GaussPy Documentation

Release 1.0

R. Lindner, C. Vera-Ciro, C. Murray, E. Bernstein-Cooper

February 28, 2017

CHAPTER

ONE

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 using 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.

CHAPTER

TWO

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 (v1.12.1)
- scipy (v0.17.0)
- lmfit (v0.9.3)
- 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

If you wish to use GaussPy's plotting capabilities you will need to install *matplotlib*:

- matplotlib (> v1.1.1)

If you wish to use optimization with Fortran code you will need

- GNU Scientific Library (GSL)

2.3 Download GaussPy

Download GaussPy using git $ git clone git://github.com/gausspy/gausspy.git

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 install via

```
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*.

CHAPTER

THREE

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 indepenent and dependent spectral arrays (e.g., channels and amplitude) and an estimate of the per-channel noise in the specrum.

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

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

where,

1. `NCOMPS` is the number of Gaussian components in each spectrum.
2. `(AMP, MEAN, FWHM)` are the amplitude, mean location, and full-width-half-maximum 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`

In Fig. 3.1 we display the spectrum with the single Gaussian described above.

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

```
# Create simple Gaussian profile with added noise
# Store in format required for GaussPy

import numpy as np
import pickle

# create a function which returns the values of the Gaussian function for a
# given x
def gaussian(amp, fwhm, mean):
    return lambda x: amp * np.exp(-4. * np.log(2) * (x-mean)**2 / fwhm**2)
```

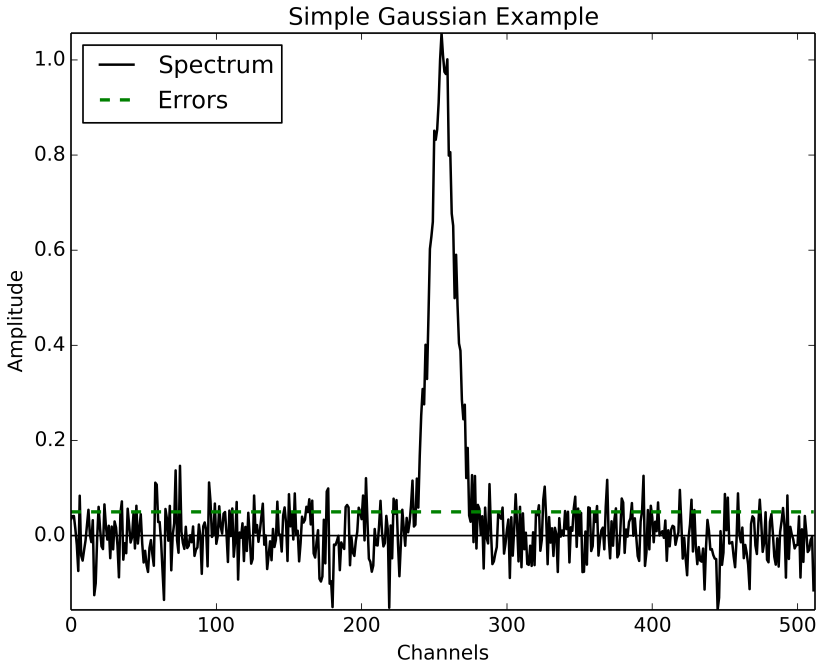


Fig. 3.1: Example spectrum containing a single Gaussian function with added spectral noise.

```
# Data properties
RMS = 0.05
NCHANNELS = 512
FILENAME = 'simple_gaussian.pickle'

# Component properties
AMP = 1.0
FWHM = 20
MEAN = 256

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

spectrum = np.random.randn(NCHANNELS) * RMS
spectrum += gaussian(AMP, FWHM, MEAN)(chan)

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

pickle.dump(data, open(FILENAME, 'w'))
```

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 α (see Behind the Scenes chapter for more details). For now, we will guess a value of $\log \alpha = 1$. In later chapters we will discuss training the AGD algorithm to select the optimal value of α.

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:

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

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

# Specify necessary parameters
alpha1 = 1.
snr_thresh = 5.
FILENAME_DATA = 'simple_gaussian.pickle'
FILENAME_DATA_DECOMP = '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)

# Run GaussPy
data_decomp = g.batch_decomposition(FILENAME_DATA)

# Save decomposition information
pickle.dump(data_decomp, open(FILENAME_DATA_DECOMP, 'w'))
```

After AGD determines the Gaussian decomposition, GaussPy then performs a least squares fit of the inital 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 data_decomp.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` : uncertainities in the fitted Gaussian parameters, determined from the least-squares fit.

GaussPy also stores the reduced χ^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. `FILENAME_DATA`: the filename containing the dataset to-be-decomposed.
2. `FILENAME_DATA_DECOMP`: 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(-4. * np.log(2) * (x-mean)**2 / fwhm**2)

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

FILENAME_DATA = 'simple_gaussian.pickle'
FILENAME_DATA_DECOMP = 'simple_gaussian_decomposed.pickle'

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

data_decomp = pickle.load(open(FILENAME_DATA_DECOMP))
means_fit = unravel(data_decomp['means_fit'])
amps_fit = unravel(data_decomp['amplitudes_fit'])
fwhms_fit = unravel(data_decomp['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'$\log\alpha=1.$', 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 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 $\log \alpha$, the fit would not change significantly as the fit to a spectrum containing a single Gaussian funciton 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.

