15 – Lease Squares Fitting of Linear Models

Physics 281 – Class 15

Grant Wilson

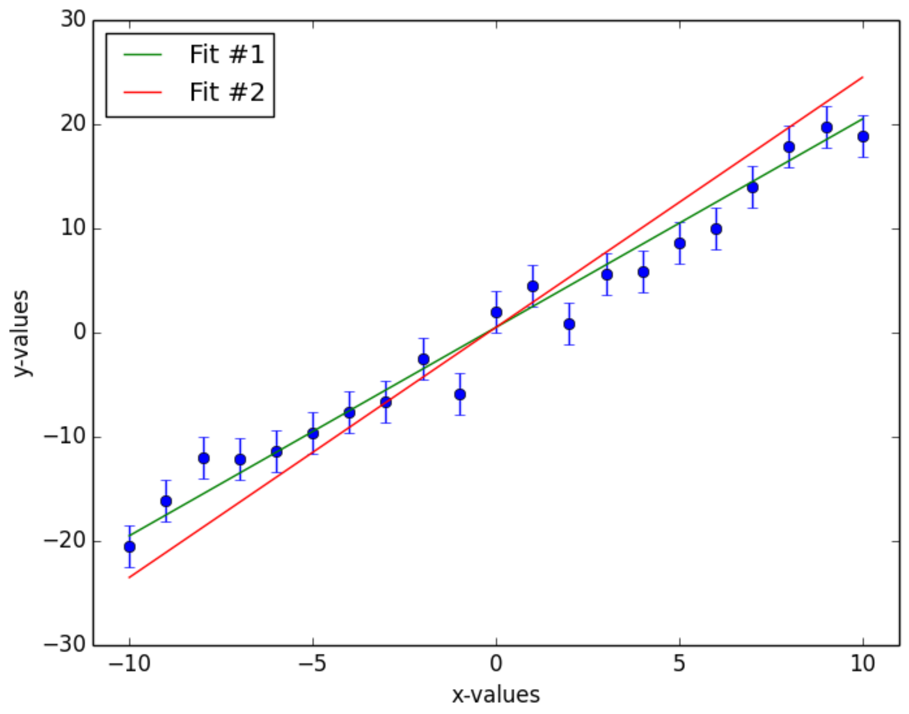
15 – Lease Squares Fitting of Linear Models

Physics 281 – Class 15

Grant Wilson

What does it mean to find the "best fit?"

- Clearly we need some kind of statistic to tell us what to do.



Recipe for Fitting Approach #1

1. write down expression for χ^2

2. find equations for minimizing χ^2 for each of our "free parameters"

3. solve the system of equations from step #2

The χ^2 Statistic

- Remember

$$\chi^2 = \sum_{i=1}^{N} \frac{(y_i - f(x_i; \vec{a}))^2}{\sigma_i^2}$$

- This is an error-weighted sum of the squares of the distances (in y) between the model and the data.

- Least Squares Fitting is the art of finding the model that **minimizes** this quantity.

Fitting Data to a Line

- Start off with something relatively easy and manageable.
- Fit a set of N-data points, (x,y) with error σ, to a line of the form

$$y = a + bx$$

- Recipe for Approach #1
  1. write down expression for χ^2
  2. find equations for minimizing χ^2 for each of our “free parameters”
  3. solve the system of equations from step #2
  4. Find the errors in the parameters and their covariance.

Exercise

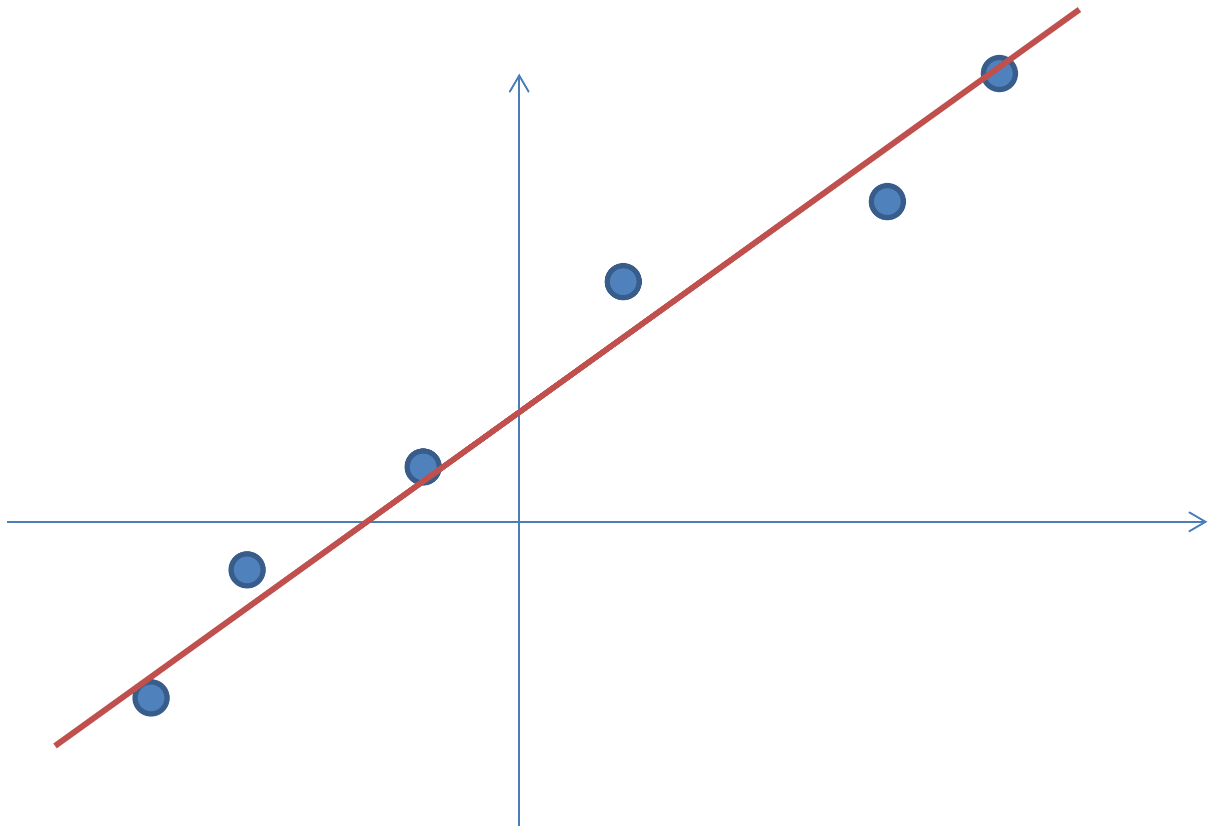
- This is a very important simulation that will give you some insight about covariance.

