Introduction to Computational Physics (PHYS105)

Lecture 6: Fitting Data

Carl Gwilliam

(C.Gwilliam@liverpool.ac.uk)

UNIVERSITY OF LIVERPOOL

Attendance code: 283985

Lecture 5: Recap

- Last week we looked at how to input and output info
  - Either to/from a user or a file
- We saw two ways of nicely formatting strings
  - f-strings or the `format()` function
  - Allows to control width, type, precision etc
- We saw how to ask the user for `input()`
  - Don't forget the result is a string by default
  - May need converting to `int`, `float`
- We saw how to `open()` and `close()` files
  - And how to read() from and `write()` to them
  - Can also read line-by-line using a `for` loop
- Can combine to e.g. read in data and make table

```
# Print out the header
print ("| {:12s} | {:10s} | {:10s} | {:10s} | {:10s} |".
       format("Measurement", "$x$ value", "$x$ error",
              "$y$ value", "$y$ error"))

# Can use times to print a string multiple times
print ("-"*68)

# Open the file
f = open("fitdata.csv")

# Print out the data, tabulated
imeas = 0
for line in f:
    imeas = imeas + 1
    # strip() removes \n from end of line
    # split(",") separates parts of the string into a list
    data = line.strip().split(",")
    print (f"| {imeas:<12d} | {float(data[0]):^10.2f} "
           f"| {float(data[1]):^10.2f} | {float(data[2]):^10.1f} "
           f"| {float(data[3]):^10.1f} |")

# Don't forget to close the file
f.close()
```

```
| Measurement  | $x$ value  | $x$ error  | $y$ value  | $y$ error  |
--------------------------------------------------------------------
| 1            |    1.50    |    0.21    |    14.3    |    2.1     |
| 2            |    2.31    |    0.11    |    20.2    |    1.7     |
| 3            |    2.78    |    0.43    |    30.1    |    3.3     |
| 4            |    3.58    |    0.13    |    36.5    |    1.1     |
| 5            |    4.08    |    0.17    |    42.7    |    0.9     |
| 6            |    4.76    |    0.18    |    47.1    |    1.1     |
| 7            |    5.62    |    0.15    |    52.9    |    1.5     |
| 8            |    6.02    |    0.19    |    78.8    |    0.9     |
| 9            |    8.45    |    0.17    |    85.2    |    1.2     |
| 10           |    9.65    |    0.11    |    99.4    |    2.9     |
```

Lecture 5: Formative problem solutions

- Exercise 1: Formatting number of eggs
  - Use curly braces
    - f-string: value entered in-place
    - format: value entered via function call
  - Format specifier after colon. Here `d` (rather than `.0f` as some people put)

```
dozen = 12
print(f"The number of eggs in a dozen ({dozen:d}) is more than ten.")
print("The number of eggs in a dozen ({:d}) is more than ten.".format(dozen))
```

```
The number of eggs in a dozen (12) is more than ten.
The number of eggs in a dozen (12) is more than ten.
```

- Exercise 2: Day and number of week, aligned in columns
  - Using tab characters (`\t`)
    - A bit trial-and-error
  - Format function
    - Specify exact column width
    - String (`s`) as type
  - f-string
    - Store values as variables or provide directly using different level of quotes

```
print("Days of the week:")
print("One \t\t two \t\t three \t\t four \t\t five \t\t six \t\t seven")
print("Monday \t\t Tuesday \t Wednesday \t Thursday \t Friday \t Saturday \t Sunday")
```

```
print("Days of the week:")
print("{:10s}{:10s}{:10s}{:10s}{:10s}{:10s}{:10s}".
      format("One", "Two", "Three", "Four", "Five", "Six", "Seven"))
print("{:10s}{:10s}{:10s}{:10s}{:10s}{:10s}{:10s}".
      format("Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"))
```

