Data_fitting

November 9, 2016

1 Data fitting

Curve fitting is the process of constructing a curve, or mathematical function, that has the best fit to a series of eperimental data points.

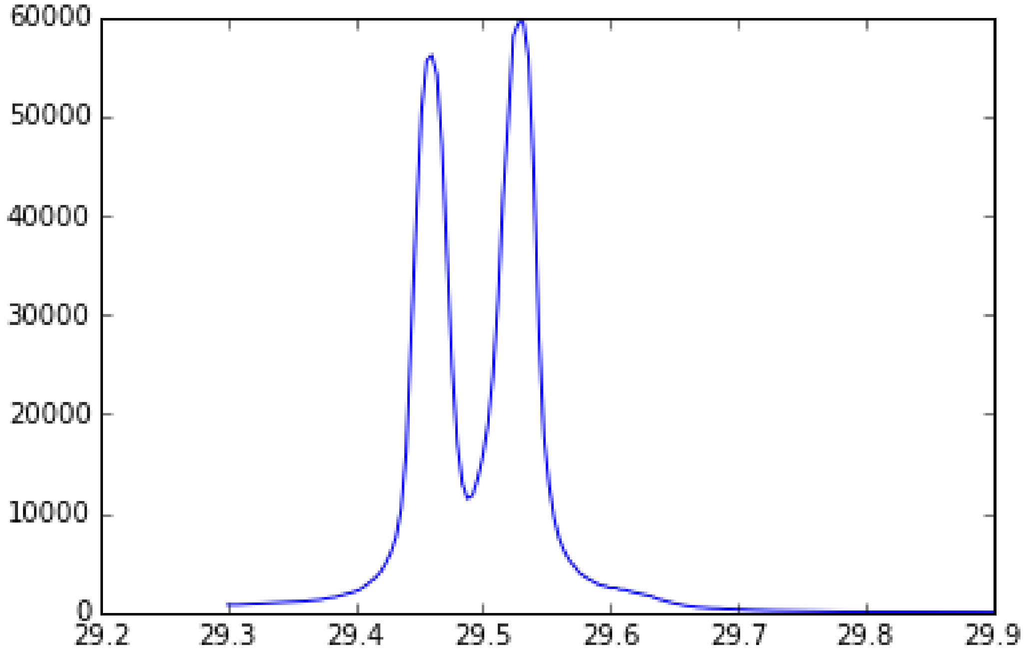
1.1 Loading data

```
In [1]: %pylab inline
```

```
Populating the interactive namespace from numpy and matplotlib
```

```
In [2]: from silx.io import spech5
        specfile = spech5.SpecH5("31oct98.dat")
        xdata = specfile["/22.1/measurement/TZ3"]
        ydata = specfile["/22.1/measurement/If4"]
        plot(xdata, ydata)
```

```
Out[2]: [<matplotlib.lines.Line2D at 0x7f1203aa0400>]
```



1.2 Use of a builtin fitting function

The fitting function is a function of the data-point (xdata) and of a sert of parameters. Silx offers a set of usual fitting functions in module `silx.math.fit.functions`.

