Practical Kinetics

Exercise 3:

Initial Rates vs. Reaction Progress Analysis

Objectives:

1. Initial Rates
2. Whole Reaction Kinetic Analysis
3. Extracting Rate Constants

Introduction

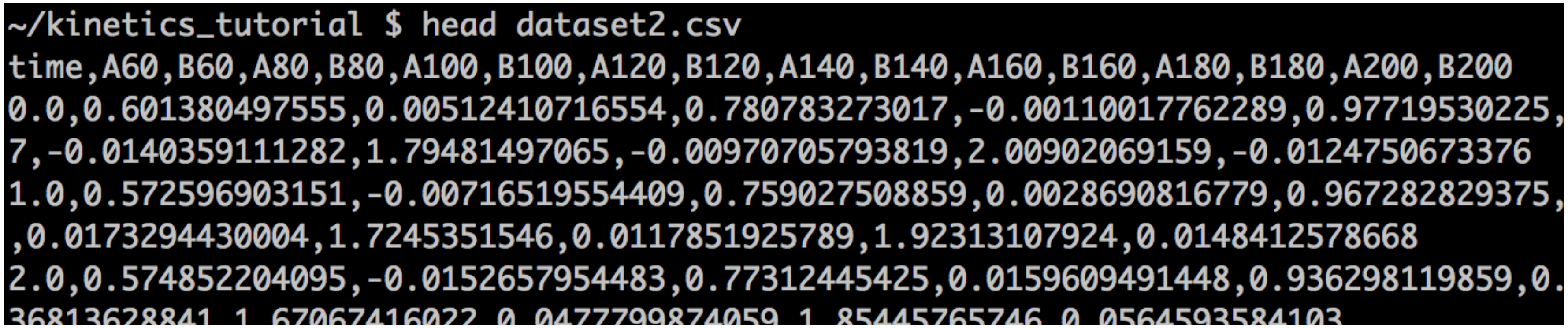
In this exercise, we will examine a concentration vs. time dataset for a hypothetical stoichiometric reaction:

$$n\,A \xrightarrow{k} B$$

1. What is the order in A?

2. What is the rate constant?

In `dataset2.csv`, you will find 8 experimental runs. Each column is labeled by the reagent name and the starting concentration of A in units of 0.01 M. For example, **A60** and **B60** are the concentrations of A and B with a starting A concentration of 0.60 M. Each run contains 1000 s worth of data in CSV format:



```
~/kinetics_tutorial $ head dataset2.csv
time,A60,B60,A80,B80,A100,B100,A120,B120,A140,B140,A160,B160,A180,B180,A200,B200
0.0,0.601380497555,0.00512410716554,0.780783273017,-0.00110017762289,0.97719530225,
7,-0.0140359111282,1.79481497065,-0.00970705793819,2.00902069159,-0.0124750673376
1.0,0.572596903151,-0.00716519554409,0.759027508859,0.0028690816779,0.967282829375,
,0.0173294430004,1.7245351546,0.0117851925789,1.92313107924,0.0148412578668
2.0,0.574852204095,-0.0152657954483,0.77312445425,0.0159609491448,0.936298119859,0.
36813628841,1.67067416022,0.0477799874059,1.85445765746,0.0564593584103
```

Step 1: Introduction to Pandas

We will need some import statements:

```
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
import pandas as pd
from pandas import Series, DataFrame
from scipy.optimize import curve_fit
from scipy.stats import linregress
```

Pandas, introduced in Exercise 2, is a data analysis package. It will let us keep track of our data in a central "spreadsheet" called a DataFrame. Each column of data is called a Series. This will be much more convenient than using many lists or numpy arrays.

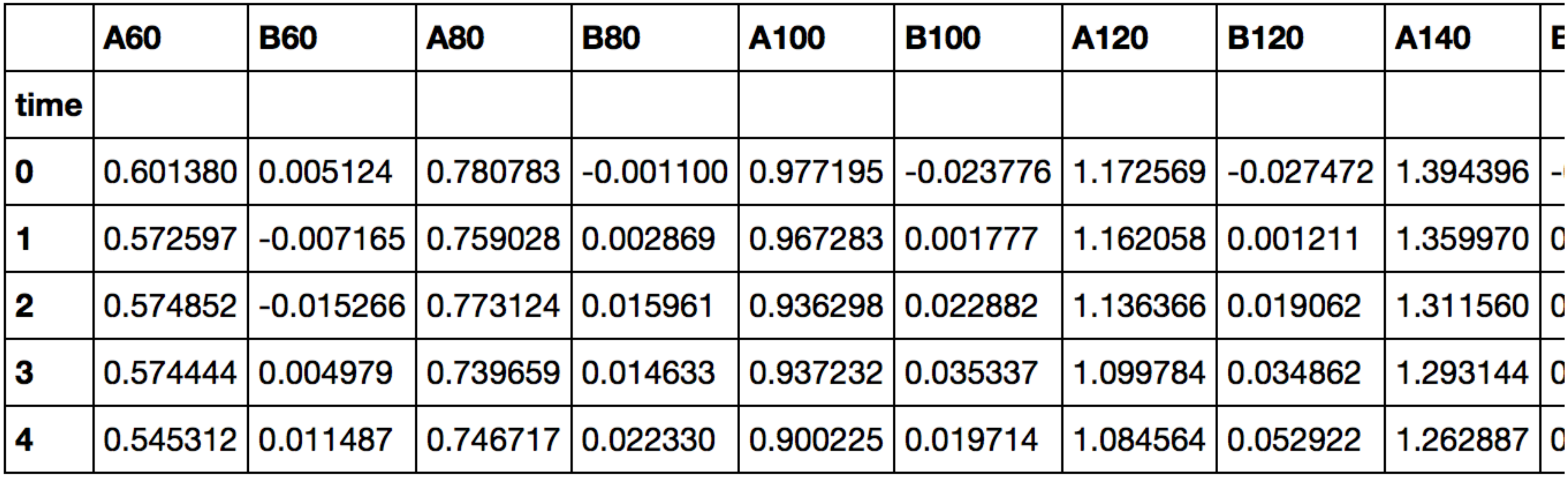
We will also need to do linear regression using **scipy.stats.linregress**.

Step 1: Introduction to Pandas

Let’s read the CSV with Pandas:

```
df = pd.read_csv("dataset2.csv")
df.set_index("time",inplace=True)
time = df.index
df.head()
```

This generates a DataFrame (`df`). We are looking at the start (head) of it:



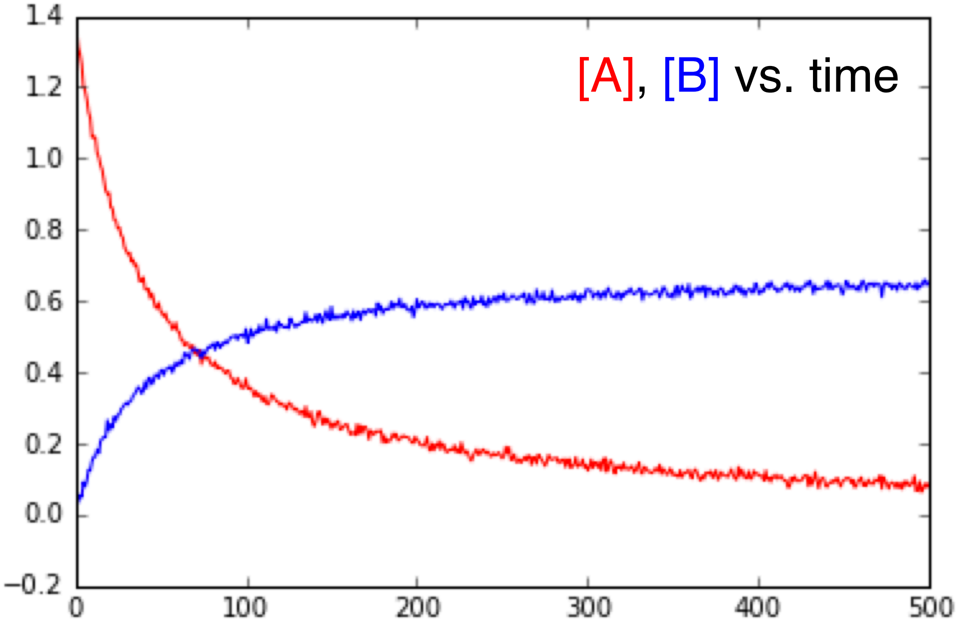
	A60	**B60**	**A80**	**B80**	**A100**	**B100**	**A120**	**B120**	**A140**	**B**
time										
0	0.601380	0.005124	0.780783	-0.001100	0.977195	-0.023776	1.172569	-0.027472	1.394396	-
1	0.572597	-0.007165	0.759028	0.002869	0.967283	0.001777	1.162058	0.001211	1.359970	0
2	0.574852	-0.015266	0.773124	0.015961	0.936298	0.022882	1.136366	0.019062	1.311560	0
3	0.574444	0.004979	0.739659	0.014633	0.937232	0.035337	1.099784	0.034862	1.293144	0
4	0.545312	0.011487	0.746717	0.022330	0.900225	0.019714	1.084564	0.052922	1.262887	0

For a discussion of indexing and the many features of Pandas:

http://byumcl.bitbucket.org/bootcamp2013/labs/pd_types.html

Step 2: A First Look

```
plt.plot(time, df.A140, "r")
plt.plot(time, df["B140"], "b")
plt.show()
```



There are two ways to access a column of a pandas DataFrame.

Method 1: We can type `df.column_name` (if there are no special characters).

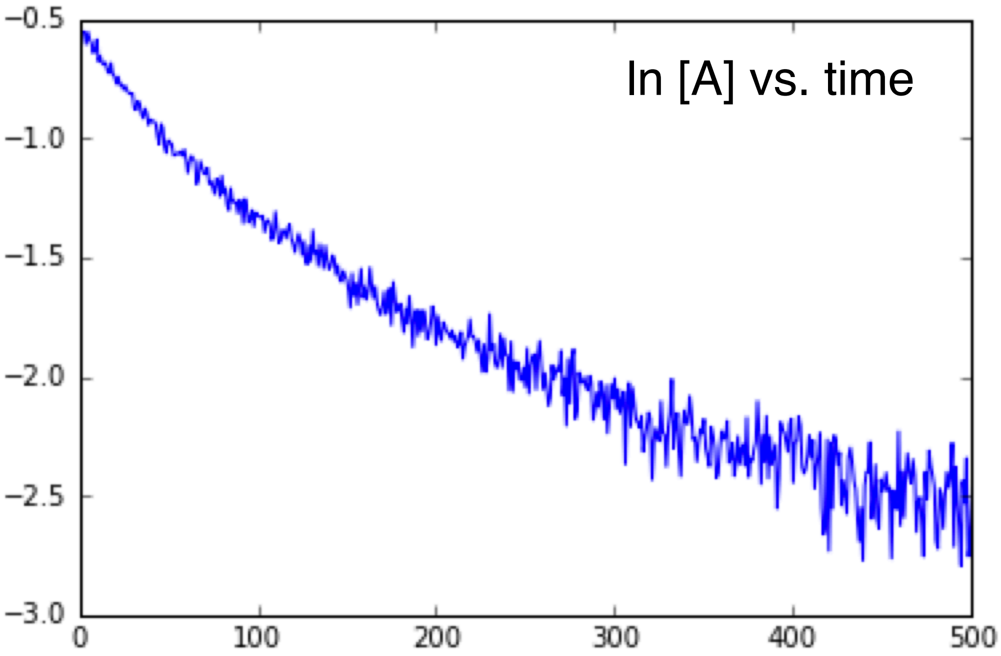
Method 2: We can also type `df[“column_name”]`. This form is handy because we can generate the necessary strings with a loop (coming up soon).

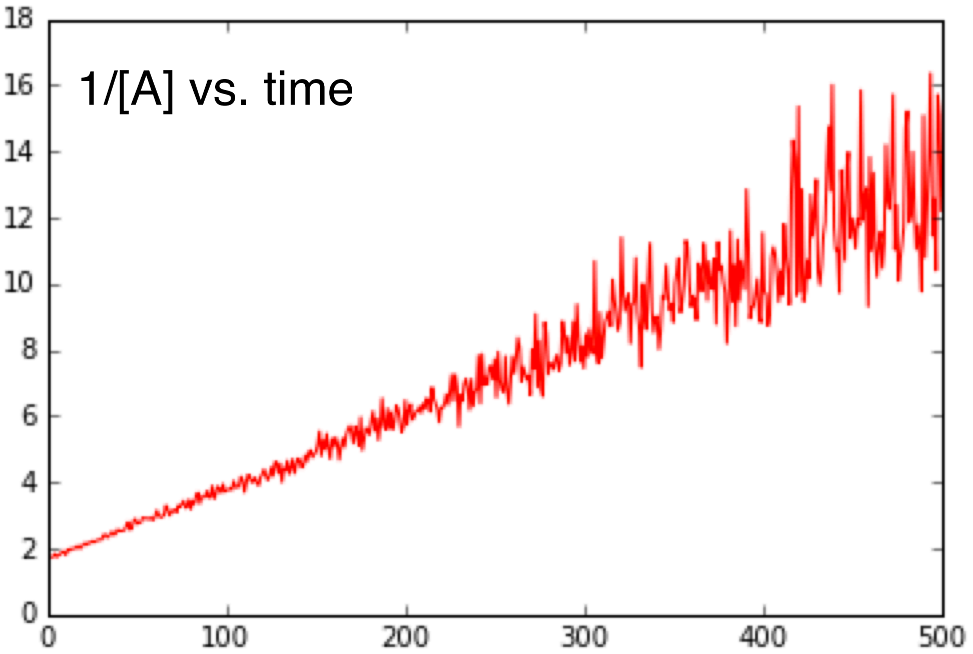
Regardless of the method used, one obtains a pandas.Series object. This is analogous to a numpy array.

Technical Note: Slicing syntax can be used on either DataFramers or Series, but the syntax is inclusive of the final index.

Step 2: A First Look

```
plt.plot(time, np.log(df.A60), "b")
plt.show()
plt.plot(time, 1.0/df.A60, "r”)
plt.show()
```





The curvature in the blue plot shows that this is not a first-order reaction.

The red plot is relatively straight, indicating a second-order reaction.

