Practical Kinetics

Exercise 2:

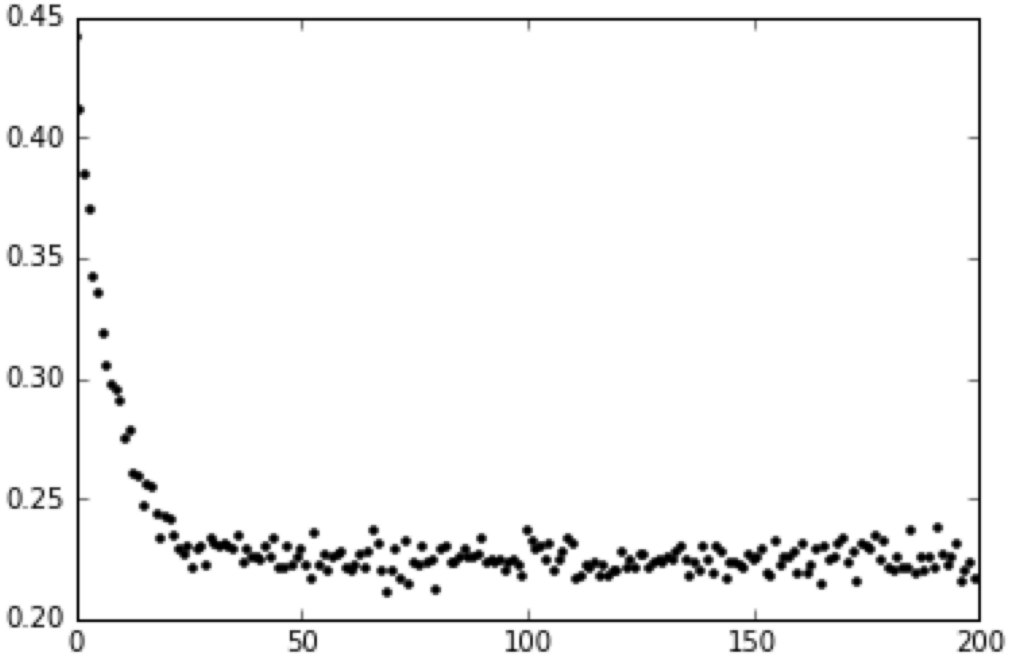
Integral Rate Laws

Objectives:

1. Import data from Excel and CSV
2. Fit to zero-, first-, and second-order rate laws
3. Error analysis

Introduction

In this exercise, we will examine some synthetic absorbance vs. time data:



What is the kinetic order in this reaction?

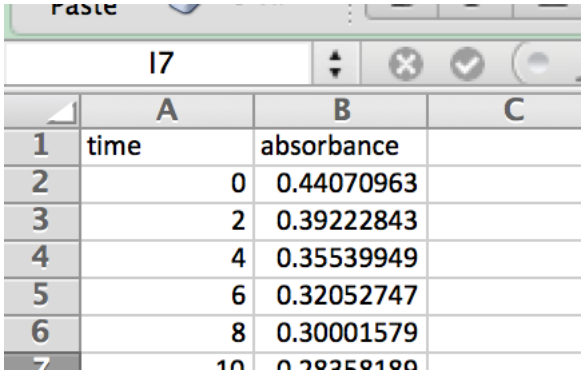
What are the rate constants? Error bars?

What if other materials absorb at this wavelength?

Step 1: Importing Data

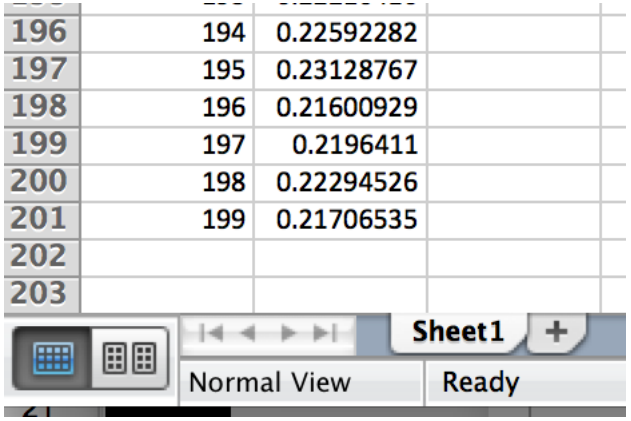
Please ensure you have `dataset1.xlsx` and `dataset1.csv` in the directory where you start your IPython Notebook.

XLSX is the file extension for Microsoft Excel. Here is what the spreadsheet looks like:



	A	B	C
1	time	absorbance	
2	0	0.44070963	
3	2	0.39222843	
4	4	0.35539949	
5	6	0.32052747	
6	8	0.30001579	

We'll deal with the simplest case, where there is only one sheet:



196	194	0.22592282	
197	195	0.23128767	
198	196	0.21600929	
199	197	0.2196411	
200	198	0.22294526	
201	199	0.21706535	
202			
203			

Step 1: Importing Data

CSV stands for **c**omma **s**eparated **v**alue. This is a common format for instrumental data. You can also save Excel spreadsheets as CSVs.

CSVs are in plain text, so you can use Notepad (PC), Textedit (Mac), or the command line to look at CSVs. As you can see, this is the exact same data:

```
~/kinetics_tutorial $ head dataset1.csv
time,absorbance
0.0,0.441875923634
1.0,0.411688006804
2.0,0.384199698532
3.0,0.369743385429
4.0,0.342148055601
5.0,0.335511711496
```

Let's convert these files directly into Python lists. You will need some imports:

```
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
from math import exp
import pandas as pd
```

You've seen some of these, but pandas is the standard Python data analysis library.

The name derives from "**pan**el **da**ta."

Step 1: Importing Data

Let's read in the Excel file first:

```
df = pd.read_excel("dataset1.xlsx")
df.head()
```

	time	absorbance
0	0	0.441876
1	1	0.411688
2	2	0.384200
3	3	0.369743
4	4	0.342148

`pd` represents pandas. `df` stands for DataFrame, which is the pandas version of a spreadsheet. To put the time and absorbances in lists:

```
time = df["time"].tolist()
absorbance = df["absorbance"].tolist()
print time[:5]
print absorbance[:5]
```

Step 1: Importing Data

```
time = df["time"].tolist()
absorbance = df["absorbance"].tolist()
print time[:5]
print absorbance[:5]
```

```
[0, 1, 2, 3, 4]
[0.44187592363442152, 0.4116880068038154, 0.38419969853187741,
 0.3697433854286537, 0.34214805560105632]
```

In English:

Take the time/absorbance column and make it a list. Print out the first five entries.