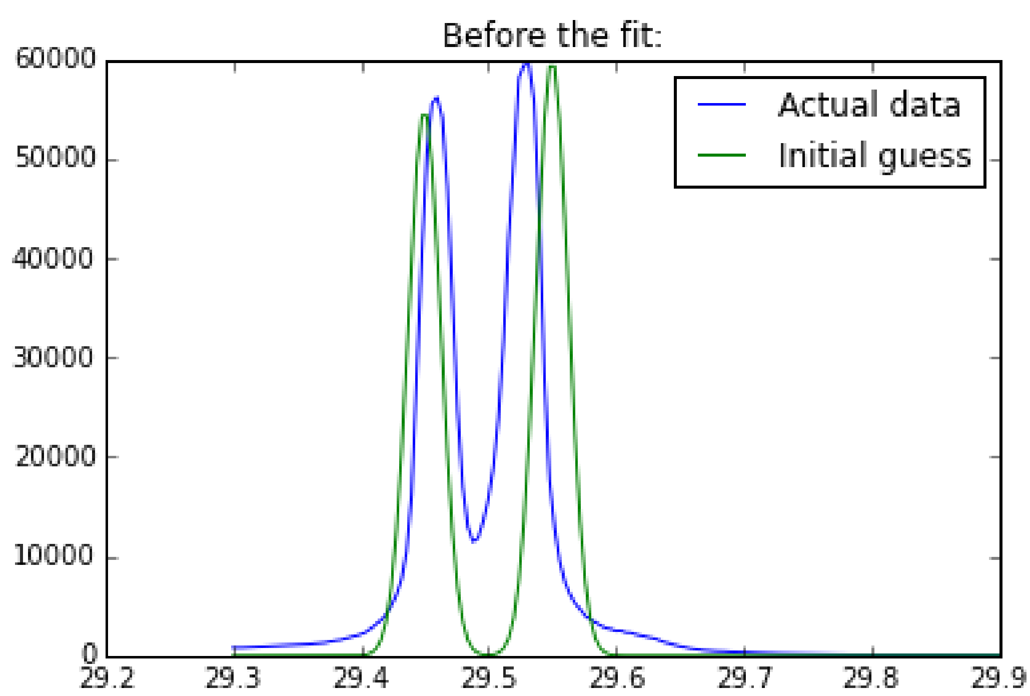
In this example, we will use a gaussian function, `sum_gauss`, whose parameters are *height*, *center* an *fwhm* (full-width at half maximum). The functions accepts a multiple number of these 3 base parameters, to fit a sum of multiple gaussian peaks.

```
In [3]: from silx.math.fit.functions import sum_gauss
```

An essential part of the iterative fitting process is the choice of the initial set of parameters p0. Our initial estimate is that we have 2 peaks, centered at $x = 29.45$ and at $x = 29.55$, with amplitude 55000 and 60000 and with a full-width at half maximum of 0.03.

```
In [4]: p0 = [55000., 29.45, 0.03,
              60000., 29.55, 0.03]
        plot(xdata, ydata, label="Actual data")
        plot(xdata, sum_gauss(xdata, *p0), label="Initial guess")
        legend()
        title("Before the fit:")
```

```
Out[4]: <matplotlib.text.Text at 0x7f1202e296a0>
```



1.3 Fitting using silx

```
In [5]: from silx.math.fit import leastsq
```

```
In [6]: p, cov_matrix = leastsq(model=sum_gauss, xdata=xdata, ydata=ydata, p0=p0)
        print(p)
```

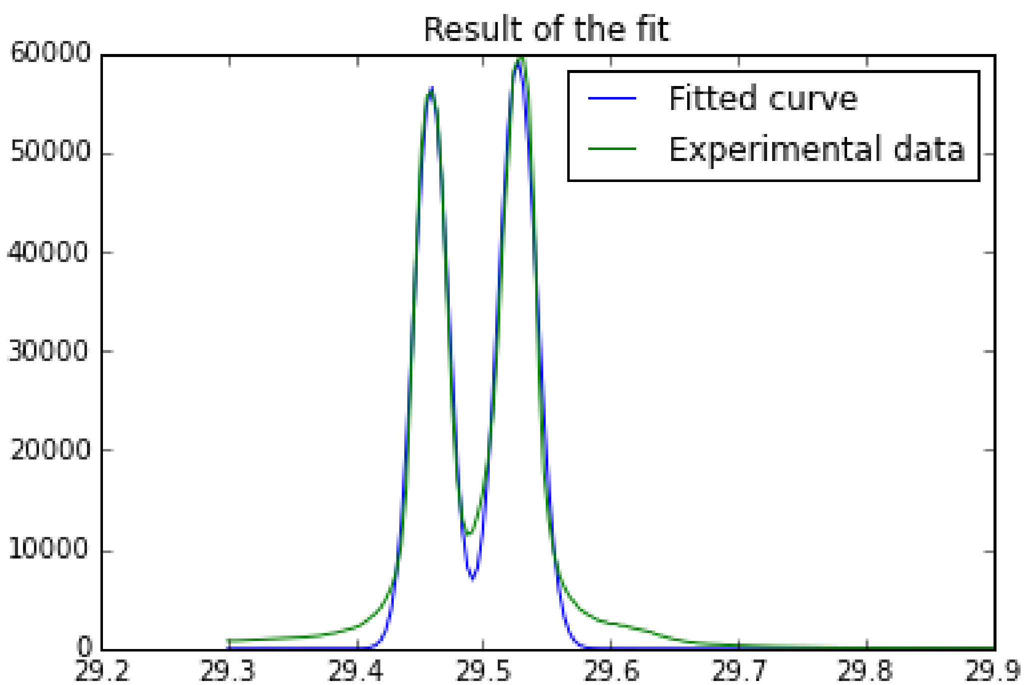
```
[  5.65660023e+04   2.94595631e+01   3.19836859e-02   5.92015970e+04
   2.95275309e+01   3.54588691e-02]
```

```
In [7]: %timeit p, cov_matrix = leastsq(model=sum_gauss, xdata=xdata, ydata=ydata,
```

```
100 loops, best of 3: 6.63 ms per loop
```

```
In [8]: #Display the result of the fit
        #plot(xdata, sum_gauss(xdata, *p0), label="Initial guess")
        plot(xdata, sum_gauss(xdata, *p), label="Fitted curve")
        plot(xdata, ydata,label="Experimental data")
        legend()
        title("Result of the fit")
```

```
Out[8]: <matplotlib.text.Text at 0x7f1202db3278>
```

