  $\alpha_i = 7$ , the training converges to  $\alpha = 6.8$  with an accuracy of 73.1% following 56 iterations. (results will vary very slightly for each test of the above code, given the random selection of component parameters in the training dataset).

## 4.6 Running GaussPy using Trained $\alpha$

With the trained value of  $\alpha$  in hand, we can proceed to decompose our target dataset with AGD. In this example, we will return to the example from the *Multiple Gaussians Tutorial* chapter. Following training, we select a value of  $\alpha = 6.8$ , which decomposed our training dataset with an accuracy of ~73%. As in the *Simple Example Tutorial* and *Multiple Gaussians Tutorial*, the important parameters to specify are:

1. `alpha1`: our choice for the value of  $\alpha$ .
2. `snr_thresh`: the signal-to-noise ratio threshold below which amplitude GaussPy will not fit a component.
3. `DATA`: the filename containing the dataset to-be-decomposed, constructed above (or any GaussPy-friendly dataset)
4. `DATA_out`: filename to store the decomposition results from GaussPy.

```
# Decompose multiple Gaussian dataset using AGD with TRAINED alpha
import pickle
import gausspy.gp as gp

# Specify necessary parameters
alpha1 = 6.8
snr_thresh = 5.

DATA = 'multiple_gaussians.pickle'
DATA_out = 'multiple_gaussians_trained_decomposed.pickle'

# Load GaussPy
g = gp.GaussianDecomposer()

# Setting AGD parameters
g.set('phase', 'one')
```

```

g.set('SNR_thresh', [snr_thresh, snr_thresh])
g.set('alpha1', alpha1)
g.set('mode', 'conv')

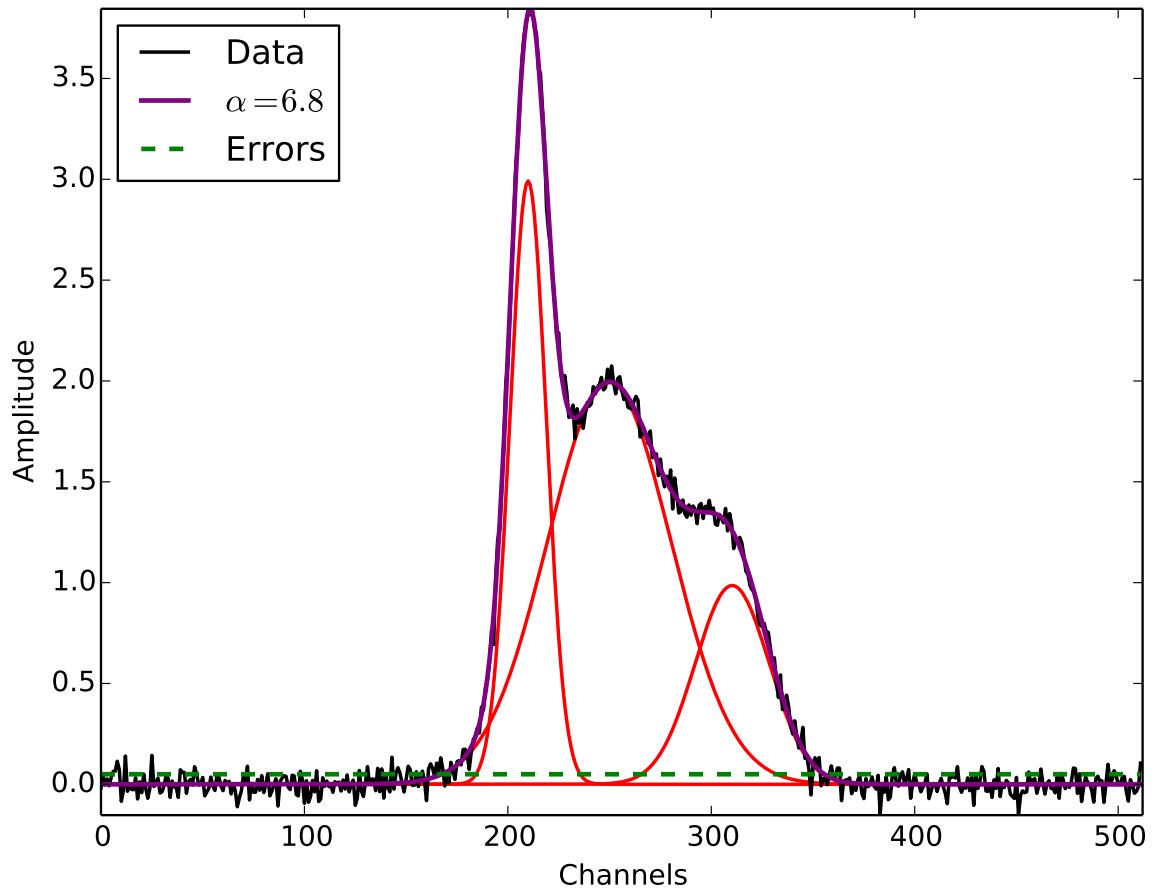
# Run GaussPy
decomposed_data = g.batch_decomposition(DATA)