There are more sophisticated ways to extract specific columns from specific sheets, but they are beyond the scope of this exercise. Please see:

http://www.gregreda.com/2013/10/26/intro-to-pandas-data-structures/

http://byumcl.bitbucket.org/bootcamp2014/labs/pandas_types.html

Step 1: Importing Data

To load the CSV data, we can use numpy:

```
data = np.genfromtxt("dataset1.csv", delimiter=",", skip_header=1)
print data[:3]
time=data[:,0]
absorbance=data[:,1]
print time[:5]
print absorbance[:5]
```

The output is:

```
[[ 0.          0.44187592]
 [ 1.          0.41168801]
 [ 2.          0.3841997 ]]
[ 0.  1.  2.  3.  4.]
[ 0.44187592  0.41168801  0.3841997   0.36974339  0.34214806]
```

In English:

Load CSV data from `dataset1.csv`, where the columns are separated by commas, and ignore the first row. Print out the first three entries.

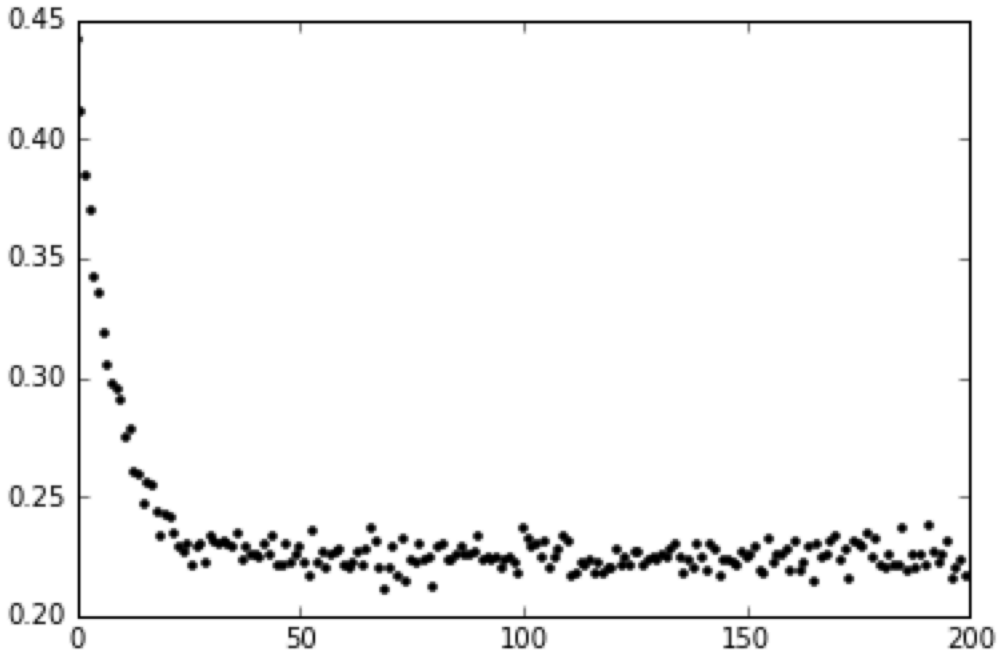
As you can see, the data get loaded in “tuple” form, so we use `[:,0]` to transpose the data into column vectors. We then print out the first five elements.

Step 2: A First Look

Let's take a look at the data:

```
plt.plot(time,absorbance,"k.")
```

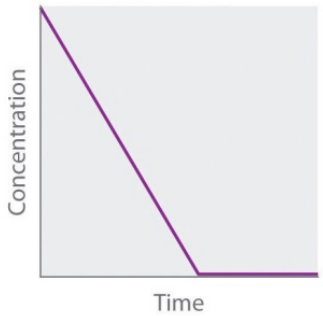
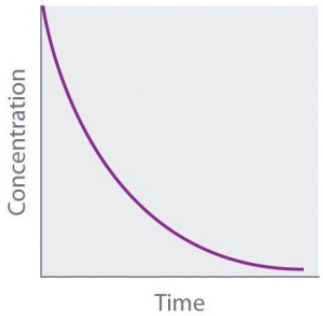
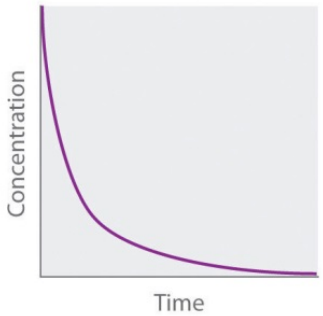
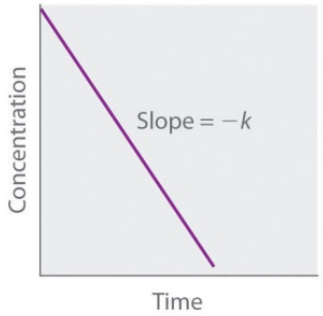
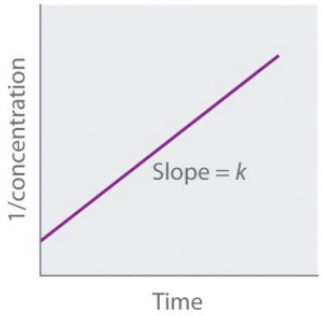
```
[<matplotlib.lines.Line2D at 0x11166dbd0>]
```



What is the kinetic order?