1. Generate fake data set #1 with 20 points:
    1. slope = 4.
    2. y-intercept = -10
    3. errors = 1.
    4. x1 = np.linspace(-20.,20.,npts)

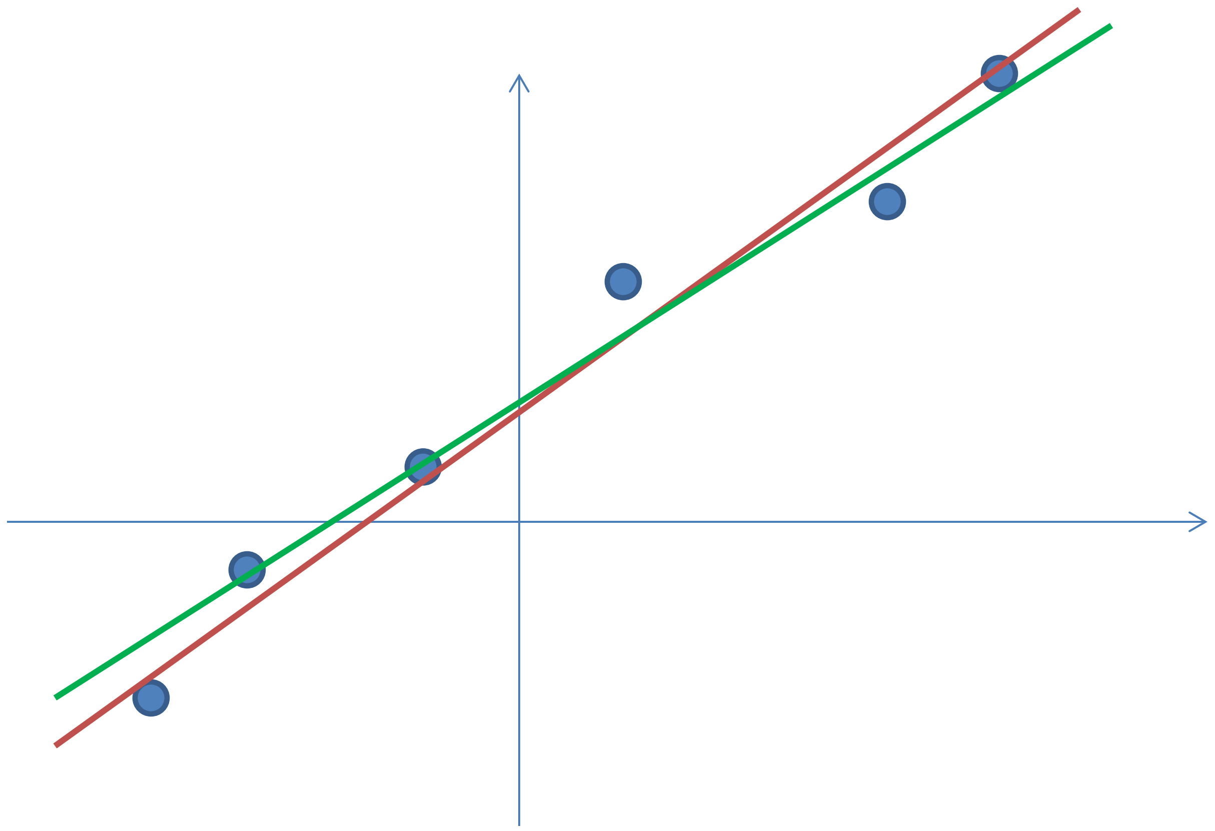
2. Generate fake data set #2 with 20 points:
    1. slope = 3.
    2. y-intercept = -20
    3. errors = 1
    4. x2 = np.linspace(30.,60.,npts)

3. Fit both synthetic data sets to a line and find the best-fit slope and y-intercept for your data. Also calculate the errors in both the slope and the y-intercept. Calculate the covariance of the two parameters as well.