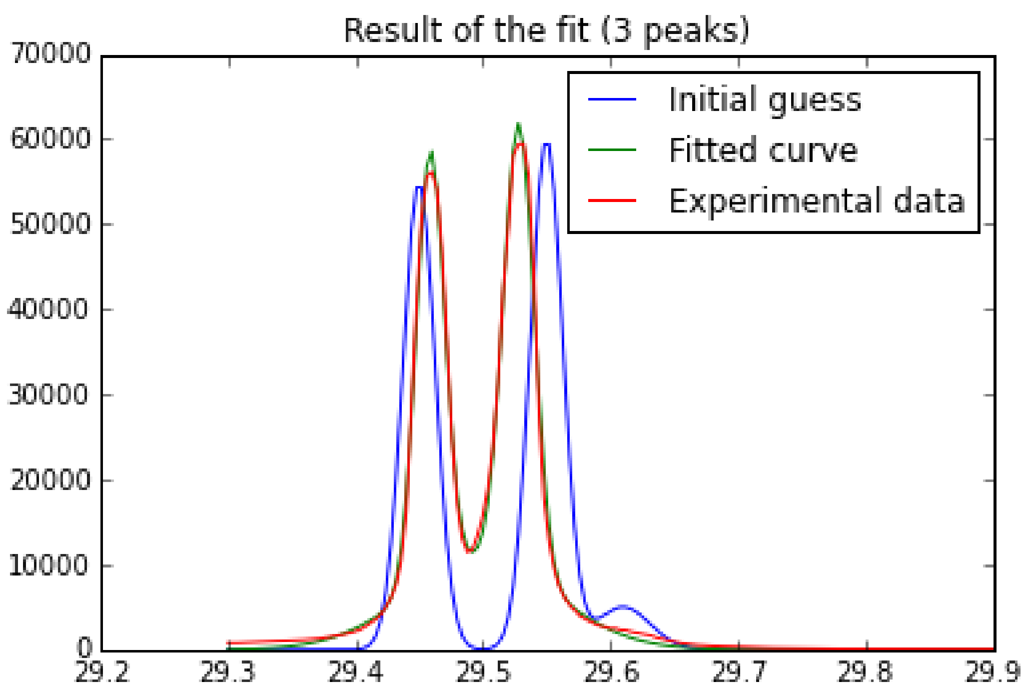


We could probably improve this with a small third peak at $x = 29.61$

```
In [9]: # estimate
        p0 = [55000., 29.45, 0.03,
              60000., 29.55, 0.03,
              5000., 29.61, 0.05]
        # fit
        p, cov_matrix = leastsq(model=sum_gauss, xdata=xdata, ydata=ydata, p0=p0)

        # plot
        plot(xdata, sum_gauss(xdata, *p0), label="Initial guess")
        plot(xdata, sum_gauss(xdata, *p), label="Fitted curve")
        plot(xdata, ydata, label="Experimental data")
        legend()
        title("Result of the fit (3 peaks)")

Out[9]: <matplotlib.text.Text at 0x7f1202d1eba8>
```



