Chapter 6:
Fitting Data with Multiple Components

- ☐ **New concepts**
  - *and their prerequisites*

- ☐ **Getting yes/no answers to questions posed by comparison statements**
  - *Section: 2.1.3 - Representing answers to yes/no questions using booleans*
- ☐ **Slicing arrays by values instead of indices using comparison expressions**
  - *Section: 1.1.1 - Operating on collections of data efficiently with Numpy arrays*
  - *Section: 6.1.2 - Having Python do basic math by evaluating mathematical expressions*
  - *Section: 2.1.3 - Representing answers to yes/no questions using booleans*
- ☐ **Slicing based on multiple conditions using `and` and `or`**
  - *Section: 1.1.1 - Operating on collections of data efficiently with Numpy arrays*
  - *Section: 6.1.2 - Having Python do basic math by evaluating mathematical expressions*
  - *Section: 2.1.3 - Representing answers to yes/no questions using booleans*
- ☐ **Visualizing fit results using `codechembook.quickPlots.plotFit()`**
  - *Section: 2.1.7 - Adding `codechembook` to your Python installation*
  - *Section: 5.1.9 - Fitting models to data with `lmfit`*
- ☐ **Learning to use libraries by reading documentation**
- ☐ **Combining `lmfit` models**
  - *Section: 5.1.9 - Fitting models to data with `lmfit`*
- ☐ **Accessing the number of times a loop has iterated using `enumerate()`**
  - *Section: 0.1.2 - Accessing capabilities beyond basic math using functions*
  - *Section: 3.1.11 - Performing repetitive operations using `for` loops*

6.0 Problem

Looking at the spectra for your initial and final titration points, you notice that there is some structure in the MLCT band. Specifically, for the band around 600nm in the spectrum at the initial point, you wonder how many features are present and what their central positions, widths, and intensities are are. The band clearly has two partially resolved features, but there also appears to be a shoulder on the high energy side that indicates that the band is really three underlying features. Seeing that you can now fit equations to data, you want to fit this band to try to identify the underlying components. Because this is a solution phase UV-vis spectrum, you reason that any features are heterogeneously broadened and should be treated as Gaussian lineshapes. Thus, you would really like to fit the band to three Gaussian bands, plus a background. If you treat the background as linear, you already know how to use the linear model in lmfit[1]. So, you wonder if there might be a similar model for a Gaussian and a way to combine multiple instances of it.

6.1 Solution

The solution in your mind is pretty simple: figure out how to use the lmfit **library** to fit the overall shape of your MLCT band to the sum of three Gaussians and a linear background. Then you can make a plot showing the three individual Gaussian components and annotate it wiht their positions.

6.1.0 Planning

Your labmate says that basically what you want to do is to:

1. Load in the spectrum you want to fit.
2. Isolate the region of the spectrum you want to fit.
3. Make a composite model that is a linear background + 3 Gaussians.
4. Fit the model to the desired region of the spectrum and extract parameters for the Gaussians.
5. Plot the results.

They point out that you already know how to do #1, #4, and #5. They say they can teach you how to do #2 quickly, but that they is leaving on a backpacking trip with friends and so you might have to figure out #4 by reading the online documentation. They say you should have enough knowledge to read the documentation, and so you should give it a try. But first, they launch into a quick tutorial on how to isolate the spectral region of interest.

"This relies on combining logical operations with array slicing, so let's see each in turn."

6.1.1 Getting yes/no answers to questions posed by comparison statements

When looking for a region of a spectrum, you really mean you are looking for wavelengths that are between two values. In other words, you want to find values that are greater than a specific value *and* also less than another value. "We can use logical operations to test for this," your labmate says. "These operations evaluate to one of the two **boolean** values: `True` or `False`,[2] so you can use them to decide which data points are in the range you want."

```
>>> 1 > 0
```

[1] See Chapter 5

[2] This is the same **data type** introduced in Section 2.1.3.

```
True
```

```
>>> 0 > 0
False
```

"You can also test that something is less than (<), equal to (==), and so on.[3] Note that the test for 'equal to' uses two equal signs. This done to distinguish it from the assignment operator (=)."

"We have encountered the **keyword `in`** when building **`for` loop**. This was used to access the elements in a **collection** (e.g., **list**, **array**, etc.). We can also use this in a logical test, to check if something exists in a collection."

```
>>> mylist = [1, "one", 2, 5]
>>> 3 in mylist
False
```

"You can see that `in` checks if the item on the left is in the collection on the right. If it is, it will return `True` and if it is not it will return `False`. You could try to change the entries in `mylist` to make the `in` statement `True`."

6.1.2 Selecting a subset of a Numpy array using comparison expressions

"The logical tests we just saw can be applied to a Numpy array. When you do so, an array of the same length will be returned, but this array will only contain the values `True` and `False`. For instance, suppose you had a **Numpy array** of numbers from 1 to 20:"

```
>>> import numpy as np
>>> nums = np.linespace(1, 21, 1)
```

"You can see which elements are greater than 5, using:"

```
>>> nums > 5
array([False, False, False, False, False,  True,  True,  True,  True, True,  ↩
    ↪True,  True,  True,  True,  True,  True,  True,  True, True,  True])
```

You see that this returns an array of the same length as `nums`, but where each entry specifies if that number was greater than 5.

