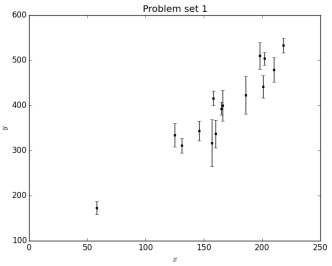
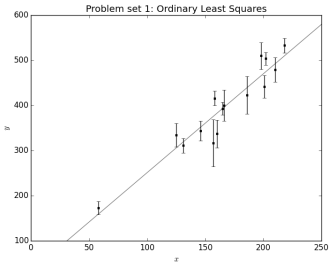
Problem Set 1: Fitting a Line to Data

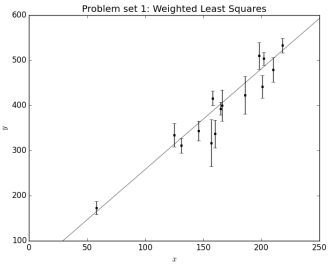
Problem 1: Getting started. Get the Python sample code `ps1.py`. Run it and look at the plots that are produced:



`p1-data.png`



`p1-ols.png`



`p1-wls.png`

Read through the code to see how we read the data, run the standard least squares fitting procedure using linear algebra, and plot the results.

Problem 2: Likelihood. The sample code contains a data set $\{x_i, y_i\}_{i=1}^N$. Imagine that each y_i value is drawn from a Gaussian distribution with mean $m\,x_i + b$ and variance σ_{yi}^2. Assume the x_i and σ_{yi} values are known exactly.

Write a Python function that returns the probability distribution function of the "observations" $\mathbf{y} \equiv \{y_i\}_{i=1}^N$ given a particular set of "parameter" values (m, b). That is, evaluate the product of the N Gaussians. This value is called the "likelihood" of the parameters (m, b) given the data $\mathbf{y}$. Your function should look like:

```
def straight_line_gaussian_likelihood(x, y, sigmay, m, b):
    '''
    Return the likelihood of drawing data values *y* at
    known values *x* given Gaussian measurement noise with standard
    deviation with known *sigmay*, where the "true" y values are
    *y_t = m * x + b*

    x: list of x coordinates
    y: list of y coordinates
    sigmay: list of y uncertainties
    m: scalar slope
    b: scalar line intercept

    Returns: scalar likelihood
    '''
    # compute it!
    return likelihood
```

Run the function for the data set you've been given (in the sample code `ps1.py`) for the parameter values $(m, b) = (2.2, 30)$. Write your name and the output on the board.

Problem 3: MCMC. "Markov-Chain Monte-Carlo" is a general name for a class of algorithms that act locally but produce a globally fair sampling of a posterior probability distribution function in a parameter space. MCMC algorithms proceed by taking steps randomly and accepting or rejecting them according to a probabilistic formula. The Metropolis–Hastings MCMC algorithm is the following step iterated many times:

```
def metropolis_hastings_step(old_params, data):
    new_params = sample_new_params(old_params)
    new_prob = posterior_probability(new_params, data)
    old_prob = posterior_probability(old_params, data)
    if (new_prob / old_prob) > uniform_random_number():
        return new_params
    else:
        return old_params
```