Step 2: A First Look

As a reminder:

	Zeroth Order	First Order	Second Order
Differential rate law	$\text{Rate} = -\frac{\Delta[A]}{\Delta t} = k$	$\text{Rate} = -\frac{\Delta[A]}{\Delta t} = k[A]$	$\text{Rate} = -\frac{\Delta[A]}{\Delta t} = k[A]^2$
Concentration vs. time			
Integrated rate law	$[A] = [A]_0 - kt$	$[A] = [A]_0 e^{-kt}$ or $1n[A] = 1n[A]_0 - kt$	$\frac{1}{[A]} = \frac{1}{[A]_0} + kt$
Straight-line plot to determine rate constant			

Step 2: A First Look

Plot the linearized forms of the zero-, first-, and second-order integral rate laws:

```
corrected_absorbance = absorbance - np.min(absorbance)
log_absorbance = np.log(corrected_absorbance)
one_over_time = 1.0/time
one_over_absorbance = 1.0/absorbance

plt.plot(time[:30],absorbance[:30],"k.")
plt.show()
plt.plot(time[:30],log_absorbance[:30],"r.")
plt.show()
plt.plot(one_over_time[:30],one_over_absorbance[:30],"b.")
plt.show()
```

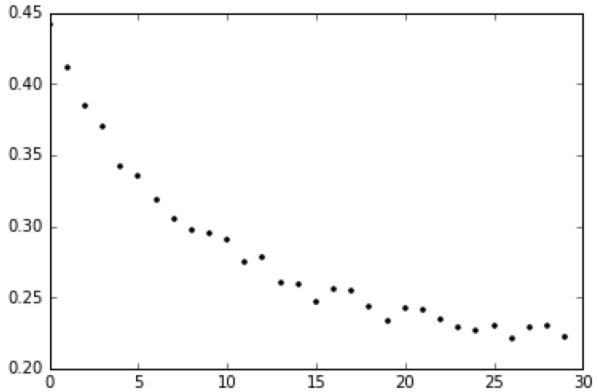
`np.min` finds the minimum of a list. If we don't do this, then we will get some negative numbers when we take the natural log with `np.log`.

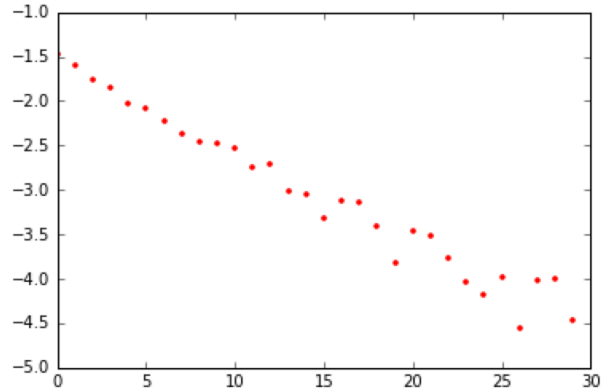
Since most of the run was noise, I'm only showing the first thirty points. In reality, it would not be a good idea to collect data past five half-lives.

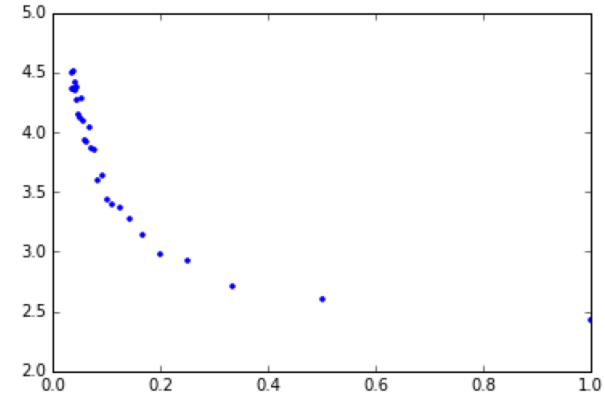
For now, let's ignore the fact that absorbance is not equal to concentration.

Step 2: A First Look

Plot the linearized forms of the zero-, first-, and second-order integral rate laws:







(In your browser, these will appear one after another instead of side by side.)

Clearly, the first-order fit is best. If we call the starting material "A," then:

$$[A] = [A]_0 - \exp(-kt)$$

If the system obeys Beer's law for the experimental concentration (*c*) range, then:

$$absorbance = \varepsilon bc = \varepsilon'[A]$$

where I combined the path length (*b*) and the extinction coefficient (ε) into ε'.

Step 3: First-Order Fit

Previously (lectures 1 and 2), we showed that when other species also absorb, we pick up a constant *c*. (This could also account for any background absorbance, like that of the container.)

Therefore:

$$\begin{aligned} absorbance &= \varepsilon'[A] + c \\ &= \varepsilon'[A]_0 - \varepsilon'\exp(-kt) + c \\ &= c_1 - c_2 \exp(-kt) \end{aligned}$$

where c_1 represents the infinity value, c_2 is ε', and *k* is the rate constant.

As you can see, we don't need to know the extinction coefficient to determine *k*.

Now, let's fit the data to this three-parameter function. As in Exercise 1, we'll use `scipy.optimize.curve_fit`, a least squares tool.

Step 3: First-Order Fit

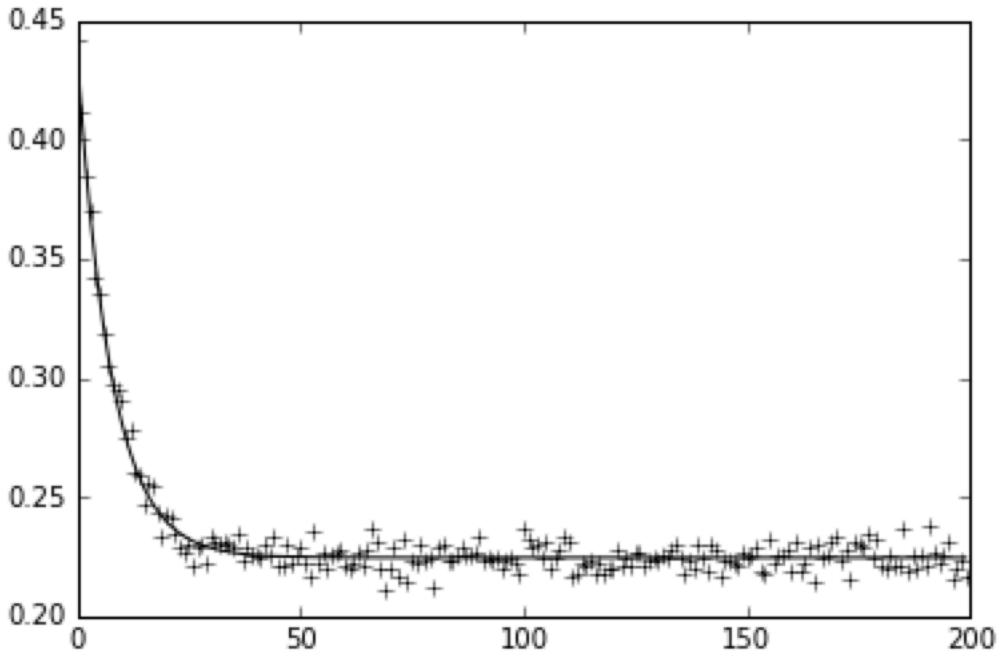
```
from scipy.optimize import curve_fit

def first_order_function(t, pre_factor, rate_constant, offset):
    return pre_factor*np.exp(-rate_constant*t) + offset

popt, pcov = curve_fit(first_order_function, time, absorbance)
errors = np.sqrt(np.diag(pcov))
print "pre_factor: %7.4f ± %6.4f" % (popt[0], errors[0])
print "rate_const: %7.4f ± %6.4f" % (popt[1], errors[1])
print "offset:     %7.4f ± %6.4f" % (popt[2], errors[2])

fitted_absorbance = [ first_order_function(t, popt[0], popt[1], popt[2]) for t
in time ]

plt.plot(time,absorbance,"k+")
plt.plot(time,fitted_absorbance,"k")
```

