Lab 3 Individual Analysis - Energy Measurements

This exercise will introduce (and beat to death) fitting a line to data. You should also learn the utility of transforming data sets so that they display as a line, the most easily recognizable function.

In [117]:

```
import numpy as np  # import the numpy library functions.
import scipy.constants as const # import physical constants
import matplotlib.pyplot as plt # plotting functions
import uncertainties as unc # Uncertainties package.  Good for simple err
or propagation
# directive below puts the plots in the notebook
%matplotlib inline
```

PMT Gain Analysis

From the collected data of peak height vs. high-voltage bias, carry out the following analysis:

1. Convert each peak voltage range to a median and uncertainty. For example if the listed voltage is "12.3 - 14.5 mV" convert it to (12.3+14.5)/2 plus or minus (14.5-12.3)/2, i.e., 13.4 +/- 1.1. Accomplish this by putting the numbers into NumPy arrays which are then used to create new arrays.
2. Plot the median voltage vs HV bias on a double-log scale. If the model for PMT gain is mostly correct, the result should be points lying along a line.
3. Fit the data to obtain the slope of the line on the double-log plot. The slope should be a number close to the number of dynode stages.

Step 1. Enter and reduce the data set

Create the dat set arrays:

In [132]:

```
# From the data set, create arrays to manipulate.
# One suggested way is below, where the "low" and "hi" are the low and high
 values for
# each peak.

# You are welcome to use other ways to create the data sets

HV_data = np.arange(400.,950.,50.)
peak_heights_low = np.array([1.46, 3.68, 8.52, 18.6, 38.2, 71.4, 132.5, 237.
,
                              395., 646., 1010.])
peak_heights_hi  = np.array([1.68, 4.12, 9.80, 20.8, 42.6, 83.2, 155.0, 267.
5,
                              449., 730., 1155.])
len(HV_data)
```

Out[132]:

```
11
```

Manipulate them to obtain midpoints and uncertainty. Then print out using the code at the bottom (or your version of it).

In [133]:

```
# Create new arrays, according to the calculations above

peak_heights_mid = (peak_heights_low + peak_heights_hi)/2
peak_heights_delta = peak_heights_hi - peak_heights_low

# Use the following to print out.  Note the use of the "zip()" function
for V, peak, delta_peak in zip(HV_data, peak_heights_mid, peak_heights_delt
a):
    print('{:.0f} V: {:7.2f} +/- {:5.2f} mV'.format(V,peak,delta_peak))
```

```
400 V:    1.57 +/-  0.22 mV
450 V:    3.90 +/-  0.44 mV
500 V:    9.16 +/-  1.28 mV
550 V:   19.70 +/-  2.20 mV
600 V:   40.40 +/-  4.40 mV
650 V:   77.30 +/- 11.80 mV
700 V:  143.75 +/- 22.50 mV
750 V:  252.25 +/- 30.50 mV
800 V:  422.00 +/- 54.00 mV
850 V:  688.00 +/- 84.00 mV
900 V: 1082.50 +/- 145.00 mV
```

Step 2. Plot the peak heights versus HV values

Use the `plt.errorbar()` plotting form to allow error bars to be plotted along with the points. The error bars should be proportional to the "uncertainty" array values. (Search for `pyplot.errorbar()` to see how to use this form.)

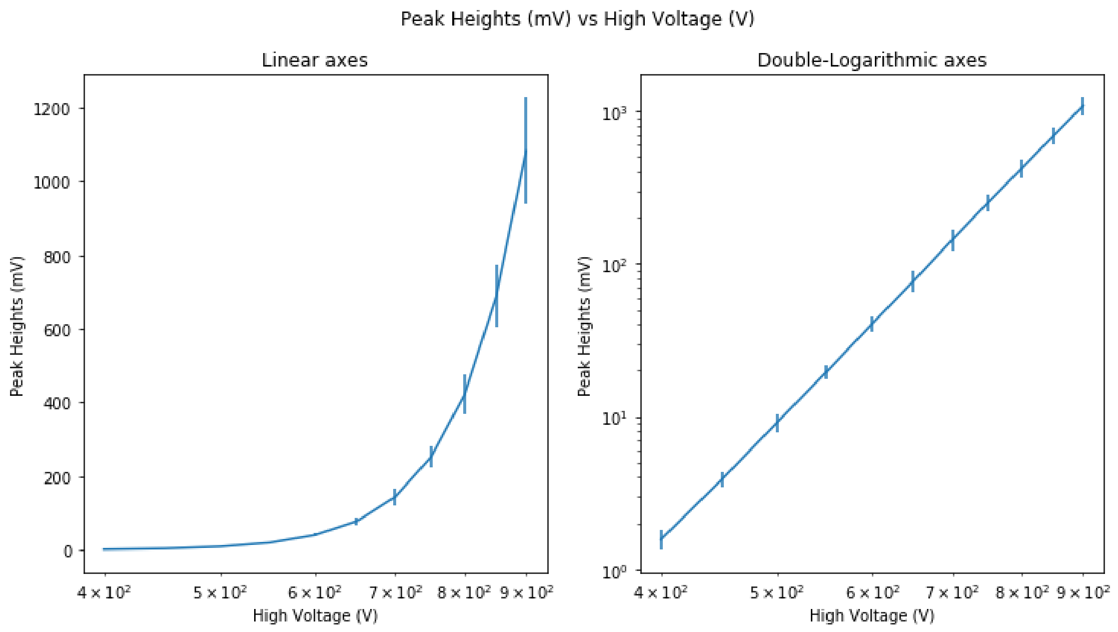
Plot with both linear and double-logarithmic axes to see how the data become "linearized." (Hint: `plt.yscale('log')`)