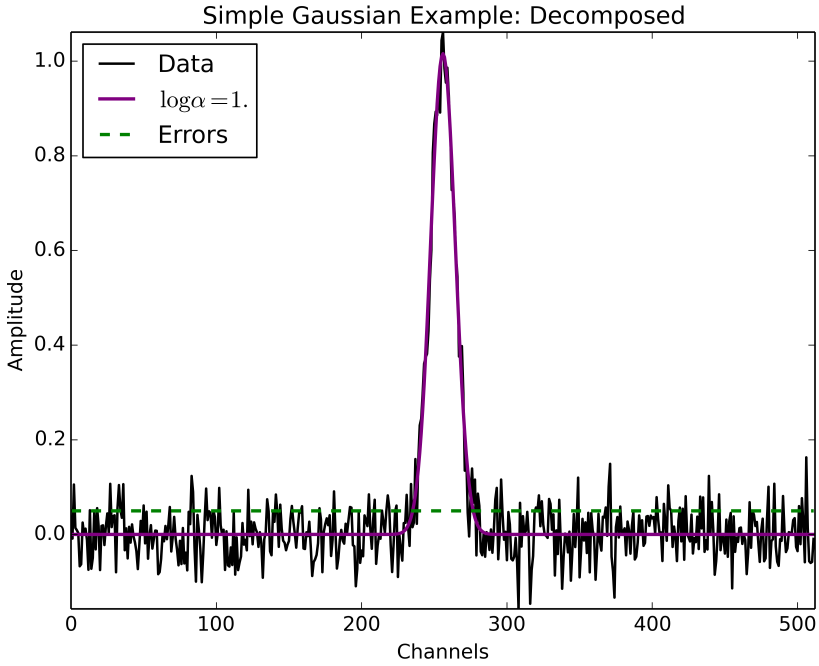


Fig. 3.2: Example spectrum containing a single Gaussian function with added spectral noise, decomposed using GaussPy.

In the ensuing chapters, we will move on from this simple example to consider spectra of increased complexity, as well as the effect of different values of α on the decomposition.

CHAPTER

FOUR

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 uncertainity 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 = [20,50,40]` : FWHM (in channels) of the included Gaussian functions
4. `MEANS = [220,250,300]` : 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(-4. * np.log(2) * (x-mean)**2 / fwhm**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 = [20,50,40] # channels
MEANS = [220,250,300] # channels
```

```
# Data properties
RMS = 0.05
NCHANNELS = 512

# Initialize
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
data['data_list'] = data.get('data_list', []) + [spectrum]
data['x_values'] = data.get('x_values', []) + [chan]
data['errors'] = data.get('errors', []) + [errors]

pickle.dump(data, open(FILENAME, 'w'))
```

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

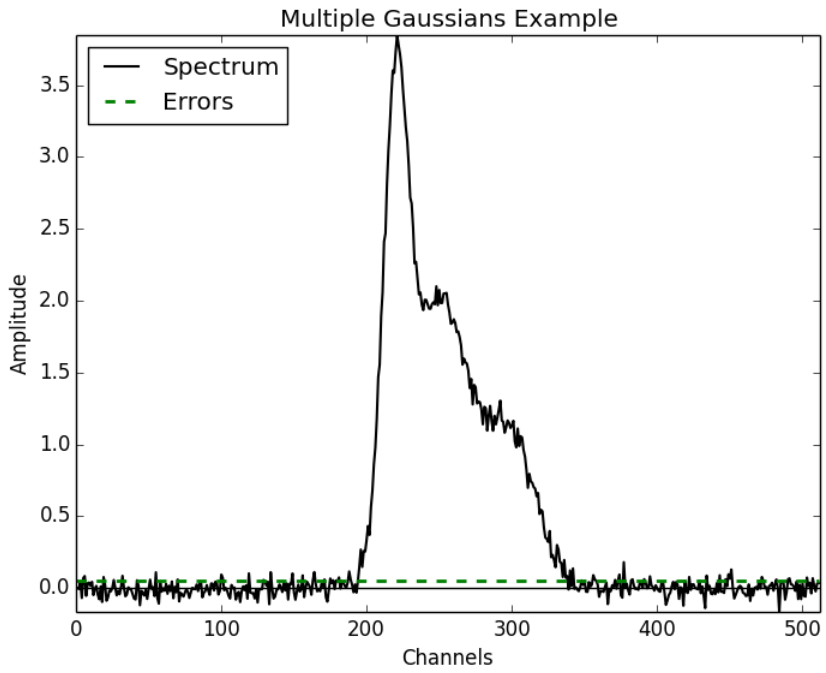


Fig. 4.1: Example spectrum containing multiple Gaussian functions with added spectral noise.

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 α to use in the decomposition. In this example, we will select $\log \alpha = 0.5$ to begin with. As before, the important parameters to specify are:

1. `alpha1`: our choice for the value of $\log \alpha$.
2. `snr_thresh`: the signal-to-noise ratio threshold below which amplitude GaussPy will not fit a component.
3. `FILENAME_DATA`: the filename containing the dataset to-be-decomposed, constructed above (or any GaussPy-friendly dataset)
4. `FILENAME_DATA_DECOMP`: the 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 = 0.5
snr_thresh = 5.
FILENAME_DATA = 'multiple_gaussians.pickle'
FILENAME_DATA_DECOMP = 'multiple_gaussians_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)

# Run GaussPy
data_decomp = g.batch_decomposition(FILENAME_DATA)

# Save decomposition information
pickle.dump(data_decomp, open(FILENAME_DATA_DECOMP, 'w'))
```

4.3 Plot Decomposition Results

Following the decomposition by GaussPy, we can explore the effect of the choice of α on the decomposition. In Fig. 4.2, we have run GaussPy on the multiple-Gaussian dataset constructed above for three values of α, including $\log \alpha = 0.5, \log \alpha = 2.5$ and $\log \alpha = 1.5$ and plotted the results.