We use `np.sqrt(np.diag(pcov))` to extract the uncertainties in the optimized parameters (`popt`).

This form of the print statement allows us to specify the number of decimal places to print out. `%7.4f` means: “Print out this **f**loating point number with seven characters, using 4 decimal places.” The arguments after the final % are read in order: the first `%7.4f` refers `popt[0]`, the second `%6.4f` refers to `errors[0]`.

Step 3: First-Order Fit

```
pre_factor:   0.2131 ± 0.0033
rate_const:   0.1320 ± 0.0032
offset:       0.2245 ± 0.0004

[<matplotlib.lines.Line2D at 0x112733210>]
```

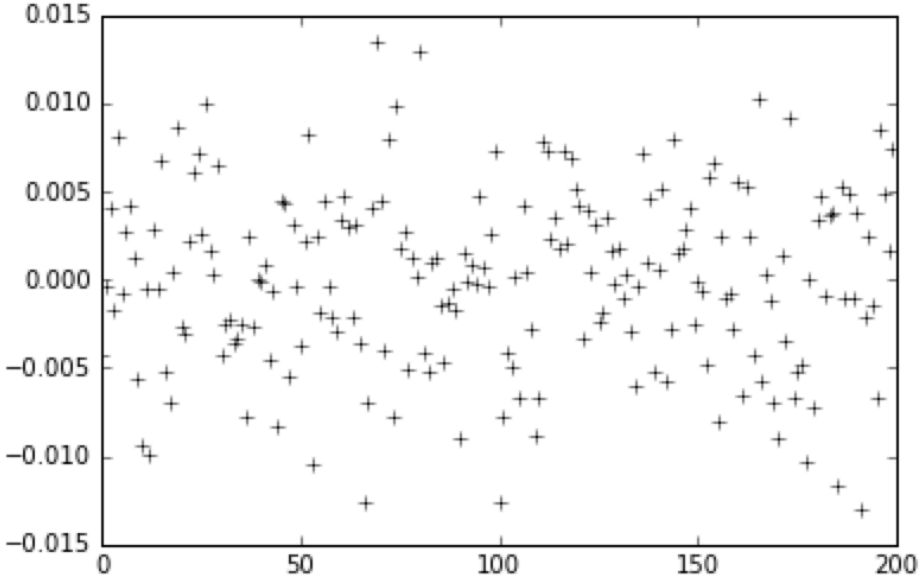


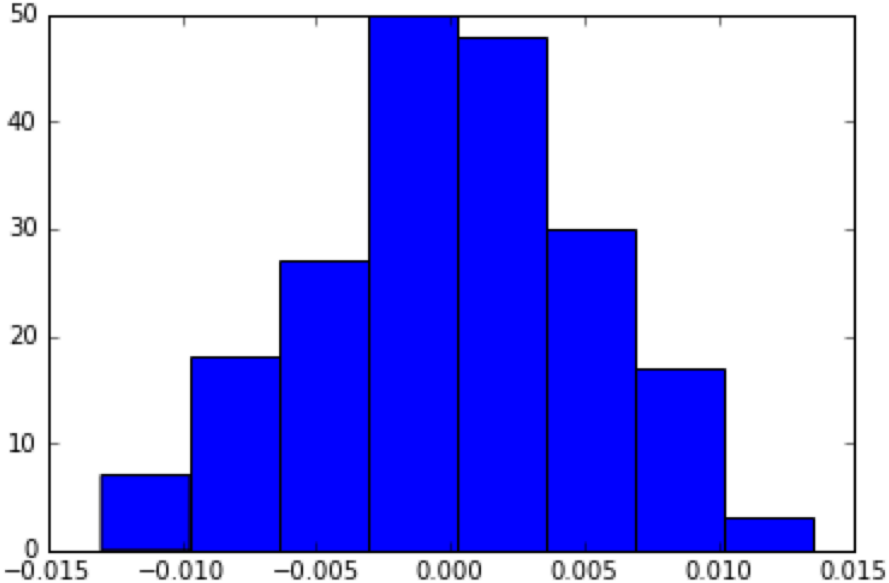
Visually, this looks like a good fit. But just how good is it actually?

We should look at the **residuals**, the difference between the observed and predicted values.