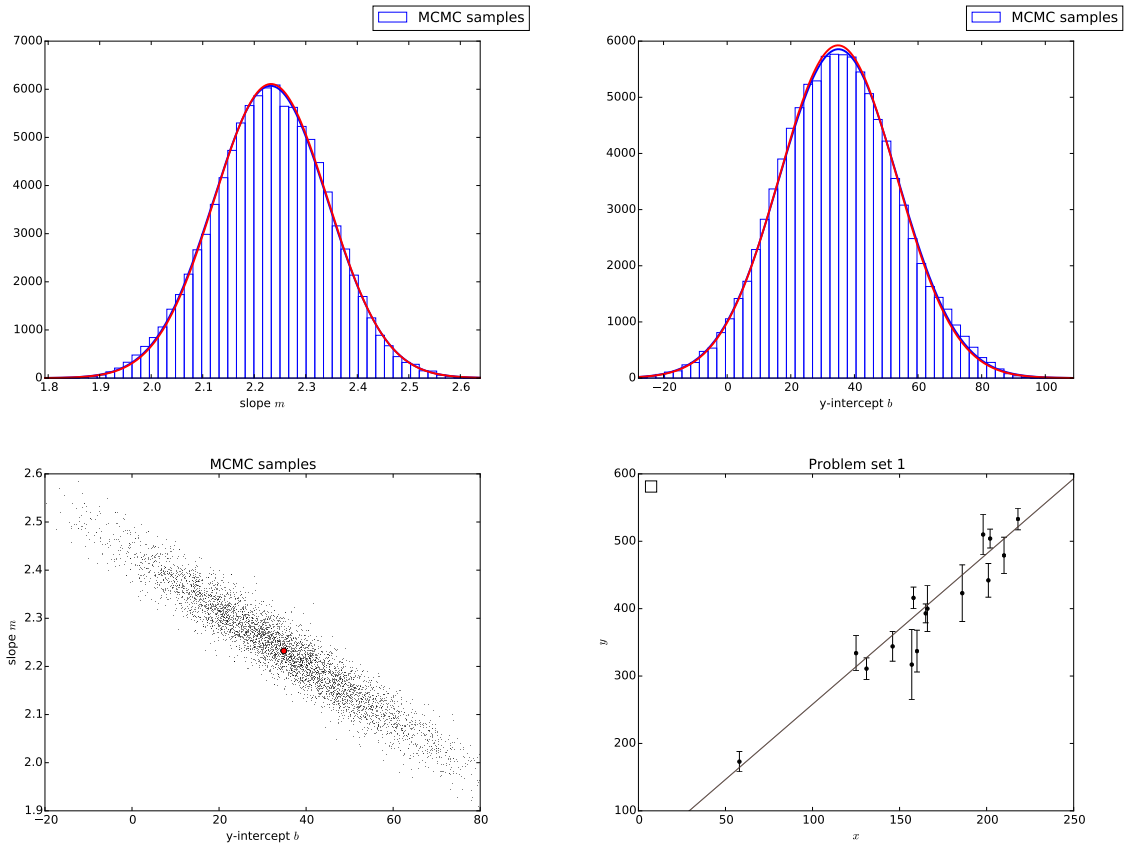
where the function `sample_new_params(params)` takes a local step in parameter space from the input parameters `params`, the `posterior_probability_function` returns the likelihood of the input parameters (because we are ignoring priors for now), the `uniform_random_number()` function returns a random number in the range $[0, 1]$, and we have assumed that the `sample_new_params` function is appropriately symmetric in parameter space[1]

Write a Metropolis–Hastings sampler and use for the posterior probability the likelihood function you wrote in the previous problem. For the parameter sampling function, use small Gaussian steps in (m, b). If your step-size is about right, the function will accept the new parameters about half the time. Try to write your code to reduce the number of likelihood function calls, and try to log relevant metrics.

Run your sampler for 10^5 steps, and plot the one-dimensional histograms of m and b generated by the chain. Also plot the two-dimensional scatter plot of m against b. Also plot the fit line with highest posterior probability on an x, y plot of the data points. If your acceptance fraction is significantly above or below one-half, adjust your step sizes and re-run.

You should get plots like this:

[1] If you read the Wikipedia article on "Metropolis–Hastings", this means that $Q(x, x') = Q(x', x)$.



Problem 4: emcee. Try using the *emcee* sampler rather than your Metropolis–Hastings sampler. Note that you will have to write a *log* posterior function. Compare the results as you vary the number of walkers, the number of steps, and the initialization.

Problem 5: Linear least squares. Run a standard linear least square fit on the data, using properly the errors $\{\sigma_{yi}\}$. Overplot a Gaussian on each of the one-dimensional histograms you plotted with the MCMC in the previous problem, where the mean of the overplotted Gaussian is the best fit from the LLS, and the variance is the uncertainty variance from the LLS. You will have to scale the amplitudes of the Gaussians appropriately.

Overplot on the two-dimensional scatterplot the one-sigma ellipse implied by the LLS best-fit value and output covariance matrix.

If there are any significant discrepancies between the MCMC sampling output and the LLS output, there is a bug in your code.

Problem 6: Challenge: outliers. Note that in the `ps1.py` code, the function `get_data_no_outliers` there is a range `[5:]` applied to the data. This excludes the first five data points from the `data1` variable. Remove this data exclusion and re-run your code. Observe that the results are totally thrown off by the outliers.

Now assume that each observation y_i is either drawn from a "foreground" distribution (a Gaussian with mean $m\,x_i + b$ and variance σ_{yi}^2, as before), or a "background" distribution. That is, each observation is either an "inlier" or an "outlier". This is called a "mixture

model". Replace your likelihood function with a likelihood function that has two (weighted) Gaussians, one Gaussian for the foreground and one for the background. For the background model, you can just use something simple like the population mean and variance of the $\mathbf{y}$ values. Run your MCMC code to sample the parameters of this new likelihood function: m and b for the foreground, plus the mixing fraction α, where the "weight" of the foreground model is α and the weight of the background is $1 - \alpha$. Do the best-fit m and b values become reasonable again? How does the best α you find compare with the fraction of outliers in the data?

Problem 7: Challenge: explanation. Explain in words (some math is permitted) why the Metropolis–Hastings local step algorithm leads (in the limit) to a fair sampling of the posterior probability distribution function.