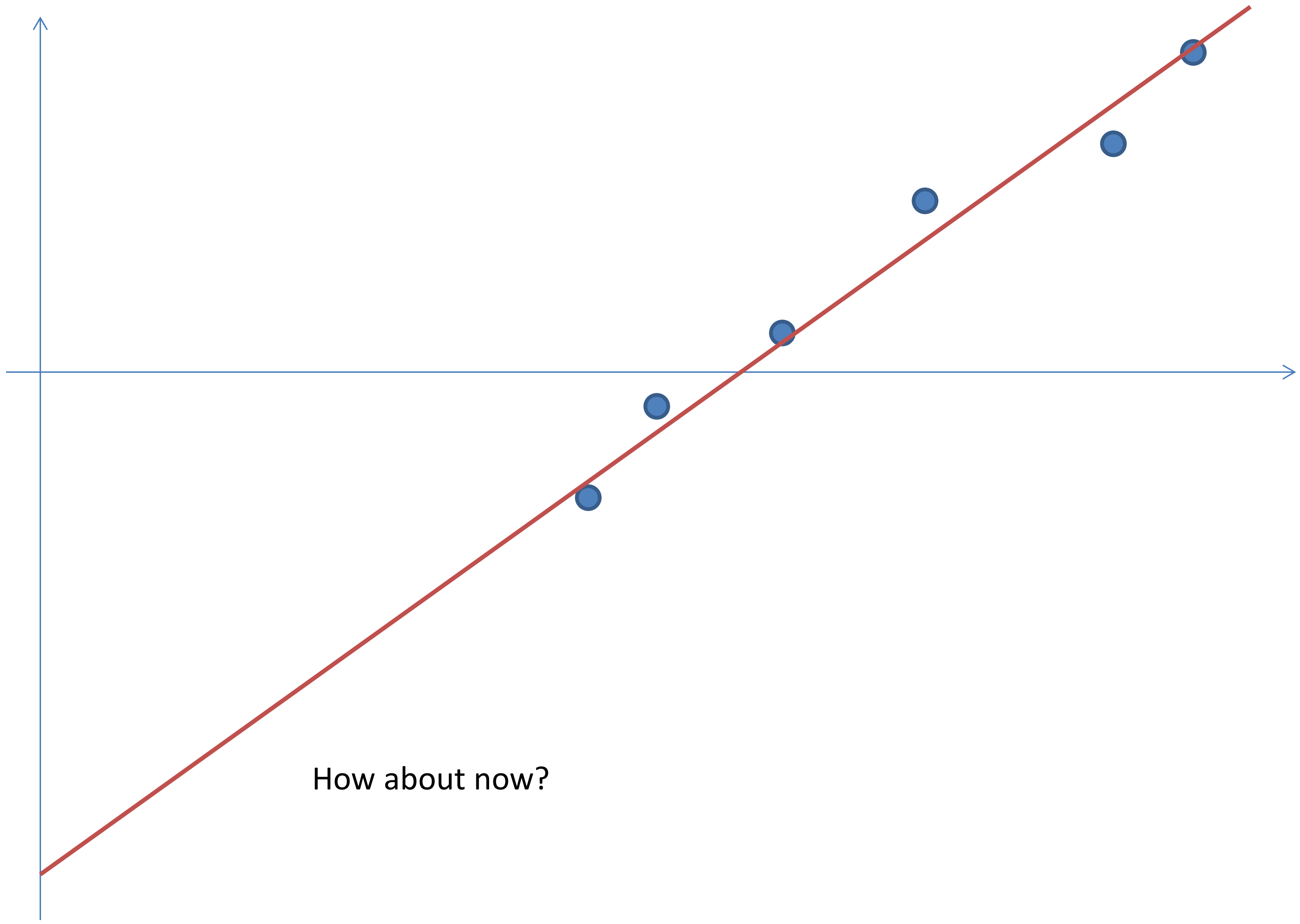
4. Now repeat steps 1-3 1000 times. Plot the resulting best-fit y-intercepts versus the best-fit slopes. What do you see in the two cases? How does this compare to your parameter errors and the covariance?



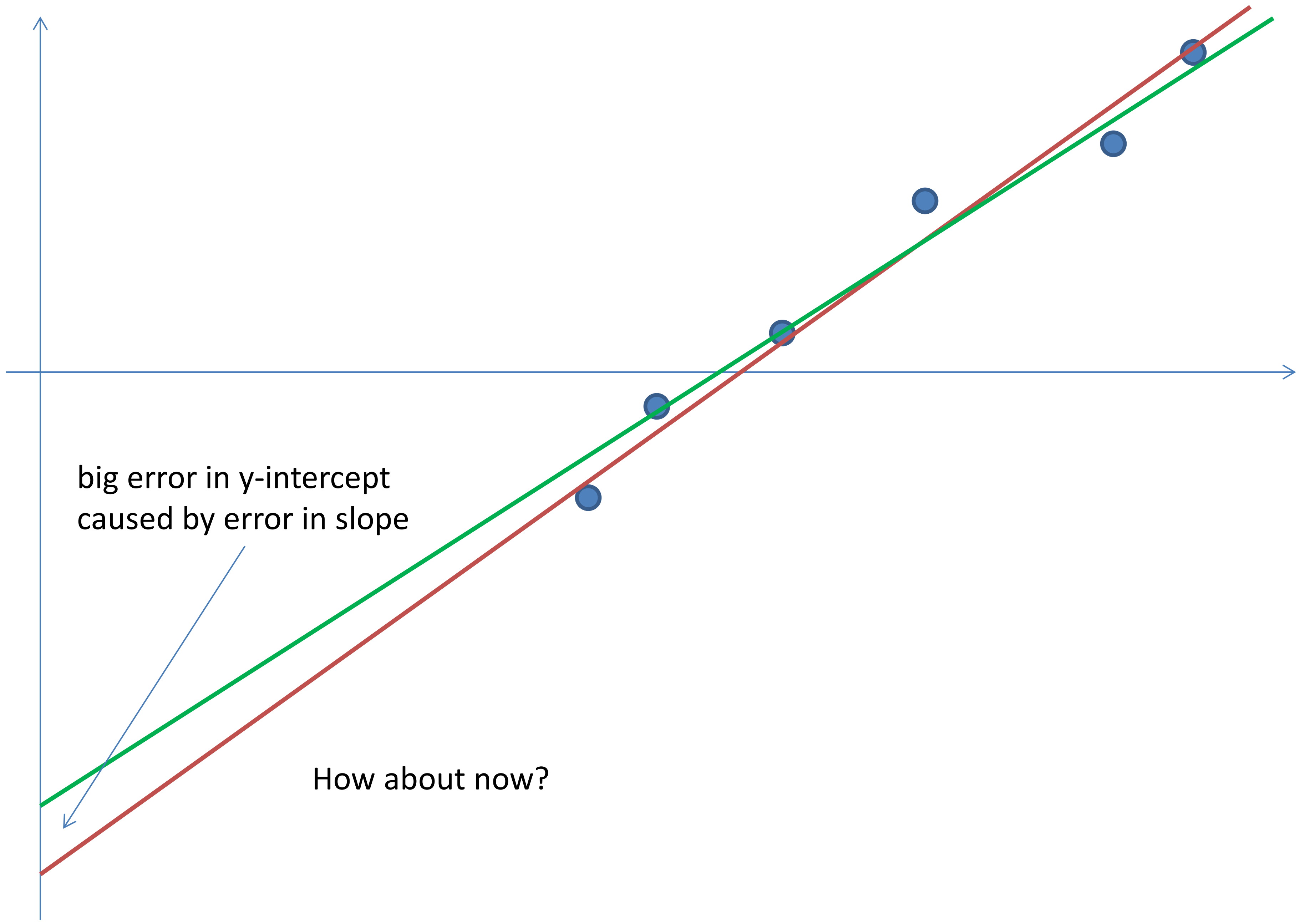
Imagine an error in the slope of this line. How would that affect the y-intercept of the line?