[5]:
```
print("Days of the week:")
print(f"{'One':10s}{'Two':10s}{'Three':10s}{'Four':10s}{'Five':10s}{'Six':10s}{'Seven':10s}")
print(f"{'Monday':10s}{'Tuesday':10s}{'Wednesday':10s}{'Thursday':10s}{'Friday':10s}{'Saturday':10s}{'Sunday':10s}")
```

Lecture 5: Formative problem solutions (2)

- Exercise 3: fix the error in the code
  - By default input variables are read as strings
  - But subtracting a string from a number doesn't make sense → you need to convert them to `int` to be able to do maths with them.

```
retirementAge = 67

name = input("What's your name?")
print("Nice to meet you " + name + "!")
age = int(input("How old are you?"))

retireIn = retirementAge - age
if retireIn > 0:
    print("I guess you will retire in",retireIn,"years,",name)
else:
    print("I guess you retired",-retireIn,"years ago,",name)
```

- Exercise 4: check age range 3 times
  - Use a `while` (or `for`) loop to ask for input 3 times
    - Don't forget to increment the count
  - `If` in the correct range you can just `break`
    - Won't run rest of loop so doesn't ask again
    - Otherwise, it will go round the loop asking again
  - After the while loop we check the final count
    - If count <= 3, a correct value has been entered → calculate the retirement age and print out
    - Else print an error message
  - Some people did `if age in range(0,120)`
    - Works but inefficient

```
retirementAge = 67
name = input("What's your name?")
print("Nice to meet you " + name + "!")

count = 1
maxCount = 3
while count <= maxCount:
    age = input("How old are you?")
    if int(age) > 0 and int(age) < 120:
        break

    count = count + 1

if count <= maxCount:
    retireIn = retirementAge - int(age)
    if retireIn > 0:
        print("I guess you will retire in",retireIn,"years,",name,"\b.")
    else:
        print("I guess you retired",-retireIn,"years ago,",name,"\b.")
else:
    print("Too many incorrect inputs!")
```

Lecture 5: Formative problem solutions (3)

- Write out table of numbers, with squares and cubes, to a file and then read it back in
  - Open the file
  - Use the write function to output the string both for the header and each of the lines
    - Don't forget the new line (`\n`)
  - Loop over the `range` of numbers using `for`
  - Format with either f-string or format function
    - Specifying integer (`d`) and width (here 6)
  - Open the file again and call `read()`
  - Don't forget to `close()` the file each time

```
ftable = open("table.txt", "w")

nMax = 10
nMin = 0

ftable.write(f"{'Number':>6s} {'Square':>6s} {'Cube':>6s}\n")
for n in range(nMin, nMax):
    ftable.write("{:6d} {:6d} {:6d}\n".format(n, n**2, n**3))

ftable.close()

ftablein = open("table.txt", "r")
print(ftablein.read())
ftablein.close()
```

```
Number Square   Cube
     0      0      0
     1      1      1
     2      4      8
     3      9     27
     4     16     64
     5     25    125
     6     36    216
     7     49    343
     8     64    512
     9     81    729
```

The course so far …

- So far we have been focusing on learning how to use the basics of python.

- We have seen ….
  - How to use jupyter notebooks to write python programs
  - How to store data in data structures, particularly NumPy arrays
  - How to use functions, either existing ones or our own, to analyse the data
  - How to control the flow of our programs via loops and conditional expressions
  - How to make plots to visualise our data using matplotlib
  - How to deal with input and output

- From now on we are going to focus on pulling these together to analyse physics problems
  - This will allow you to practice your python skills

- We will see how to break down the problems so that we can solve them computationally
  - In doing so, we will introduce a new python module: SciPy
  - We will also look at how to debug issues in more depth (next week)

How confident do you feel with programming in python?