"Something else that is interesting is that one can use an array of booleans to specify which elements of a Numpy array to select. This is used similarly to specifying indices—that is you place this between two square brackets, after the array. If we assign the array of booleans to a variable, then we can use this variable to select a subset of the array."

```
>>> mask = nums > 5
>>> nums[mask]
array([ 6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20])
```

"Using an array of booleans to select a subset of an array is known as **mask**ing and it creates a *copy* (not a **view**) of the array.

You think that maybe you could skip the intermediate step of assigning the result of a logical operation to a mask, so you try another version of the code that perform the logical test directly within the square brackets.

```
>>> nums[nums > 5]
```

[3] A complete list of such operators is found in Chapter B.

```
array([ 6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20])
```

It worked, and looks very clean! So, masking is a powerful way to select specific elements in an array.

Then, you think this could be useful for **list**s, too. So you try the same thing:

```
>>> num_list = list(range(1, 21))
>>> num_list > 5
Traceback (most recent call last):

  Cell In[17], line 1
    num_list>5

TypeError: '>' not supported between instances of 'list' and 'int'
```

Your labmate explains, "Masking only works because of the **broadcasting** behavior of Numpy arrays. If you tried this on something like a **list**, you will get an error."

6.1.3 Selecting array subsets based on multiple conditions using `and` and `or`

You see that **mask**ing is great, but it still doesn't do what you want. You can test whether a point is greater than the minimum wavelength that you want to keep, or if it is less than the maximum, but not both. Your labmate says, "Sometimes we want to apply multiple tests to an object. For instance, we want to know if a point x is greater than 7 *and* less than 10. Mathematically, we would just write $10 > x > 7$, but we would understand that this is really two distinct logical operations: $10 > x$ *and* $x > 7$. For the expression to evaluate to true, x must satisfy the first *and* the second."

"In Python, the same behavior is accomplished using the `and` operator,[4] but Python allows the same shortcuts: `10 > x > 7` is a shorthand for `10 > x and x > 7`. More complicated tests can be created using multiple operations (e.g., `and` and `or` statements), like `10 > x and x > 7 or x == 0`. For these statements, there is an order of operations that is followed, but it is usually better to use parentheses to make it very clear what order you intend the conditionals to be evaluated. For example: `(10 > x and x > 7) or x == 0` may evaluate to a different result than `10 > x and (x > 7 or x == 0)`."

"So, you **mask**ed **Numpy array**s using a single **conditional expression**, but in this case, you want to take a subset of the arrays that corresponds to a range of values in the middle of the array. A quick test reveals the problem:"

```
>>> wavenumber = np.arange(400, 4000, 1)
>>> sliced_wavenumber = wavenumber[1900 > wavenumber > 800]
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    sliced_wavenumber = wavenumber[1900 > wavenumber > 800]
ValueError: The truth value of an array with more than one element is ↩
   ↪ambiguous. Use a.any() or a.all()
```

Trying

```
>>> sliced_wavenumber = wavenumber[1900 > wavenumber and wavenumber > 800]
```

[4] This is true of any boolean operator, like or or xor.

yields the same result.

Your labmate explains, "You gave Python a mathematically reasonable statement, but since `wavenumber` is a Numpy array, Python evaluates one conditional and returns an array of equal size but containing only `True` and `False` values at each **index**. When you link two conditionals like this, Python interprets it as you looking for values that satisfy the first condition *and* the second one, simultaneously, and implicitly inserts the `and` **operator** for you. Unfortunately, the `and` operator is only designed to evaluate to a single `True` or `False` value, and cannot handle the **vectorization** that operations between Numpy arrays typically rely on. Basically, it is asking "are these arrays both true?" rather than "are the i^{th} elements in each of these arrays both true?" This obviously is a problem."

"What you really want to do is an "element-wise" logical operation, performing a logical operation on two corresponding elements in two different arrays to form a new array holding the result of that logical operation. For this, use the 'elementwise and' operator ('`&`'):"

```
>>> sliced_wavenumber = wavenumber[(wavenumber > 800) & (wavenumber < 1900)]
```

"Basically, you use `&` rather than the Python `and` command and everything else works as expected. Similarly, `or` can be replaced by '`|`'.[5] Note that *the parentheses here are important to set the order of operations correctly*. If you leave them out, the line of code will fail because it evaluates the operators in order from left to right and the expression `wavenumber > 80 & wavenumber < 1900` will not work. This is because the statement is evaluated left to write and this is the equivalent of `((wavenumber > 80) & wavenumber) < 1900`, which will reduce to comparing the `True/False` values in an array produced by the first logical test to the numerical data contained in the `wavenumber` array. This will either fail, or produce results that you do not expect."

6.1.4 Accessing the number of times a loop has iterated using `enumerate()`

"We saw that we could **iterate** through multiple iterable **collection**s together using `zip`.[6] Sometimes you need to know what number iteration you are on (corresponding to an index of the collection). Or, sometimes the arrays or lists that you want to iterate over are not the same length. In either of these cases, there is a better solution: the `enumerate()` function."

"The enumerate **function** accepts an **iterable object** as a **positional argument** and **return**s a **tuple** of two values for each element in the object. The first value is the iteration number, and the second is the element at the index corresponding to the iteration number. These values can be **unpacked** and assigned to two **variable**s. In some sense then `enumerate(mylist)` is equivalent to `zip(range(0, len(mylist)), mylist)`. The behavior can be seen in the following:"

```
1 for i, name in enumerate(["one", "two", "three"]):
2     print(f"the index is {i} and the value at that index is {name}")
```

```
the index is 0 and the value at that index is one
the index is 1 and the value at that index is two
the index is 2 and the value at that index is three
```

"This can be a very powerful way to keep track of where you are in a list."