In [134]:

```
fig, (ax0, ax1) = plt.subplots(nrows=1, ncols=2, sharex=True,
                               figsize=(12, 6))

fig.suptitle('Peak Heights (mV) vs High Voltage (V)')

ax0.errorbar(HV_data, peak_heights_mid, yerr=peak_heights_delta)
ax0.set_title('Linear axes')
ax0.set_xlabel('High Voltage (V)')
ax0.set_ylabel('Peak Heights (mV)')

ax1.errorbar(HV_data, peak_heights_mid, yerr=peak_heights_delta)
ax1.set_xscale('log')
ax1.set_yscale('log')
ax1.set_title('Double-Logarithmic axes')
ax1.set_xlabel('High Voltage (V)')
ax1.set_ylabel('Peak Heights (mV)')
```

Out[134]:

```
Text(0, 0.5, 'Peak Heights (mV)')
```



Step 3. Fit the log of the data sets to a line, and find the slope

SOMETHING NEW: The **lmfit** package is a useful add-on to the SciPy fitting functions. This package simplifies fitting data to a variety of standard functions. See the Lmfit Documentation (https://lmfit.github.io/lmfit-py/index.html) for a full discussion. The package is quite powerful, but for basic fitting with common functions, it is very easy to use.

Example: Fitting a line

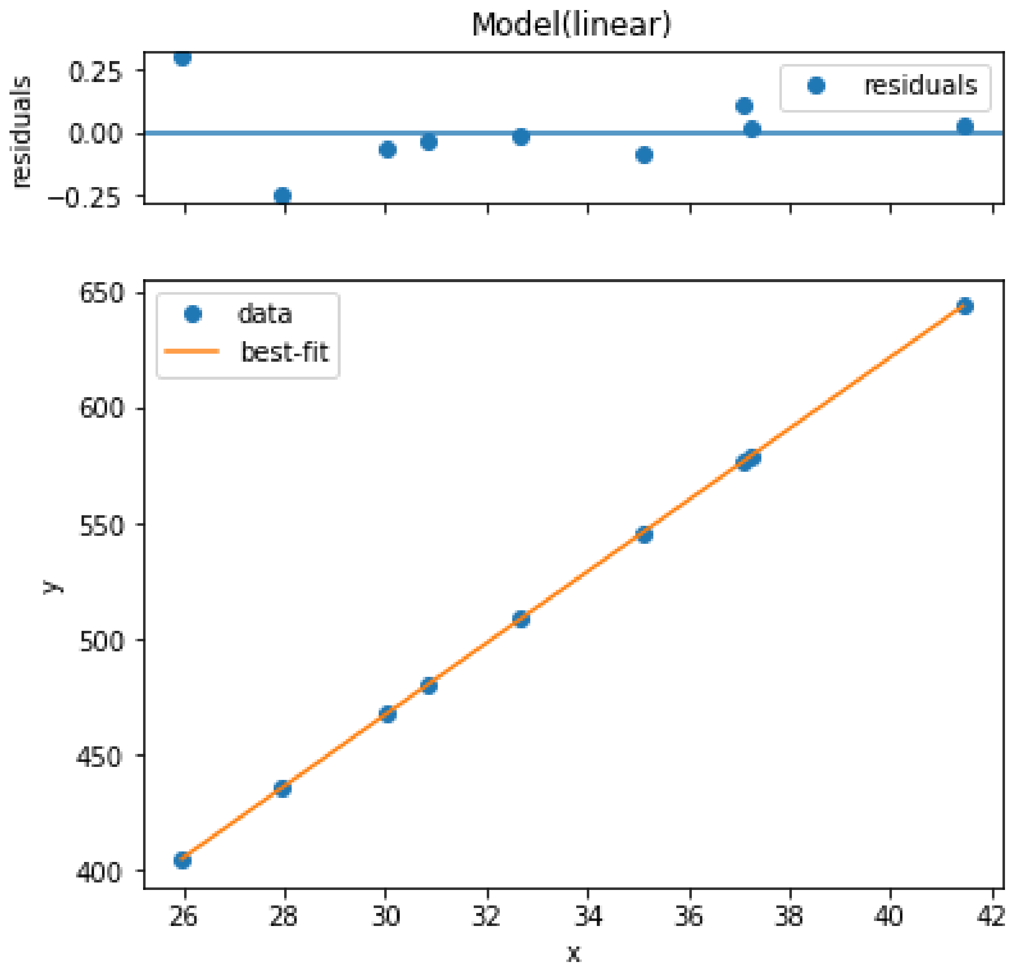
The example below shows how to use the package to fit data to a line, obtain the fit parameters along with uncertainties, and then plot the data and fit. Execute the cell and study how it works. (Note: the data come from a calibration problem in physics 331)

In [135]:

```
# Calibration Data
# First column is wavelength (nm), second is carriage poisition (cm)
#
Cal_data = np.array([
    [643.85, 41.43],
    [579.07, 37.24],
    [576.96, 37.11],
    [546.08, 35.10],
    [508.58, 32.68],
    [479.99, 30.83],
    [467.81, 30.04],
    [435.83, 27.96],
    [404.66, 25.98]])

# Array slicing separates x (position) and y (wavelength)
# Goal of calibration is to be able to feed in a position and obtain a wavel
ength
wavelength = Cal_data[:,0]
position = Cal_data[:,1]

# imports a linear fitting model from lmfit
from lmfit.models import LinearModel

# create an instance of the model
line = LinearModel()

# One must have a guess of the parameters. The guess() method works with mos
t of the standard
# lmfit models
param_guess = line.guess(wavelength, x=position)

# The line below executes the fitting process.  The results are returned to
 "line_fit"
line_fit = line.fit(wavelength, param_guess, x=position)

# This prints the results in an easy to read form
print(line_fit.fit_report())

# The parameters and uncertainties are accessible as follows, for example:
print('¥nSlope = ',line_fit.params['slope'].value,'+/-',line_fit.params['slo
pe'].stderr)
```

```
#Then you can plot the results quickly just to see how it looks using the pl
ot() method
line_fit.plot();
```

```
[[Model]]
    Model(linear)
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 3
    # data points      = 9
    # variables        = 2
    chi-square         = 0.18041902
    reduced chi-square = 0.02577415
    Akaike info crit   = -31.1872805
    Bayesian info crit = -30.7928313
[[Variables]]
    slope:      15.4639413 +/- 0.01139807 (0.07%) (init = 15.4
6394)
    intercept:  3.20598011 +/- 0.38164201 (11.90%) (init = 3.2
0598)
[[Correlations]] (unreported correlations are < 0.100)
    C(slope, intercept) = -0.990

Slope =  15.463941344548298 +/- 0.011398066752229 34
```