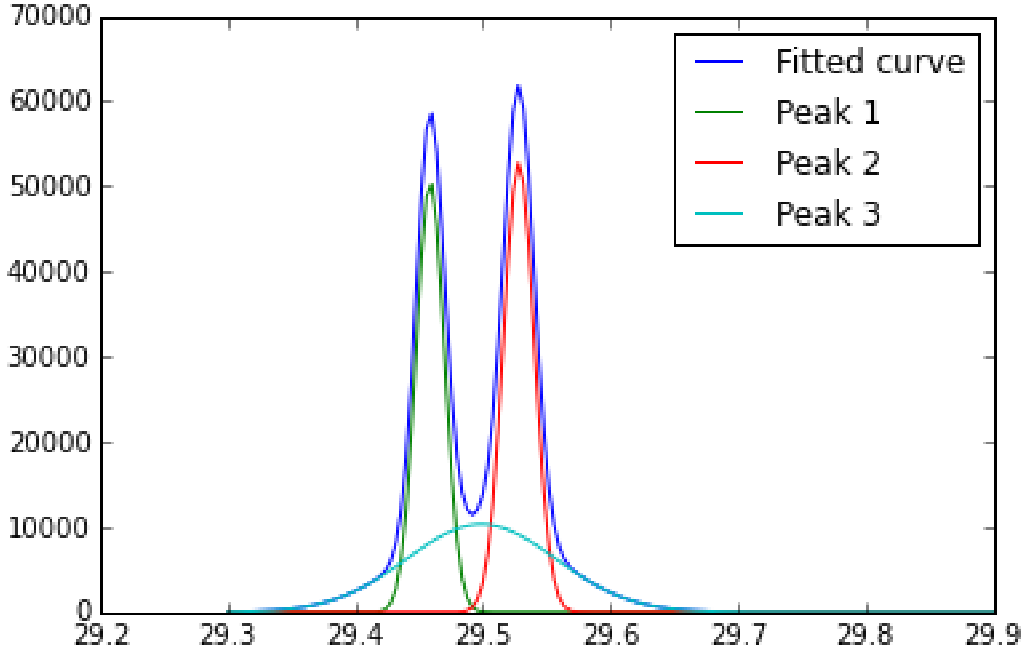
It turns out that we were wrong regarding the position of our third peak, and about the amplitudes of all 3 peaks. The fit converged towards a much better 3-peak solution:

```
In [10]: print(p)
         plot(xdata, sum_gauss(xdata, *p), label="Fitted curve")
         for i in range(3):
             plot(xdata, sum_gauss(xdata, *p[3*i:3*i+3]), label="Peak %d" % (i+1))
         legend()

[  5.05170978e+04   2.94588874e+01   2.62330656e-02   5.27459197e+04
   2.95283488e+01   2.84759182e-02   1.03471962e+04   2.94988854e+01
```

```
 -1.37750880e-01]
```

```
Out[10]: <matplotlib.legend.Legend at 0x7f1202cae860>
```



2 Conclusion

Fitting curves requires the following steps:

- get the data points x_data and y_data
- define the fitting function as y_data = function (x_data, param)
- choose an initial guess for the set of parameters p0
- run the optimizer
- check the result.

3 Fitting data with a background

Beyond simple curve fitting, *silx* offers more builtin fitting tools, to handle initial parameter estimation (including peak detection) and background estimation.

We need to import the `FitManager` class and the module defining builtin fit theories (`silx.math.fit.fittheories`).