[5] On a U.S. keyboard, you get this character by holding shift and pressing the button above the Enter key
[6] See Chapter 4.

6.1.5 Visualizing fit results using `codechembook.quickPlots.plotFit()`

You had previously created a figure with two subplots that plotted data and a fit to this data in the top plot, and a plot of the residuals (data - fit) in the bottom plot. This allowed you to easily visualize the fit and how it deviates from the data, but it was a fair amount of coding.

This approach is generally useful enough that the **`codechembook` library** includes a **function** to accomplish this quickly: `fitPlot()`. To use this function, you simple provide a lmfit fit **result object** as the only **positional argument**, and this function will generate a plot with data and fit on top and residual on the bottom. There are also a few useful **keyword argument**s for this function:

- `residual`. The default value for this is `False` and the default behavior is to plot only the top potion (data and fit). Passing a value of `True` results in showing the residual below the data. If `True` is used, then the data and fit subplot will be 4 times taller than the residual subplot—a behavior that makes it very easy to see the residual, even when it is small. However, passing `residual = 'scaled'` to the function will produce a plot in which the scale of the y-axis on the top and the bottom are the same, meaning that one can directly compare the residuals to the data.
- `components`. The default value for this is `False` and the default behavior is to plot only the final fit line. If this keyword is set to `True`, then the plot will also include individual components of the model. For instance, if you had a model that was the sum of a line and a Gaussian profile, this will show the sum, as well as just the line and just the Gaussian. This can be useful for understanding the behavior of your model.

There are a few other keywords, and you can consult the codechembook library to find them, but for now, this is sufficient to solve your current problem.

6.1.6 Learning to use libraries by reading documentation

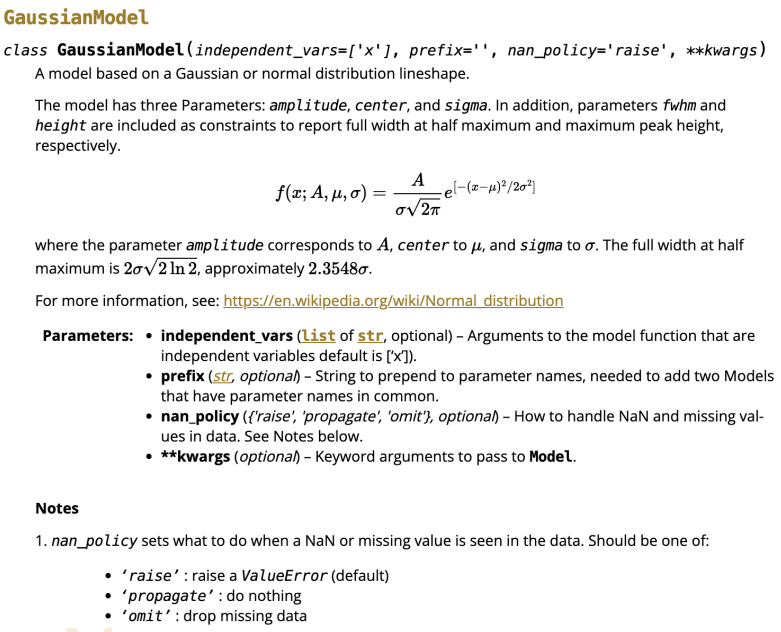
"Alright, I have to run!" Your labmate says, grabbing their messenger bag. "Remember, you know what you need to in order to read the documentation for **lmfit**. Here is a hint, start by Googling what you already know: 'lmfit linear model'."

Following your labmate 's advice, you find a page titled 'Built-in fitting models of the models module.' Just a short time ago, this would have sounded like nonsense to you, but now you know that a **module** is just part of a **library**, and so the page is describing how that part of the lmfit library works. Scrolling through this page, you see the linear model you have been using. And you also see a Gaussian model! The documentation as of the writing of this book is reproduced in Figure 6.1.

Reading this documentation, you find that the model is actually a **class**. You also find a discussion of the **parameter**s used by the Gaussian model function. You find that they are all optional, but decide that you will want to supply the `independent_vars` (the x-values), just as you did when using the linear model. You also see other optional parameters, but aren't totally sure what they are used for now. What you don't see is anything about how to pass the parameters that you want to fit (which the model says are amplitude, center, and sigma). So, next you need to find this, as well as how to combine models.

6.1.7 Combining `lmfit` models

Scrolling more on the same page with the documentation for the GaussianModel, you find a section called "Example 3: Fitting Multiple Peaks—and using Prefixes" (Figure 6.2). The first part sounds exactly like what



GaussianModel

class **GaussianModel**(*independent_vars=['x']*, *prefix=''*, *nan_policy='raise'*, ***kwargs*)

A model based on a Gaussian or normal distribution lineshape.

The model has three Parameters: *amplitude*, *center*, and *sigma*. In addition, parameters *fwhm* and *height* are included as constraints to report full width at half maximum and maximum peak height, respectively.

$$f(x; A, \mu, \sigma) = \frac{A}{\sigma\sqrt{2\pi}} e^{[-(x-\mu)^2/2\sigma^2]}$$

where the parameter *amplitude* corresponds to A, *center* to μ, and *sigma* to σ. The full width at half maximum is $2\sigma\sqrt{2\ln 2}$, approximately 2.3548σ.