Adapt the above to your problem

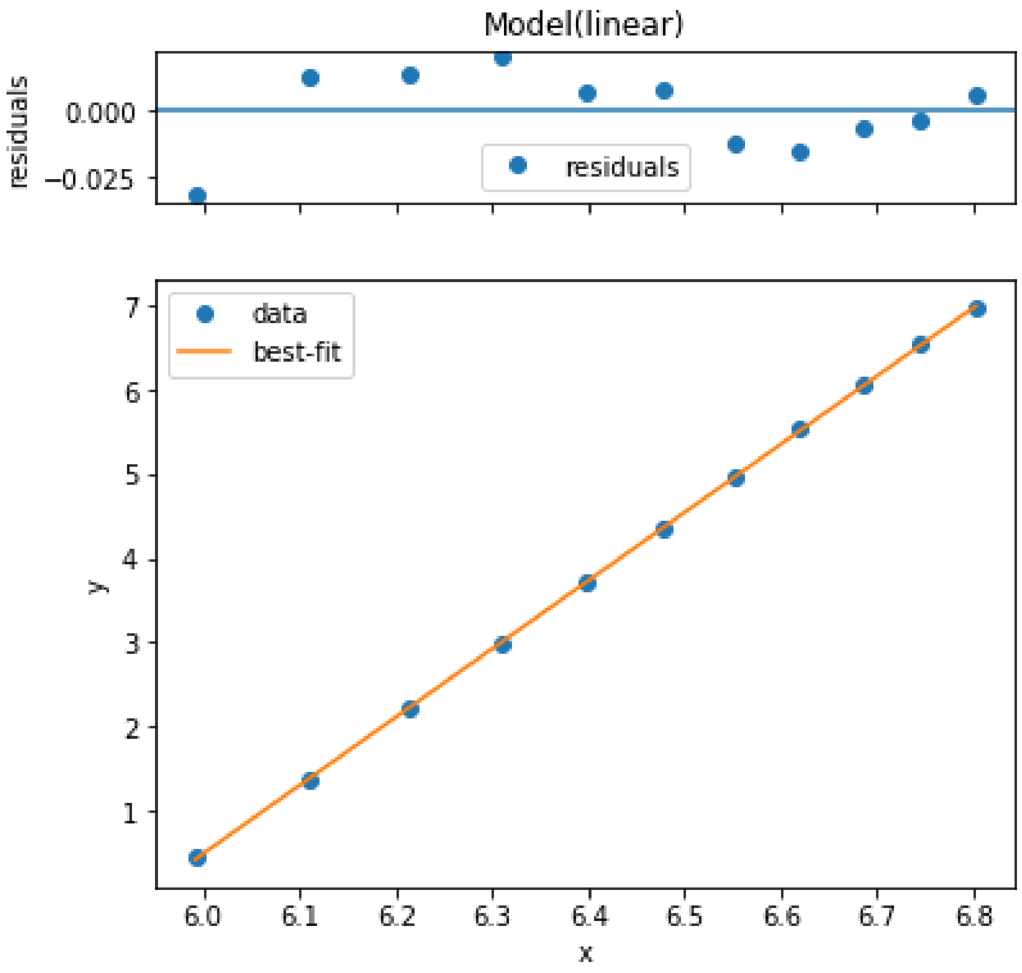
To adapt the linear model, you would need to transform the data with a `log()` function, run the fit, and then transform the fit back to plot against the original data. Alternately, you could fit to the power law model, available in the lmfit package. You decide. Either way, determine the exponent n in the PMT gain model

$$G = \delta^n = (KV_d)^n$$

In [141]:

```
# You can feed np.log(peak_heights_mid), and similar directly to the fitting functions.
# There is no need to create a second set of arrays
log_peak_heights = np.log(peak_heights_mid)
log_HV_data = np.log(HV_data)
from lmfit.models import LinearModel

line = LinearModel()

param_guess = line.guess(log_peak_heights, x=log_HV_data)

line_fit = line.fit(log_peak_heights, param_guess, x=log_HV_data)

print(line_fit.fit_report())

print('¥nSlope = ',line_fit.params['slope'].value,'+/-',line_fit.params['slope'].stderr)

line_fit.plot();
```

```
[[Model]]
    Model(linear)
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 3
    # data points      = 11
    # variables        = 2
    chi-square         = 0.00236020
    reduced chi-square = 2.6224e-04
    Akaike info crit   = -88.9159693
    Bayesian info crit = -88.1201787
[[Variables]]
    slope:      8.10727834 +/- 0.01921503 (0.24%) (init = 8.10
7278)
    intercept: -48.1552947 +/- 0.12395023 (0.26%) (init = -48.
15529)
[[Correlations]] (unreported correlations are < 0.100)
    C(slope, intercept) = -0.999

Slope =  8.107278336637254 +/- 0.019215028074135942
Help on method eval in module lmfit.model:

eval(params=None, **kwargs) method of lmfit.model.ModelResult
instance
    Evaluate model function.

    Parameters
    ----------
    params : Parameters, optional
        Parameters to use.
    **kwargs : optional
        Options to send to Model.eval()

    Returns
    -------
    out : numpy.ndarray
       Array for evaluated model.
```