Imagine an error in the slope of this line. How would that affect the y-intercept of the line?



How about now?



How about now?

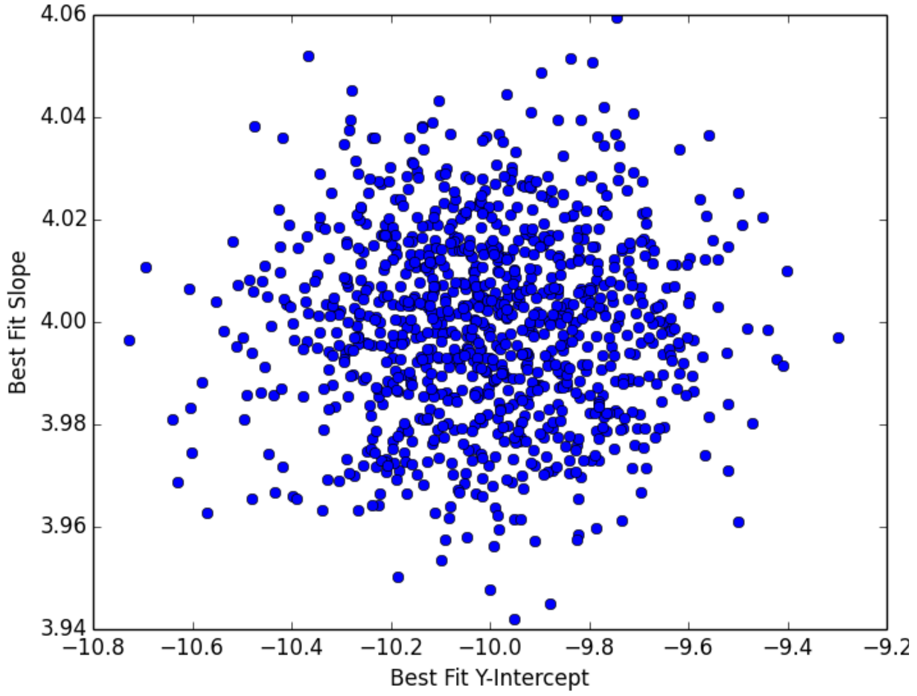
Exercise Answer

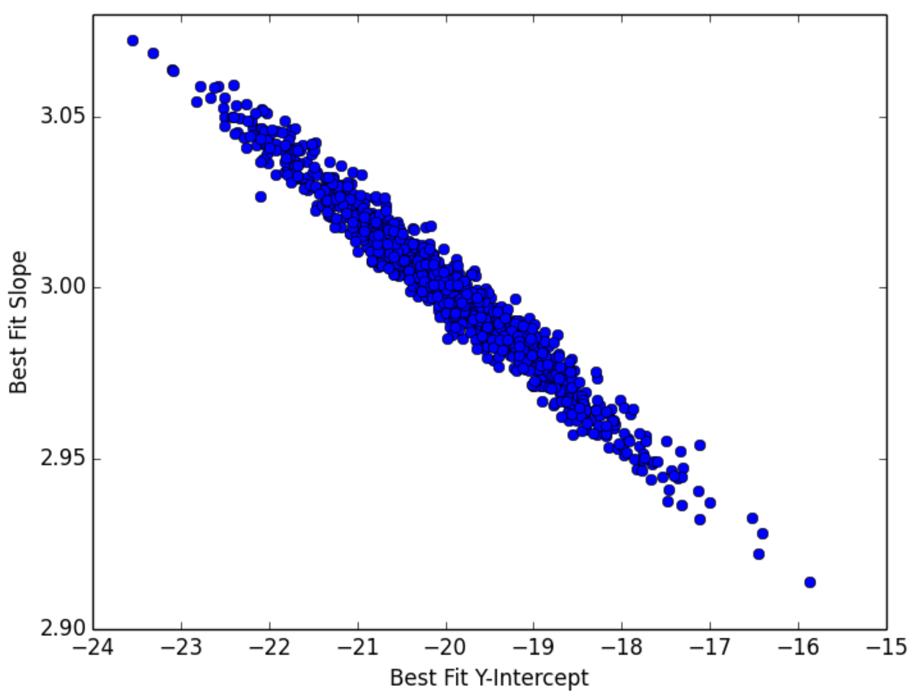
```
npts = 20.
m1 = 4., b1 = -10., sig1 = 1.
x1 = np.linspace(-20.,20.,npts)
m2 = 3., b2 = -20.,  sig2 = 1.
x2 = np.linspace(30.,60.,npts)

nTrials = 1000
best_slope1 = np.zeros(nTrials)
best_slope2 = np.zeros(nTrials)
best_int1 = np.zeros(nTrials)
best_int2 = np.zeros(nTrials)

for i in range(nTrials):
    y1 = b1 + m1*x1 + np.random.randn(npts)*sig1
    y2 = b2 + m2*x2 + np.random.randn(npts)*sig2

    #calculate the best-fit parameters here
     …….
```


Best Fit Slope
Best Fit Y-Intercept
4.06
4.04
4.02
4.00
3.98
3.96
3.94
−10.8
−10.6
−10.4
−10.2
−10.0
−9.8
−9.6
−9.4
−9.2


Best Fit Slope
Best Fit Y-Intercept
3.05
3.00
2.95
2.90
−24
−23
−22
−21
−20
−19
−18
−17
−16
−15

A More General Approach

- We have N-data points (20 in the last exercise) and we are looking for the best-fit line.