For more information, see: https://en.wikipedia.org/wiki/Normal_distribution

Parameters:
- **independent_vars** (list of str, optional) – Arguments to the model function that are independent variables default is ['x']).
- **prefix** (*str, optional*) – String to prepend to parameter names, needed to add two Models that have parameter names in common.
- **nan_policy** (*{'raise', 'propagate', 'omit'}, optional*) – How to handle NaN and missing values in data. See Notes below.
- ****kwargs** (*optional*) – Keyword arguments to pass to **Model**.

Notes

1. *nan_policy* sets what to do when a NaN or missing value is seen in the data. Should be one of:
   - *'raise'* : raise a *ValueError* (default)
   - *'propagate'* : do nothing
   - *'omit'* : drop missing data

Figure 6.1: Screenshot of the online documentation for the GaussianModel **class** within the lmfit **library**.

you are wanting to do, and the second part references "prefixes," which you recall is one of the optional **parameter**s of the Gaussian Model. Looking at the other built-in models described on the page, you see that "prefix" is an optional parameter for all of them. Looking over the text of the example, you find that you can take existing models and add them together to make new models! Specifically, on line 24, you see: `mod = gauss1 + gauss2 + exp_mod`, and you note that `gauss1`, `gauss2`, and `exp_mod` are all variables that were given to calls of the built-in models. So they appear to be adding together models to make a combined model that has 2 Gaussians and an exponential background.

You also notice that, while the `guess()` **method** of the `ExponentialModel` was used to get initial **parameter** values for that model (and store them in a variable), this was not done for the Gaussian models. Instead, it seems these were added directly to the same variable using what looks to be a **method**: `update()`. You think this is a method, because it follows the **syntax** for using methods `<object>.method()`. Given that this is holding parameters to be fit, you suspect this must be something like a 'parameter object.'

You notice on the top of the web page that there is a link to a page called "Parameters" and so you click on that and find out that there is, indeed, a Parameters **class** (Figure 6.3 shows part of this documentation), which means there are objects associated with it. Thus, the `guess()` method created a parameter object, that `update()` added to.

Looking at Example 3 more (Figure 6.2), you find that parameter values were supplied using the `make_params()` method of the Gaussian model object. That is, first a Gaussian model object is created (and assigned to either the variable `gauss1` or `gauss2`) and then `make_params()` was used on this to set values of the parameters to be fit (`center`, `sigma`, and `amplitude`).

Comparing this to the description of the `GaussianModel` (Figure 6.1), you see that Gaussian model has three parameters: amplitude, center, and sigma, and that these are all used as the name of keyword arguments in the `make_params()` method in the example (Figure 6.2). Moreover, each of these keyword arguments accepted

Example 3: Fitting Multiple Peaks – and using Prefixes

As shown above, many of the models have similar parameter names. For composite models, this could lead to a problem of having parameters for different parts of the model having the same name. To overcome this, each **Model** can have a `prefix` attribute (normally set to a blank string) that will be put at the beginning of each parameter name. To illustrate, we fit one of the classic datasets from the NIST StRD suite involving a decaying exponential and two Gaussians.

```
# <examples/doc_builtinmodels_nistgauss.py>
import matplotlib.pyplot as plt
import numpy as np

from lmfit.models import ExponentialModel, GaussianModel

dat = np.loadtxt('NIST_Gauss2.dat')
x = dat[:, 1]
y = dat[:, 0]

exp_mod = ExponentialModel(prefix='exp_')
pars = exp_mod.guess(y, x=x)

gauss1 = GaussianModel(prefix='g1_')
pars.update(gauss1.make_params(center=dict(value=105, min=75, max=125),
                               sigma=dict(value=15, min=0),
                               amplitude=dict(value=2000, min=0)))

gauss2 = GaussianModel(prefix='g2_')
pars.update(gauss2.make_params(center=dict(value=155, min=125, max=175),
                               sigma=dict(value=15, min=0),
                               amplitude=dict(value=2000, min=0)))

mod = gauss1 + gauss2 + exp_mod

init = mod.eval(pars, x=x)
out = mod.fit(y, pars, x=x)

print(out.fit_report(correl_mode='table'))
```

Figure 6.2: Screenshot of the online documentation for the lmfit **library**, showing an example of how different model **object**s can be combined.

The `Parameter` class

class **Parameter**(*name*, *value=None*, *vary=True*, *min=-inf*, *max=inf*, *expr=None*, *brute_step=None*, *user_data=None*)

A Parameter is an object that can be varied in a fit.

It is a central component of lmfit, and all minimization and modeling methods use Parameter objects.

A Parameter has a *name* attribute, and a scalar floating point *value*. It also has a *vary* attribute that describes whether the value should be varied during the minimization. Finite bounds can be placed on the Parameter's value by setting its *min* and/or *max* attributes. A Parameter can also have its value determined by a mathematical expression of other Parameter values held in the *expr* attribute. Additional attributes include *brute_step* used as the step size in a brute-force minimization, and *user_data* reserved exclusively for user's need.

After a minimization, a Parameter may also gain other attributes, including *stderr* holding the estimated standard error in the Parameter's value, and *correl*, a dictionary of correlation values with other Parameters used in the minimization.

