GaussPy Documentation

Release 1.0

Robert Lindner, Carlos Vera-Ciro

February 22, 2016

CONTENTS

Contents:

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

- Python 2.7
- Numpy
- Scipy
- h5py
- GNU Scientific Library (GSL)

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.

To obtain GSL:

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

2.2 Download GaussPy

Download GaussPy from...

2.3 Installing Dependencies

You will need several libraries which the GSL, h5py, and scipy libraries depend on.

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

Install pip for easy installation of python packages:

$ sudo apt-get install python-pip $ sudo pip install –upgrade pip

Install *scipy*:

$ sudo pip install -I scipy==0.17.0

Install *lmfit*

$ sudo pip install lmfit

Install *numpy*

$ sudo apt-get install python-numpy -y

Install *matplotlib*

$ sudo apt-get install -qq python-matplotlib -y

Install *h5py*

$ sudo apt-get install -qq python-h5py -y

2.4 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

CHAPTER

THREE

QUICKSTART TUTORIAL

3.1 Simple Example

3.1.1 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:

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

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.

```
# Create simple Gaussian profile with 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)

# Data properties
```

```
RMS = 0.05
NCHANNELS = 512
FILENAME = 'simple_gaussian.pickle'
TRAINING_SET = True

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

# Initialize
simple_gaussian = {}
chan = np.arange(NCHANNELS)
errors = chan * 0. + RMS # Constant noise for all spectra

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

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

pickle.dump(simple_gaussian, open(FILENAME, 'w'))
print 'Created: ', FILENAME
```

3.1.2 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. 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 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 guess for the value of α.
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 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 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` : 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.

3.1.3 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

def gaussian(amp, fwhm, mean):
    sigma = np.float(fwhm / (2. * np.sqrt(2. * np.log(2))))
    return lambda x: amp * np.exp(-(x-mean)**2/2./sigma**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 = np.array(unravel(data['data_list']))
chan = np.array(unravel(data['x_values']))
errors = np.array(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 = chan * 0.

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, color='black', linewidth=1.5)
ax.plot(chan, model, color='purple', linewidth=2.)
ax.plot(chan, errors, color='green', linestyle='dashed', linewidth=2.)
ax.set_xlabel('Channels')
ax.set_ylabel('Amplitude')

plt.show()
```

Given the speed an efficiency of AGD, we can use the above examples to vary the complexity of the spectra to be decomposed, as well as the effect of different values of α on the decomposition.

3.2 Training AGD to Select α

3.2.1 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 the function

$$S(x_i) = \sum_{k=1}^{\mathtt{NCOMPS}} \mathtt{AMP}_k \exp\left[-\frac{4\ln 2(x_i - \mathtt{MEAN}_k)^2}{\mathtt{FWHM}_k^2}\right] + \mathtt{NOISE}, \qquad i = 1, \cdots, \mathtt{NCHANNELS} \tag{3.2}$$

where

1. `NSPECTRA` is then the number of synthetic spectra to be created
2. `NCOMPS` is the number of components in each synthetic spectrum
3. `(AMP, MEAN, FWHM)` are the parameters of each Gaussian component
4. `NOISE` is the level of noise introduced in each spectrum

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

1. $\mathrm{NOISE} \sim N(0, \mathrm{RMS}) + f \times \mathrm{RMS}$ with `RMS=0.05` and $f = 0$
2. `NCOMPS = 4`

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. AMP $\sim \mu(5\mathrm{RMS}, 25\mathrm{RMS})$, 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. FWHM $\sim \mu(10, 35)$ and MEAN $\sim \mu(0.25, 0.75) \times$ 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.

```
# GaussPy Example 1
# 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 = 1000

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

# Estimate the mean number of Gaussian functions to add per spectrum (NCOMPS)
NCOMPS = 4

# Specify the min-max range of possible properties of the Gaussian function paramters:
# Amplitude (AMP)
AMP_lims = [RMS * 5, RMS * 25]
# Full width at half maximum in channels (FWHM)
FWHM_lims = [10, 35] # channels
# Mean channel position (MEAN)
MEAN_lims = [0.25 * NCHANNELS, 0.75 * NCHANNELS]

# Indicate whetehre 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 = 'agd_data_science.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. If we want this to be a training set (`TRAINING_SET = True`), the "true" decomposition answers for estimating the accuracy of a decomposition are also stored in the output file. For example, to construct a synthetic dataset:

```
# GaussPy Example 1
# Create spectra with Gaussian profiles -cont-

# Initialize
agd_data = {}
chan = np.arange(NCHANNELS)
errors = chan * 0. + RMS # Constant noise for all spectra

# Begin populating data
for i in range(NSPECTRA):
    spectrum_i = np.random.randn(NCHANNELS) * RMS
    # Sample random components:
    amps = np.random.rand(NCOMPS) * (AMP_lims[1] - AMP_lims[0]) + AMP_lims[0]
    fwhms = np.random.rand(NCOMPS) * (FWHM_lims[1] - FWHM_lims[0]) + FWHM_lims[0]
    means = np.random.rand(NCOMPS) * (MEAN_lims[1] - MEAN_lims[0]) + MEAN_lims[0]

    # Create spectrum
    for a, w, m in zip(amps, fwhms, means):
        spectrum_i += gaussian(a, w, m)(chan)

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

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

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

3.2.2 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. `alpha1`: initial guess for α

```
import gausspy.GaussianDecomposer as gp

# Set necessary parameters
FILENAME = 'agd_data.pickle'
snr_thresh = 5.
alpha1

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', 5.)

# Train AGD starting with initial guess for alpha
g.train(alpha1_initial = alpha1, plot=False, verbose = False, mode = 'conv', learning_rate = 1.0, eps
```

GausspPy will iterate over a range of α values and compare the decomposition associated with each α value to the correct decomposition specified within the training dataset to maximize the accuracy of the decomposition.

Once the training is completed, we can view the "trained" value of alpha by looking at the attribute of our GaussianDecomposer instance.

```
# get the parameters attribute of g, which is a dictionary of important
# variables
print(g.p['alpha1'])
```

3.3 Running AGD using Trained α

With the trained value of α in hand, we can proceed to decompose our target dataset with AGD.

CHAPTER

FOUR

BEHIND THE SCENES

A description of what AGD does ...

CHAPTER FIVE

INDICES AND TABLES

- genindex
- modindex
- search