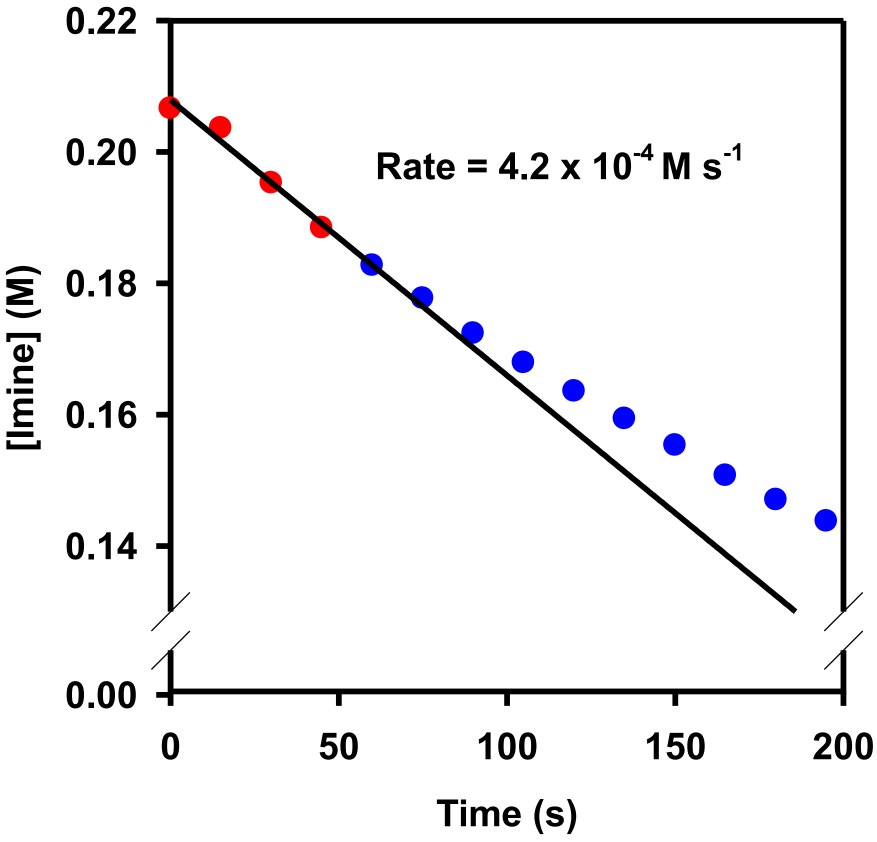
The righthand portion of the graph is noisy because the reciprocal exaggerates the loss of signal to noise in the later stages of the reaction.

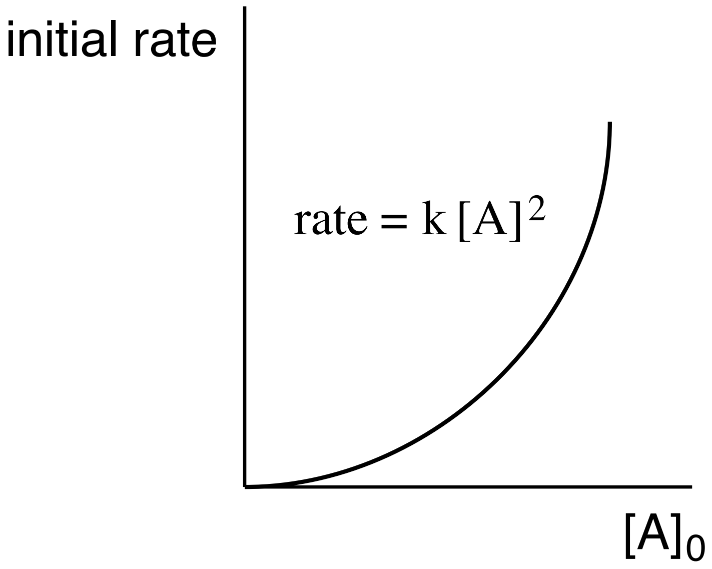
As a reminder, “A60” means the [A] for the experiment in which $[A]_0$ = 0.60 M.

You can perform vector operations like `1.0/df.A60`.

Step 3: Initial Rates Analysis

Recall that in an initial rates analysis, we assume the concentration vs. time plot can be treated as a straight line during the initial stages of the reaction:





By obtaining initial rates for various starting concentrations, one can derive the rate constant from the resulting parabolic plot of initial rate vs. $[A]_0$.

Step 3: Initial Rates Analysis

```
max_conversion = 0.20

initial_concentrations=[]
initial_rates=[]

def initial_rate(index):

    x = []
    y = []

    initial_concentration = index / 100.0
    initial_concentrations.append(initial_concentration)

    min_concentration = initial_concentration * (1.0 - max_conversion)



    for t, c in zip(time, df["A%d" % index]):
        if c < min_concentration:
            break
        x.append(t)
        y.append(c)

    x = np.array(x)
    y = np.array(y)
```

(function to be continued)

Step 3: Initial Rates Analysis

```
max_conversion = 0.20
```

Define the initial stages of the reaction to mean 20% conversion.

```
initial_concentrations=[]
initial_rates=[]
```

Initialize two arrays that will hold the initial concentrations and rates.

```
def initial_rate(index):
```

Make a function that will calculate the initial rates for a particular run. Index is the starting concentration of A in units of 0.01 M.

```
    x = []
    y = []
```

For each run, make two lists for time and concentration.

```
    initial_concentration = index / 100.0
    initial_concentrations.append(initial_concentration)
```

Convert the index to a concentration in M.

```
    min_concentration = initial_concentration * (1.0 - max_conversion)
```

Once the concentration of A drops below this number, we will have passed the initial stages of the reaction.

```
    for t, c in zip(time, df["A%d" % index]):
        if c < min_concentration:
            break
        x.append(t)
        y.append(c)
```

Gather the (time, concentration) points for the initial stages of the reaction.

“A%d” % index creates a string like A60: index replaces %d. This “method 2” for accessing a DataFrame.

zip(x,y) combines two lists, x and y, into a single list: [(x1, y1), (x2, y2), ...] so that they can be iterated in parallel.

break quits the loop.

```
    x = np.array(x)
    y = np.array(y)
```

Convert to numpy arrays to allow “broadcasting” math.

(function to be continued)

Step 3: Initial Rates Analysis

(function, continued)

```
    m, b, r, p, err = linregress(x,y)



    rate = -m/2.0
    initial_rates.append(rate)


    fitted_y = m*x + b


    print "%d, rate = %.4f, corr. coeff. = %.4f" % (index, rate, r)

    plt.plot(x, y, "ko")
    plt.plot(x, fitted_y, "b")
    plt.show()


for i in np.arange(60.0,220.0,20.0):
    initial_rate(i)

initial_concentrations = np.array(initial_concentrations)
initial_rates = np.array(initial_rates)

plt.plot(initial_concentrations, initial_rates, "ko")
```

Step 3: Initial Rates Analysis

(function, continued)

```
    m, b, r, p, err = linregress(x,y)
```

Perform linear regression to get the initial rate.
m = slope, b = intercept
r = correlation coefficient
p = p-value, err = standard error of estimate

```
    rate = -m/2.0
    initial_rates.append(rate)
```

The rate is half the slope because this is a second-order reaction. Store this initial rate.

```
    fitted_y = m*x + b
```

Compute the linear fit so we can graph it. Because x is a numpy array, we can use this kind of “broadcasting” math.

```
    print "%d, rate = %.4f, corr. coeff. = %.4f" % (index, rate, r)

    plt.plot(x, y, "ko")
    plt.plot(x, fitted_y, "b")
    plt.show()
```

Plot concentration vs. rate (and the fit) for each run.

```
for i in np.arange(60.0,220.0,20.0):
    initial_rate(i)
```