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

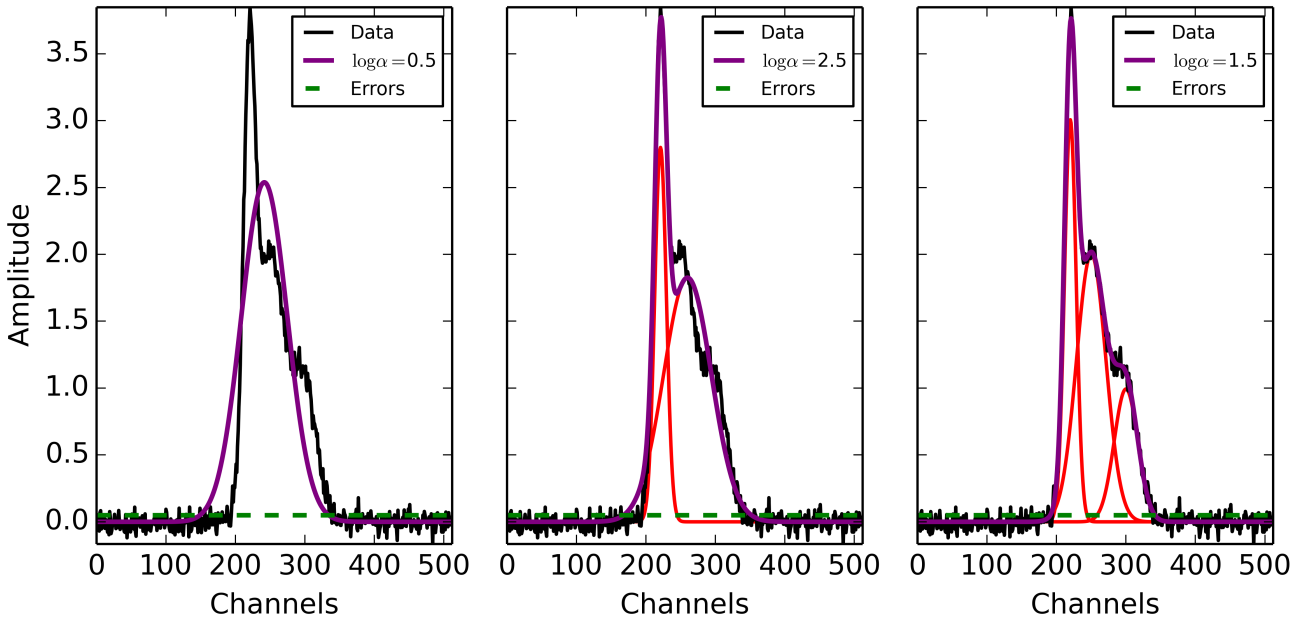


Fig. 4.2: Example spectrum containing multiple Gaussian functions with added spectral noise, decomposed using GaussPy for three values of the smoothing parameter $\log \alpha$.

CHAPTER

FIVE

TRAINING AGD

5.1 Creating a Synthetic Training Dataset

To select the optimal value of the smoothing parameter α, 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 α 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 $\sim N(0, \mathrm{RMS}) + f \times \mathrm{RMS}$ with `RMS=0.05` and $f = 0$
2. `NCOMPS = 3`
3. `NCHANNELS = 512` : the number of channels per spectrum
4. `RMS = 0.05` : RMS noise per channel.
5. `NSPECTRA = 200` : number of synthetic spectra to create for the training dataset.
4. AMP $\sim \mu(0.5, 4)$: the possible range of amplitudes to be included in each synthetic spectrum. Spectra with a more dominant contribution from the noise can also be generated and used as training sets for AGD.
5. FWHM $\sim \mu(20, 80)$ and MEAN $\sim \mu(0.25, 0.75) \times$ NCHANNELS : the possible range of FWHM and mean positions of Gaussian functions to be included in each synthetic spectrum.
6. `TRAINING_SET` : True, determines whether the decomposition "true answers" are sorted 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 paramters:
AMP_lims = [0.5, 4]
FWHM_lims = [20, 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" decompositon), 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
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)
        fwhms.append(w)
        means.append(m)

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

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

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

5.2 Training the Algorithm

Next, we will apply GaussPy to the real or synthetic training dataset and compare the results with the known underlying decompositon to determine the optimal value for the smoothing parameter α. 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_initial`: initial choice for $\log \alpha$

```
# 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_initial = 1.

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])

# Train AGD starting with initial guess for alpha
g.train(alpha1_initial = alpha_initial)
```

GausspPy will decompose the training dataset with the initial choice of $\alpha_{\rm initial}$ 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 $\alpha_{\rm initial}$ and recompute the decomposition until the process converges.The accuracy of the decomposition associated with the converged value of α is a description of how well GaussPy can recover the true underlying decomposition.

The above training dataset parameters were selected with the *Multiple Gaussians Tutorial* in mind. As we saw in that example, the choice of α has a significant effect on the GaussPy decomposition. In the training above, when we choose an initial value of $\log \alpha_{\rm initial} = 1.0$ the training process converges to $\log \alpha = 1.58$ with an accuracy of 68.4%, and required 33 iterations.

