Kernel: Python 3 (system-wide)

PHAS0029 Session 2: Fitting to arbitrary functions

Updated: 13/01/2025

Intended learning outcomes:

By the end of this session, you should be able to:

- fit data to any arbitrary function using scipy.optimize.curve_fit;
- quantitatively evaluate the goodness of fit;
- reach physical conclusions based on these results.

You should already know how to fit histograms to a Gaussian, and how to use a polynomial to fit a set of data. In this session we will practice performing a fit to an arbitrary function.

In these examples, we'll be looking at whether a Lorentzian or Gaussian functions provide a better fit to some optical lineshape data. However, you can use the same method to fit *any* function, provided you can write a suitable Python function to describe your target "fit" function.

Context for this example

The data we'll use for this session is taken from the Lab 3 Zeeman effect experiment, which some of you will do yourselves in PHAS0051.

The Zeeman effect occurs when a spectral line is split into different components by a magnetic field. The physics of the Zeeman effect will be covered in detail in PHAS0023 "Atomic and Molecular Physics".

The Lab 3 experiment examines how the lines in the emission spectrum of a mercury discharge lamp split under a magnetic field. The student records the spectrum using a CCD camera, which yields data in the form of recorded intensity (in counts per second) vs pixel position (in pixels).

We're not going to be considering the *positions* of the spectral lines in this task, instead we're going to be looking at the *lineshapes*. Rather than a spectral line with a single energy, the line is broadened into a wider peak by various physical effects. For example, the uncertainty principle leads to broadening which has a Lorentzian form, while there will also be thermal broadening effects, which are Gaussian in nature (there are also several other sources of broadening, with different effects). In theory, for this experiment, Lorentzian broadening is expected to dominate.

In this task we will look at an experimental spectral line recorded by a student in the Lab 3 experiment, fit it to both a Gaussian and Lorentzian, and determine which provides a better fit.

Getting started with the code

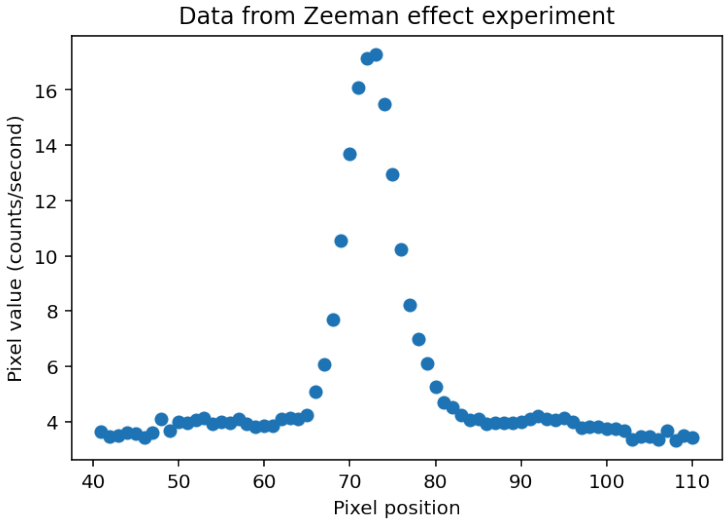
First, we'll import the modules we will need. This includes import of curve_fit from the scipy.optimize library - more on this later, when we come to use it.

In [125]:
```
%matplotlib inline

import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stats
from scipy.optimize import curve_fit # we're importing just this one
function from scipy.optimize

#plt.style.use('dark_background')

plt.rcParams["patch.force_edgecolor"] = True # include outlines on
histograms etc
```

Now we can import the csv (Comma Separated Value) file with the data the student collected, and plot it. In CoCalc the file is saved in the same directory as your notebook and it is called Zeeman_data.csv.

In [126]:
```
# import the data...
xdata,ydata = np.loadtxt('Zeeman_data.csv',delimiter=",",unpack=True) #
reminder: need to set delimiter for csv files

# ...and plot it.
plt.figure()
plt.plot(xdata,ydata, 'o')
plt.xlabel("Pixel position")
plt.ylabel("Pixel value (counts/second)")
plt.title("Data from Zeeman effect experiment");
```

Out[126]:



We can see that we have a single peak with a constant background level. It looks feasible to attempt fitting this to a Gaussian.