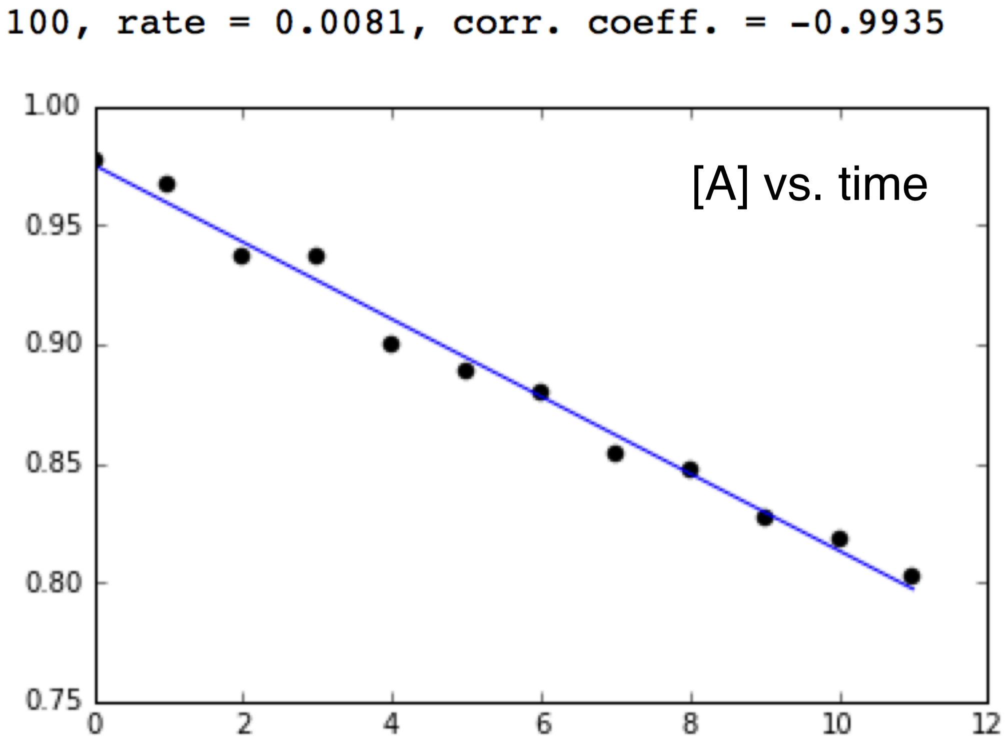
Run the function for each of our 8 datasets.

```
initial_concentrations = np.array(initial_concentrations)
initial_rates = np.array(initial_rates)

plt.plot(initial_concentrations, initial_rates, "ko")
```

Plot initial concentration vs. initial rate.

Step 3: Initial Rates Analysis

For each run, a plot of concentration vs. time is output, along with the regression parameters:



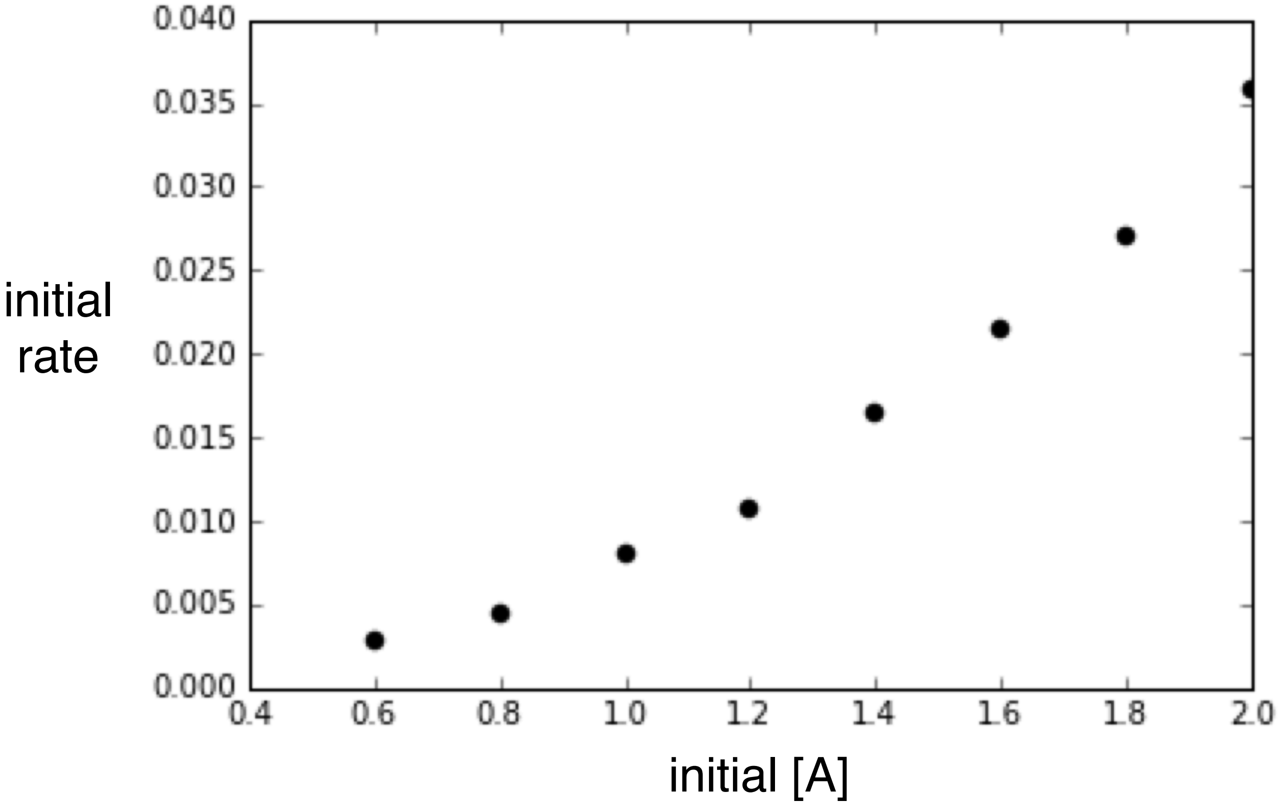
```
100, rate = 0.0081, corr. coeff. = -0.9935
```

The rate is half the slope because this is a second-order reaction.

The points follow a straight line because this is a small slice of a slowly curving function.

Step 3: Initial Rates Analysis

The final plot is of rate vs. intial concentration. It is parabolic:



Our next task will be to extract the rate constant, *k*, by fitting to rate = k $[A]_0^2$.

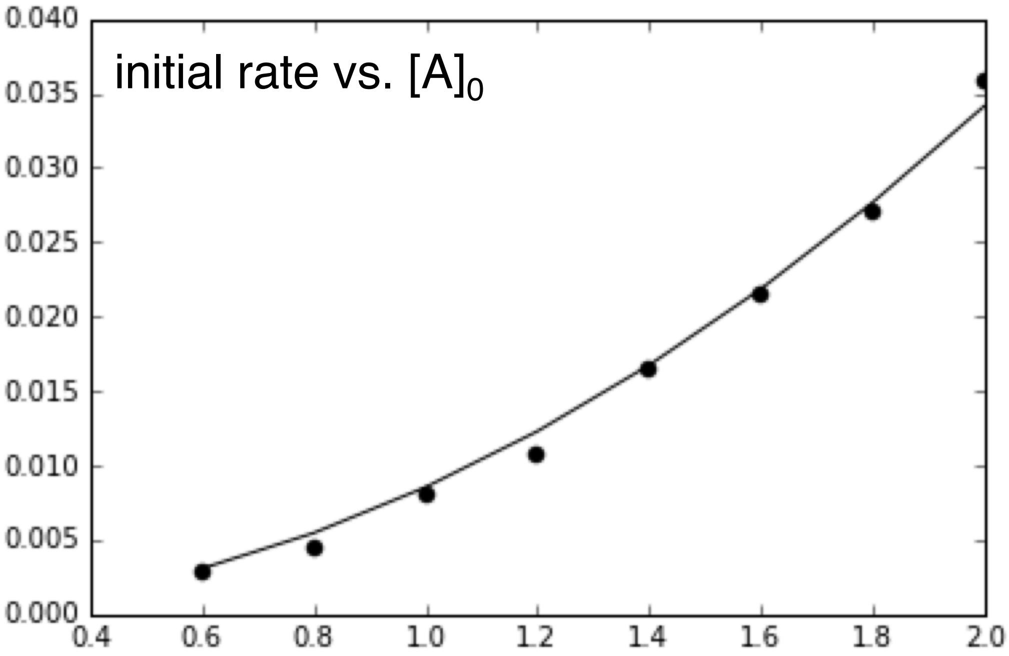
Step 4: Getting the Rate Constant

```
def second_order(A,k):
    return k*A*A
```