- Consider:

$$y_1 = a + bx_1$$
$$y_2 = a + bx_2$$

Could you solve these for a and b?

A More General Approach

- We have N-data points (20 in the last exercise) and we are looking for the best-fit line.

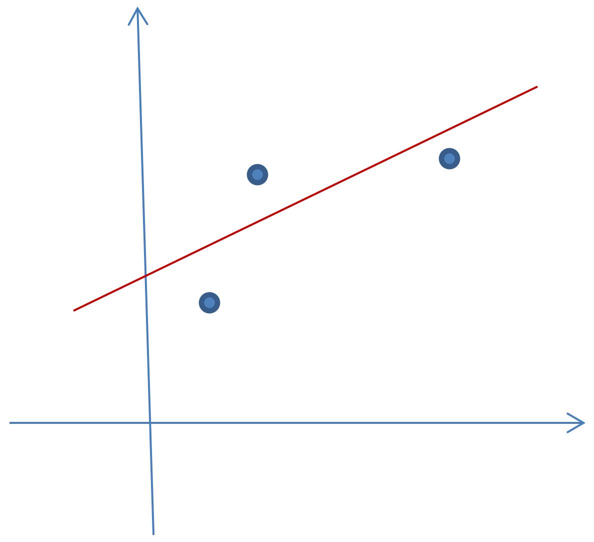
- Consider:

$$y_1 = a + bx_1$$
$$y_2 = a + bx_2$$

- Could you solve these for a and b? How about:

$$y_1 = a + bx_1$$
$$y_2 = a + bx_2$$
$$y_3 = a + bx_3$$

- We already know the problem here. There is no line guaranteed to pass through a general set of 3 points. Instead we search for the “least squares” solution.



Let’s generalize

(temporarily ignoring σ):

$$\begin{aligned} y_1 &= a + bx_1 \\ y_2 &= a + bx_2 \\ y_3 &= a + bx_3 \\ &\vdots \\ y_N &= a + bx_N \end{aligned} \longrightarrow \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_N \end{pmatrix} = \begin{pmatrix} 1 & x_1 \\ 1 & x_2 \\ 1 & x_3 \\ \vdots & \vdots \\ 1 & x_N \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix}$$

or even more simply:

$$\vec{y} = M\vec{p}$$

A little linear algebra

1. talk about the shape of a matrix
2. show how to multiply two matrices by hand
3. show the identify matrix
4. talk about the inverse of a matrix
5. talk about the transpose of a matrix

Let’s generalize:

$$\begin{aligned} y_1 &= a + bx_1 \\ y_2 &= a + bx_2 \\ y_3 &= a + bx_3 \\ &\vdots \\ y_N &= a + bx_N \end{aligned} \longrightarrow \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_N \end{pmatrix} = \begin{pmatrix} 1 & x_1 \\ 1 & x_2 \\ 1 & x_3 \\ \vdots & \vdots \\ 1 & x_N \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix}$$

or even more simply:

$$\vec{y} = M\vec{p}$$

With Linear Algebra

- Our system of N-equations is:

$$\vec{y} = M\vec{p}$$

- Multiply both sides by transpose of M:

$$M^T\vec{y} = M^T M\,\vec{p}$$

- The matrix $M^T M$ is square so we can possibly take its inverse, $(M^T M)^{-1}$. Multiply both sides of eq (2) above by the inverse to get $\vec{p}$

$$\vec{p} = (M^T M)^{-1} M^T \vec{y}$$

- The Gauss-Markov Theorem states that $\vec{p}$ is the least squares estimate of a and b.

- The covariance matrix is simply $(M^T M)^{-1}$

Exercise

- Use your code from last class to find the best-fit line for these data points:
  - x = np.array([0.,1.,4.])
  - y = np.array([0.05,0.88,4.13])

- Now try the same thing using the linear algebra approach
  - Construct M by hand.
  - The transpose of M, M^T, is M.transpose() in Python
  - Build alpha = $(M^T M)$
  - Find the inverse of alpha with: np.linalg.inv(alpha)

Fitting a Quadratic

$$\begin{aligned} y_1 &= a + bx_1 + cx_1^2 \\ y_2 &= a + bx_2 + cx_2^2 \\ y_3 &= a + bx_3 + cx_3^2 \\ &\vdots \\ y_N &= a + bx_N + cx_N^2 \end{aligned} \longrightarrow \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_N \end{pmatrix} = \begin{pmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \\ \vdots & \vdots & \\ 1 & x_N & x_N^2 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \end{pmatrix}$$

again:

$$\vec{y} = M\vec{p}$$

All the math is identical to before

- Our system of N-equations is:

$$\vec{y} = M\vec{p}$$

- Multiply both sides by transpose of M:

$$M^T\vec{y} = M^T M\,\vec{p}$$

- The matrix $M^T M$ is square so we can possibly take its inverse, $(M^T M)^{-1}$. Multiply both sides of eq (2) above by the inverse to get $\vec{p}$

$$\vec{p} = (M^T M)^{-1} M^T \vec{y}$$

- The Gauss-Markov Theorem states that $\vec{p}$ is the least squares estimate of a and b.