In order to use `curve_fit` to fit this to a Gaussian, we need to write a "target" function to fit to, which in this case will be

$$f(x) = y_0 + h \exp\left(\frac{-(x - x_0)^2}{2\sigma^2}\right)$$

(This is a slightly different definition from the one we used when we were fitting histograms to Gaussians in PHAS0028. Can you see why?)

The parameters for our Gaussian fit will be the mean value (`x0`), the standard deviation (`sigma`), the background value `y0` and the peak height, `h` . Here is a function that will do exactly this.

In [127]:
```
def gaussian(x,x0,sigma, y0, h):
    '''Returns a single value or 1D array of Gaussian function values
for
    - input x-value or array of x-values: x
    - mean value of distribution: x0
    - standard deviation of distribution: sigma
    - background value y0
    - peak height, h (measured from background level y0)'''
    gauss = h * np.exp(-(x-x0)**2/(2*sigma**2)) + y0 # the gaussian
itself
    return gauss
```

curve_fit

The three parameters, x0, y0 and sigma, are (as yet) unknown. To find them, we use the scipy.optimize.curve_fit function. The full documentation for this is here: http://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html#scipy.optimize.curve_fit

We're going to do this in the simplest way possible for the moment, by just sending curve_fit the target function (our " `gaussian` " function), the independent variable (`xdata`) and the dependent variable (`ydata`). We can also, optionally, choose to send an initial guess of the parameters, as well as weightings for each of the ydata data points, but for the moment we won't do that.

Recall that the `curve_fit` routine returns two arrays.

- The first of these is an array of the fitted parameters - in our case this array will have four elements, as we have four parameters, `x0` , `sigma` , `y0` , and `h` .
- The second is the matrix of covariance - an indication of the goodness of fit. We covered this in Session 3 when we were doing polynomial fitting.

Let's do this, and see what results we get back:

In [128]:
```
#popt: Optimized parameters
#pcov: matrix of covariance.
popt,pcov = curve_fit(gaussian,xdata,ydata)

print ("popt :\n", popt)
print ("pcov :\n", pcov)
```

Out[128]:
```
popt :
 [1.         1.         5.30499999 1.        ]
pcov :
 [[inf inf inf inf]
 [inf inf inf inf]
 [inf inf inf inf]
 [inf inf inf inf]]
```

```
/usr/local/lib/python3.10/dist-packages/scipy/optimize/_minpack_py.py:1010:
OptimizeWarning: Covariance of the parameters could not be estimated
  warnings.warn('Covariance of the parameters could not be estimated',
```

We can see that this hasn't worked so well - `curve_fit` hasn't been able to find a fit to the data. This is due to the nature of non-linear regression. Since there are mulitple local minima in the error space, the optimisation algorithm may converge to the wrong minimum (local instead of global) or not converge at all if the initial parameters are not optimised. (For more details on this see your notes from PHAS0028 and /or *Measurements And Their Uncertainties: A practical guide to modern error analysis* by I. Hughes)

Instead, we'll try to make life easier for `curve_fit` by giving an initial guess for the parameters. From looking at the plot of the data, we can see that the peak is at around $x = 75$, and the background around $y = 3.5$. We'll try a value of 10 for σ. These values need to be given in the form of a python list of numbers, in the same order as parameters are given to our " `gaussian` " function. Remember - in Python we use `[` square brackets `]` to define a list, with the elements separated by commas.