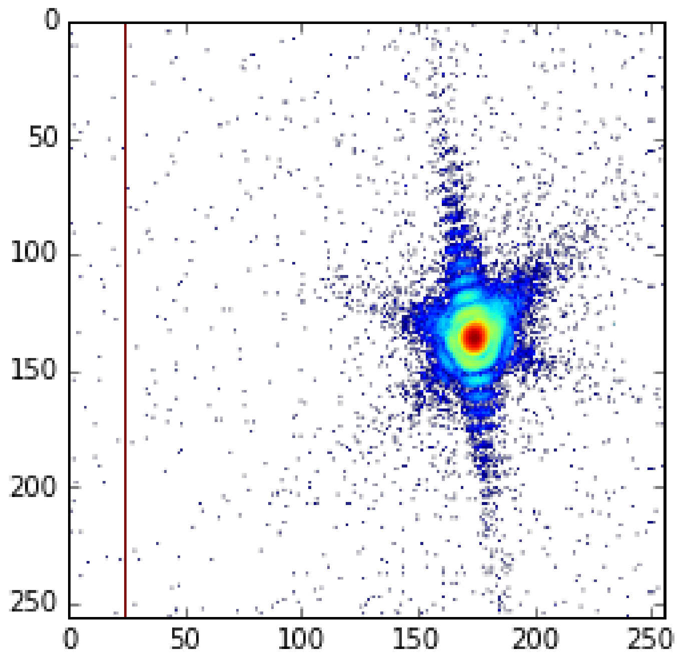
3.1 Loading data

Lets load an image, and sum all samples along the horizontal dimension to get a 1D curve.

```
In [11]: import fabio
         img = fabio.open("medipix.edf").data
         imshow(np.log(img))
```

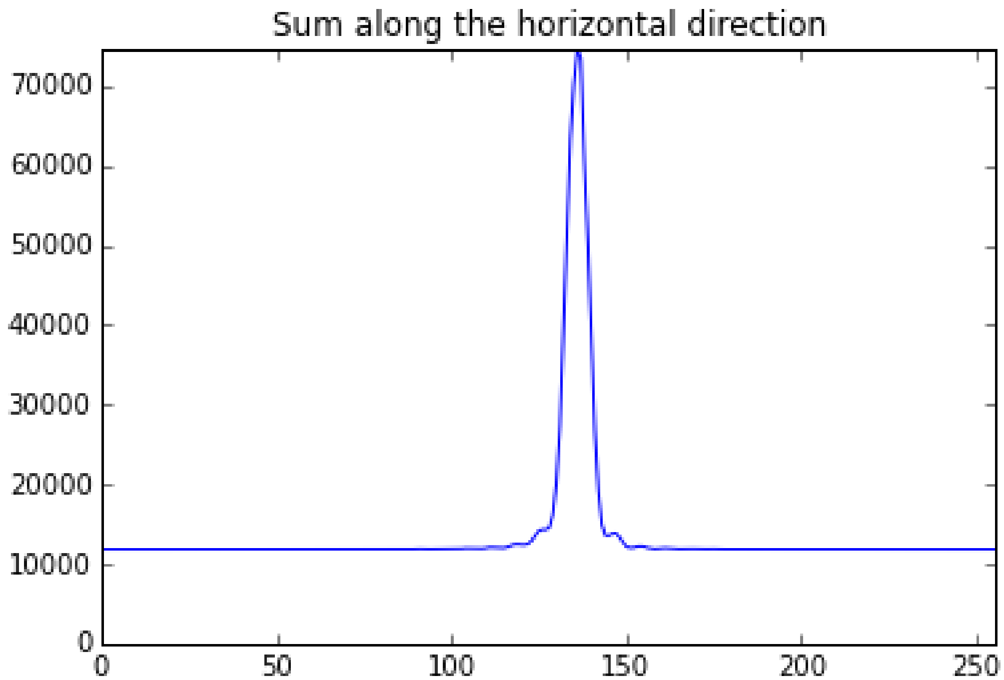
```
/users/knobel/.local/lib/python3.4/site-packages/ipykernel/__main__.py:3: RuntimeWa
  app.launch_new_instance()
```

```
Out[11]: <matplotlib.image.AxesImage at 0x7f1201866dd8>
```




```
In [12]: ydata = img.sum(axis=1)
         xdata = numpy.arange(len(img),dtype="float")
         plot(xdata, ydata)
         axis([0, xdata.max(), 0, ydata.max()])
         title("Sum along the horizontal direction")
```

```
Out[12]: <matplotlib.text.Text at 0x7f1201833f28>
```