To ensure that the training converges on the optimal value of α and not a local maximum, it is useful to re-run the training process for several choices of $\alpha_{\rm initial}$. When we run the above example with an initial choice of $\log \alpha_{initial} = 3$, AGD converges to a value of $\log \alpha = 1.58$ with an accuracy of 68.4% and required 33 iterations. However, this is a relatively simple example and therefore the converged value of alpha is not very sensitive to $\alpha_{\rm initial}$. In the Prepping a Datacube chapter, we will discuss the effects of added complexity.

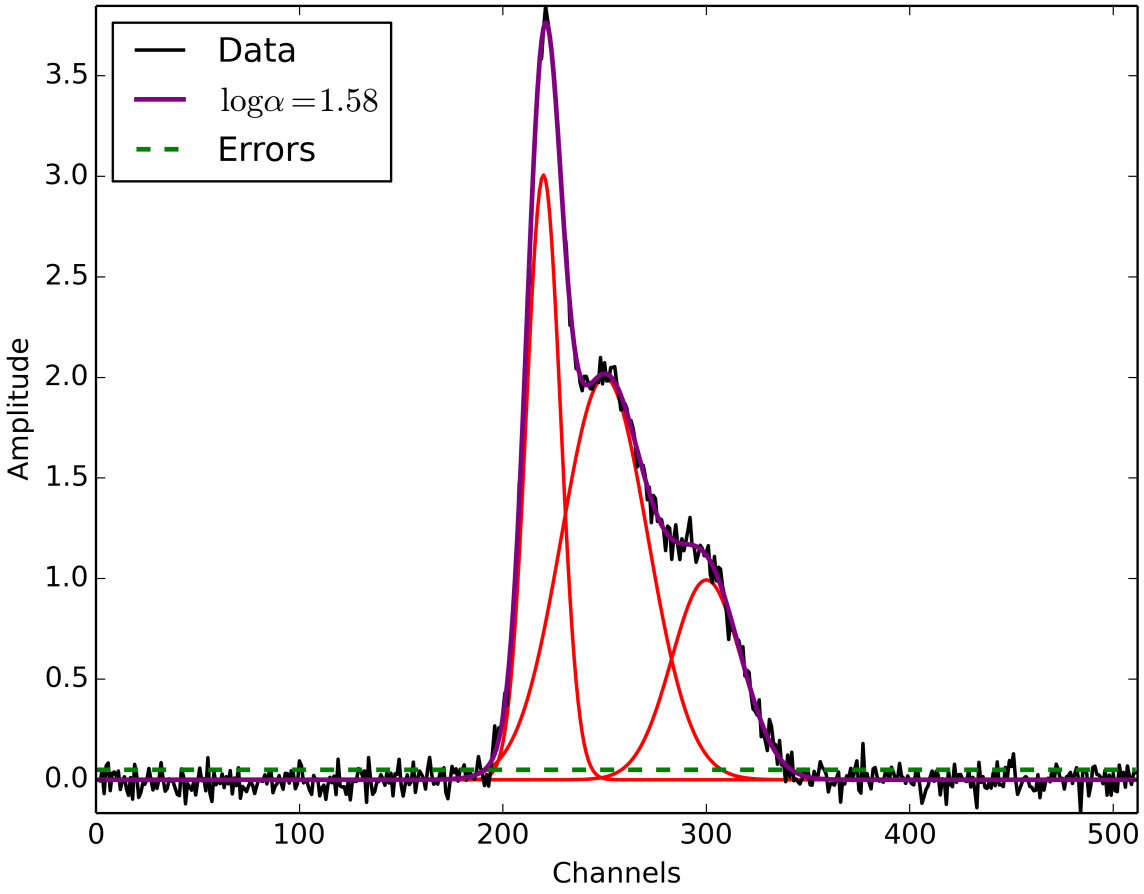
5.3 Running GaussPy using Trained α

With a trained value of α 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 $\log \alpha = 1.58$, which decomposed our training dataset with an accuracy of 68.4%. As in the *Simple Example Tutorial* and *Multiple Gaussians Tutorial*, the important parameters to specify are:

1. `alpha1`: our choice for the value of $\log \alpha$
2. `snr_thresh`: the signal-to-noise ratio threshold below which amplitude GaussPy will not fit a component
3. `FILENAME_DATA`: the filename containing the dataset to-be-decomposed, constructed above (or any GaussPy-friendly dataset)
4. `FILENAME_DATA_DECOMP`: 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 = 1.58
snr_thresh = 5.

FILENAME_DATA = 'multiple_gaussians.pickle'
FILENAME_DATA_DECOMP = '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)