```

#### 4.6.1 <<<<<< HEAD

#### 4.6.2 Running AGD using Trained Alpha

Fig. 4.6.2 displays the result of fitting the “Multiple Gaussians” spectrum with a trained value of  $\alpha = 6.8$ .



## TWO-PHASE DECOMPOSITION

>>>>>>> e9b9f3ecc1505f858a999f72e6a46e85f56fe223

In the <<<<<<< *HEAD* chapter, we learned how to “train” AGD to select the optimal value of the smoothing parameter  $\alpha$  using a training dataset with known underlying decomposition. This trained value is essentially tuned to find a particular type of Gaussian shape within the data. However, when more than one family or phase of Gaussian shapes is contained within a spectrum, one value of  $\alpha$  is not enough to recover all important spectral information. For example, in radio astronomical observations of absorption by neutral hydrogen at 21 cm, we find narrow and strong lines in addition to wide, shallow lines indicative of two different populations of material, namely the cold and warm neutral media.

For GaussPy to be sensitive to two types of Gaussian functions contained within a dataset, we must use the “two-phase” version of AGD. The two-phase decomposition makes use of two values of the smoothing parameter  $\alpha$ , one for each “phase” contained within the dataset.

### 5.1 Training for Two Phases: $\alpha_1$ and $\alpha_2$

## PREPPING A DATACUBE

In this example we will download a datacube to decompose into individual spectra.

## BEHIND THE SCENES

### 7.1 Basic concepts

`GaussPy` is a Python implementation of the AGD algorithm described in [Lindner et al. \(2015\)](#), AJ, 149, 138. At its core, AGD is a fast, automatic, extremely versatile way of providing initial guesses for fitting Gaussian components to a function of the form  $f(x) + n(x)$ , where  $n(x)$  is a term modeling possible contributions from noise. It is important to emphasize here that although we use terminology coming from radio-astronomy all the ideas upon which the code is founded can be applied to any function of this form, moreover, non-Gaussian components can also be in principle extracted with our methodology, something we will include in a future release of the code.

Ideally, if blending of components was not an issue and  $n(x) = 0$  the task of fitting Gaussians to a given spectrum would be reduced to find local maxima of  $f(x)$ . However, both of these assumptions dramatically fail in practical applications, where blending of lines is an unavoidable issue and noise is intrinsic to the process of data acquisition. In that case, looking for solutions of  $df(x)/dx = 0$  is not longer a viable route to find local extrema of  $f(x)$ , instead a different approach must be taken.

AGD uses the fact that a local maximum in  $f(x)$  is also a local minimum in the curvature. That is, the algorithm looks for points  $x^*$  for which the following conditions are satisfied.

- The function  $f(x)$  has a non-trivial value

$$f(x^*) > \epsilon_0. \quad (7.1)$$

In an ideal situation where the contribution from noise vanishes we can take  $\epsilon_0 = 0$ . However, when random fluctuations are added to the target function, this condition needs to be modified accordingly. A good selection of  $\epsilon_0$  thus needs to be in proportion to the RMS of the analyzed signal.