3.2 Setting up a FitManager

If we want to fit a single gaussian to this data, we clearly need to remove the constant background signal, at about $y = 12000$.

```
In [13]: from silx.math.fit import FitManager, fittheories
         fitman = FitManager()
         fitman.setdata(x=xdata, y=ydata)
         fitman.loadtheories(fittheories)
         fitman.settheory('Gaussians')
         fitman.setbackground("Constant")
```

In *silx 0.3*, background theories are hardcoded in `FitManager`. In the next release, background theories will be handled the same way as fit theories::

```
from silx.math.fit import bgtheories
...
fitman.loadbgtheories(bgtheories)
```

This will enable adding custom background theories, the same way as we can add custom fit theories.

3.3 Estimating initial parameters

Now that the FitManager is set-up, we need to estimate the initial parameters before running the fit. This is done by simply running the `estimate` method of fitmanager. This method uses the estimation function defined in fit theories. The results are stored internaly in the FitManager instance as attribute `fit_results`.

This attribute stores a list of dictionaries, one per parameter to be fitted.

```
In [14]: fitman.estimate()

         for param in fitman.fit_results:
             print("Estimated value for param %s: %f" % (param["name"], param["esti

Estimated value for param Constant: 11810.000000
Estimated value for param Height1: 63176.944734
Estimated value for param Position1: 135.955454
Estimated value for param FWHM1: 7.078567
```

The estimation function managed to find 1 peak and correctly estimated its gaussian parameters and the background level.

3.4 Final fit

Running the fit is as simple as running the estimation.

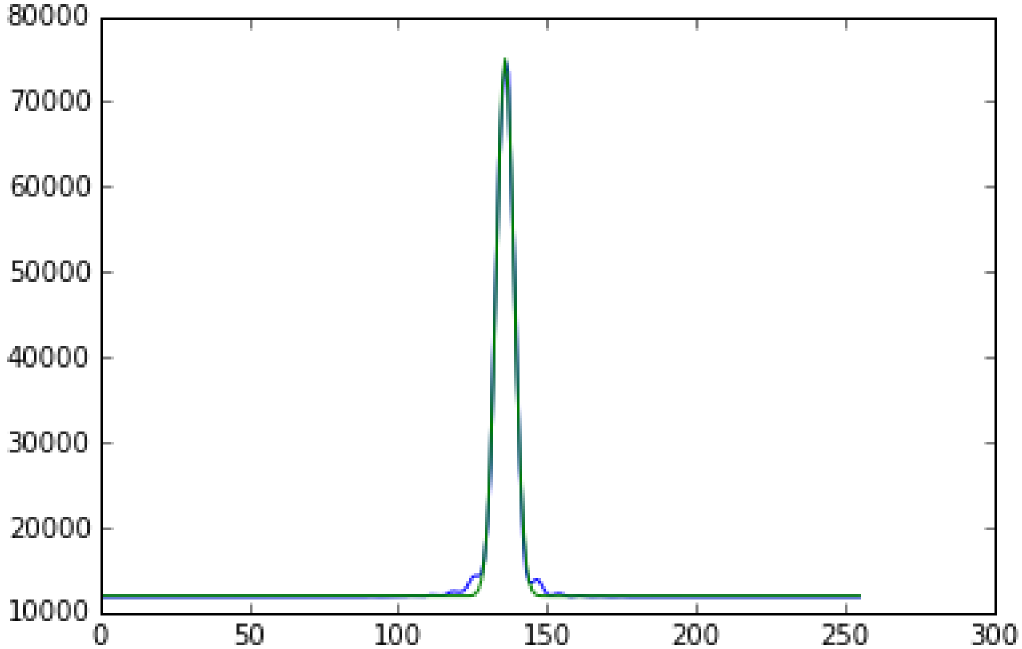
```
In [15]: params, uncertainties, infodict = fitman.runfit()

         for param in fitman.fit_results:
             print("Fitted value for param %s: %f" % (param["name"], param["fitresu

         plot(xdata, ydata, label="Original data")
         plot(xdata, params[0] + sum_gauss(xdata, *params[1:]), label="Fitted data'

Fitted value for param Constant: 11918.694707
Fitted value for param Height1: 63103.126632
Fitted value for param Position1: 135.955545
Fitted value for param FWHM1: 7.061779


Out[15]: [<matplotlib.lines.Line2D at 0x7f1201854fd0>]
```



As you can see, parameters can be accessed as the first value returned by the `runfit` method, a simple list of raw values, or again as attribute `fitman.fit_results`.