Not confident at all - I am still struggling to understand the basics

0%

Somewhat confident - I understand the majority of the python features we have covered

0%

Confident - I understand the features and am happy writing simple programs

0%

Very confident - I am happy bring together different features into more complex programs

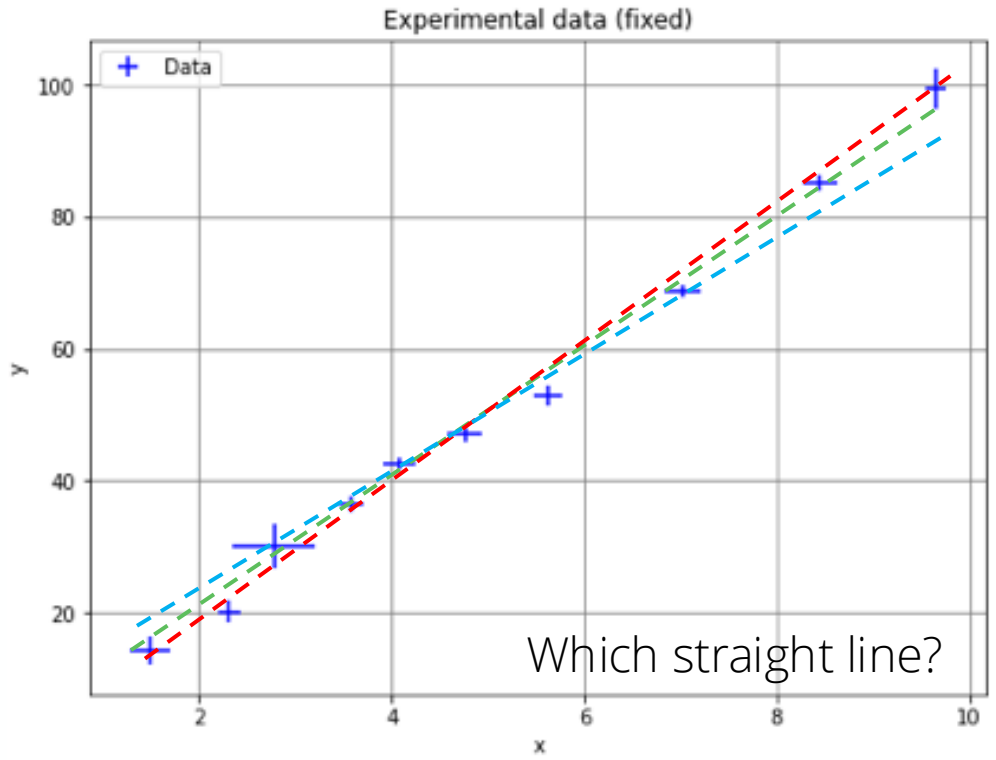
0%

Tackling physics problems

- There are many different types of physics problem, which differ in the approach & tools needed
  - We will look at a range of problems in the coming weeks to develop the techniques needed
- Fitting a functional form to data to extract the best-fit parameters (this lecture)
- Numerical solutions of equations (lecture 8+9)
  - Finding the roots
  - Integrating and differentiating
  - Solving differential equations
    - E.g. projectile motion with air resistance
- Using random numbers to produce Monte Carlo models (lecture 10)
  - Integration e.g. calculating π
  - Simulation e.g. bed availability in a hospital, motion of gas particles
- Finally, we will see how to bring your code together in your own reusable module (lecture 11)
  - That you can then use in labs to fit data going forward

Line fitting

- When we take experimental data, we usually want to determine the relationship between the dependent variable, y, and the independent variable, x: $y = f(x)$
  - E.g. straight line, polynomial, exponential, power law, …
- Often we know the expected functional form, for example from an underlying theory
  - But we have some unknown parameters β: $y = f(x, \beta)$
- In order to find these, we need to "fit" the expected functional form to the data to find out:
  - If the expected function form is compatible with our data, within the measurement uncertainties
  - If so, what are the values of the free parameters that best describe the data we have taken
- So, how do we fit the function to the data?
  - E.g. which line is the "best fit" to these points?
  - First, we need to quantify the "best fit"



Finding the best fit (1)