Step 4: Plot the Residuals

```
residual = fitted_absorbance - absorbance
plt.plot(time, residual, "k+")
plt.show()
plt.hist(residual,bins=8)
plt.show()
standard_deviation = np.std(residual)
print "standard deviation: %5.2E" % standard_deviation
```





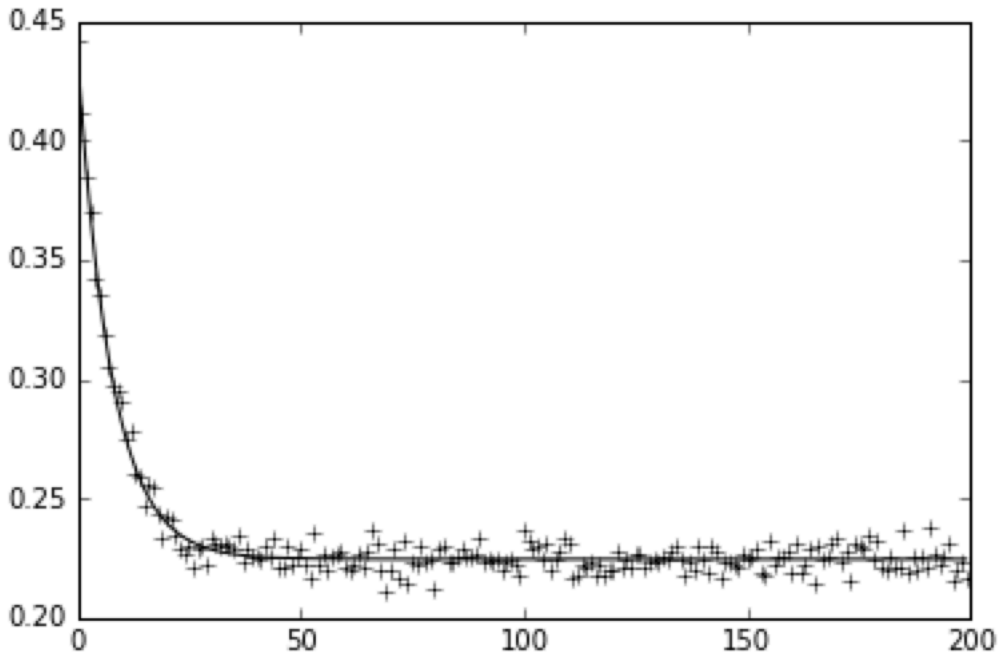

```
standard deviation: 5.11E-03
```

The residuals are normally distributed, which is a sign of a good fit.

We made a histogram with `plt.hist`. We also printed out the standard deviation in scientific notation (`%5.2E`). We will need this value in a moment.

Step 5: Fit with Error Bars

Here is our fit again:



Notice that the latter part of the experiment is all noise. It doesn't make sense to fit each point with equal weight! As a result, the error bars in the parameters we got are meaningless:

```
pre_factor:   0.2131 ± 0.0033
rate_const:   0.1320 ± 0.0032
offset:       0.2245 ± 0.0004
```

Step 5: Fit with Error Bars

Let's fit again, but weight each point by the signal to noise ratio:

```
weights = [ (i/standard_deviation)**2 for i in absorbance ]

popt, pcov = curve_fit(first_order_function, time, absorbance, sigma=weights)
errors = np.sqrt(np.diag(pcov))

print "pre_factor: %7.4f ± %6.4f" % (popt[0], errors[0])
print "rate_const: %7.4f ± %6.4f" % (popt[1], errors[1])
print "offset:     %7.4f ± %6.4f" % (popt[2], errors[2])

fitted_absorbance2 = [ first_order_function(t, popt[0], popt[1], popt[2]) for
t in time ]

plt.plot(time,absorbance,"k+")
plt.plot(time,fitted_absorbance2,"k")
```

This kind of square weighting is standard.

Note that the result is now going into `fitted_absorbance_2`.

Step 5: Fit with Error Bars

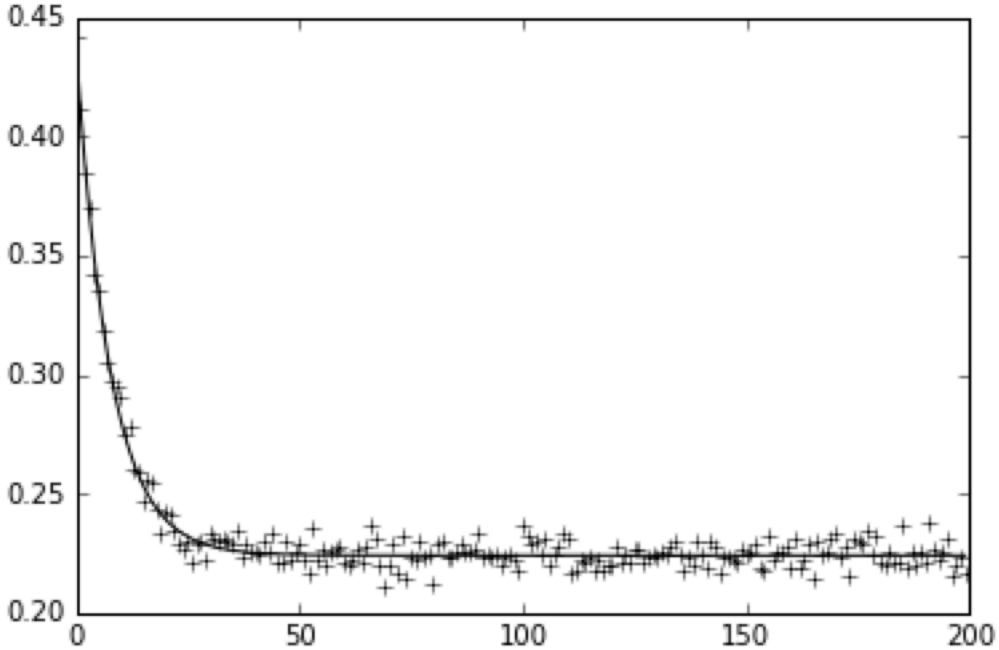
The fit has changed slightly, but the error bars are now meaningful:

```
pre_factor:   0.2139 ± 0.0090
rate_const:   0.1321 ± 0.0053
offset:       0.2241 ± 0.0004
```

```
pre_factor:   0.2131 ± 0.0033
rate_const:   0.1320 ± 0.0032
offset:       0.2245 ± 0.0004
```

(old)

```
[<matplotlib.lines.Line2D at 0x112904b50>]
```



Step 6: Chi-Square

A more accurate indication of the goodness-of-fit is the chi-square statistic:

$$\chi^2 = \sum \frac{\left(\text{observed} - \text{expected}\right)^2}{\text{expected}^2}$$

Just as the normal distribution describes the sampling distribution of the mean, the chi-square distribution describes the sampling distribution of the variance.

In our case, we have a series of normally distributed random variables with means μ_i and standard deviations σ_i, chi-square is defined as:

$$\chi^2 = \sum \frac{\left(x_i - \mu_i\right)^2}{\sigma_i^2}$$

Since the fluctuation about each mean is on the order of σ, we expect this to sum to about the number of data points.