Calculate a fit line

Make a fitline in the original data space by taking antilogs (np.exp()) of the line fit values. You need to supply voltages to calculate the fitted values

In [147]:

```
# Make an array of arbitrary values between 350 an 1000 volts
V_values = np.arange(400, 950, 50)

#Then use the model to evaluate the fit values from these.  Hint: use line_f
it.eval() to generate y values
y_vals = np.exp(line_fit.eval())
print(y_vals)
fl_param_guess = line.guess(y_vals, x=V_values)
fitline = line.fit(y_vals, fl_param_guess, x=V_values)
```

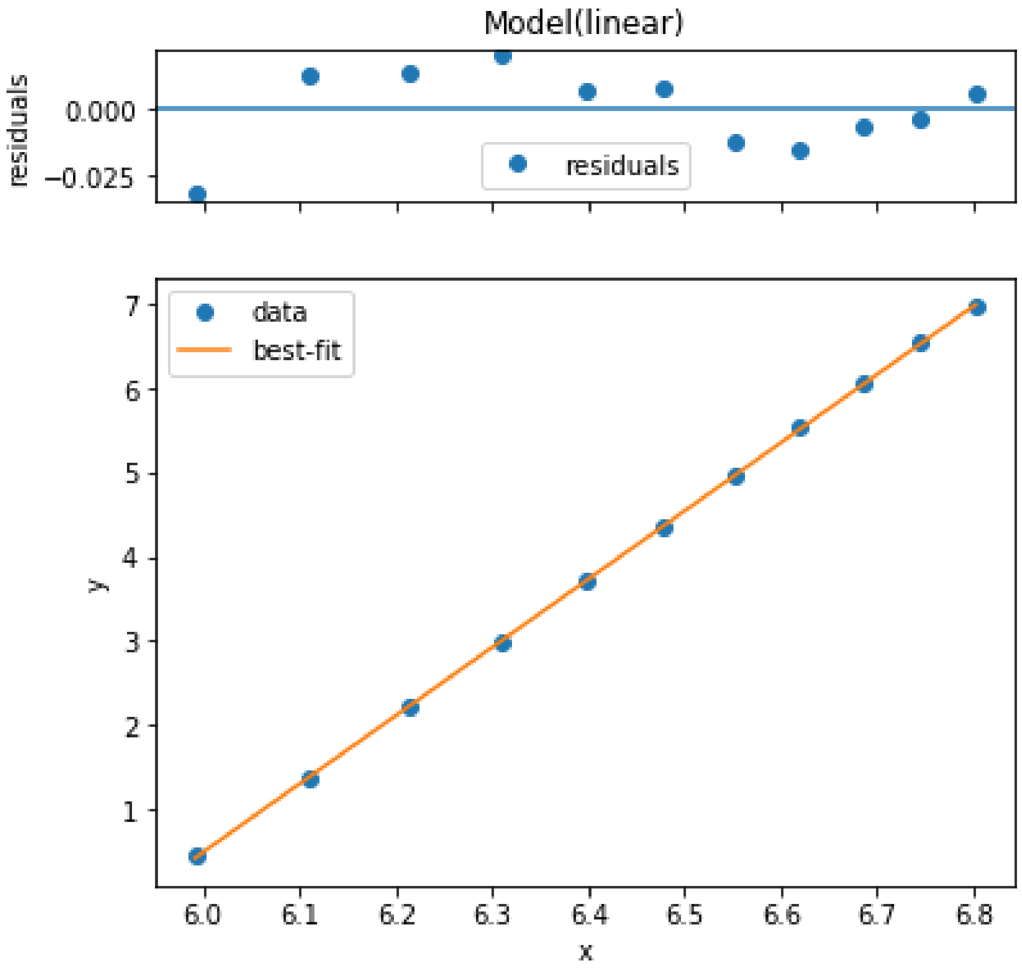
```
[   1.520708      3.95142341    9.28372529   20.10501033   40.
70677359
   77.89198161  142.0440148   248.50957733  419.35303872  685.
54478091
 1089.65127523]
```

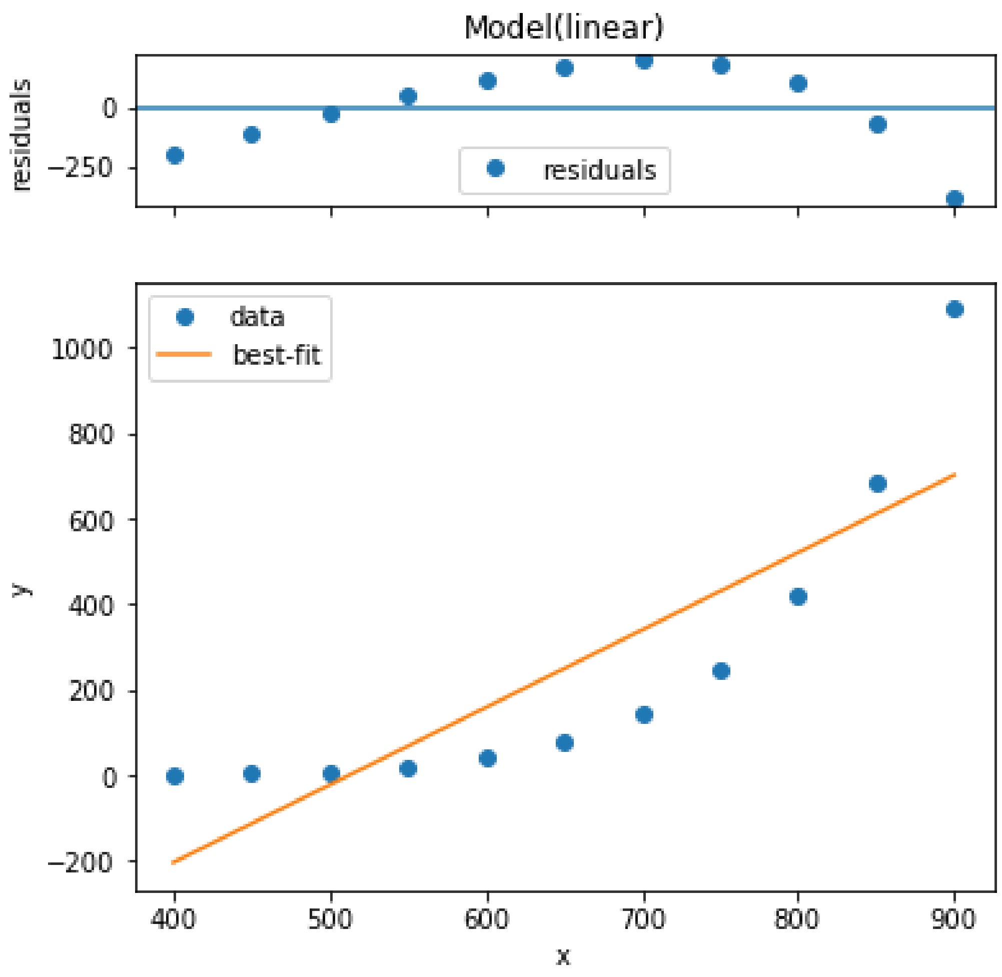
Redo the plot above in Step 2, including the fitline.