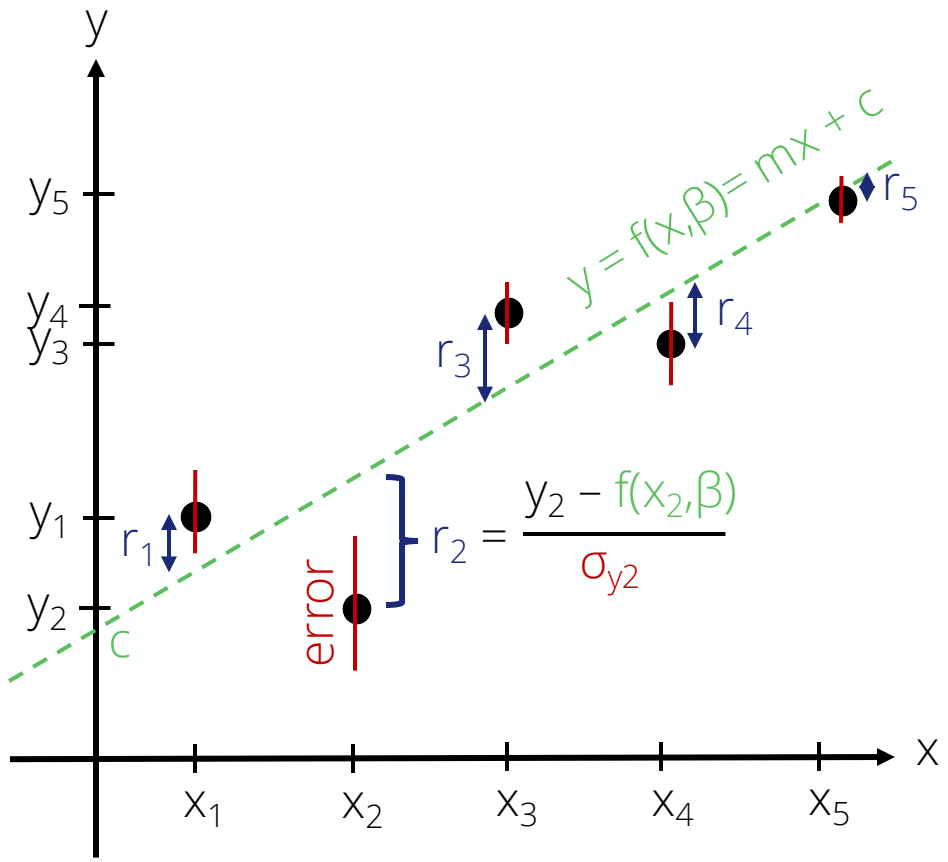
- In simple terms, the best fit is the one that lies closest to the set of data points overall
  - This is where the sum of the distance between the prediction and each data point is smallest
  - This distance is called the "residual" and for each point i is given by $r_i = y_i - f(\beta, x_i)$

- The most common method is the least-squared method
  - So-called because it minimises the squares of the residuals

$$S = \sum_{i=0}^{N} r_i^2 = \sum_{i=0}^{N} [y_i - f(\beta, x_i)]^2$$

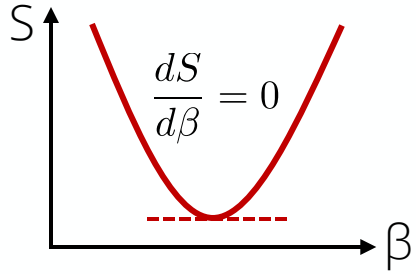
- The above formula assumes that we measure the data points exactly, but in reality, they have an uncertainty σ_{yi}
  - The larger the uncertainty the less we trust that point → points with smaller error have a larger weight
  - We take this into account by measuring the residuals in terms of the size of the uncertainty on the point

$$S = \sum_{i=0}^{N} r_i^2 = \sum_{i=0}^{N} \left[\frac{y_i - f(\beta, x_i)}{\sigma_{y_i}} \right]^2$$