# Run GaussPy
data_decomp = g.batch_decomposition(FILENAME_DATA)

# Save decomposition information
pickle.dump(data_decomp, open(FILENAME_DATA_DECOMP, 'w'))
```

Fig. 5.3 displays the result of fitting the "Multiple Gaussians" spectrum with a trained value of $\log \alpha = 1.58$.


Data
$\log\alpha = 1.58$
Errors
Amplitude
Channels
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0
100
200
300
400
500

CHAPTER

SIX

TWO-PHASE DECOMPOSITON

In the *Training AGD* chapter, we learned how to "train" AGD to select the optimal value of the smoothing parameter α 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 α 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 α, one for each "phase" contained within the dataset.

6.1 Training for Two Phases: α_1 and α_2

Using the example from the *Multiple Gaussians Tutorial*, we will train AGD to allow for two different values of α. This gives GaussPy enough flexibilty to use appropriate values of α to fit both narrow and wide features simultaneously. We will use the same training dataset constructed in *Training AGD*. We must set the following parameters:

1. `FILENAME_TRAIN`: 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. `alpha1_initial, alpha2_initial``: initial choices for $\log \alpha_1$ and $\log \alpha_2$

The training will be the same as in *Training AGD*, however we will set the GaussPy parameter *phase* equal to *two* instead of *one* to indicate that we would like to solve for two different values of α.

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

import gausspy.gp as gp

# Set necessary parameters
FILENAME_TRAIN = 'training_data.pickle'
snr_thresh = 5.
alpha1_initial = 0.5
alpha2_initial = 2.

g = gp.GaussianDecomposer()

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

# Set GaussPy parameters
```

```
g.set('phase', 'two')
g.set('SNR_thresh', [snr_thresh, snr_thresh])

# Train AGD starting with initial guess for alpha
g.train(alpha1_initial = alpha1_initial, alpha2_initial = alpha2_initial)
```

Following training, GaussPy converges on values of $\log \alpha_1 = 0.39$ and $\log \alpha_2 = 2.32$ in 39 iterations, with an accuracy of 76.0%. Clearly, the two-phase decomposition improves the accuracy of the decomposition, of course at the expense of introducing a second free parameter in the decomposition. In general, for datasets containing more than one type of component (corresponding to different physical sources, for example), two-phase decomposition will maximize the decompositon accuracy.

CHAPTER

SEVEN

PREPPING A DATACUBE

In this example we will download a datacube to decompose into individual spectra. The example cube we will use is from the GALFA-HI emission survey at the Arecibo Observatory, specifically the M33 datacube from Putman et al. 2009. You can directly download the cube from here:

http://www.astro.columbia.edu/~mputman/M33only.fits.gz

7.1 Storing Data cube in GaussPy-Friendly Format

Before decomposing the datacube, we must store the data in a format readable by GaussPy. The following code provides an example of how to read a fits-formatted datacube and store the spectral information. The necessary parameters to specify here are:

1. `FILENAME_DATA`: the fits filename of the target data cube
2. `FILENAME_DATA_GAUSSPY`: the filename to store the GaussPy-friendly data in
3. `RMS`: estimate of the RMS uncertainty per channel for constructing the error arrays

```
# Read fits datacube and save in GaussPy format
import numpy as np
import pickle
from astropy.io import fits

# Specify necessary parameters
FILENAME_DATA = 'M33only.fits'
FILENAME_DATA_GAUSSPY = 'cube.pickle'
RMS = 0.06

hdu_list = fits.open(FILENAME_DATA)
hdu = hdu_list[0]
cube = hdu.data

# initialize
data = {}
errors = np.ones(cube.shape[0]) * RMS
chan = np.arange(cube.shape[0])

# cycle through each spectrum
for i in xrange(cube.shape[1]):
    for j in xrange(cube.shape[2]):

        # get the spectrum
        spectrum = cube[:, i, j]
```

```
        # get the spectrum location
        location = np.array((i, j))

        # Enter results into GaussPy-friendly dataset
        data['data_list'] = data.get('data_list', []) + [spectrum]
        data['x_values'] = data.get('x_values', []) + [chan]
        data['errors'] = data.get('errors', []) + [errors]
        data['location'] = data.get('location', []) + [location]

# Save decomposition information
pickle.dump(data, open(FILENAME_DATA_GAUSSPY, 'w'))
```

The output pickle file from the above example code contains a python dictionary with four keys, including the independent and dependent arrays (i.e. channels and spectral values), an array per spectrum describing the uncertainty per channel, and the (x,y) pixel location within the datacube for reference.

7.2 Creating a Synthetic Training Dataset

Before decomposing the target dataset, we need to train the AGD algorithm to select the best values of $\log \alpha$ in two-phase decomposition. First, we construct a synthetic training dataset composed of Gaussian components with parameters sampled randomly from ranges that represent the data as closely as possible.

1. `RMS`: root mean square uncertainty per channel
2. `NCHANNELS`: number of channels per spectrum
3. `NSPECTRA`: number of spectra to include in the training dataset
4. `NCOMPS_lims`: range in total number of components to include in each spectrum
5. `AMP_lims, FWHM_lims, MEAN_lims`: range of possible Gaussian component values, amplitudes, FWHM and means, from which to build the spectra
6. `TRAINING_SET` : True or False, determines whether the decomposition "answers" are stored along with the data for accuracy verification in training
7. `FILENAME_TRAIN` : filename for storing the training data

```
# Create training dataset with Gaussian profile
import numpy as np
import pickle

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

# Estimate of the root-mean-square uncertainty per channel (RMS)
RMS = 0.06

# Specify the number of spectral channels (NCHANNELS)
NCHANNELS = 680

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

# Estimate the number of components
NCOMPS_lims = [3,6]

# Specify the min-max range of possible properties of the Gaussian function paramters:
```

```
AMP_lims = [0.5,30]
FWHM_lims = [20,150] # channels
MEAN_lims = [400,600] # 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_TRAIN = 'cube_training_data.pickle'

# Initialize
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 = []

    ncomps = np.random.choice((np.arange(NCOMPS_lims[0],NCOMPS_lims[1]+1)))

    for comp in xrange(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)
        fwhms.append(w)
        means.append(m)

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

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

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

7.3 Training AGD to Select α values

With a synthetic training dataset in hand, we train AGD to select two values of $\log \alpha$ for the two-phase decomposition, $\log \alpha_1$ and $\log \alpha_2$. The necessary parameters to specify are:

1. `FILENAME_TRAIN`: the pickle file containing the training dataset in GaussPy format
2. `snr_thresh`: the signal to noise ratio below which GaussPy will not fit a component
3. `alpha1_initial, alpha2_initial` initial choices of the two $\log \alpha$ parameters