Step 6: Chi-Square

We can then employ the following test procedure:

1. Calculate chi-square.

2. Calculate the degrees of freedom as the number of data points minus the number of model parameters.

3. Divide chi-square by the number of degrees of freedom to get the “goodness of fit” or “normalized chi-square.”

The expected value of this statistic is 1.0, such that:

goodness of fit >> 1: poor fit

goodness of fit = 1: good fit

goodness of fit << 1: error bars are underestimated

For a more detailed analysis: http://maxwell.ucsc.edu/~drip/133/ch4.pdf

For another Python implementation:
http://www.physics.utoronto.ca/~phy326/python/ (curve fit to data)

Step 6: Chi-Square

In Python:

```
dof = len(time)-3

def chi_square(observed, expected, stdev):
    chi_squared_value = 0.0

    for i in range(len(observed)):
        o = observed[i]
        e = expected[i]
        chi_squared_value += ((o-e) / standard_deviation)**2

    return chi_squared_value

goodness_of_fit =
chi_square(absorbance,fitted_absorbance2,standard_deviation) / dof
print "chi_squared / dof = %.4f" % goodness_of_fit
```

We find a good fit:

```
chi_squared / dof = 1.0233
```

Step 7: Zeroth-Order Fit

For a zeroth-order process:

$$
\begin{aligned}
[A] &= [A]_0 - kt \\
absorbance &= \varepsilon'[A] + c \\
&= \varepsilon'\left([A]_0 - kt\right) + c \\
&= c_1 - kt
\end{aligned}
$$

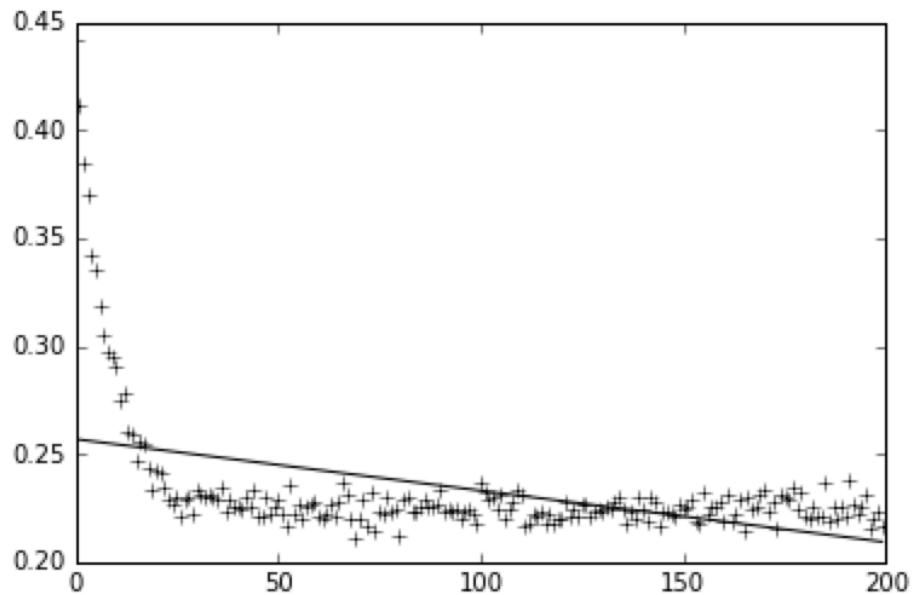
In the last step, I collapsed the first and third terms into one constant.

Again, we can determine what the rate constant is without knowing what the extinction coefficient is.

Step 7: Zeroth-Order Fit

```
def zero_order_function(t, c_1, rate_constant):
    return c_1 - rate_constant*t

popt, pcov = curve_fit(zero_order_function, time, absorbance)
errors = np.sqrt(np.diag(pcov))
print "c_1:        %7.4f ± %6.4f" % (popt[0], errors[0])
print "rate_const: %7.4f ± %6.4f" % (popt[1], errors[1])

fitted_absorbance3 = [ zero_order_function(t, popt[0], popt[1]) for t in
time ]

plt.plot(time,absorbance,"k+")
plt.plot(time,fitted_absorbance3,"k")
```

```
c_1:        0.2569 ± 0.0039
rate_const: 0.0002 ± 0.0000

[<matplotlib.lines.Line2D at 0x11b6b1b50>]
```



We defined a new function, `zero_order_function`, to represent the zeroth order rate law.

The fit is terrible.

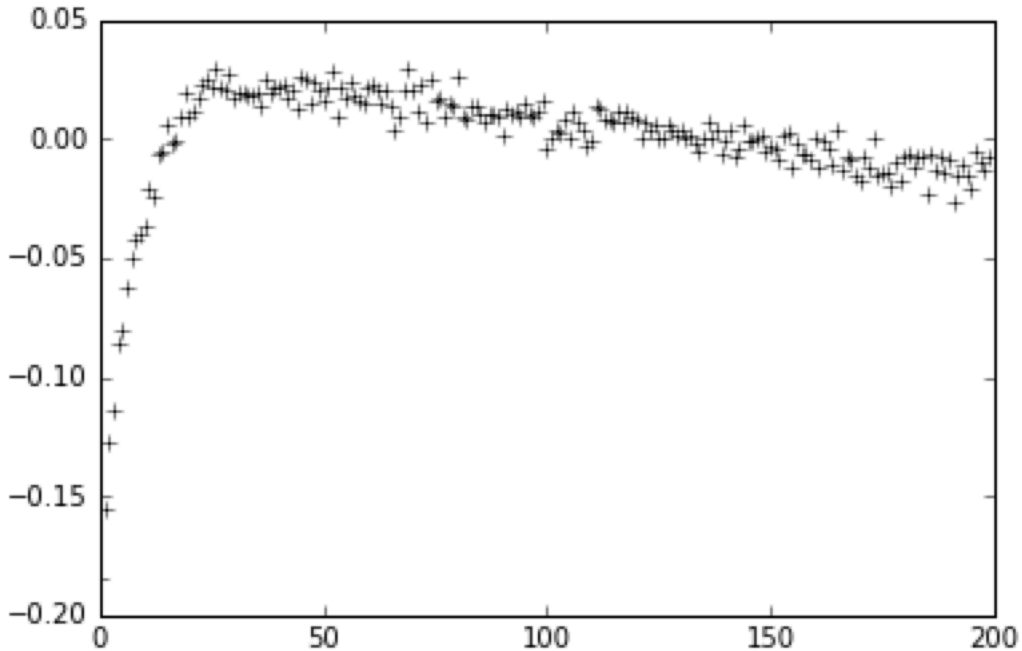
For the chi-square analysis, we need to adjust the degrees of freedom for this 2-parameter fit.