In [129]:
```
guess = [75,10,3.5,18] # list of initial guess parameters
# what type of object does the variable "guess" represent?
print ("The variable 'guess' is a ", type(guess) )
```

Out[129]:
```
The variable 'guess' is a  <class 'list'>
```

Now we can retry the fit:

In [130]:
```
popt,pcov = curve_fit(gaussian,xdata,ydata,p0=guess)
print ("popt :\n", popt)
print ("pcov :\n", pcov)
```

Out[130]:
```
popt :
 [72.50930905  3.01525268  3.85742572 13.40680375]
pcov :
 [[ 2.02507205e-03 -3.60862663e-10 -3.23142700e-12  1.62635824e-09]
 [-3.60862663e-10  2.22561786e-03 -6.30519921e-04 -4.05620892e-03]
 [-3.23142700e-12 -6.30519921e-04  1.98236679e-03 -1.40174495e-03]
 [ 1.62635824e-09 -4.05620892e-03 -1.40174495e-03  3.10175059e-02]]
```

This has worked (or it should have done)! We can use the information from the matrix of covariance to calculate the error on each parameter, just as we did in the PHAS0028 exercises for the polynomial coefficients. Remember, the error on the parameters are given by the *square roots* of the diagonal elements of the matrix of covariance.

A python aside / hint:

When dealing with an array like `popt` that contains numbers each representing different variables, it's sometimes useful to be able to "unpack" the array into different variables - we've already seen examples of this in the code cell above and in the second code cell when unpacking the data from the file. To unpack `popt`, we could use a line of code like:

```
x0_fit, sigma_fit, y0_fit, h_fit = popt
```

If we wanted to then calculate the fitted line at a given x-value (in this case at x = 65), we could then use something like:

```
fitted_point = gaussian(65, x0_fit, sigma_fit, y0_fit, h_fit)
```

or, if we don't want/need to assign individual variable names to the elements of `popt` (or whichever array we are dealing with), we could use:

```
fitted_point = gaussian(65, popt[0], popt[1], popt[2],
popt[3])
```

This is a bit unwieldy though, so sometimes it's useful to be able to unpack the array automatically when calling a function by using * syntax, like this:

```
fitted_point = gaussian(65, *popt)
```

This is much easier to deal with! You can find a fuller discussion of this in Hill: Learning Scientific Programming with Python section 2.4.3 (page 49).

The code cell below demonstrates that the two methods do give identical results:

In [131]:
```
# specifying the elements by hand:
print("At x = 65 our fitted Gaussian has a value of: ", gaussian(65,
popt[0], popt[1], popt[2], popt[3]))

# use *syntax to unpack the elements of popt automatically:
print("Calculating the same value using * syntax:    ",
gaussian(65,*popt))
print("Both give the same result!")
```

Out[131]:
```
At x = 65 our fitted Gaussian has a value of:  4.460698389042042
Calculating the same value using * syntax:     4.460698389042042
Both give the same result!
```

Fitting the data

Task 1: Fit data to Gaussian

In the cell below, write a function called `g_fit` which will:

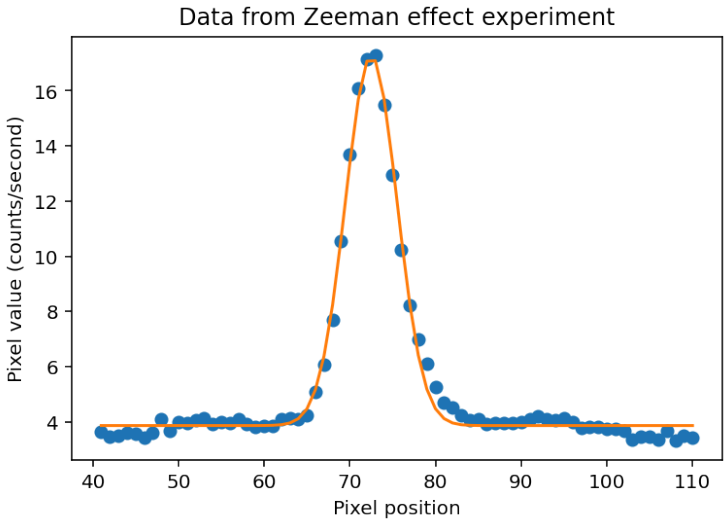
- fit Gaussian to provided x and y data at input based on a guess array (also given as the third input)
- calculate the errors on the fitted parameters
- output fitted parameters and their errors
- plot the original data and the fitted line on a single, appropriately labelled graph.