- Next we require that the function  $f(x)$  has a “bump” in  $x^*$

$$\left. \frac{d^2 f}{dx^2} \right|_{x=x^*} < 0, \quad (7.2)$$

this selection of the inequality ensures also that such feature has negative curvature, or equivalently, that the point  $x^*$  is candidate for being the position of a local maximum of  $f(x)$ . Note however that this is not a sufficient condition, we also need to ensure that the curvature has a minimum at this location.

- This is achieved by imposing two additional constraints on  $f(x)$

$$\left. \frac{d^3 f}{dx^3} \right|_{x=x^*} = 0 \quad (7.3)$$

$$\left. \frac{d^4 f}{dx^4} \right|_{x=x^*} > 0 \quad (7.4)$$

These 4 constraints then ensure that the point  $x^*$  is a local minimum of the curvature. Furthermore, even in the presence of both blending and noise, these expressions will yield the location of all the points that are possible candidates for the positions of Gaussian components in the target function. Fig. 7.1 is an example of a function defined as the sum of three gaussians for which the conditions Eq. (7.1) - (7.4) are satisfied and the local minima of curvature are successfully found, even when blending of components is relevant.

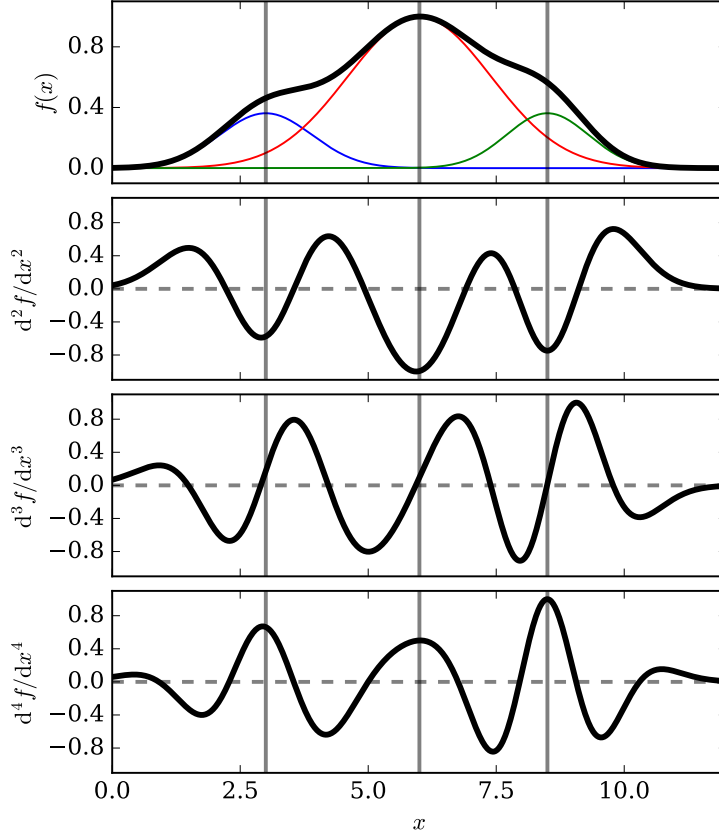


Fig. 7.1: Example of the points of negative curvature of the function  $f(x)$ . In this case  $f(x)$  is the sum of three independent Gaussian functions (top). The vertical lines in each panel show the conditions imposed on the derivatives to define the points  $x^*$ .

## 7.2 Dealing with noise

The numeral problem related to the solution shown in the previous section comes from the fact that calculating Eq. (7.2) - (7.4) is not trivial in the presence of noise. For instance, if the top panel of Fig. 7.1 is sampled with 100 channels, and in each panel a random uncorrelated noise component is added at the 10% level, a simple finite difference prescription to calculate the derivative would lead to variations of the order  $\sim 1/dx \sim 10$ . That is, the signal would be buried within the noise!