- The covariance matrix is simply $(M^T M)^{-1}$

Add in Measurement Errors

- Let our measurement errors be σ_i
- Our Matrix M becomes:

And our vector $\vec{y}$ transforms to:

$$M \rightarrow \begin{pmatrix} \frac{1}{\sigma_1} & \frac{x_1}{\sigma_1} & \frac{x_1^2}{\sigma_1} \\ \frac{1}{\sigma_2} & \frac{x_2}{\sigma_2} & \frac{x_2^2}{\sigma_2} \\ \frac{1}{\sigma_3} & \frac{x_3}{\sigma_3} & \frac{x_3^2}{\sigma_3} \\ & \vdots & \vdots \\ \frac{1}{\sigma_N} & \frac{x_N}{\sigma_N} & \frac{x_N^2}{\sigma_N} \end{pmatrix}$$

$$\vec{y} \rightarrow \begin{pmatrix} y_1/\sigma_1 \\ y_2/\sigma_2 \\ y_3/\sigma_3 \\ \vdots \\ y_N/\sigma_N \end{pmatrix}$$

and again:

$$\vec{y} = M\vec{p}$$

And all the math is identical to before

- Our system of N-equations is:
$$\vec{y} = M\vec{p}$$

- Multiply both sides by transpose of M:
$$M^T\vec{y} = M^T M\,\vec{p}$$

- The matrix $M^T M$ is square so we can possibly take its inverse, $(M^T M)^{-1}$. Multiply both sides of eq (2) above by the inverse to get $\vec{p}$
$$\vec{p} = (M^T M)^{-1} M^T \vec{y}$$

- The Gauss-Markov Theorem states that $\vec{p}$ is the least squares estimate of a and b.

- The covariance matrix is simply $(M^T M)^{-1}$

What Can Go Wrong?

- The biggest opportunity for catastrophic failure of this technique comes in the inversion of $(M^T M)$.

- Matrices that are “singular” (have determinate=0) can not be inverted.

- Worse, matrices that are “ill-conditioned” can be inverted but will give poor results for the inversion due to numerical errors.

- Recommendation: For problems that are not too big use an inversion technique based on “singular value decomposition”. SVD-based techniques are guaranteed to return the least-squares solution for the vector of free parameters.

- Here we will invoke the technique we’ve looked at thus far which should work well for most cases you run into.

General Fitting Recipe

1. Collect your *N*-data points, $\vec{x}$ and $\vec{y}$, in numpy arrays
2. Determine your data errors σ_i for each of the y_i
3. Write down your fitting model (here I assume *K* free parameters) in the form $f(\vec{x};\vec{p}) = p_0 g_0(\vec{x}) + p_1 g_1(\vec{x}) + p_2 g_2(\vec{x}) \cdots + p_{K-1} g_{K-1}(\vec{x})$
4. Construct the design matrix *M*
5. Construct the matrix $\alpha = M^T M$
6. Construct the inverse of alpha: $\alpha^{-1} = (M^T M)^{-1}$
7. Construct the vector $\vec{y}'$ with elements y_i/σ_i
8. The best fit parameters, p, are then given by
$$\vec{p} = (M^T M)^{-1} M^T \vec{y}'$$
9. The best fit parameters are then given by
$$p_i(best) = p_i \pm \alpha^{-1}[i,i]$$
10. Make a plot of the residuals. That is, plot $\vec{x}$ vs. $f\big(\vec{x};\vec{p}(best)\big)$.

Fitting Example (all together)

1. Collect your *N*-data points, $\vec{x}$ and $\vec{y}$, in numpy arrays
2. Determine your data errors σ_i for each of the y_i

- Download the file ‘fitting_data.npz’ from the Moodle site and extract the vectors, x, y, and sigma.
  - remember:
    - data = np.load(<filename>)
    - x = data[‘x’]

Fitting Example (all together)

3. Write down your fitting model (here I assume *K* free parameters) in the form

$$f(\vec{x};\vec{p}) = p_0 g_0(\vec{x}) + p_1 g_1(\vec{x}) + p_2 g_2(\vec{x}) \cdots + p_{K-1} g_{K-1}(\vec{x})$$

For this example, the fitting function is

$$f(\vec{x};\vec{p}) = p_0 \cos(x) + p_1 x^3$$

4. Construct the design matrix, *M*:

$$M = ?$$

Fitting Example (all together)

3. Write down your fitting model (here I assume *K* free parameters) in the form

$$f(\vec{x};\vec{p}) = p_0 g_0(\vec{x}) + p_1 g_1(\vec{x}) + p_2 g_2(\vec{x}) \cdots + p_{K-1} g_{K-1}(\vec{x})$$

For this example, the fitting function is

$$f(\vec{x};\vec{p}) = p_0 \cos(x) + p_1 x^3$$

4. Construct the design matrix, *M*:

$$M = \begin{pmatrix} \cos(x_0)/\sigma_0 & x_0^3/\sigma_0 \\ \cos(x_1)/\sigma_1 & x_1^3/\sigma_1 \\ \vdots & \\ \cos(x_N)/\sigma_N & x_N^3/\sigma_N \end{pmatrix}$$

Fitting Example (all together)

3. Write down your fitting model (here I assume *K* free parameters) in the form

$$f(\vec{x};\vec{p}) = p_0 g_0(\vec{x}) + p_1 g_1(\vec{x}) + p_2 g_2(\vec{x}) \cdots + p_{K-1} g_{K-1}(\vec{x})$$

For this example, the fitting function is

$$f(\vec{x};\vec{p}) = p_0 \cos(x) + p_1 x^3$$

4. Construct the design matrix, *M*:

```
>>> M = np.zeros((npts,2))
>>> for i in range(npts):
>>>     M[i,0] = np.cos(x[i])/sigma[i]
>>>     M[i,1] = x[i]**2/sigma[i]
```