HINT: make sure you include enough points in your fitted line to give a smooth curve, particularly at the peak. Also, make sure you use the `gaussian` function defined before.
HINT 2: all parameters should be clearly annotated and be output as part of the plot. Click here to learn how to insert a text box into a plot. You may find it easier to define position of the text in figure coordinates, see the `fig.transFigure` method here.

In [132]:

```
def g_fit(x,y,guess):
    """The following function plots data nd fits them with a gaussian
fit
    """


    #plotting figure
    plt.figure()
    plt.plot(x,y, 'o')
    plt.xlabel("Pixel position")
    plt.ylabel("Pixel value (counts/second)")
    plt.title("Data from Zeeman effect experiment");

    #gaussian fit

    popt,pcov = curve_fit(gaussian,x,y,p0=guess)
    gauss = popt[3] * np.exp(-(x-popt[0])**2/(2*popt[1]**2)) + popt[2]
    perr = np.sqrt(np.diag(pcov))
    plt.plot(x,gauss,'-')

    return popt,perr

guess = [75,10,3.5,18]


a,b =g_fit(xdata,ydata,guess)
a,b
```

Out[132]:
```
(array([72.50930905,  3.01525268,  3.85742572, 13.40680375]),
 array([0.0450008 , 0.04717645, 0.04452378, 0.17611788]))
```



Checkpoint 1: Run the next cell to check if your function `g_fit` works correctly.

In [133]:
```
%run -i c2checkpoint1-patch.py
```

Out[133]:
```
************
 Well done! All test passed. You can move to the next part of the task.
************

<Figure size 432x288 with 0 Axes>
```

Task 2: Define Lorentzian

First, write a properly formatted python function called `lorentzian`, similar in form to the `gaussian` function above, that will return a Lorentzian function for these parameters.
HINT: your function should now have 5 inputs but still one output. The expected order of inputs should be x, x_0, b, y_0 and h.

HINT 2: Lorentzian function has a form of $f(x) = h/(1 + ((x - x_0)/b)^2) + y_0$

In [134]:
```
# YOUR CODE HERE
def lorentzian(x,x0,b, y0, h):
    """The following function generates lorentzian fit
    """

    lorentz = h/(1 + ((x-x0)/b)**2) + y0

    return lorentz
```

Checkpoint 2: Run the next cell to check if your function `lorentzian` works correctly.

In [135]:
```
%run -i c2checkpoint2-patch.py
```

Out[135]:
```
************
 Well done! All test passed. You can move to the next part of the task.
************
```

Task 3: Double fit

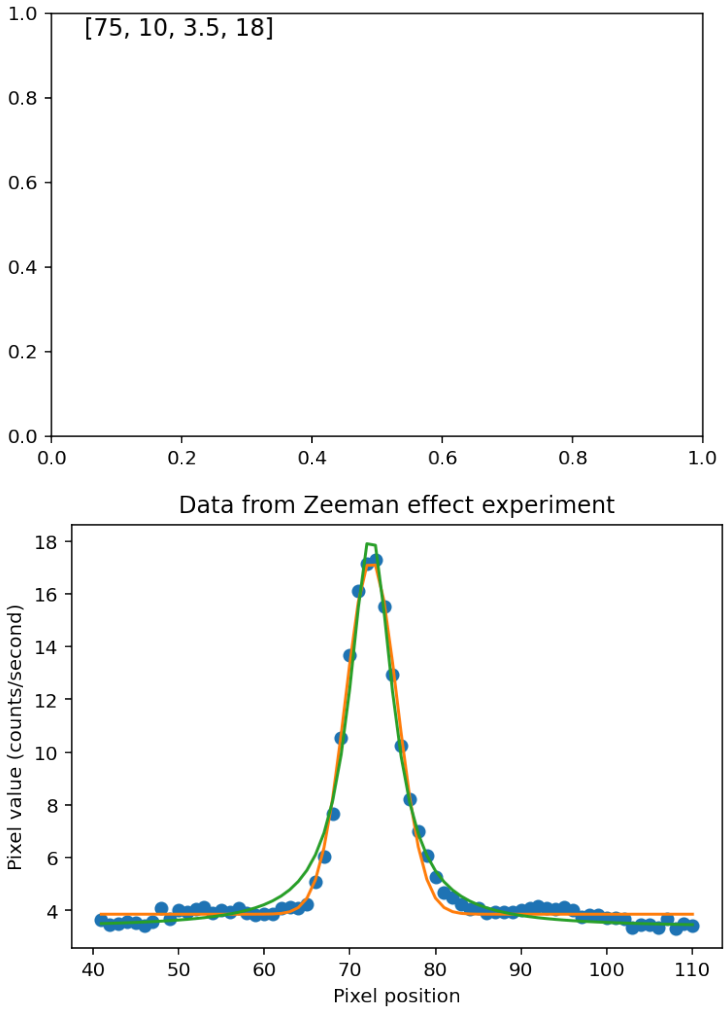
Now write a function named `double_fit` which:

- uses `curve_fit` to calculate the best Gaussian and best Lorentzian fit for input data provided initial guesses for their parameters
- calculates the error on each parameter
- outputs each parameter with its error (as four arrays: gaussian coefficients, their errors, lorentzian coefficients, their errors)
- plots the data, **the fitted Gaussian** and the fitted Lorentzian, all on the same labelled graph. Your graph should contain all parameters with their errors in a suitable format (e.g. the value for $x_0 = 45 \pm 3$)

HINT: you can re-use most of the code from your previous function `g_fit`.
HINT 2: your function will need two arrays of guessed parameters - one for each fit. The expected input should be: x data, y data, guess for Lorentzian and guess for Gaussian.
HINT 3: use plt.text to tidy up your plot.

In [136]:
```
def double_fit(x,y,lorentz_guess,guess):
    """function fits and plots a gaussian fit and a lorentzian"""



    #plotting figure
    plt.figure()
    plt.plot(x,y, 'o')
    plt.xlabel("Pixel position")
    plt.ylabel("Pixel value (counts/second)")
    plt.title("Data from Zeeman effect experiment");

    #gaussian fit

    popt,pcov = curve_fit(gaussian,x,y,p0=guess)
    gauss = popt[3] * np.exp(-(x-popt[0])**2/(2*popt[1]**2)) + popt[2]
    perrt = np.sqrt(np.diag(pcov))
    plt.plot(x,gauss,'-')

    #lorentzian fit
    poptl,pcov1 = curve_fit(lorentzian,x,y,p0=lorentz_guess)
    w = lorentzian(x,*poptl)
    perrl = np.sqrt(np.diag(pcov1))
    plt.plot(x,w,'-')

    return popt,perrt, poptl,perrl

plt.text(0.05,0.95,guess ,fontsize = 12)

lorentz_guess = [72,3.9,3,18]
poptg,perrg,poptl,perrl=double_fit(xdata,ydata,lorentz_guess,guess)
```

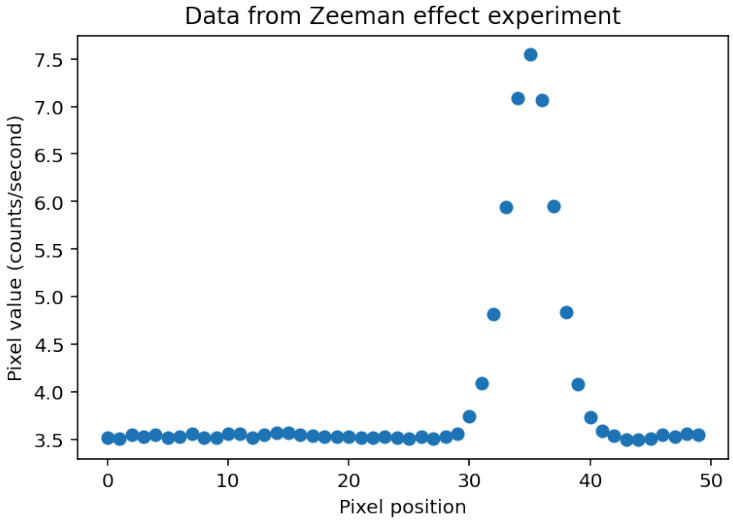
Out[136]:

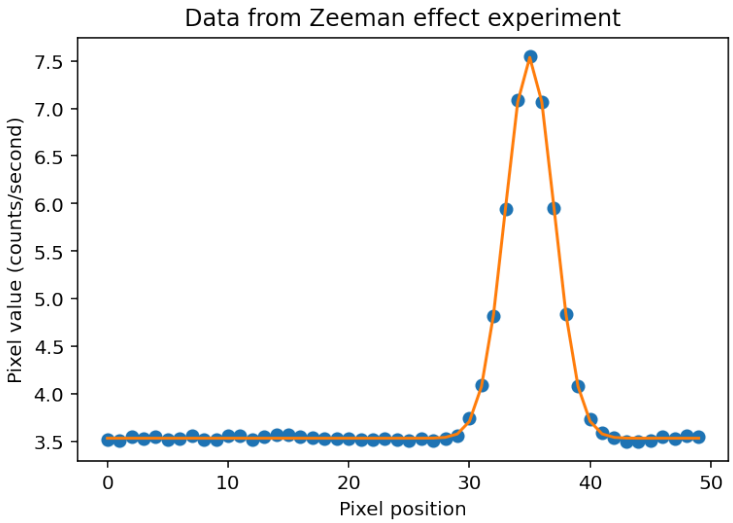


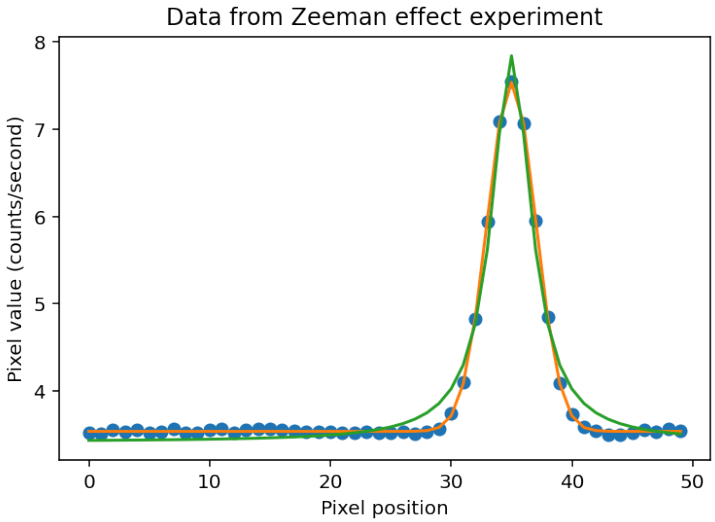
Checkpoint 3: Run the next cell to check if your function `double_fit` works correctly.

In [137]:
```
%run -i c2checkpoint3-patch.py
```

Out[137]:
```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
File ~/Assignment C2 - non-linear regression/c2checkpoint3-patch.py:131
    124 with patch('matplotlib.pyplot.text') as mock_fn:
    125     double_fit(xt, yt, guesst, guesst)
--> 131 assert mock_fn.called, 'Make sure you use the plt.text method to
print your parameters on the plot or next to it.'
    134 ##CHANGES ADDED: plot checking
    135 from unittest.mock import patch
AssertionError: Make sure you use the plt.text method to print your
parameters on the plot or next to it.
```








```
<Figure size 432x288 with 0 Axes>
```

Evaluating the goodness of fit

So now we have two potential fits to our data. Looking at them, it's hard to tell which one provides the better fit. We can do this quantitatively by calculating χ^2 for each fit, in the same way as we did in PHAS0028.

We'll also need to know the y-error on the data points for this - which for this experiment were estimated to be ± 1 counts/second.