Note that the independent variable, A, must come first.

```
popt,pcov = curve_fit(second_order, initial_concentrations, initial_rates)
print "k = %.4f" % popt[0]
fitted_rate = second_order(initial_concentrations, popt[0])
plt.plot(initial_concentrations, initial_rates, "ko")
plt.plot(initial_concentrations, fitted_rate, "k")
plt.show()
```

```
k = 0.0085
```


initial rate vs. $[A]_0$

The fit is very good. The rate constant we obtain is in reasonable agreement with the actual value of 0.01.

(We can’t calculate the error in the estimates because we haven’t used error bounds in the data.)

Step 5: Rate vs. Concentration

The initial rates approach was very easy, but it throws away the data from 80% of the reaction. Can we get a more accurate answer?

The last ten slides were concerned with converting concentration vs. time to rate vs. initial concentration. Now, we'll try to get to rate vs. concentration for the entire reaction by differentiating:

```
polynomial_order = 7
measured_concentration = df.A60
poly_coeff = np.polyfit(time, measured_concentration, polynomial_order)
polynomial = np.poly1d(poly_coeff)
fitted_concentration = polynomial(time)
plt.plot(time, measured_concentration, "b.")
plt.plot(time, fitted_concentration, "r.")
plt.show()
residual = measured_concentration - fitted_concentration
plt.plot(time,residual)
plt.show()
RMSE = np.sqrt(np.mean(np.square(residual)))
print RMSE
```

Step 5: Rate vs. Concentration

The initial rates approach was very easy, but it throws away the data from 80% of the reaction. Can we get a more accurate answer?

The last ten slides were concerned with converting concentration vs. time to rate vs. initial concentration. Now, we'll try to get to rate vs. concentration for the entire reaction by differentiating:

```
polynomial_order = 3
measured_concentration = df.A60
poly_coeff = np.polyfit(time, measured_concentration, polynomial_order)
polynomial = np.poly1d(poly_coeff)
fitted_concentration = polynomial(time)
plt.plot(time, measured_concentration, "b.")
plt.plot(time, fitted_concentration, "r.")
plt.show()
residual = measured_concentration - fitted_concentration
plt.plot(time,residual)
plt.show()
RMSE = np.sqrt(np.mean(np.square(residual)))
print RMSE
```

"Fit the $[A]_0$ = 0.60 M data to a third order polynomial. Use the polynomial derivative to compute the rate as a function of time. Plot the result. Calculate the residual and root-mean-square-error of the fit."

Step 5: Rate vs. Concentration

```
polynomial_order = 3
```

Fit to a cubic polynomial. The choice of a polynomial function is arbitrary. Any smooth and differentiable interpolation would do.

```
measured_concentration = df.A60
```

Use the data from the $[A]_0$ = 0.60 M run. This is DataFrame “access method 1.”

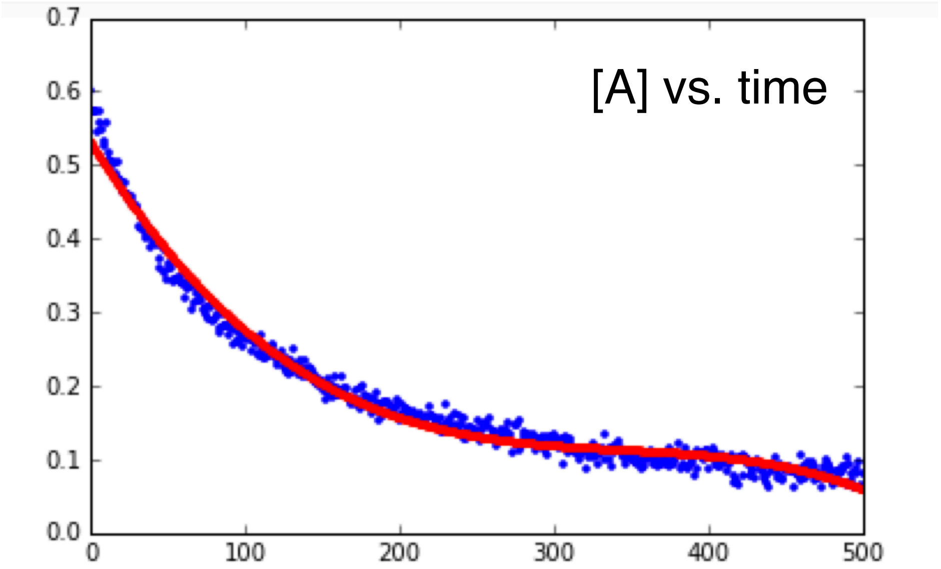
```
poly_coeff = np.polyfit(time, measured_concentration, polynomial_order)
polynomial = np.poly1d(poly_coeff)
fitted_concentration = polynomial(time)
```

Perform the polynomial fit. (The syntax is polyfit(x, y, polynomial_order)). Then, take the derivative of the polynomial and calculate the rate with broadcasting.

```
plt.plot(time, measured_concentration, "b.")
plt.plot(time, fitted_concentration, "r.")
plt.show()

residual = measured_concentration - fitted_concentration

plt.plot(time,residual)
plt.show()

RMSE = np.sqrt(np.mean(np.square(residual)))
print RMSE
```

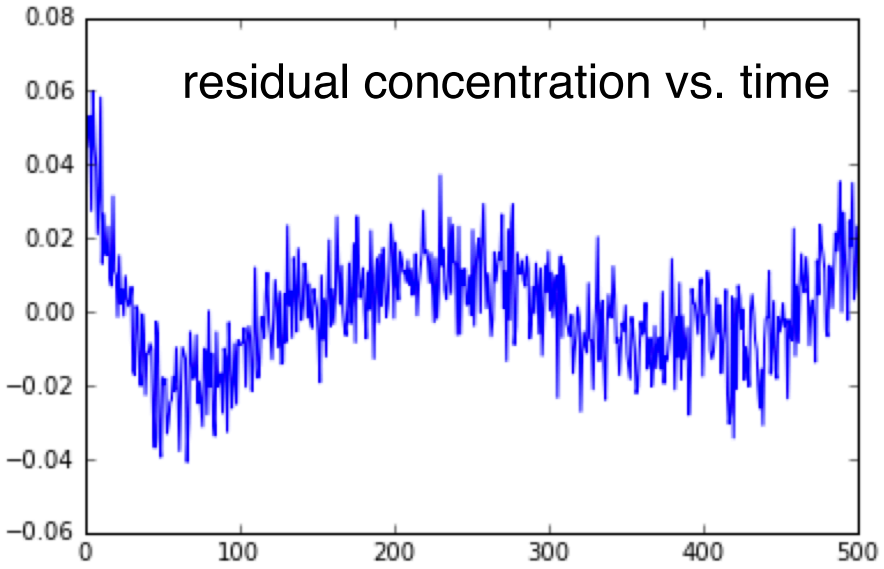
Calculate the root-mean-square error.