$\leftarrow$ make an empty matrix

$\leftarrow$ fill in the values

Fitting Example (all together)

5. Construct the matrix $\alpha = M^T M$

This is a matrix multiplication that you can use the numpy.dot() method to do:

```
>>> Mt = M.transpose()
>>> alpha = np.dot(Mt,M)
```

Check the size of alpha, it should be KxK where K is the number of free parameters.

```
>>> print alpha.shape
```

Fitting Example (all together)

6. Construct the inverse of alpha: $\alpha^{-1} = (M^T M)^{-1}$

We will use scipy.linalg.inv() (so do this: from scipy import linalg.inv)

```
>>>  alpha_inverse = linalg.inv(alpha)
```

Double check that the inverse worked out:

```
>>> iden = np.dot(alpha_inverse,alpha)
>>> print iden
```

Fitting Example (all together)

7. Construct the vector $\vec{y}'$ with elements $\frac{y_i}{\sigma_i}$

```
>>> yprime = y/sigma
```

8. The best fit parameters, p, are then given by

$$\vec{p} = \alpha^{-1} M^T \vec{y}'$$

```
>>> b = np.dot(Mt,yprime)
>>> p = np.dot(alpha_inverse,b)
```

Fitting Example (all together)

9. The best fit parameters are then given by

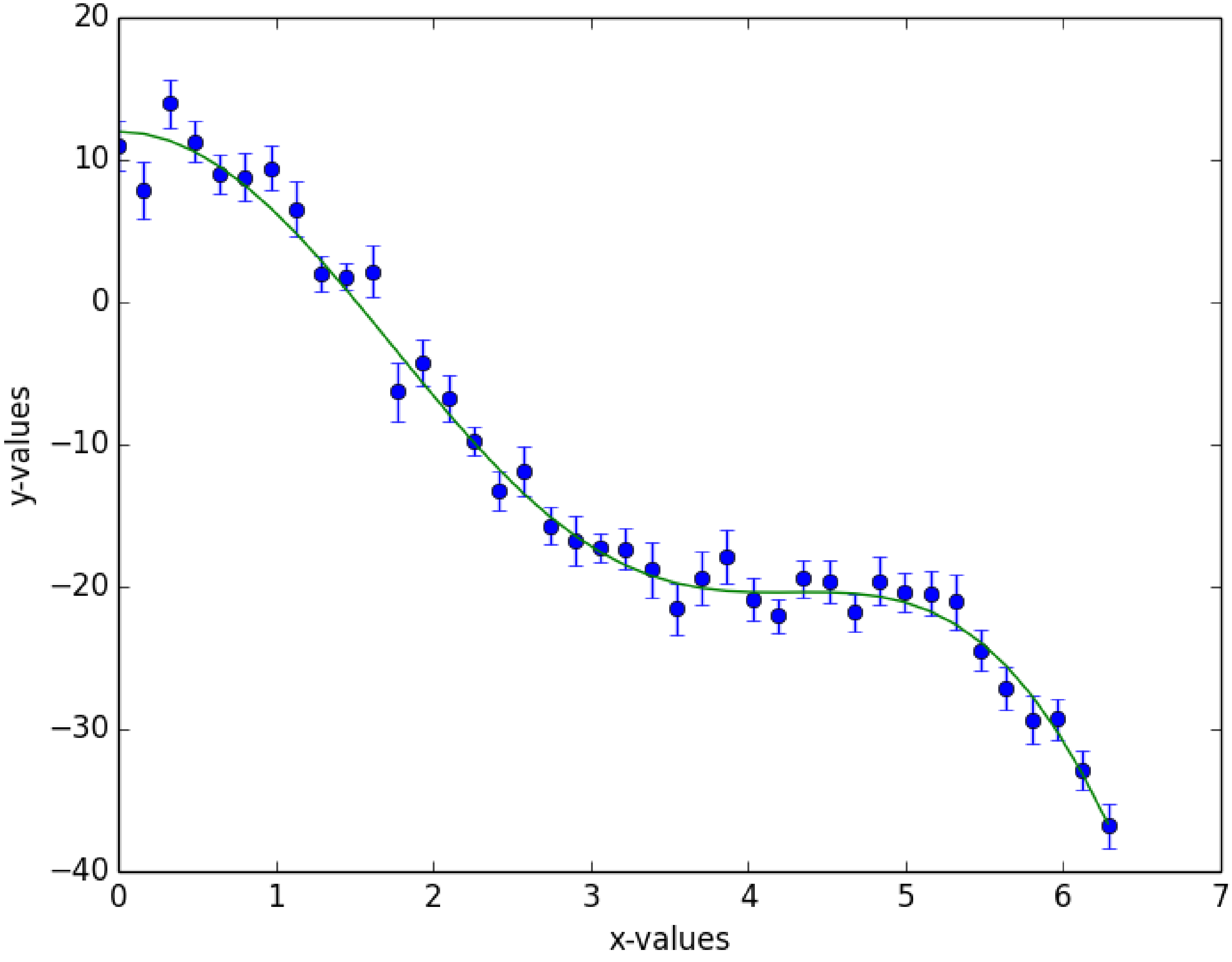
$$p_i(best) = p_i \pm \alpha^{-1}[i,i]$$

```
>>> for i in range(len(p)):
>>>         print ‘p[‘+str(i)+’]=‘+str(p[i])+’ +/- ‘+str(alpha_inverse[i,i])
```