In order to solve this problem AGD uses a regularized version of the derivative (Vogel (2002)). If  $u = df(x)/dx$ , then the problem we solve is  $u = \arg \min_u \{R[u]\}$  where  $R[u]$  is the functional defined by

$$R[u] = \int |Au - f| + \alpha \int \sqrt{(Du)^2 + \beta^2}, \quad (7.5)$$

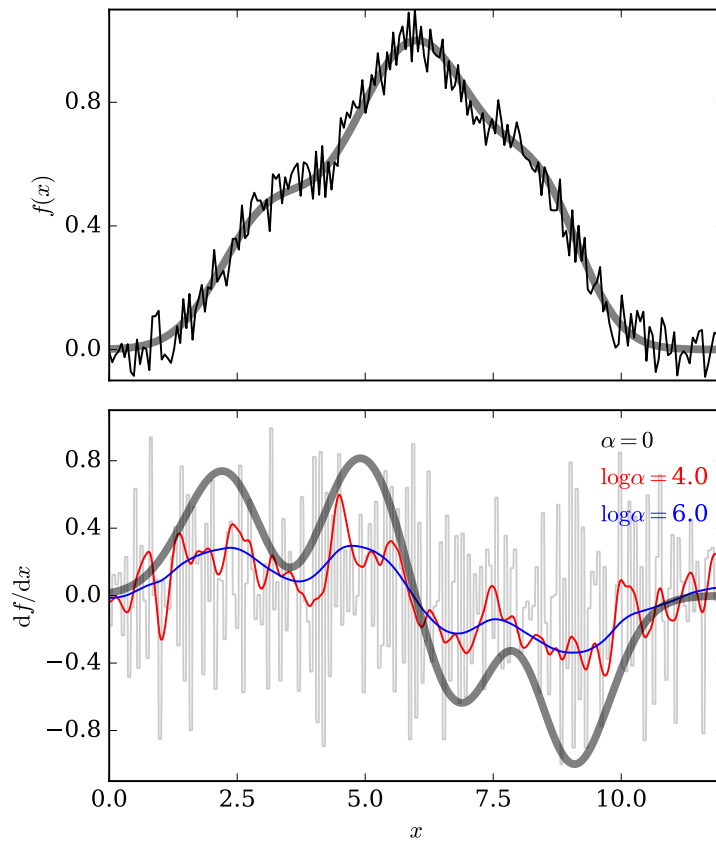


Fig. 7.2: The top panel shows the same function used in Fig. 7.1 but now random noise has been added to each channel. In the bottom panel we show various estimates of the first derivative.  $\alpha = 0$  corresponds to the finite differences method, larger values of  $\alpha$  makes the function smoother.

where  $Au = \int dx \, u$ . Note that if  $\alpha = 0$  this is equivalent to find the derivative of the function  $f(x)$ , since we will be minimizing the difference between the integral of  $u = df(x)/dx$  and  $f(x)$  itself. This, however, has the problem we discussed in the previous paragraph. Fig. 7.2 shows this case, it is clear that this simple approach fails to recover the behavior of the target function. If, on the other hand,  $\alpha > 0$  an additional weight is added to the inverse problem in Eq. (7.5), now the differences between successive points in  $u(x)$  are taken into account.

The parameter  $\alpha$  then controls how smooth the derivative is going to be. The risk here is that overshooting the value of this number can erase the intrinsic variations of the actual derivative. What is the optimal value of  $\alpha$ ? This question is answered by GaussPy through the training process of the algorithm. We refer the reader to the example chapters to learn how to use this feature.

## 7.3 Two phases

Within GaussPy is built-in the ability to automatically choose the best value of  $\alpha$  for any input data set. Special caution has to be taken here. If a component is too narrow it can be confused with noise and smoothed away by the algorithm!

In order to circumvent this issue GaussPy can be trained in “two-steps”. One for narrow components, and one for broad components. The result then is two independent values  $\alpha_1$  and  $\alpha_2$  each giving information about the scales of different features in the target function.



## INDICES AND TABLES

- `genindex`
- `modindex`
- `search`