Parameters:
- **name** (*str*) – Name of the Parameter.
- **value** (*float, optional*) – Numerical Parameter value.
- **vary** (*bool, optional*) – Whether the Parameter is varied during a fit (default is True).
- **min** (*float, optional*) – Lower bound for value (default is `-numpy.inf`, no lower bound).
- **max** (*float, optional*) – Upper bound for value (default is `numpy.inf`, no upper bound).
- **expr** (*str, optional*) – Mathematical expression used to constrain the value during the fit (default is None).
- **brute_step** (*float, optional*) – Step size for grid points in the *brute* method (default is None).
- **user_data** (*optional*) – User-definable extra attribute used for a Parameter (default is None).

Figure 6.3: Screenshot of the online documentation of the Parameter **class** for the lmfit **library**.

a dictionary, where `value` was always specified, and sometimes `min` and `max`.

Going back to the parameter page (Figure 6.3) you decide the **dictionary** was specifying the starting value (`value`), the lower bound for the fit range (`min`), and the maximum value for the fit range (`max`). You can see there are other arguments that could be supplied, such as if you want the parameter to vary at all (`vary`), but for now, you think this is probably all you need.

Reading this, you also realize that what `make_params()` must be doing is taking the dictionary for each parameter, and then passing this to the `pararmeter()` function to define each parameter.

The last thing from the example (Figure 6.2) that you notice is that you can directly use the `fit()` method on the composite model (assigned to the variable `mod`). So, you now think you now understand how to set up and carry out the fit.

There remains one last problem: you want to get the values of the parameters of the final fit—specifically the center positions and their uncertainties. You could go through the same process as above, but in the interest of space, we present the following information, assuming a usage like: `result = <model>.fit()`, so that the final result is stored in a variable called "result":

- The final parameter values are an **attribute** of the fit result object. So we can get all parameter information using `result.params`
- You can treat the above attribute like a dictionary, so if you want information on a single parameter, you can use `result.params["<parameter name>"]`
- The above contains all the accessible information about that parameter, and you can access the value of the parameter and its uncertainties using the attributes `value` and `stderr`, respectively

With that information, you can sit down and both fit and extract the information from that fit.

6.2 Code

You start by drawing up a flowchart for the process you want to code.

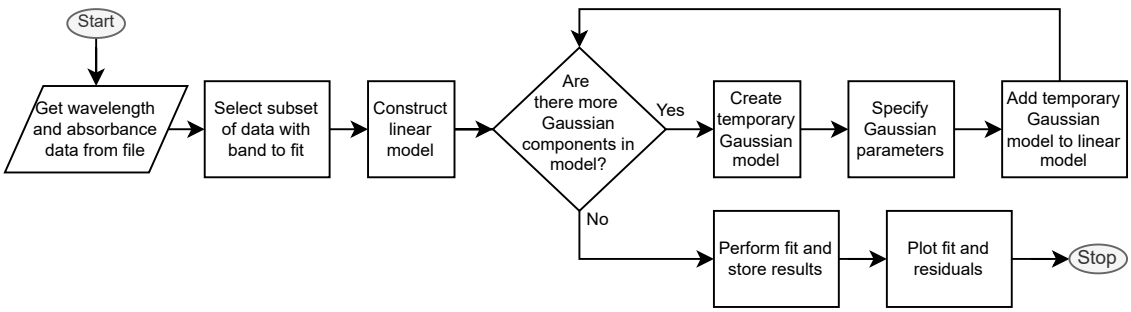


Figure 6.4: Flowchart for the code that extracts spectral data from a csv file, builds up a model of this spectrum using multiple Gaussian line shapes and a linear background, and then fits this model to the data, displaying the results.

```
'''
Fit data to multiple gaussian components and a linear background
    Requires: a .csv file with col 1 as wavelength and col 2 as intensity
    Written by: Ben Lear and Chris Johnson (authors@codechembook.com)
    v1.0.0 - 250207 - initial version
```

```
'''
import numpy as np
from lmfit.models import LinearModel, GaussianModel
from codechembook.quickPlots import plotFit
import codechembook.symbols as cs
from codechembook.quickTools import quickOpenFilename, quickPopupMessage

# Ask the user for the filename containing the data to analyze
quickPopupMessage(message = 'Select the file 0.001.csv')
file = quickOpenFilename(filetypes = 'CSV Files, *.csv')

# Read the file and unpack into arrays
wavelength, absorbance = np.genfromtxt(file, skip_header=1, unpack = True,
    delimiter=',')

# Set the upper and lower wavelength limits of the region of the spectrum to
    analyze
lowerlim, upperlim = 450, 750

# Slice data to only include the region of interest
trimmed_wavelength = wavelength[(wavelength >= lowerlim) & (wavelength < upperlim
    )]
trimmed_absorbance = absorbance[(wavelength >= lowerlim) & (wavelength < upperlim
    )]

# Construct a composite model and include initial guesses
final_mod = LinearModel(prefix='lin_') # start with a linear model and add more
    later
pars = final_mod.guess(trimmed_absorbance, x=trimmed_wavelength) # get guesses
    for linear coefficients
c_guesses = [532, 580, 625] # initial guesses for centers
s_guess = 10 # initial guess for widths
a_guess = 20 # initial guess for amplitudes
for i, c in enumerate(c_guesses): # loop through each peak to add corresponding
    gaussian component
    gauss = GaussianModel(prefix=f'g{i+1}_') # create temporary gaussiam model
    pars.update(gauss.make_params(center=dict(value=c), # set initial guesses for
     parameters
                                  amplitude=dict(value=a_guess, min = 0),
                                  sigma=dict(value=s_guess, min = 0, max = 25)))
    final_mod = final_mod + gauss # add each peak to the overall model

# Fit the model to the data and store the results
result = final_mod.fit(trimmed_absorbance, pars, x=trimmed_wavelength)

# Create a plot of the fit results but don't show it yet
plot = plotFit(result, residual = True, components = True, xlabel = 'wavelength /
    nm', ylabel = 'intensity', output = None)

# Add best fitting value for the center of each gaussian component as annotations
for i in range(1, len(c_guesses)+1): # loop through components and add
    annotations with centers
```

```
48      plot.add_annotation(text = f'{result.params[f'g{i}_center'].value:.1f} {cs.
        math.plusminus} {result.params[f'g{i}_center'].stderr:.1f}',
49                          x = result.params[f'g{i}_center'].value,
50                          y = i*.04 + result.params[f'g{i}_amplitude'].value / (
        result.params[f'g{i}_sigma'].value * np.sqrt(2*np.pi)),
51                          showarrow = False)
52
53  plot.show('png') # show the final plot
```