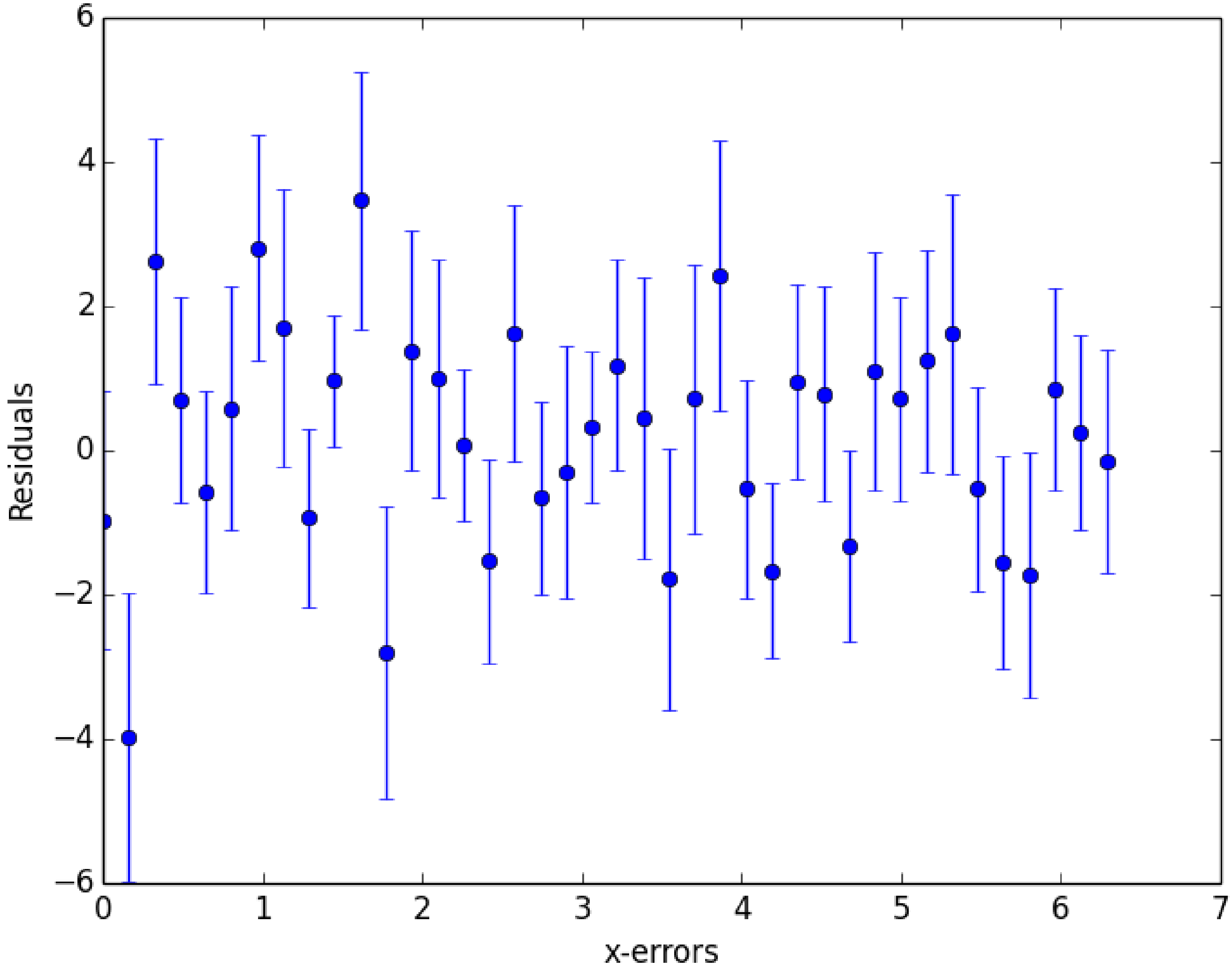
10. Now plot out the residuals

```
>>> plt.clf()
>>> plt.errorbar(x,y-f(x,p),yerr=sigma,fmt='o')
>>> plt.xlabel(‘x-values’)
>>> plt.ylabel(‘Residuals’)
```

What do you expect to see?



20
10
0
−10
−20
−30
−40
0
1
2
3
4
5
6
7
x-values
y-values


Residuals
x-errors
6
4
2
0
−2
−4
−6
0
1
2
3
4
5
6
7

Recap – Fitting Linear Models

- We fit linear models by asking our data, “What set of parameters minimizes the squared distance between the model and the data points?”

- Casting the problem into matrix form makes the math straightforward. So we write a matrix equation that describes the problem.

$$\vec{y} = M\vec{p}$$

- Our goal is then to solve this equation for $\vec{p}$ which will be the vector of best-fit parameters.

- The errors in the parameters come from our covariance matrix, α^{-1}.

- I don’t know of a robust python program to do this for you so make sure your own code works!

Exercise

- In order to focus the LMT, we must make a small map of a source at different positions of the z-position of the secondary mirror (M2). We measure the amplitude of the source in the map at different z-positions and then derive the best-fit quadratic to the data.
- Use the matrix technique of least-squares fitting to find the best-fit position (in z) of the secondary mirror (and its error!) assuming this focus data:
  - z_offset = [-1., 0., -2., 2., 1.]
  - amplitude = [2.23 , 3.02, 0.982, 2.00, 2.92]
  - amplitude_error = [0.03, 0.03, 0.025, 0.029, 0.03]

Exercise

Insulin is a key hormone for the regulation of the production of glucose in our blood. Diabetics have a problem metabolizing glucose and 5-10% of people with diabetes are unable to produce insulin at all. Consequently, these people must take an injection of insulin regularly.

Insulin in blood breaks down with time and so the concentration of insulin is a strong function of time obeying the relation

$$C = Ae^{-\tau/t}$$

where C is the concentration and t is measured in minutes.

Using the data from ‘insulin.npz’, find the best-fit values for A and τ and their associated errors.

(Hint: you will need to cleverly re-write the equation above to cast it in a form suitable for a linear fit to the parameters. Remember, your data is **your** data and you are welcome to transform it as you wish to make your life easier.)