Step 7: Zeroth-Order Fit

The goodness of fit is bad as well:

```
dof = len(time) - 2
residual3 = fitted_absorbance3 - absorbance
plt.plot(time,residual3,"k+")
goodness_of_fit =
chi_square(absorbance,fitted_absorbance3,standard_deviation) / dof
print "chi_squared / dof = %.4f" % goodness_of_fit
```

```
chi_squared / dof = 28.6341
```



Curved residuals are a strong visual indicator of a poor fit.

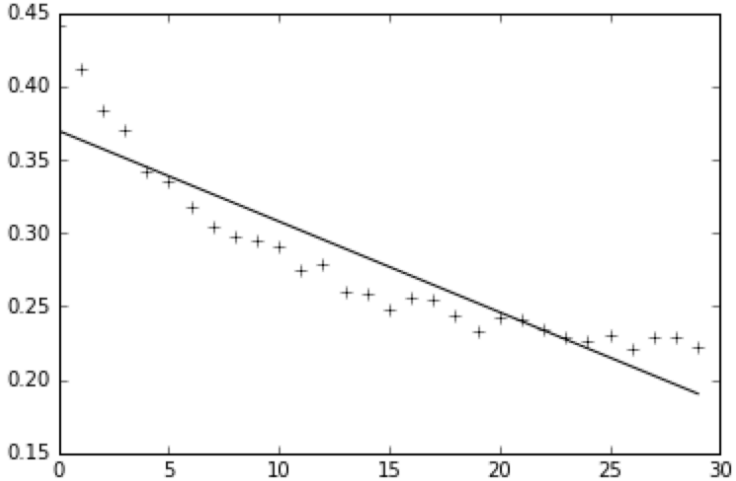
Step 7: Zeroth-Order Fit

The fit is artifically bad because we fitted a lot of points at very high conversions:

```
popt, pcov = curve_fit(zero_order_function, time[:30], absorbance[:30])
errors = np.sqrt(np.diag(pcov))
print "c_1:        %7.4f ± %6.4f" % (popt[0], errors[0])
print "rate_const: %7.4f ± %6.4f" % (popt[1], errors[1])

fitted_absorbance3 = [ zero_order_function(t, popt[0], popt[1]) for t in
time[:30] ]

goodness_of_fit = chi_square(absorbance[:
30],fitted_absorbance3,standard_deviation) / dof
print "chi_squared / dof = %.4f" % goodness_of_fit

plt.plot(time[:30],absorbance[:30],"k+")
plt.plot(time[:30],fitted_absorbance3[:30],"k")
```

This uses `[:30]` to deal with the first thirty points only.

```
c_1:          0.3697 ± 0.0090
rate_const:   0.0062 ± 0.0005
chi_squared / dof = 3.4891
```



Step 8: Second-Order Fit

For a second-order reaction:

$$\frac{1}{[A]} = \frac{1}{[A]_0} + kt$$

$$[A] = \frac{1}{\frac{1}{[A]_0} + kt}$$

$$\begin{aligned} absorbance &= \varepsilon'[A] + c \\ &= \frac{\varepsilon'}{\frac{1}{[A]_0} + \frac{kt[A]_0}{[A]_0}} + c \\ &= \frac{\varepsilon'[A]_0}{1 + k[A]_0 t} + c \\ &= \frac{c_1}{1 + c_2 t} + c_3 \end{aligned}$$

This is a three parameter fit. We cannot determine what the rate constant is unless we know what the initial concentration is because c_2 contains both k and $[A]_0$.

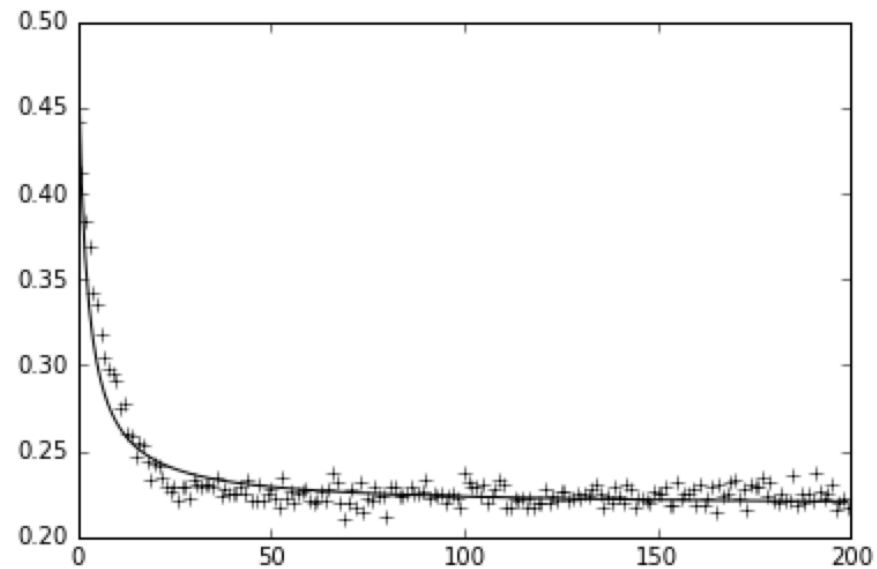
Step 8: Second-Order Fit

```
def second_order_function(t, c_1, c_2, c_3):
    return (c_1 / ( 1.0 + c_2*t )) + c_3

popt, pcov = curve_fit(second_order_function, time, absorbance,sigma=weights)
errors = np.sqrt(np.diag(pcov))
print "c_1: %7.4f ± %6.4f" % (popt[0], errors[0])
print "c_2: %7.4f ± %6.4f" % (popt[1], errors[1])
print "c_3: %7.4f ± %6.4f" % (popt[2], errors[2])

fitted_absorbance4 = [ second_order_function(t, popt[0], popt[1], popt[2]) for
t in time ]

plt.plot(time,absorbance,"k+")
plt.plot(time,fitted_absorbance4,"k")
```

```
c_1:  0.2813 ± 0.0222
c_2:  0.4697 ± 0.0570
c_3:  0.2177 ± 0.0007

[<matplotlib.lines.Line2D at 0x112b7c3d0>]
```

This fit actually looks decent! Qualitatively, first- and second-order reactions are very similar, but second-order reactions are more senstitive to concentration.