In [150]:

```
# Add the line to the previous plot
log_peak_heights = np.log(peak_heights_mid)
log_HV_data = np.log(HV_data)
from lmfit.models import LinearModel

line = LinearModel()

param_guess = line.guess(log_peak_heights, x=log_HV_data)

line_fit = line.fit(log_peak_heights, param_guess, x=log_HV_data)

print(fitline.fit_report())
line_fit.plot();
fitline.plot();
```

```
[[Model]]
    Model(linear)
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 3
    # data points      = 11
    # variables        = 2
    chi-square         = 340545.125
    reduced chi-square = 37838.3472
    Akaike info crit   = 117.744484
    Bayesian info crit = 118.540275
[[Variables]]
    slope:      1.81006920 +/- 0.37093664 (20.49%) (init = 1.8
10069)
    intercept: -927.584768 +/- 248.139698 (26.75%) (init = -92
7.5848)
[[Correlations]] (unreported correlations are < 0.100)
    C(slope, intercept) = -0.972
```





Don't forget to obtain n

After calculating, look up the specifications for the 9266KB tube (see the Data Sheets and Equipment Information link on the canvas page). Compare your result to what you expect in your group document.

In [11]:

```
10
```

Out[11]:

```
10
```

Calculate the compton edges for the γ sources

Write a Python function to calculate the "Compton edges" for the main γ emissions of the Cs-137, Na-22, and Co-60 isotopes. You will find a formula in Leo, Section 2.7.2, and in the lecture notes on Energy Measurements.

Use the scipy.constants library to obtain the necessary physical constants in your function. Energy should be expressed as MeV

Then apply your formula, and create a table showing each isotope, the photopeak energy, and the Compton edge.

Enter the data for the gamma energies

Create some data structure to hold the known energies of the gamma ray sources. A Python "dictionary" is a good option, but a simple array or list will work, if you keep track of which energy belongs to which nuclide.

In [12]:

```
# Data from table of radioactive sources by Browne
# Energies in MeV
RA_energies = {'Ba-133' : [0.081, 0.356, 0.031],
              'Co-60' : [1.173, 1.333],
              'Cs-137' : [0.662],
              'Na-22' : [0.511, 1.275]}
print(RA_energies.get('csasd'))
#help(dict)
```

```
None
```

Write your function

Write a function to return the Compton edge energy for a given gamma-emission energy (in MeV).

In [13]:

```
def compton_edge(sources, src, energy):
    if (src in sources) and (energy in sources.get(src)):
        m_e = const.m_e
        c = const.c
        e_E = m_e * c**2 * 6.242e12
        #compton_edge = (2 * energy**2) / (m_e * c**2 + 2 * energy)
        compton_edge = energy * ((2 * (energy / e_E)) /
                                 (1 + 2 * (energy / e_E)))
        return compton_edge
```

Print a table

Print out a table listing the source, the gamma-emission energy, and the Compton edge energy for all values.

In [17]:

```
#print(compton_edge(RA_energies, 'Ba-133', 0.081))

from prettytable import PrettyTable

table = PrettyTable()
table.field_names = ['source', 'gamma-emission energy (MeV)', 'Compton edge
 energy (MeV)']
for key in RA_energies.keys():
    for energy in RA_energies.get(key):
        edge = compton_edge(RA_energies, key, energy)
        table.add_row([key, energy, edge])

print(table)
```

```
+--------+-----------------------------+---------------------
-----+
| source | gamma-emission energy (MeV) | Compton edge energy
(MeV) |
+--------+-----------------------------+---------------------
-----+
| Ba-133 |            0.081            |     0.019496637223084
|
| Ba-133 |            0.356            |    0.2072476596810825
8     |
| Ba-133 |            0.031            |   0.00335404662117008
22    |
| Co-60  |            1.173            |     0.963185263715350
8     |
| Co-60  |            1.333            |     1.118581748761081
9     |
| Cs-137 |            0.662            |     0.477639947648868
4     |
| Na-22  |            0.511            |     0.340657968535570
7     |
| Na-22  |            1.275            |     1.062139309087616
8     |
+--------+-----------------------------+---------------------
-----+
```

Create a calibration curve from the measurements

From your group's fits to the main photopeaks in the pulse height spectra of the four sources, create three arrays:

- `peaks` : the peak locations in channel number
- `delta-peaks` : the full-width at half-maximum of the photopeaks, in channel number
- `energies` : the known energies, in MeV of the photopeaks

In [83]:

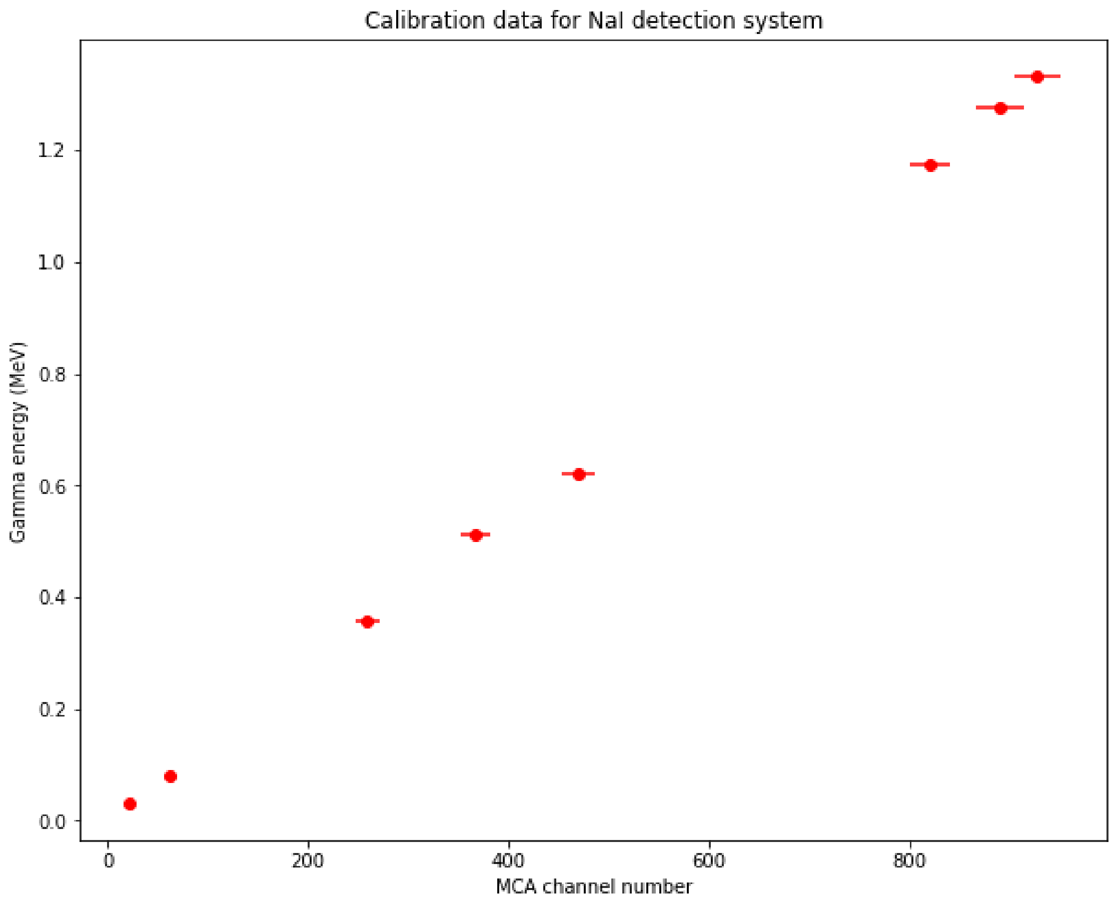
```
peaks = np.array([470, 928, 820, 21, 61, 259, 367, 891])
delta_peaks = np.array([32.8, 45.9, 40.6, 4.7, 6.6, 22.9, 29.1, 47.6])
energies = np.array([0.622, 1.333, 1.173, 0.031, 0.081, 0.356, 0.511, 1.275
])
```

Make a sanity-check plot

Execute the code below to see if you have a nice line. If not? Fix the problem.

In [84]:

```
# Execute this to run a sanity-check plot
plt.figure(figsize=(10,8))
plt.xlabel('MCA channel number')
plt.ylabel('Gamma energy (MeV)')
plt.title('Calibration data for NaI detection system')
plt.errorbar(peaks,energies,xerr=delta_peaks/2,fmt='ro');
```



Fit the calibration data to a line

Fit the calibration so that so that you obtain the coefficients necessary to convert a channel to an energy.

Then create a calibration function from these coefficients.

Optional: Use the uncertainties package to calculate the uncertainty in the energy from the uncertaities in the coefficients.