```
# Train AGD using synthetic dataset
import numpy as np
import pickle
import gausspy.gp as gp
reload(gp)

# Set necessary parameters
FILENAME_TRAIN = 'cube_training_data.pickle'
snr_thresh = 5.
alpha1_initial = 4
alpha2_initial = 12

g = gp.GaussianDecomposer()

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

# Set GaussPy parameters
g.set('phase', 'two')
g.set('SNR_thresh', [snr_thresh, snr_thresh])

# Train AGD starting with initial guess for alpha
g.train(alpha1_initial = alpha1_initial, alpha2_initial = alpha2_initial)
```

Training: starting with values of $\log \alpha_{1,\,\mathrm{initial}} = 3$ and $\log \alpha_{2,\,\mathrm{initial}} = 12$, the training process converges to $\log \alpha_1 = 2.87$ and $\log \alpha_2 = 10.61$ with an accuracy of 71.2% within 90 iterations.

7.4 Decomposing the Datacube

With the trained values in hand, we now decompose the target dataset:

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

# Specify necessary parameters
alpha1 = 2.87
alpha2 = 10.61
snr_thresh = 5.0

FILENAME_DATA_GAUSSPY = 'cube.pickle'
FILENAME_DATA_DECOMP = 'cube_decomposed.pickle'

# Load GaussPy
g = gp.GaussianDecomposer()
```

```
# Setting AGD parameters
g.set('phase', 'two')
g.set('SNR_thresh', [snr_thresh, snr_thresh])
g.set('alpha1', alpha1)
g.set('alpha2', alpha2)

# Run GaussPy
decomposed_data = g.batch_decomposition(FILENAME_DATA_GAUSSPY)

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

And plot the results for an example set of 9 spectra, randomly selected, to see how well the decomposition went.

```
# Plot GaussPy results for selections of cube LOS
import numpy as np
import pickle
import matplotlib.pyplot as plt

# load the original data
FILENAME_DATA_GAUSSPY = 'cube.pickle'
data = pickle.load(open(FILENAME_DATA_GAUSSPY))

# load decomposed data
FILENAME_DATA_DECOMP = 'cube_decomposed.pickle'
data_decomposed = pickle.load(open(FILENAME_DATA_DECOMP))

index_values =  np.argsort(np.random.randn(5000))

# plot random results
fig = plt.figure(0,[9,9])

for i in range(9):
    ax = fig.add_subplot(3, 3, i)

    index = index_values[i]
    x = data['x_values'][index]
    y = data['data_list'][index]

    fit_fwhms = data_decomposed['fwhms_fit'][index]
    fit_means = data_decomposed['means_fit'][index]
    fit_amps = data_decomposed['amplitudes_fit'][index]

    # Plot individual components
    if len(fit_amps) > 0.:
        for j in range(len(fit_amps)):
            amp, fwhm, mean =  fit_amps[j], fit_fwhms[j], fit_means[j]
            yy = amp * np.exp(-4. * np.log(2) * (x-mean)**2 / fwhm**2)
            ax.plot(x,yy,'-',lw=1.5,color='purple')

    ax.plot(x, y, color='black')
    ax.set_xlim(400,600)
    ax.set_xlabel('Channels')
    ax.set_ylabel('T_B (K)')