You run this code, select the file 0.001.csv from the folder Titration/UVvis in your data directory, and wait for about a second. It produces the output shown in Figure 6.5.

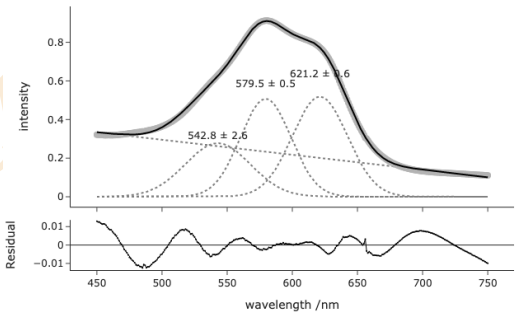


Figure 6.5: A plot of the area of the spectrum saved as 0.001.csv, including indications of the background and the three Gaussian components of the fit and their centers. The key new concepts demonstrated here are the construction of a composite fit model and the use of methods and attributes of the `lmfit ModelResult` object.

6.2.0 Line-by-line explanation

8–11: We import the libraries (or parts of libraries) we will be using. New to this chapter are: `GaussianModel()` (used to fit Gaussian lineshapes to data), `plotFit()` (which we will use to plot the final result of our fitting), and `quickPopupMessage()` (used to generate messages to remind the user of what they need to do next).

14: We use the function `quickPopupMessage()` to tell the user what we want them to do. This will bring up a native operating system message box that shows the message. Including such helpful hints can be nice for users who might wait a long time between using the code, and so might forget what they are supposed to be doing. In this case, we want them to select the 0.001.csv file, since that is the data that this fit is going to be set up to handle. If you want, you can try a different file, and see what happens. By the time you get to the final titration point, this code will fail, since the initial guesses and bounds provided for the parameters below will not work for that data.

15–16: We identify the Path to the data we wish to fit, using the `quickOpenFilename()` function to specify that we can only open up csv files. As a reminder, this will return a pathlib Path **object**.

18: Using Numpy's `genfromtxt()` function, we read the data file, and return to **Numpy array**s that have our `x` (wavelength) and `y` (absorbance) values of interest.

21: We define the lower and upper limits for our fitting. This is the range of wavelengths we want to consider in our fitting.

24: We mask the wavelength array such that we only get values that are greater than or equal to our lower limit and less than our upper limit, and assign the resulting sliced Numpy array to a new variable.

25: We can use the same mask expression and apply it to the absorbance values. Thus, we use the logic test on the wavelength arrays (which is the only place it makes sense, given the limits we specify). This logic test tells us which indices to include. But, since we want the same indices kept in the absorbance array, we can just apply the same mask to it. The end result is that we have a trimmed absorbance that matches the trimmed wavelength Numpy array.

28: Our final model is going to be the sum of three Gaussians and a linear background. We will see below that we can simply add together model objects. So, to start with, we create a linear model object, just as we did in the last chapter. There is one difference: we now have used the `prefix` keyword argument. This will append the string that is provided to the front of all parameters. Though not truly needed quite yet, we will find this behavior useful as we build up the multiple Gaussian contributions below.

29: Also, just as we did in the last chapter, we have Python provide guesses for the parameters of the linear model.

30: We are going to have three Gaussian components. It is possible to ask Python to guess the starting parameters, just as we did for the linear model. However, when we have three closely overlapping Gaussians the guesses may not be good, and it is useful to learn how to provide guesses for the parameters manually. Here we create a list that contains the guesses we will use for the center positions of the three Guassians. This will allow us to iterate through them.

31: We will use a single guess for all three widths, which we assign to a variable.

32: This variable holds a number we will use as a guess for the amplitude for all three Gaussians. Though we don't anticipate all three Gaussians to be the same amplitude, starting with the same amplitude will be a good enough guess to start the fitting.

33–38 This is the **`for` loop** that will allow us to iterate though our guesses for the center of the Gaussians. We also use `enumerate()` function in order keep track of the interation number, which also happens to correspond to the index for each element we have retrieved within the loop. Basically, for each Gaussian, we'll make a temporary model object with only one Gaussian model, set its parameters, and then add it to the final composite model.

- 34: We produce a Gaussian model object. We again use the `prefix` keyword argument to supply a string that is prepended on all parameter names. We use an f-string to provide a prefix that starts with a g (for Gausian) and then has a number associated with it, based on the iteration we are on within the loop. We add 1 to this, so that we are going to number our Gaussians g1, g2, and g3.

