Astro518_HW_02

September 13, 2014

In[3]:
```
import numpy as np
import matplotlib.pyplot as plt
import prettyplotlib as ppl
```

1 Problem 1, Linear Fit

1.1 Ignoring the first four points

- linear fit to function $y = mx + b$ be done with the function:

$$\begin{bmatrix} b \\ m \end{bmatrix} = [A^T C^{-1} A]^{-1} [A^T C^{-1} Y] \tag{1}$$

in which

$$A = \begin{bmatrix} 1 & x_1 \\ 2 & x_2 \\ \vdots & \vdots \\ n & x_n \end{bmatrix} \tag{2}$$

$$C = \begin{bmatrix} \frac{1}{\sigma_1^2} & 0 & \cdots & 0 \\ 0 & \frac{1}{\sigma_2^2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{1}{\sigma_n^2} \end{bmatrix} \tag{3}$$

In[4]:
```
## read data
(data_id, x, y, dy, dx, rho_xy) = np.loadtxt('sample_data.txt', comments = '%', unpack = True)
```

In[5]:
```
## linear fitting using the method in Hogg et. al.
## '_1' indicates the removal of first 4 lines of the orginal data

x_1 = x[4:]
y_1 = y[4:]
dy_1 = dy[4:]

# Different from matlab, numpy array is a different data type from matrix.
# * operator in numpy is similar to .* operator in matlab
# to carry out matrix multiply, the numpy array has to be converted to matrix using function np.mat()

Y_vec_1 = np.mat(y_1).T
```

```
A_1 = np.mat([np.ones(len(x_1)), x_1]).T
C_1 = np.mat(np.diagflat(dy_1**2))


## calculate b and m
(b_1, m_1) = (A_1.T *C_1**(-1) * A_1)**(-1) * (A_1.T * C_1**(-1) * Y_vec_1)

b_1 = b_1.flat[0]
m_1 = m_1.flat[0] # convert matrix to number

print 'Fitting result for y = mx + b'
print 'm = {0:.2f}'.format(m_1)
print 'b = {0:.2f}'.format(b_1)
```

```
Fitting result for y = mx + b
m = 2.24
b = 34.05
```

- The uncertainties of fitting parameter could be calculated with following equation

$$\begin{bmatrix} \sigma_b^2 & \sigma_{mb} \\ \sigma_{bm} & \sigma_m^2 \end{bmatrix} = [A^T C^{-1} A]^{-1} \tag{4}$$

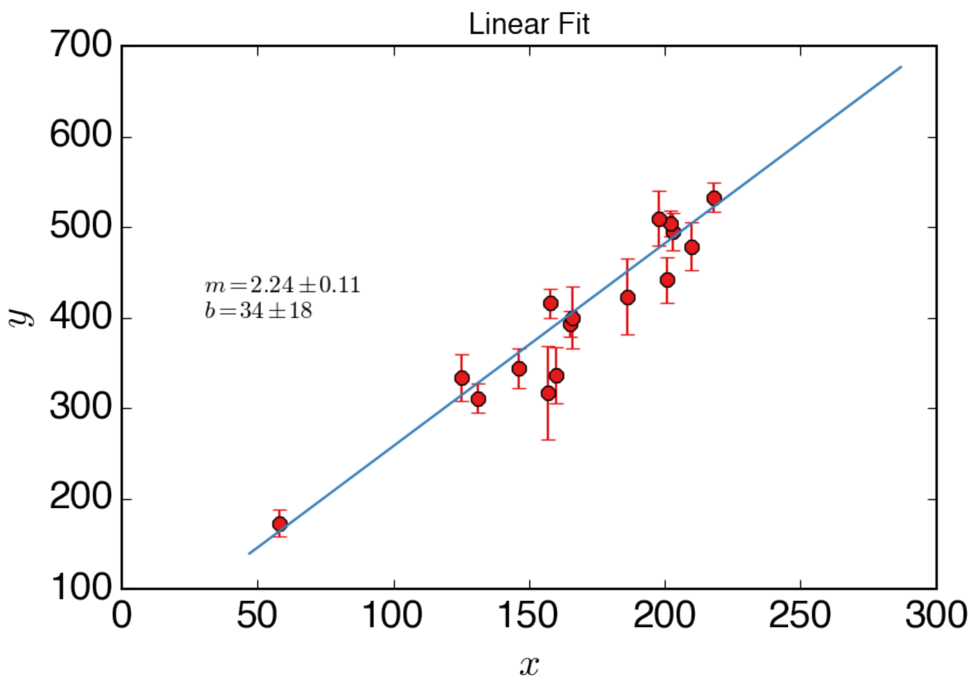
In [96]:
```
var_b_1, var_m_1 = np.diag((A_1.T * C_1**(-1) * A_1)**(-1))
dm_1 = np.sqrt(var_m_1)
db_1 = np.sqrt(var_b_1)

print 'The uncertainties:'
print 'sigma_m = {0:.2f}'.format(dm_1)
print 'sigma_b = {0:.2f}'.format(db_1)
```

```
The uncertainties:
sigma_m = 0.11
sigma_b = 18.25
```