plt.show()
```

Fig. 7.1 displays an example set of spectra from the data cube and the GaussPy decomposition using trained values of $\log \alpha_1 = 2.87$ and $\log \alpha_2 = 10.61$.

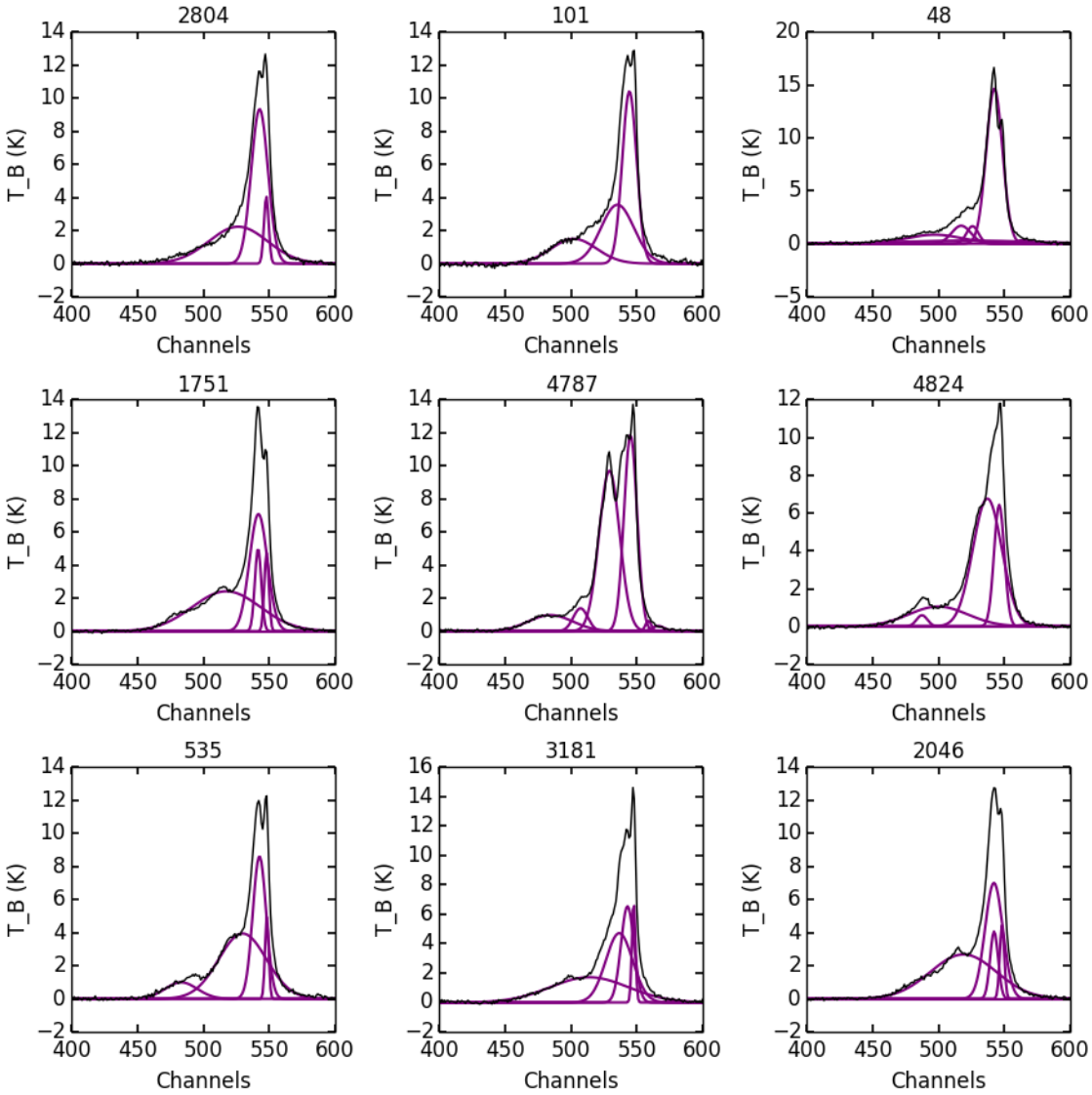


Fig. 7.1: Example spectra from the GALFA-HI M33 datacube, decomposed by GaussPy following two-phase training.

CHAPTER

EIGHT

BEHIND THE SCENES

8.1 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 $\mathrm{d}f(x)/\mathrm{d}x = 0$ 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

$$f(x^*) > \epsilon_0. \tag{8.1}$$

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 ϵ_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^*

$$\left.\frac{\mathrm{d}^2 f}{\mathrm{d}x^2}\right|_{x=x^*} < 0, \tag{8.2}$$

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 $f(x)$

$$\left.\frac{\mathrm{d}^3 f}{\mathrm{d}x^3}\right|_{x=x^*} = 0 \tag{8.3}$$

$$\left.\frac{\mathrm{d}^4 f}{\mathrm{d}x^4}\right|_{x=x^*} > 0 \tag{8.4}$$

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. 8.1 is an example of a function defined as the sum of three gaussians for which the conditions Eq. (8.1) - (8.4) are satisfied and the local minima of curvature are successfully found, even when blending of components is relevant.

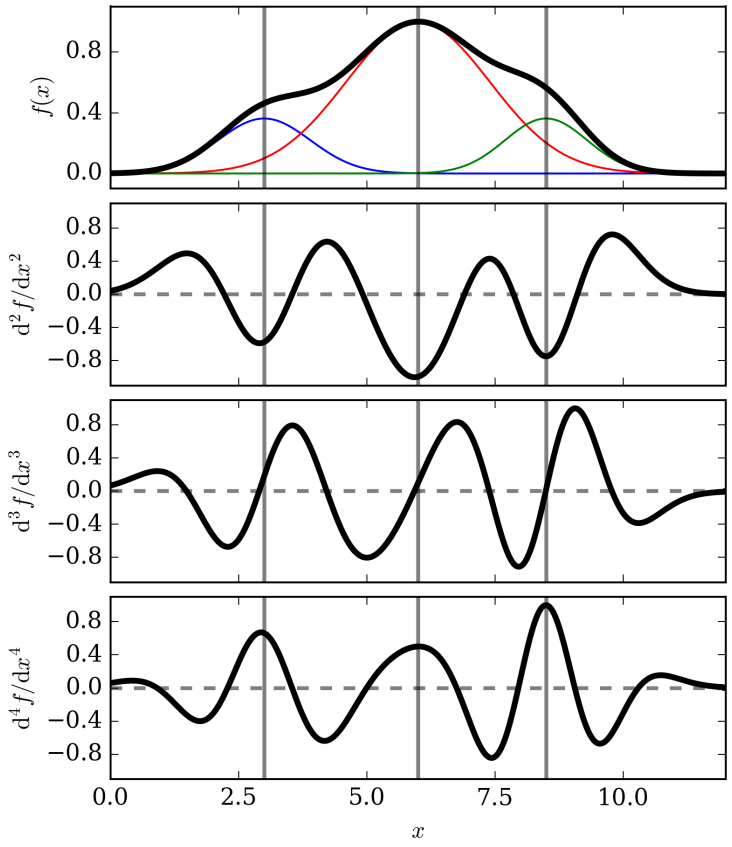


Fig. 8.1: Example of the points of negative curvature of the function $f(x)$. In this case $f(x)$ is the sum of three independent Gaussian functions (top). The vertical lines in each panel show the conditions imposed on the derivatives to define the points x^*.

8.2 Dealing with noise

The numeral problem related to the solution shown in the previous section comes from the fact that calculating Eq. (8.2) - (8.4) is not trivial in the presence of noise. For instance, if the top panel of Fig. 8.1 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 $\sim 1/\mathrm{d}x \sim 10$. 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 $u = \mathrm{d}f(x)/\mathrm{d}x$, then the problem we solve is $u = \arg\min_u\{R[u]\}$ where $R[u]$ is the functional defined by

$$R[u] = \int |Au - f| + \alpha \int \sqrt{(Du)^2 + \beta^2}, \tag{8.5}$$

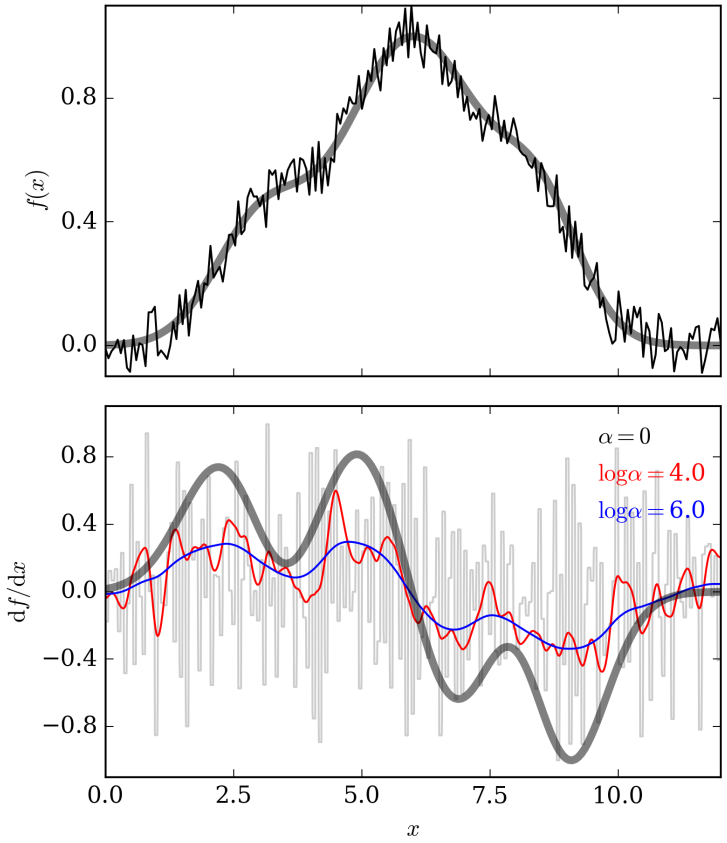


Fig. 8.2: The top panel shows the same function used in Fig. 8.1 but now random noise has been added to each channel. In the bottom panel we show various estimates of the first derivative. $\alpha = 0$ corresponds to the finite differences method, larger values of α makes the function smoother.

where $Au = \int \mathrm{d}x\ u$. 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 $u = \mathrm{d}f(x)/\mathrm{d}x$ and $f(x)$ itself. This, however, has the problem we discussed in the previous paragraph. Fig. 8.2 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. (8.5), now the differences between successive points in $u(x)$ are taken into account.