Task 4: Chi squared

In the cell below, define a new function `eval_fit` that uses output of your `double_fit` function and:

- calculates the residuals divided by the y-error
- calculates the number of degrees of freedom
- hence calculates χ^2 for the fit
- **works for either Gaussian or Lorentzian fit**

Refer back to the PHAS0028 notes if you need a reminder of any of the definitions. Your function should have **five** inputs: fit function (`lorentzian` or `gaussian`) x- and y-data, fit coefficients and y-errror. The outputs should be: the residuals (normalised), number of degrees of freedom and reduced χ^2.
HINT: set a variable to use for the y-error, then you'll be able to experiment with this value later.

In [138]:
```
def eval_fit(fit_function,x,y,coeff,y_err):
    """calculates the reduced chu-squared"""

    y_fit = fit_function(x,*coeff)
    residuals = (y - y_fit)
    nor_res = residuals/y_err
    square = nor_res ** 2
    chi = np.sum(nor_res ** 2)
```

```
    dof = len(y)- len(coeff)
    red_chi = chi/dof

    return nor_res, dof, red_chi
```

Checkpoint 4: Run the next cell to check if your function `eval_fit` works correctly.

In [139]:
```
%run -i c2checkpoint4-patch.py
```

Out[139]:
```
************
 Well done! All test passed. You can move to the next part of the task.
************

<Figure size 432x288 with 0 Axes>
```

Comparison and discussion

Run your evaluation function for both distributions in the cell below and compare their goodness of fit.
Are these the results you'd expect? Discuss briefly in a text cell.

In [141]:
```
# comparing both fits
a=eval_fit(lorentzian,xdata,ydata,poptl,1.0)
b=eval_fit(gaussian,xdata,ydata,poptg,1.0)
print('For a Gaussian fit the reduced chi squared is ', b[2],'.\nFor a
Lorentzian fit the reduced chi squared is ',a[2])
```

Out[141]:
```
For a Gaussian fit the reduced chi squared is  0.10698194422306931 .
For a Lorentzian fit the reduced chi squared is  0.19277556427136613
```

The reduced chi squared for gaussian is lower than that for Lorentzian fit. This shows that the gaussian fit is a better fit due to the reduced chi squared being of a lower value. This is expected, as looking at the two fits, the gaussian follows the trend of the data points more than the Lorentzian at the peak (Lorentzian peak goes to a much higher max y value than data points), and just as the y-values start dramatically increasing in value.

Task 5: Analysing the residuals

Another way of verifying the validity of our fits is to check the distribution of the residuals, and see if they follow a normal (Gaussian) distribution. Again, follow a similar procedure as we did in the PHAS0028 and check the distribution of the calculated residuals for both fits.

Create a function `fit_residuals` that fits an input array of residuals to a Gaussian distribution using `stats.norm.fit` method and returns x_0, σ and a histogram with a fitted line.

- Make sure your plot contains x_0 and σ in the title.

In [167]:
```
def fit_residuals(resid):
    """Plots residuals"""



# do the fit for the highest order polynomial we calculated
```

```
# others will give similar results
    x0,sigma = stats.norm.fit(resid) # fit the residuals to a normal
distribution
    xvals = np.linspace(-3,3,100) # generate array of x-values
    gaussian = 1/(np.sqrt(2*np.pi)*sigma) * np.exp(-(xvals-
x0)**2/(2*sigma**2)) # generate line for fitted gaussian
# gaussian = stats.norm.pdf(x,x0,sigma) # alternative / easier way of
generating line

# plot the histogram of the residuals and the fitted Gaussian
    plt.figure()
    title_label=(r'Residuals fitted with Gaussian $x_0$ = {0:3.2e},
$\sigma$ = {1:3.2e}'.format(x0,sigma))
    plt.suptitle(title_label)
    plt.xlabel("x ")
    plt.ylabel("residual")
    plt.hist(resid,bins=10,density=True,alpha=0.25)
    plt.plot(xvals,gaussian);

    return x0, xvals
```

Checkpoint 5: Run the next cell to check if your function `fit_residuals` works correctly.