Step 5: Rate vs. Concentration



The fit is shown in red. Notice that it looks wavy, particularly near the edges of the dataset.

This occurs because the shape of the polynomial is different from the shape of the data. The “ringing” at the edge of datasets when fitting is a kind of Gibbs phenomenon.

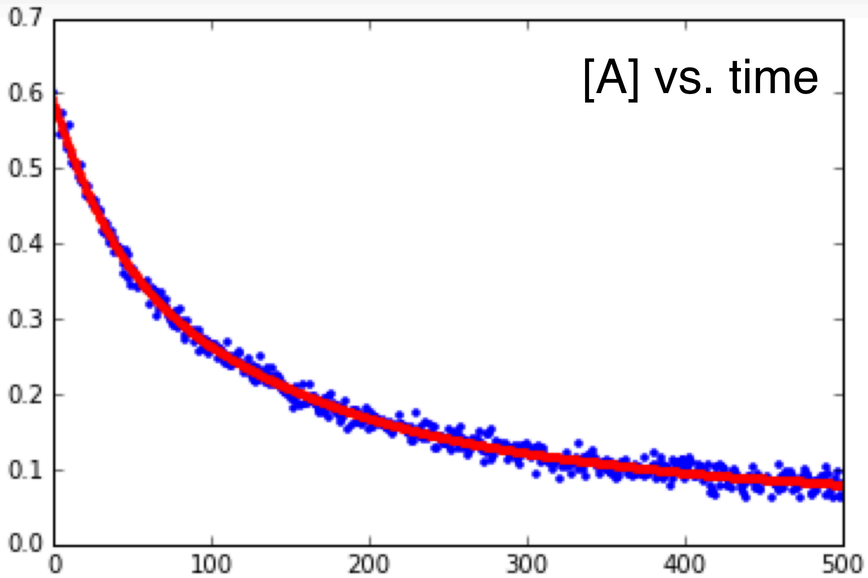


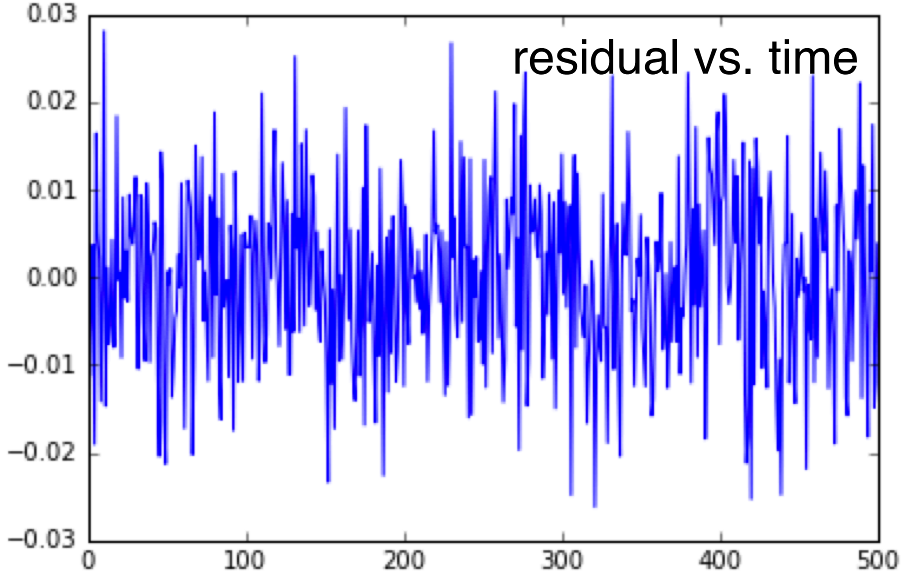
```
0.0155993625219
```

The residuals emphasize the poor fit at the edges of the dataset, despite the relatively good RMSE (printed in molar).

We can reduce the ringing by using a higher order polynomial. Set `polynomial_order` to 7 and try again.

Step 5: Rate vs. Concentration





```
0.0100648461229
```

This time, the fit is much better.

If you look carefully, the rate the beginning and end is still a bit anomalous.

How did we know to use a 7th order polynomial?

One rule is to use the smallest degree polynomial that fits the data to avoid overfitting. (One can apply various statistical tests as well.)

In this case, we can take advantage of the fact that we have multiple runs of data.

Step 5: Rate vs. Concentration

Let’s repeat the process for all the datasets at the same time. To avoid repeating ourselves, we’ll make a function:

```
polynomial_order = 7

def estimate_rate(index):
    concentration = df["A%d" % index]

    poly_coeff = np.polyfit(time, concentration, polynomial_order)
    polynomial = np.poly1d(poly_coeff)
    fitted_concentration = polynomial(time)
    derivative = np.polyder(polynomial)

    rate_vector = -0.5*derivative(time)

    df["rate%d" % index]=Series(rate_vector, index=time)

    plt.plot(concentration, rate_vector)




for i in range(60,220,20):
    estimate_rate(i)


plt.show()
```

Step 5: Rate vs. Concentration

Let’s repeat the process for all the datasets at the same time. To avoid repeating ourselves, we’ll make a function:

```
polynomial_order = 7
```

The same polynomial order will be used for every dataset.

```
def estimate_rate(index):
    concentration = df["A%d" % index]
```

Pull out the relevant dataset using method 2.

```
    poly_coeff = np.polyfit(time, concentration, polynomial_order)
    polynomial = np.poly1d(poly_coeff)
    fitted_concentration = polynomial(time)
    derivative = np.polyder(polynomial)
```

This is the same polynomial fitting as before.

```
    rate_vector = -0.5*derivative(time)
```

Remember, the rate is half of the derivative.

```
    df["rate%d" % index]=Series(rate_vector, index=time)

    plt.plot(concentration, rate_vector)
```

Store the result in the master DataFrame. You can see new columns like `rate60` have been added if you type `df.head()`.

```
for i in range(60,220,20):
    estimate_rate(i)
```