- 35–37: We explicitly add parameters to our Gaussian model. We do this by nesting the `make_params()` method of the model object inside the `update()` method of the parameter object we created in line 29. Within the `make_params()` method, we specify the name of the parameter, and then several other items within a dictionary. We have to at least provide a value for the starting guess, which we do using the `value` key.

  - 35: We assign the values for the center of the Gaussian distribution, using the value we took from the `c_guesses` list.
  - 36: We set up the `amplitude` parameter, providing a starting value, but also another keyword: `min`. This allows us to specify a value below which the parameter is not allowed to go, when fitting. The starting guess must, of course, be higher than this value. Sometimes it is quite useful to set such limits, because the fitting procedure will vary the parameter value until it fits the data best, without regard to if it makes sense. Setting the lower limit of the amplitude allows us to avoid negative amplitudes, which we do not expect in our UV-vis experiment.
  - 37: We assign the value for the `sigma` parameter (the width of the distribution) using the same approach. However, now we provide `min` and `max`, which specify the lower and upper limits for this parameter, respectively. We are telling the fitting program that it can vary the value of `sigma`, but it cannot ever be smaller than 0 or greater than 25. While there might be a good physical reason to not allow the width of the feature to be less than 0, there is not really an excellent physical justification for this upper limit. Instead, we are setting this, because our chemical intuition tells us that we might expect all three bands to have similar widths, and this prevents one band from having very different widths from the other two. When restricting parameters to ranges, it is important to understand this may result in a non-optimal fit, so use such constraints with caution. Good practice would be to remove the upper limit after the first fit, re-run the fit with the best-fitting values as the initial values to see if you get different results, and assess how different they are.

  It is important to realize that each of the parameters we create above will be prepended with the prefix specified in line 34, as it is added to the parameter object. This will be important later, as we extract information from the fit result object.

- 38: We now add the Gaussian model object we created to the final model object and then assign this to the final model object. In this way, we are ending each `for` loop by adding the newly created Gaussian model to our existing model. By the time all iterations are completed, our final model will consist of 4 components: 1 linear and 3 Gaussian.

41: We use the `fit()` method of the final model object to fit data. As we did last chapter, we supply the y-values, parameters, and x-values to this function, and pass the result of the fit to a variable.

44: We use the `plotFit()` function to plot the results of our fit. We supply the result as a **positional argument** and then specify several keyword arguments that (in turn) tell the function that we want the residuals to be plotted, that we want to show all components of the model (in this case, liner and 3 Gaussians), the title of the x-axis, the title of the y-axis, and we also tell the function that it doesn't need to show the results. The default behavior would have output the results, but we are going to annotate the figure a bit before we show it. The `plotFit()` function also returns the Plotly Figure object it creates, so we can assign this to a variable so that we can continue to modify it.

47–51: We are going to include annotations on the plot, specifically the final center positions of the Gaussians and their uncertainties. To do so, we need access to the values of the individual components of the model, which are instances of Parameter objects that are entries in the dictionary stored in `params` object. Since the **keys** in this dictionary are numbered according to which Gaussian we are working with, it is convenient to get numbers for them. We can generate them using the `range()` function. The Gaussian components start at 1, we start at 1. Since the range function is exclusive for the final number, so we add 1 to the length of the `c_guesses`, which corresponds to the number of Gaussian components we have. Iterating through the prefixes defined with the `range()` function will provide the numbers we need to access the correct dictionary entries. Once we have the corresponding Parameter object, we can use its attributes to get its value (`value`) and uncertainty (`stderr`).

48: We access the values and standard errors (uncertainties) of our parameters in turn, using the numbers that we generated in the `range()` function above. We include these in an f-string that is passed to the text keyword argument for use in the annotation.

49 & 50: Add the annotation to the Plotly figure to specify the `x` and `y` positions of the annotation.

51: Finally, we state that we don't want an arrow drawn for these annotations.

53: We show the final assembled plot.

6.3 Exercises

Targeted exercises

Exercises for Section 6.1.1

0. What do the following conditional expressions evaluate to (`True` or `False`)? First, write down what you think it should be before running the code, then run it to see if you are right. If you were wrong, explain what you did wrong. Suppose the following variables are already defined:
   ```
   trial_number = 4; pH = 2.53; acid = 'HCl'; conc_stock = 1.0 # M
   ```
   a) `pH >= 2`
   b) `acid != 'HF'`
   c) `trial_number > 2`
   d) `pH > 1`
   e) `conc_stock / 1000 < 0.001`
   f) `acid == 'HF'`
   g) `'H' in acid`
1. Write a conditional expression to evaluate the following tests. Use the variable name given in parentheses for your answer. Define the following variables:
   `pKa = 7.6; atomic_number = 19; pH = 4.4; atom_symbol = 'K'; solvent_name = 'Isopropanol';` 'formula = '(CH3)2CHOH'; `solvent_polarity = 3.9;`
   example: Is the pK_a (`pKa`) less than 5.0?: `pKa < 5.0`
   a) Is the atomic number equal to 11?
   b) Is the pH at least 4 but less than 6?
   c) Is the atom symbol 'K'
   d) Is the solvent 'Methanol' or 'Acetonitrile'? (note: do this with a single comparison)
   e) Is the string 'OH' in 'formula'?
2. For the dictionary of solvent properties created in Exercise 0 of Chapter 5, write a code that determines if an alcohol is in the dictionary by testing to see if `'ol'` is in any of the keys. You can do this by getting a list of the keys using `list(<dictionaryname>.keys())` and then iterating through that list.