- plot the result

In [7]:
```
fig = plt.figure()
ax = fig.add_subplot(111)
ax.errorbar(x_1, y_1, yerr = dy_1, fmt = 'o', mew = 0.6)
x_sample = np.linspace(min(x), max(x), 100)
ax.plot(x_sample, x_sample * m_1 + b_1)
ax.set_title('Linear Fit')
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
ax.text(30, 400, '$m = 2.24\pm 0.11$\n$b = 34 \pm 18$')
```

Out[7]:
```
<matplotlib.text.Text at 0x10f84e990>
```



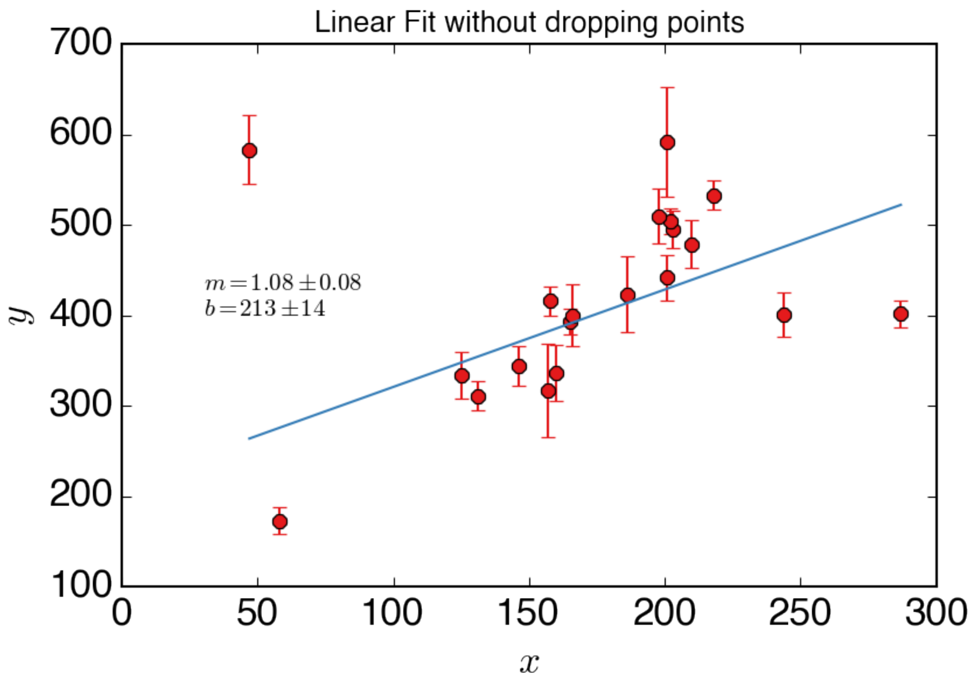
1.2 Include the first 4 points

In [8]:
```
Y_vec = np.mat(y).T ## turn the array into matrix
A = np.mat([np.ones(len(x)), x]).T
C = np.mat(np.diagflat(dy**2))

(b, m) = (A.T *C**(-1) * A)**(-1) * (A.T * C**(-1) * Y_vec)

b = b.flat[0]
m = m.flat[0]

print 'After including the first 4 points, fitting result for y = mx + b'
print 'm = {0:.2f}'.format(m)
print 'b = {0:.2f}'.format(b)

## and the uncertainties

var_b, var_m = np.diag((A.T * C**(-1) * A)**(-1))
dm = np.sqrt(var_m)
db = np.sqrt(var_b)

print 'And the uncertainties are:'
print 'sigma_m = {0:.2f}'.format(dm)
print 'sigma_b = {0:.2f}'.format(db)
```

```
After including the first 4 points, fitting result for y = mx + b
m = 1.08
b = 213.27
And the uncertainties are:
sigma_m = 0.08
sigma_b = 14.39
```

```
In[9]: fig = plt.figure()
       ax = fig.add_subplot(111)
       ax.errorbar(x, y, yerr = dy, fmt = 'o', mew = 0.6)
       ax.plot(x_sample, x_sample * m + b)
       ax.set_title('Linear Fit without dropping points')
       ax.set_xlabel('$x$')
       ax.set_ylabel('$y$')
       ax.text(30, 400, '$m = 1.08\pm 0.08$\n$b = 213 \pm 14$')
```

```
Out[9]: <matplotlib.text.Text at 0x10f98de90>
```