Finding the best fit (2)

- Now that we have a measure of how good a fit is (S) we want to find the best fit
  - By varying the free parameters β to get the minimal value of S



- In principle, this is simple: differentiate wrt β and set to zero: dS/dβ = 0
  - And for a straight line, y = mx + c, this can be done analytically
- But, in general, there is no analytical solution, so we need to use a numerical method
  - Starting from an initial guess of the solution and then iterating to improve
- We can use the third-party Scientific Python (SciPy) library to do this
  - Provides a wide range of mathematical & scientific routines
  - Works with NumPy arrays, of course!

- Includes an optimisation module that provides a least_squares fitting routine
  - (We will see in the later weeks how simple numerical methods work)

Least-squares: input

- SciPy's least_squares function can take many arguments, but for our needs it looks like

```
scipy.optimize.least_squares(fun, x0, args=())
```

- The first arg is the function (`fun`) that that computes the residuals and will be minimised
  - Note, this must simply return a NumPy array of individual residuals per point
  - SciPy takes care of squaring and summing these internally

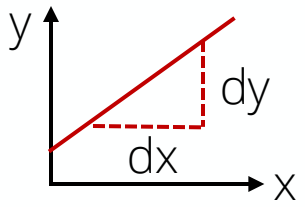
```
def straight_line_diff(params, xdata):
    "Differential of function for a straight line"

    df = params[1]
    return df
```

$$r_i = \frac{y_i - f(\beta, x_i)}{\sigma_{y_i}}$$

```
def straight_line(params, xdata):
    "Function for a straight line"

    # y = c + mx
    f = params[0] + params[1]*xdata
    return f
```

```
def residuals(params, xdata, ydata, xerr, yerr):
    """
    Function to calculate array of residuals: difference between each y point and its prediction
    by the function, divided by the sum in quadrature of the error on y, both from the y error
    and from the related error in x.
    """

    residuals = (ydata - straight_line(params, xdata)) / (np.sqrt(yerr**2 + straight_line_diff(params, xdata)**2 * xerr**2))

    return residuals
```

Total error on y



Least-squares: input (2)

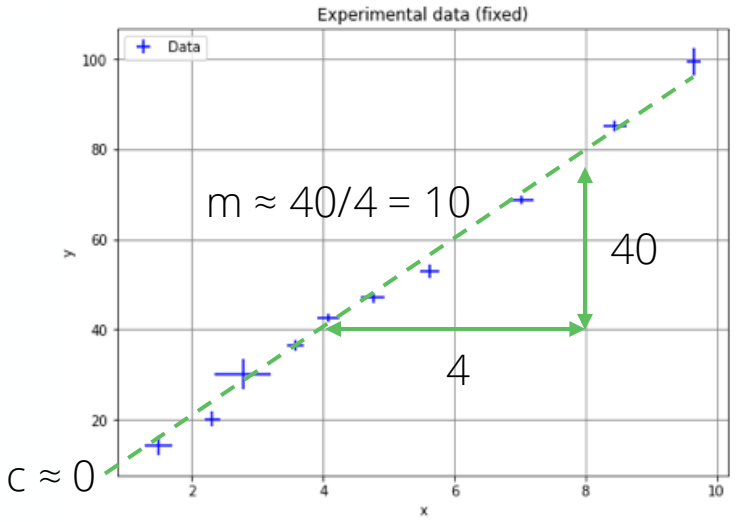
- SciPy's least_squares function can take many arguments, but for our needs it looks like

```
scipy.optimize.least_squares(fun, x0, args=())
```

- The second arg is an initial guess (`x0`) of the free parameters β
  - This is a sequence (list, tuple, array) of numbers

```
init_params = [0.0, 10.0]
```

  - Its length must equal the number of parameters
  - Can usually be determined by eye from a plot of the data