Exercises for Section 6.1.2

3. Create a sine wave with period of 1 with an `x`-axis that goes from 0 to 20 with 100 elements. Make a plot of this sine wave where positive values are green and negative values are blue using slicing to produce two different `y` arrays and two corresponding `x` arrays that you can plot separately.

Exercises for Section 6.1.3

4. What do the following conditional expressions evaluate to (`True` or `False`)? First, write down what you think it should be before running the code, then run it to see if you are right. If you were wrong, explain what you did wrong. Suppose the following variables are already defined:
   ```
   trial_number = 4;pH = 2.53;acid = 'HCl';conc_stock = 1.0 # M;
   ```
   a) `pH >= 2 and pH < 7`
   b) `acid != 'HF' and acid != 'HBr'`
   c) `trial_number > 2 and acid != 'HCl'`
   d) `pH > 1 and pH < 7`
   e) `acid == 'HCl' or acid == 'HF'`
   f) `(pH > 2 and acid == 'HF') or (pH > 3 and acid == 'HCl')`
5. Write a conditional expression to evaluate the following tests. Use the variable name given in parentheses for your answer. Define the following variables:
   `pKa = 7.6;atomic_number = 19;pH = 4.4;atom_symbol = 'K';solvent_name = 'Isopropanol';`'formula = '(CH3)2CHOH'; `solvent_polarity = 3.9;`
   a) Is the solvent anything other than 'Toluene' or 'Hexanes' and is the polarity less than 5?:
   b) Is the pH greater than 4 and the pK_a greater than 4?
   c) Does the formula contain a methyl group and an alcohol group?
   d) Is the atom potassium and the pH less than 2?

Exercises for Section 6.1.5

6. Modify the code from Chapter 5 to use the `plotFit()` function from codechembook and confirm that it works.

Exercises for Section 6.1.6

7. Read the documentation for the `plotFit()` function and make plots for the code you created for Exercise 6 but with the following changes:
   a) Confidence intervals of 65% and 95%.
   b) Hide the components of the fit.
   c) Hide the residual.
   d) Show the residual but scale it to the same scale as the main plot.
8. Sometimes you may want to have a string representation of a whole array. Read the documentation for `numpy.array2string()` and use it to create a string that represents the sin wave from Exercise 3 rounded to the nearest tenth. Make sure that the output does not contain any newline (`\n`) characters using keywords for the function.

Exercises for Section 6.1.7

9. There are times where the behavior of a system is best described by the sum of the equations for two lines. For instance you could have a reaction mechanism with distinct rate-determining steps, where one dominates at early times and another takes over later. You might also encounter this in thermal expansion of multi-phase materials. Though the model needed to describe such behavior is simple, it would be somewhat challenging to use it in a fit within Excel and extracted parameter estimates and their uncertainties. Write the code to produce a lmfit model that is made up of two linear models.

Exercises for Section 6.1.4

10. If you had two lists `[ 1, 2, 3]` and `[4, 5, 6, 7]` use `enumerate()` to add the first list to the first three elements of the second list.
11. For the list `[5, 1, 9, -3]` construct a new list using a nested `for` loop. This new list must hold only the unique 6 pairwise products.

Comprehensive exercises

12. It is a good idea to check how robust the fitting results are—that is, how different can your initial guesses be, and still get the same final fit values. The fit for the Gaussians used in the final code of this chapter has three kinds of initial guesses: those for the position of the Gaussian, the intensity of the Gaussian, and the width of the Gaussian. For each of these parameter groups:
    a) Change the initial guesses in a way that you still get the same final fit (within the errors of the parameters).
    b) Change the initial guesses in a way that you do not get the same final fit (within the errors of the parameters).
    c) Comment on wow robust this fitting seems to you.
13. Using the final code for the chapter, modify this so that there is no upper bound on the Gaussian width parameter. When you run this version, do you get a different result? If so, discuss why you might, or might not use this upper bound, and if you could justify using it.
14. You think that there may be a small 4^{th} Gaussian contribution in the data that was fit in the final code of this chapter, and it looks like it would be around 500 nm. Change the code to allow for fitting of this 4^{th} peak. Discuss the results of this new fit with respect to the one that was presented at the end of the chapter.
15. Modify the code we produced at the end of this chapter to add annotations that label the peaks of the actual signal (instead of the Gaussian components) with the best fitting peak centers. The annotation should include an arrow that points to this exact position in the data, which can be accomplished directly in the `add_annotation()` method of the Plotly figure.
16. It is technically not a good idea to fit bands to Gaussian functions when your x-axis is in wavelength. You should really be working in an energy unit, since the effects that give rise to peak shapes depend on energy rather than wavelength, and intensities in the two are related by a Jacobian transformation (see Exercise 28 of Chapter 2 for details). Rewrite the code at the end of this chapter to first convert the x-axis to electron volts (eV), then perform the fit in those units. Note: you will need to convert the guesses as well to get a good start. How do the results compare between these two approaches?
17. In Chapter 3, we plotted many of the same spectra that we fit here. Generate a script that automatically fits each of these spectra and produces a single Plotly plot for each spectrum that was fit, each in its own subplot. Do not include the residuals. You may find that this code eventually 'breaks.' This is likely be because the guesses for parameters are no longer good places to start, for later titration points. As a bonus, determine how to change the guesses as the spectra evolve, though this is not necessary for successful completion of this problem.