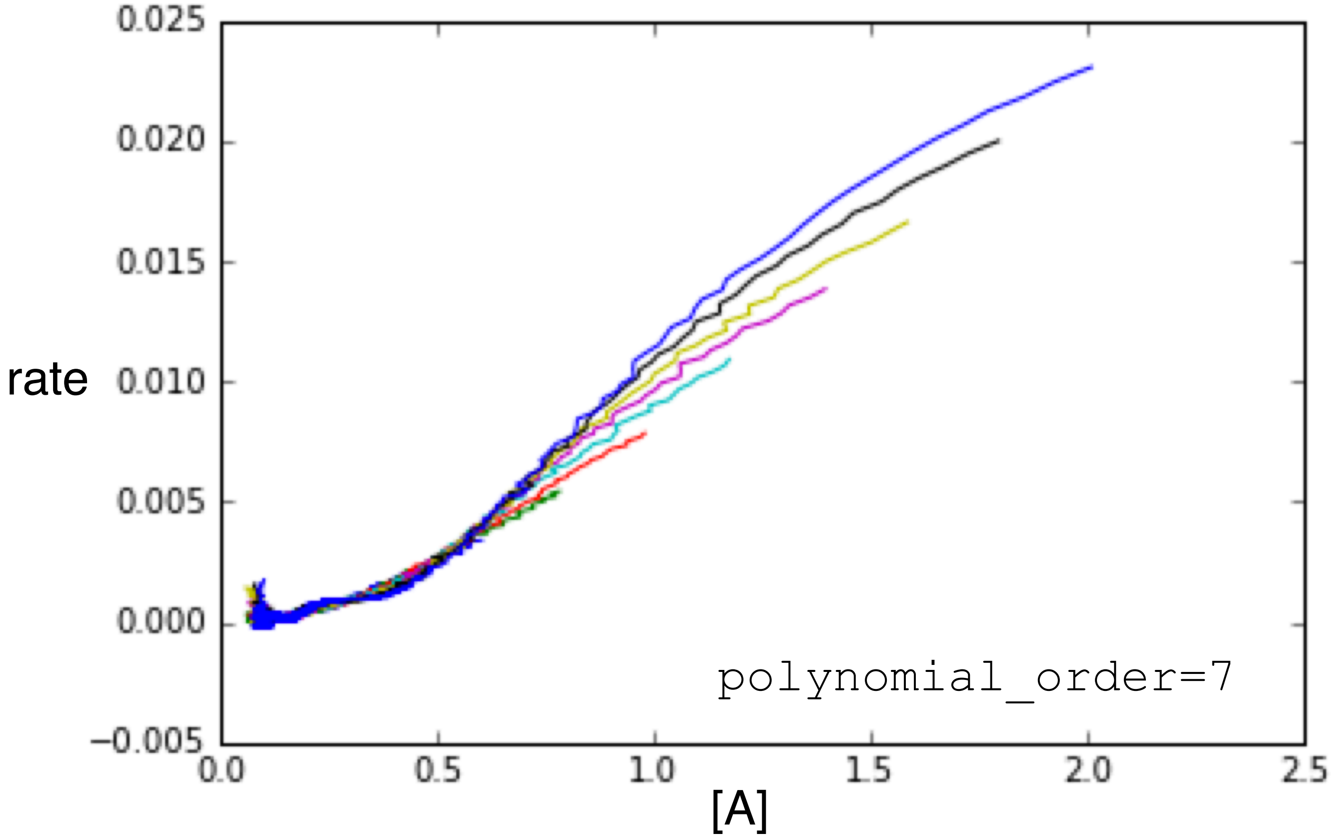
Run the function once per dataset. Remember, the range function does not include the endpoint of the interval in the returned list. That means we have to put the endpoint as 220, even though we only have data up to 200.

```
plt.show()
```

This will overlay all the rate vs. concentration plots on one graph.

Step 5: Rate vs. Concentration

Each fit is given a different color. Theoretically, they should all overlay:

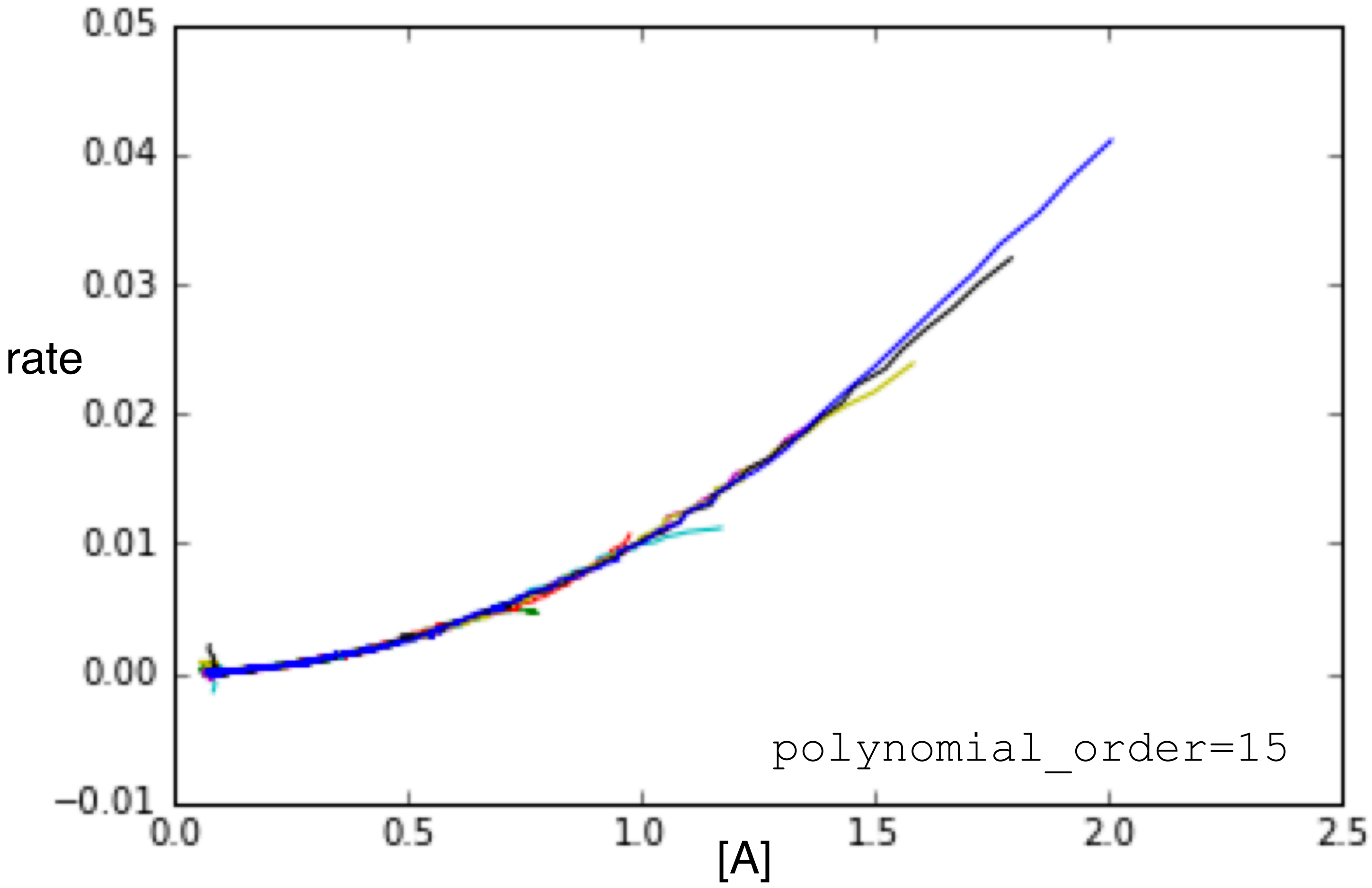


The fact that the curves don't overlay (and aren't parabolic) means the rate is being estimated poorly. Notice how the overlay is good at moderate concentrations. That's because the middle of each dataset is where the rate is estimated best.

Try again with `polynomial_order=15`.