In [168]:
```
%run -i c2checkpoint5-patch.py
```

Out[168]:
```
************
 Well done! All test passed. You can move to the next part of the task.
************

<Figure size 432x288 with 0 Axes>
```

Drawing conclusions

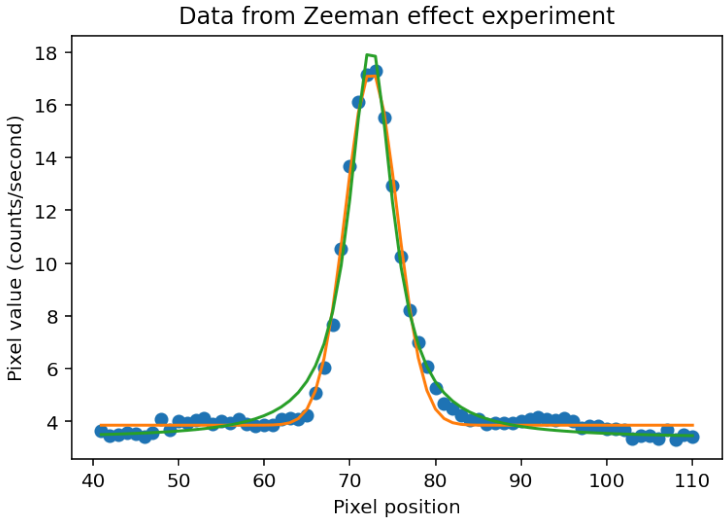
Task 7: Final discussion

Use the cell below to provide a concise summary of the fits. Try changing your value of the yerror in the data (that you used to calculate the χ^2) to the σ you obtain here. What does this tell you?
Use a text cell to discuss what you conclude from these results.

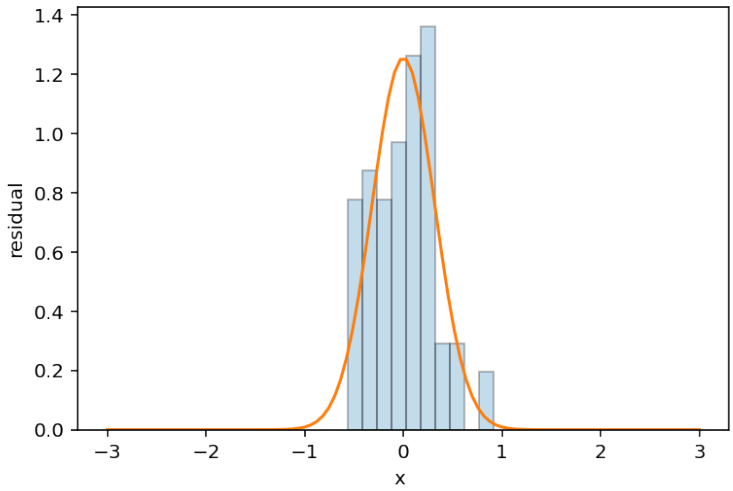
In [171]:
```
#code combining all functions from this notebook to analysie the Zeeman
data
#define initial variables
yerr==1
lorentz_guess = [72,3.9,3,18]
guess = [75,10,3.5,18]

#perform and evaluate fits
poptg,perrg,poptl,perrl=double_fit(xdata,ydata,lorentz_guess,guess);
resl,dofl,chi2l=eval_fit(lorentzian,xdata,ydata,poptl,yerr);
resg,dofg,chi2g=eval_fit(gaussian,xdata,ydata,poptg,yerr);
plt.show() #this line forces the plots from double_fit to be printed
before the next line; try removing it and see what changes!


fit_residuals(resg);
```

```
plt.show()

fit_residuals(resl);
```

Out[171]:



Residuals fitted with Gaussian x_0 = -4.49e-10, σ = 3.18e-01



Residuals fitted with Gaussian x_0 = 3.73e-11, σ = 4.26e-01