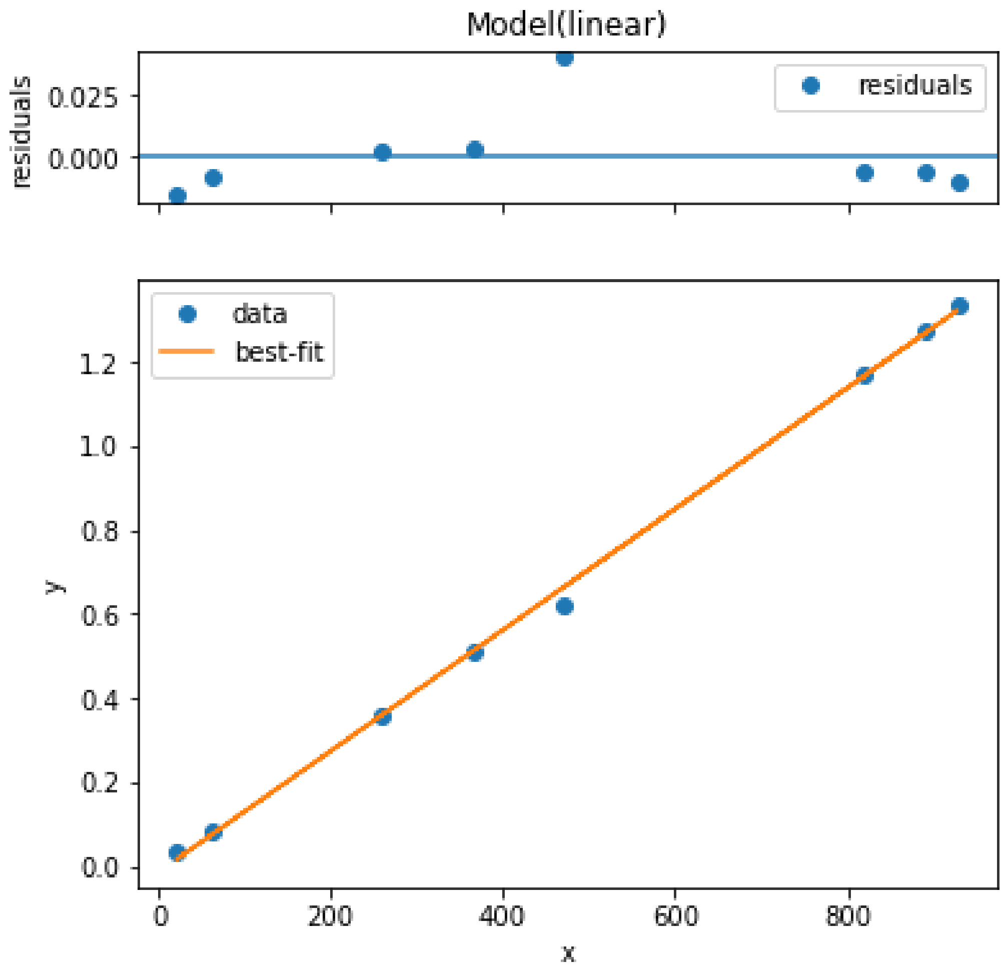
First make the fit

You have done this above. Do it again.

In [85]:

```
line = LinearModel()

param_guess = line.guess(energies, x=peaks)

line_fit = line.fit(energies, param_guess, x=peaks)

print(line_fit.fit_report())

print('¥nSlope = ',line_fit.params['slope'].value,'+/-',line_fit.params['slo
pe'].stderr)

line_fit.plot();

#help(line)
```

```
[[Model]]
    Model(linear)
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 4
    # data points      = 8
    # variables        = 2
    chi-square         = 0.00214079
    reduced chi-square = 3.5680e-04
    Akaike info crit   = -61.8081937
    Bayesian info crit = -61.6493107
[[Variables]]
    slope:      0.00144109 +/- 1.9554e-05 (1.36%) (init = 0.00
1441088)
    intercept: -0.01482930 +/- 0.01147373 (77.37%) (init = -0.
0148293)
[[Correlations]] (unreported correlations are < 0.100)
    C(slope, intercept) = -0.813

Slope =  0.001441088391559042 +/- 1.9554385786869535e-05
```



Create and test your calibration function

Write a function that takes channel number and produces an energy in MeV.

In [88]:

```
slope = line_fit.params['slope']
intercept = line_fit.params['intercept']

def chToEnergy(channel, m = slope, b = intercept):
    mev = m * channel + b
    return mev
```

Study model of ΔE vs E

According to a simple model of detector resolution, the width of the photopeak should be proportional to the square root of the energy. The simplest way to view this is to plot $(\Delta E)^2$ versus E. If the model holds the points should lie on a line.

Try it out: make such a plot.

Then fit the data to a line, and extract the slope. From this, extract the energy factor Fw (See the lecture notes and lab instructions for explanation.)

Make the plot

of E vs $(\Delta E)^2$. Run the `peaks` and `delta_peaks` through your calibration function.

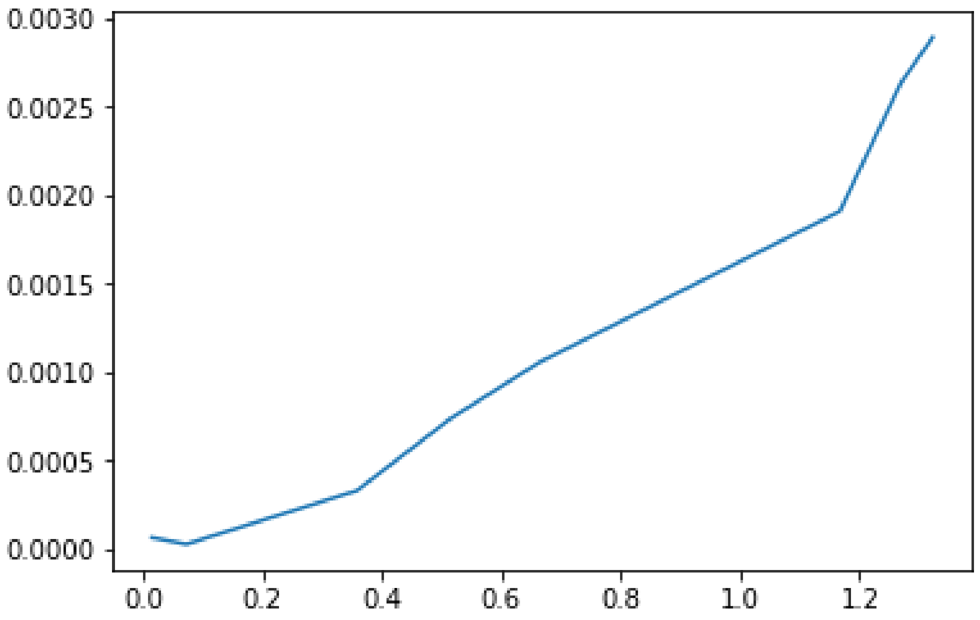
Note: if your calibration function returns a `ufloat` uncertainty object, these cannot be passed directly to the `plt.plot()` function.