- Lastly, `args` is a a tuple of the input "data" that our function will be minimised with respect to
  - In our case four NumPy arrays (`xdata, ydata, xerrors, yerrors`), often read from a file

```
xdata, xerror, ydata, yerror = np.loadtxt("fitdata.csv", delimiter = ',', unpack = True)
print(f"x  = {xdata}\ndx = {xerror}\ny  = {ydata}\ndy = {yerror}")
```

```
x  = [1.5  2.31 2.78 3.58 4.08 4.76 5.62 7.02 8.45 9.65]
dx = [0.21 0.11 0.43 0.13 0.17 0.18 0.15 0.19 0.17 0.11]
y  = [14.3 20.2 30.1 36.5 42.7 47.1 52.9 68.8 85.2 99.4]
dy = [2.1 1.7 3.3 1.1 0.9 1.1 1.5 0.9 1.2 2.9]
```

New: unpack into individual arrays rather than 2D array

  - Gets passed indirectly into the residual function during minimisation

Least-squares: output

- Once read the data and defined the residuals function & initial params, we can perform the fit

```
from scipy.optimize import least_squares
result = least_squares(residuals, init_params, args=(xdata, ydata, xerror, yerror))
```

  - Get a result object, containing several useful pieces of information …

- `success`: a boolean that tells us if the fit succeeded
  - This is the first thing we must check

```
if not result.success:
    print ("ERROR: Fit failed with message {}".format(result.message))
else:
    print ("Fit succeeded")
```

- `x`: best fit parameters after the minimisation
  - Can be fed back to our function to get final y values

```
final_params = result.x
yfit = straight_line(final_params, xdata)
print(yfit)
```

```
[13.82880427 22.12535813 26.93940791 35.13353518 40.25486472 47.2198729
 56.02855972 70.36828245 85.01528494 97.30647585]
```

- `jac`: the Jacobian matrix (see PHYS108 in S2)
  - Contains the first order partial derivatives
  - Can be used to get the covariance matrix and hence the parameter uncertainties

$$\text{covariance} = \left|\text{jac}^2\right|^{-1} = \left|\text{jac} \times \text{jac}^T\right|^{-1} = \begin{bmatrix} \sigma_c^2 & \sigma_c\sigma_m \\ \sigma_m\sigma_c & \sigma_m^2 \end{bmatrix}$$

$$\text{errors} = \sqrt{\text{diag}(\text{covariance})}$$

```
jacobian = result.jac
jacobian2 = np.dot(jacobian.T, jacobian)
determinant = np.linalg.det(jacobian2)

if determinant < 1E-32:
    print(f"Matrix singular,error calculation failed.")
else:
    covariance = np.linalg.inv(jacobian2)
    param_errors = np.sqrt(covariance.diagonal())
    print(param_errors)
```

```
[1.74379629 0.31929604]
```

Least-squares: final result

- First thing is to check if the function fits the data well using the χ^2 per degree of freedom (i.e. reduced X^2)
  - This χ^2 is simply the S function we defined earlier, which we divide by NDF = N(data) – N(params)

$$\chi^2/\mathrm{dof} = \frac{\sum_{i=0}^{N} \left[\frac{y_i - f(\beta, x_i)}{\sigma_{y_i}}\right]^2}{\mathrm{NDF}}$$

```
chi2 = sum(result.fun ** 2)
npoints = len(xdata)
nparams = len(final_params)
reduced_chi2 = chi2 / (npoints - nparams)
print(f"chi2/dof = {reduced_chi2:.2f}")
```

```
chi2/dof = 0.82
```