Step 5: Rate vs. Concentration

Despite some obvious edge artifacts, this overlays well and is correctly parabolic:



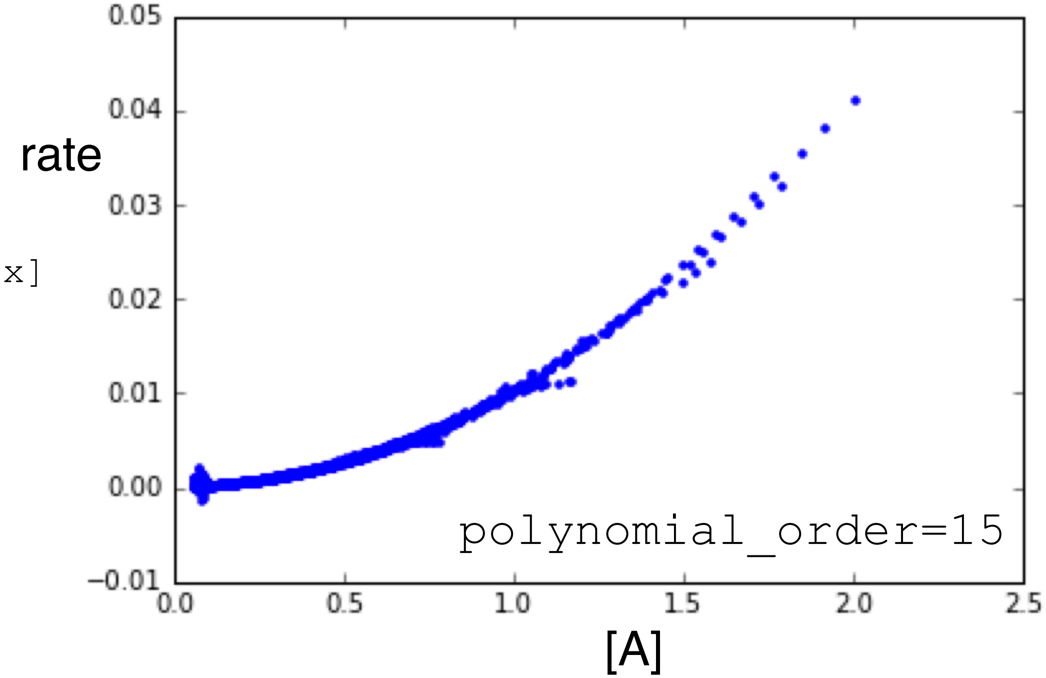
.

Remember, graphs like these should be read from right (low conversion) to left (high conversion).

Step 6: Getting the Rate Constant in Aggregate

Let’s combine all the runs into one and obtain the rate constant from the aggregate dataset.

```
x = []
y = []

for index in range(60,220,20):
    this_A = df["A%d" % index]
    this_rate = df["rate%d" % index]
    x.extend(this_A)
    y.extend(this_rate)

x = np.array(x)
y = np.array(y)
plt.plot(x,y,"b.")
```



This uses method 2 for accessing the data again.

`extend` adds one list to the end of another.

If we wanted to be more sophisticated, we could remove the outliers, but this is good enough for now. (You would perform the fitting in the next step twice, once to identify the outliers, and once to fit without the outliers.)

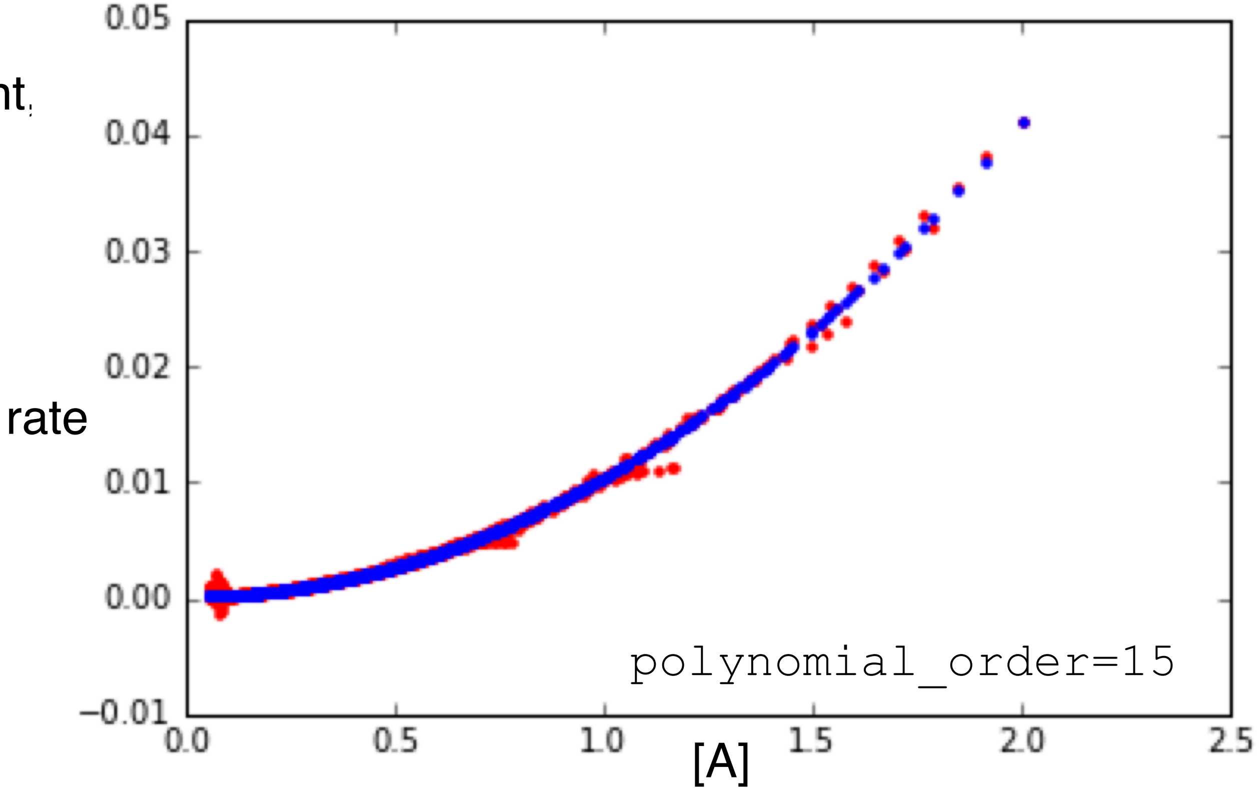
Step 6: Getting the Rate Constant in Aggregate

This is the fitting code:

```
popt,pcov = curve_fit(second_order, x, y)
print "k = %.4f" % popt[0]
fitted = second_order(x, popt[0])
plt.plot(x, y, "r.")
plt.plot(x, fitted, "b.")
plt.show()
```

```
k = 0.0102
```

By using the all of the data from each experiment, and by combining all the experiments, we get a result that agrees much more closely with the true rate constant of 0.01.



Summary

Congratulations! You now know how to perform initial rates and reaction progress analyses.

Import Statements

```
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
import pandas as pd
from pandas import Series, DataFrame
from scipy.optimize import curve_fit
from scipy.stats import linregress
```

pandas and DataFrames

```
df = pd.read_csv("dataset2.csv")
df.set_index("time",inplace=True)
time = df.index
df.head()

# access method 1 (no special chars)
df.A60

# access method 2
df[“A60”]

# access method 2 with %
key_name = “A%d” % index
df[key_name]

# iterating two lists in parallel
for t, c in zip(time, concentration):
     ...your code here...
```

Linear Regression

```
# slope, intercept, correlation coefficient
# p-value, standard error of estimate
m, b, r, p, err = linregress(x,y)
```

Polynomial Fitting

```
# returns the polynomial coefficients as a list
poly_coeff = np.polyfit(x, y, polynomial_order)

# make a numpy polynomial object
polynomial = np.poly1d(poly_coeff)

# broadcast the polynomial on x
# like a list comprehension, but more concise
fitted_y = polynomial(x)

# returns a numpy polynomial object that is
# the derivative of the original polynomial
derivative = np.polyder(polynomial)

# broadcast the derivative on x
y_prime = derivative(x)

# calculate the residual
residual = y - fitted_y

# calculate the root-mean-square error
RMSE = np.sqrt(np.mean(np.square(residual)))
print RMSE
```