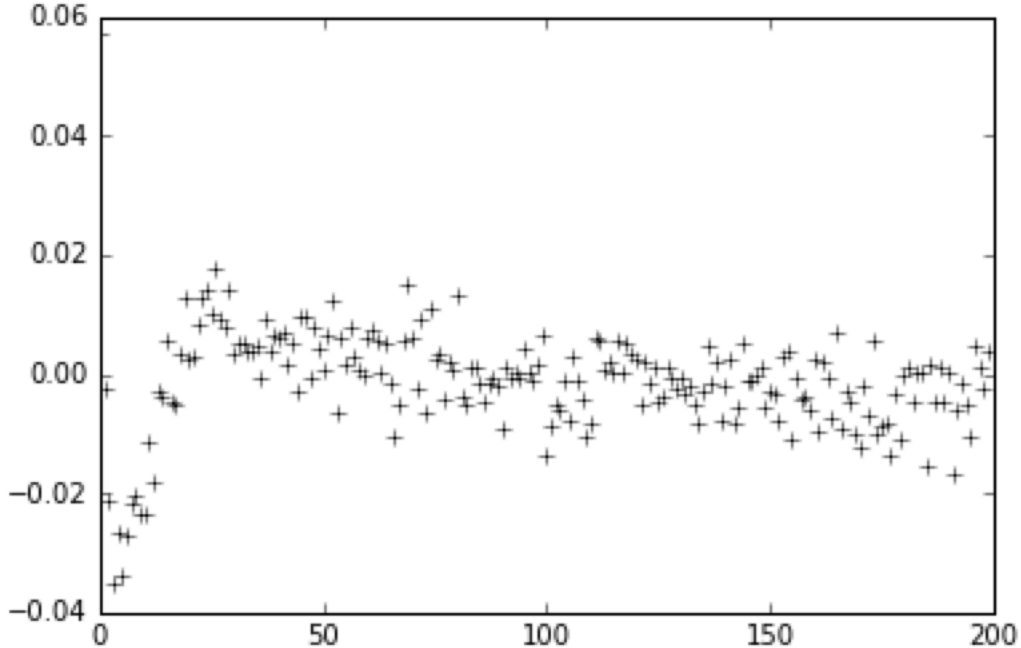
Let's look at the residuals and goodness of fit.



Step 8: Second-Order Fit

```
dof = len(time) - 3
residuals4 = fitted_absorbance4 - absorbance
plt.plot(time,residuals4,"k+")
goodness_of_fit =
chi_square(absorbance,fitted_absorbance4,standard_deviation) / dof
print "chi_squared / dof = %.4f" % goodness_of_fit
```

```
chi_squared / dof = 3.3651
```



Again, the residuals are strongly curved. The goodness of fit is significantly different from 1.0.

Summary

Useful imports for fitting:

```
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
from math import exp
import pandas as pd
from scipy.optimize import curve_fit
```

To import data from Excel:

```
import pandas as pd

df = pd.read_excel("dataset1.xlsx")
df.head()
time = df["time"].tolist()
absorbance = df["absorbance"].tolist()
print time[:5]
print absorbance[:5]
```

Summary

To import data from CSV:

```
import numpy as np
data = np.genfromtxt("dataset1.csv", delimiter=",", skip_header=1)
print data[:3]
time=data[:,0]
absorbance=data[:,1]
print time[:5]
print absorbance[:5]
```

This requires the numpy library. List slicing (i.e., the notation in the square brackets) is needed to rearrange the data into vector form.

To analyze absorbance data:

- The system should obey Beer's Law over the experimental concentration range.

- The observed absorbance at a given wavelength can reflect the absorbances of species other than the one of interest, including that of background.

- One can convert the absorbance data to concentration data before analysis using a standard curve. For zeroth- and first-order reactions, this is not necessary.

Sample Code

This code demonstrates the concepts described above in one place.

It does not depend on any of the code we just wrote. You can simply paste it into a new notebook and run it.

```
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
from math import exp
import pandas as pd
from scipy.optimize import curve_fit

data = np.genfromtxt("dataset1.csv", delimiter=",", skip_header=1)
time=data[:,0]
absorbance=data[:,1]

def first_order_function(t, pre_factor, rate_constant, offset):
    return pre_factor*np.exp(-rate_constant*t) + offset

popt, pcov = curve_fit(first_order_function, time, absorbance)
fitted_absorbance = [ first_order_function(t, popt[0], popt[1], popt[2]) for t in time ]
residual = fitted_absorbance - absorbance
standard_deviation = np.std(residual)
weights = [ (i/standard_deviation)**2 for i in absorbance ]
```

Sample Code

(continued)

```
popt, pcov = curve_fit(first_order_function, time, absorbance, sigma=weights)
errors = np.sqrt(np.diag(pcov))
print "pre_factor: %7.4f ± %6.4f" % (popt[0], errors[0])
print "rate_const: %7.4f ± %6.4f" % (popt[1], errors[1])
print "offset:     %7.4f ± %6.4f" % (popt[2], errors[2])

fitted_absorbance = [ first_order_function(t, popt[0], popt[1], popt[2]) for t in time ]

dof = len(time)-3
def chi_square(observed, expected, stdev):
    chi_squared_value = 0.0
    for i in range(len(observed)):
        o = observed[i]
        e = expected[i]
        chi_squared_value += ((o-e) / standard_deviation)**2
    return chi_squared_value
goodness_of_fit = chi_square(absorbance,fitted_absorbance,standard_deviation) / dof
print "chi_squared / dof = %.4f" % goodness_of_fit

plt.plot(time,absorbance,"k+")
plt.plot(time,fitted_absorbance,"k")
```

```
pre_factor:  0.2139 ± 0.0090
rate_const:  0.1321 ± 0.0053
offset:      0.2241 ± 0.0004
chi_squared / dof = 1.0233
```