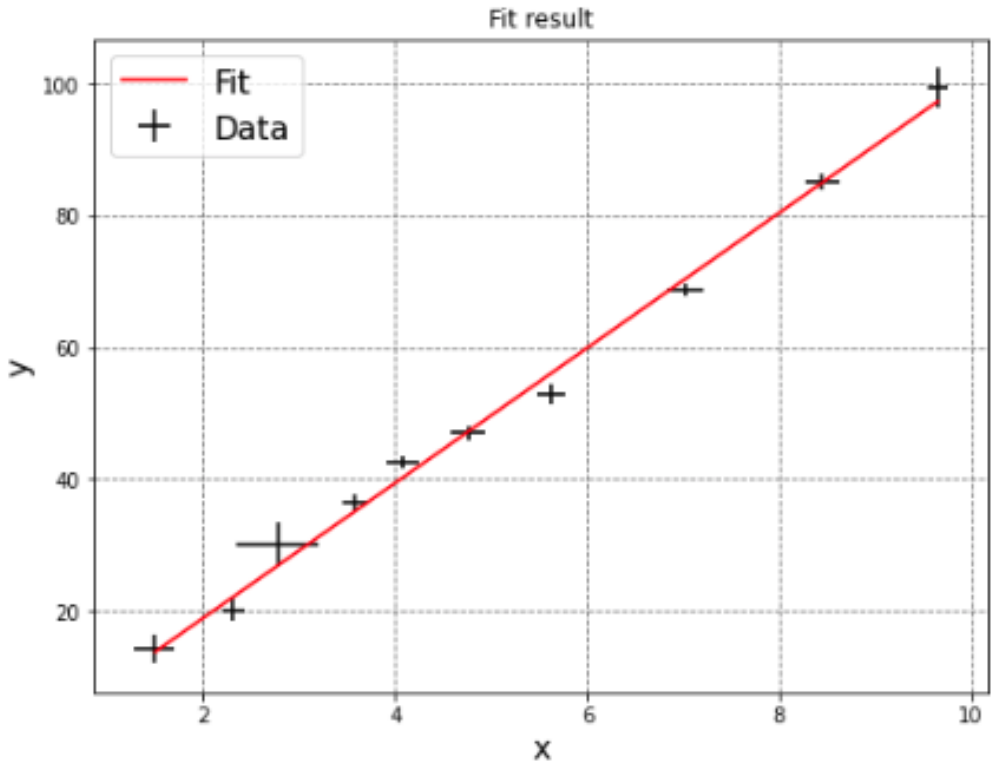
  - For a good fit, should ≈ lie in range $0.25 < \chi^2/\mathrm{NDF} < 4$

- We can then plot the best-fit line over the data and print the best-fit parameters obtained
  - Along with their uncertainties of course

```
print ("c = {:7.5g} +- {:7.5g}".format(final_params[0], param_errors[0]))
print ("m = {:7.5g} +- {:7.5g}".format(final_params[1], param_errors[1]))
```

```
c = -1.5352 +-  1.7438
m =  10.243 +-  0.3193
```

```
yfit = straight_line(final_params, xdata)

import matplotlib.pyplot as plt
plt.figure(figsize = (8, 6))
plt.title('Fit result')
plt.xlabel('x', fontsize=16)
plt.ylabel('y', fontsize=16)
plt.grid(color = 'grey', linestyle="--")

# Plot data and fit
plt.errorbar(xdata, ydata, xerr = xerror, yerr = yerror, color = 'k',
             linestyle = '', label = "Data")
plt.plot(xdata, yfit, color = 'r', linestyle = '-', label = "Fit")

# Add legend and fit params
plt.legend(loc = 2, fontsize=16)
plt.show()
```



Summary

- We have now started the second part of the course: using python to solve real physics problems
  - Pulling together and practicing the various pieces we have learnt so far
    - Variables, data structures, functions, control flow, input/output …
    - NumPy arrays and functions, Matplotlib figures, …
- This week we have looked at how we can fit a function to data to extract the parameters
  - Something you will do time-and-time again when comparing theory to data (e.g. in labs)
- In doing so, we introduced the SciPy package
  - Can perform a variety of scientific calculations
  - We only scratched the surface of what it can do
  - We will use if further in the upcoming weeks
- You will be able to practice fitting in this week's notebook
  - Going through step-by-step what we outlined here
  - Finally, applying this to real experimental data