In [89]:

```
E = chToEnergy(np.sort(peaks))
delta_E = chToEnergy(np.sort(delta_peaks))
delta_E = delta_E**2

plt.plot(E, delta_E)
```

Out[89]:

```
[<matplotlib.lines.Line2D at 0x7f7804920390>]
```



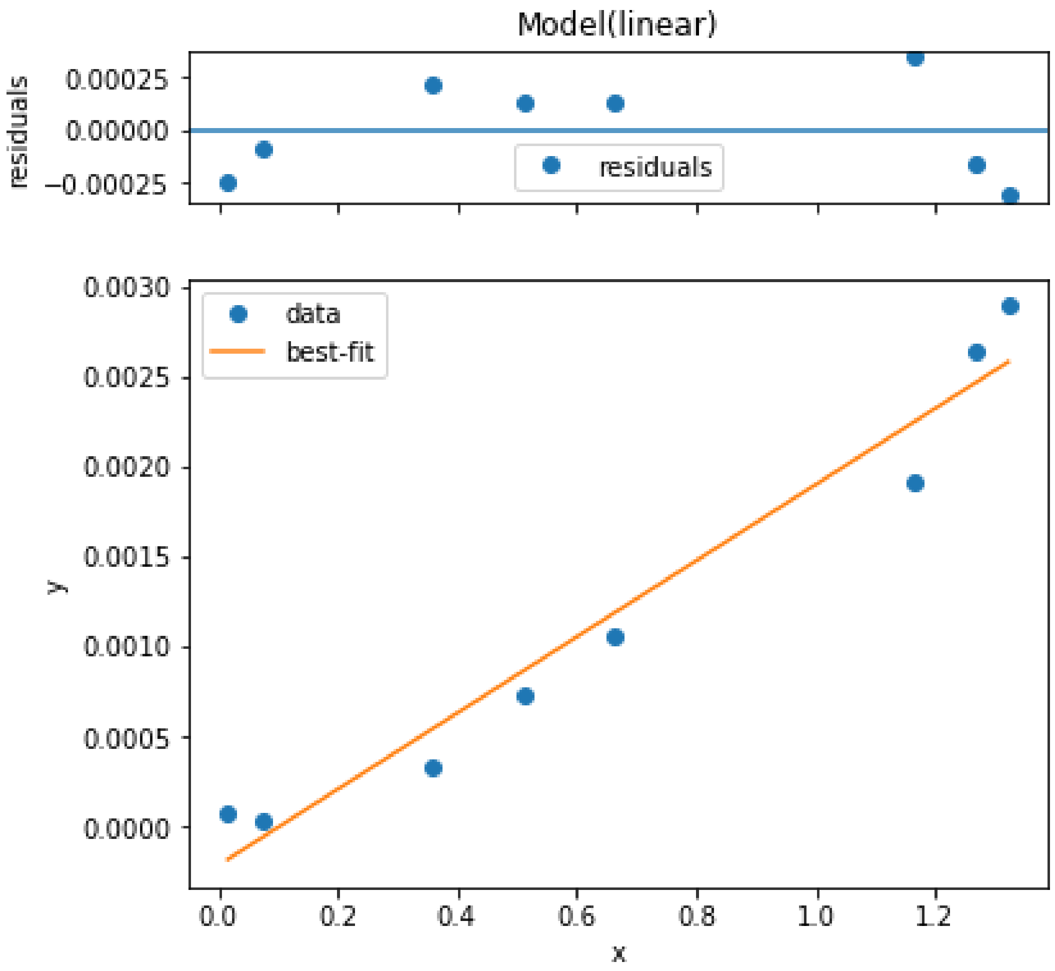
Fit the data

One more time with a linear fit.

In [90]:

```
line = LinearModel()

param_guess = line.guess(delta_E, x=E)

line_fit = line.fit(delta_E, param_guess, x=E)

print(line_fit.fit_report())

print('¥nSlope = ',line_fit.params['slope'].value,'+/-',line_fit.params['slo
pe'].stderr)

line_fit.plot();
print(line_fit.params)
```

```
[[Model]]
    Model(linear)
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 3
    # data points      = 8
    # variables        = 2
    chi-square         = 3.9240e-07
    reduced chi-square = 6.5400e-08
    Akaike info crit   = -130.643407
    Bayesian info crit = -130.484524
[[Variables]]
    slope:      0.00211620 +/- 1.8371e-04 (8.68%) (init = 0.00
2116201)
    intercept: -2.1836e-04 +/- 1.5313e-04 (70.13%) (init = -0.
0002183644)
[[Correlations]] (unreported correlations are < 0.100)
    C(slope, intercept) = -0.807

Slope =  0.0021162012138922323 +/- 0.00018370949491376272
Parameters([('slope', <Parameter 'slope', value=0.002116201213
8922323 +/- 0.000184, bounds=[-inf:inf]>), ('intercept', <Para
meter 'intercept', value=-0.00021836436627539703 +/- 0.000153,
bounds=[-inf:inf]>)])
```


Model(linear)
residuals
0.00025
0.00000
−0.00025
residuals
data
best-fit
0.0030
0.0025
0.0020
0.0015
0.0010
0.0005
0.0000
y
0.0
0.2
0.4
0.6
0.8
1.0
1.2
x

Calculate Fw

Obtain the slope and uncertainty. Use these to calculate the quantity Fw, with uncertainty, in electron volts (eV).

In [92]:

```
# After calculation, use the command below to print it out.
fw_slope = line_fit.params['slope'].value
fw_unc = line_fit.params['slope'].stderr
Fw = unc.ufloat(fw_slope, fw_unc)
print('The estimated "Fano factor X energy per photoelectron" Fw = {:.2uP} eV'.format(Fw*1e6))
```

```
The estimated "Fano factor X energy per photoelectron" Fw =
(2.12±0.18)×10³ eV
```