3.5 More FitManager features

FitManager offers more features, such as: - setting constraints for parameters - advanced background models, beyond simple constant or linear: strip or snip backgrounds - adding custom fit and background theories, with customised estimation function, customized derivatives…

4 Exercise

```
x = numpy.arange(10).astype(numpy.float64)
y = 0.001 * x**3 + 25.1 * x**2 + 1.2 * x - 25
```

1. Fit degree-2 polynomial to this data
2. Fit a degree-3 polynomial to this data
3. Fit a degree-3 polynomial to this data using FitManager.
4. Print the chi-square value and number of iterations for all previous tasks.

- Tip: For the 3-rd task, to add a model function to FitManager:

```
from silx.math.fit import FitTheory
from silx.math.fit import FitManager

def poly3(x, a, b, c, d):
    return a * x**3 + b * x**2 + c*x + d
```

```
my_poly3_theory = FitTheory(function=poly3, parameters=["a", "b", "c", "d"])

fitman = FitManager()
fitman.addtheory(name="my poly", theory=my_poly3_theory)
...
```

In [16]:
```python
# 1.
import numpy
from silx.math.fit import leastsq

x = numpy.arange(10).astype(numpy.float64)
y = 0.001 * x**3 + 25.1 * x**2 + 1.2 * x - 25


def poly2(x, a, b, c):
    return a * x**2 + b*x + c

p0 = [1., 1., 1.]

p, cov_matrix = leastsq(poly2, x, y, p0)

print("Parameters [a, b, c]: " + str(p))

# 4.
p, cov_matrix, info = leastsq(poly2, x, y, p0, full_output=True)
print(info["reduced_chisq"])
print(info["niter"])
```

```
Parameters [a, b, c]: [ 25.1135       1.15390001 -24.97480002]
0.000441257142858
4
```

In [17]:
```python
# 2.
def poly3(x, a, b, c, d):
    return a * x**3 + b*x**2 + c*x + d

p0 = [1., 1., 1., 1.]

p, cov_matrix = leastsq(poly3, x, y, p0)

print("Parameters [a, b, c, d]: " + str(p))

# 4.
p, cov_matrix, info = leastsq(poly3, x, y, p0, full_output=True)
print(info["reduced_chisq"])
print(info["niter"])
```

```
Parameters [a, b, c, d]: [  1.00000007e-03   2.51000000e+01   1.20000001e+00  -2.50
8.96788047245e-17
5
```

```
In [18]: # 3.
         from silx.math.fit import FitTheory
         from silx.math.fit import FitManager

         def poly3(x, a, b, c, d):
             return a * x**3 + b * x**2 + c*x + d

         my_poly3_theory = FitTheory(function=poly3, parameters=["a", "b", "c", "d"

         fitman = FitManager()
         fitman.addtheory("my poly", my_poly3_theory)
         fitman.settheory("my poly")

         fitman.setdata(x, y)

         # FitManager.estimate() returns an array of initial parameters
         p0 = fitman.estimate()
         # ... and as we didn't define an estimation function, it will be an array
         print("Estimated parameters: ", p0)

         # FitManager.runfit() returns the same data as leastsq(..., full_output=Tr
         p, cov, info = fitman.runfit()

         print("Parameters [a, b, c, d]: " + str(p))

         for param in fitman.fit_results:
             print("parameter %s: %f" % (param["name"], param["fitresult"]))

         # 4.
         print(fitman.chisq)
         print(fitman.niter)
```

```
Estimated parameters:  [ 1.  1.  1.  1.]
Parameters [a, b, c, d]: [  1.00000028e-03   2.51000000e+01   1.20000001e+00  -2.50
parameter a1: 0.001000
parameter b1: 25.100000
parameter c1: 1.200000
parameter d1: -25.000000
4.7203142867e-17
5
```

```
In [ ]:
```