The parameter α 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 α? 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.

8.3 Two phases

Within `GaussPy` is built-in the ability to automatically choose the best value of α 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 trainend in "two-steps". One for narrow components, and one for broad components. The result then is two independent values α_1 and α_2 each giving information about the scales of different features in the target function.

8.4 An alternative approach

There is another alternative for calculating derivatives of noise-ridden data, namely convolving the function with a low-pass filter kernel, e.g., a Gaussian filter. Although the size of the filter can be optimized by using a training routine, in a similar fashion as we did for the α scales, this technique is much more agressive and could lead to losses of important features in the signal. Indeed, the total variation scheme that `GaussPy` uses could be thought of as the first order approximation in a perturbative expansion of a Gaussian filter.

Notwhistanding this caveat `GaussPy` implements also a Gaussian filter as an option for taking the numerical derivatives. In total, there are three selectable `modes` within the package for calculating $f(x) + n(x)$

- `GaussianDecomposer.set('mode','python')`: This will execute `GaussPy` with a `Python` implementation of the total variation algorithm. The code is clean to read, easy to understand and modify, but it may perform slow for large datasets.
- `GaussianDecomposer.set('mode','C')`: This chooses a `C` implementation of the TV algorithm when calculating the derivative. It is less time-consuming that the `Python` version but comes at the price of intelligibility of some parts of the code.

  Both `Python` and `C` modes yield the same results and, therefore, are interchangable. Our suggestion is to use `Python` if you are planning to delve into the details of the code, and `C` if you are after efficiency in large dataset.
- `GaussianDecomposer.set('mode','conv')`: When this mode is set, the function is Gaussian-filtered prior to calculating the numerical derivative. In this case, the constant α is taken to be the size of the kernel

$$\tilde{f}(x) = (f \star K_\alpha)(x) \quad \text{with} \quad K_\alpha(x) = \frac{1}{\sqrt{2\pi\alpha^2}} \mathrm{e}^{-x^2/2\alpha^2}. \tag{8.6}$$

  Once this mode is selected the training for choosing the optimal size of the filter proceeds in the same way we have discussed in the previous sections, i.e., nothing else has to be changed.