- Comparison of the results of 1.a and 1.b:
- 1.a,

$$m = 2.24 \pm 0.11$$
$$b = 34 \pm 18$$

- 1.b

$$m = 1.08 \pm 0.08$$
$$b = 213 \pm 14$$

- The difference of the two sets of results are more than 5σ
- 1.b has smaller uncertainties because this fit bases on a larger data set

1.3 Include a quadratic term

matrix A changes to

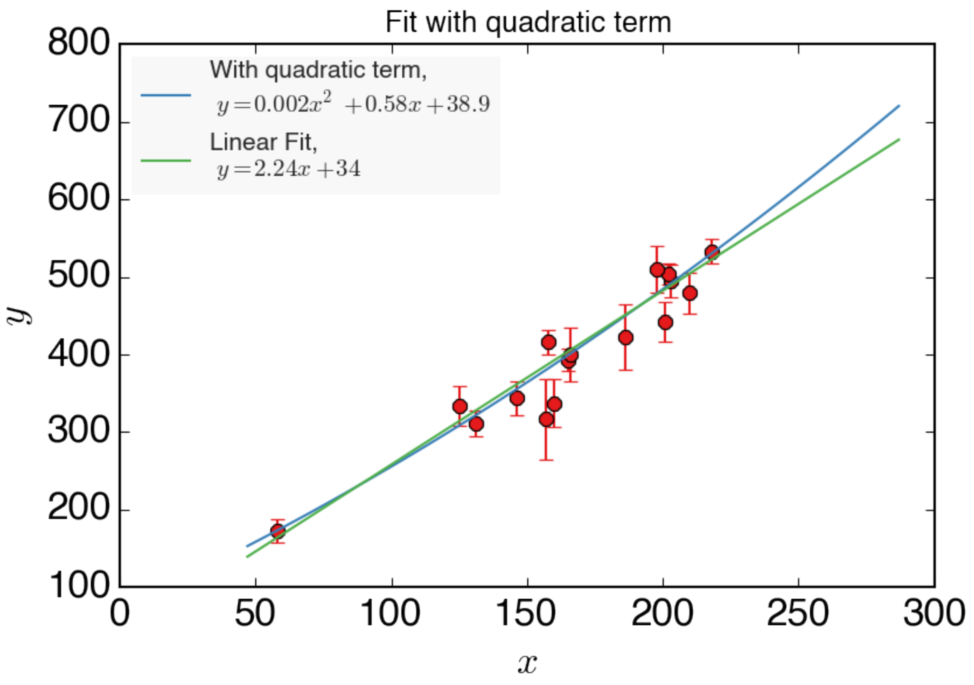
$$A_{\text{quad}} = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 \end{bmatrix} \tag{5}$$

In[10]:
```
A_quad = np.mat([np.ones(len(x_1)), x_1, x_1**2]).T


## fit y = a2*x**2 + a1*x + a0
(a0_quad, a1_quad, a2_quad) = (A_quad.T *C_1**(-1) * A_quad)**(-1) * (A_quad.T * C_1**(-1) * Y_vec_1)

a0_quad = a0_quad.flat[0]
a1_quad = a1_quad.flat[0]
a2_quad = a2_quad.flat[0]

print 'Fitting result for function y = a2*x**2 + a1*x + a0'
print 'a2 = {0:.4f}'.format(a2_quad)
print 'a1 = {0:.2f}'.format(a1_quad)
print 'a0 = {0:.2f}'.format(a0_quad)

## calculate the uncertainties
var_as = np.diag((A_quad.T * C_1**(-1) * A_quad)**(-1))
da0, da1, da2 = np.sqrt(var_as)

print 'Uncertainties:'
print 'sigma_a2 = {0:.4f}'.format(da2)
print 'sigma_a1 = {0:.4f}'.format(da1)
print 'sigma_a0 = {0:.4f}'.format(da0)
```

```
Fitting result for function y = a2*x**2 + a1*x + a0
a2 = 0.0023
a1 = 1.60
a0 = 72.89
Uncertainties:
sigma_a2 = 0.0020
sigma_a1 = 0.5797
sigma_a0 = 38.9116
```

In[95]:
```
fig = plt.figure()
ax = fig.add_subplot(111)
ax.errorbar(x_1, y_1, yerr = dy_1, fmt = 'o', mew = 0.6)
ax.plot(x_sample, x_sample**2 * a2_quad + x_sample*a1_quad + a0_quad,
        label = 'With quadratic term,\n $y=0.002x^2+0.58x+38.9$' )
ax.plot(x_sample, x_sample * m_1 + b_1, label = 'Linear Fit,\n $y=2.24x+34$')
ax.set_title('Fit with quadratic term')
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
ppl.legend(ax, loc = 'best')
```

Out[95]:
```
<matplotlib.legend.Legend at 0x10c41f2d0>
```



2 Problem 2, Goodness to fit and use χ^2

2.1 χ^2 calculation

```
In[11]: def chisq(y, y_mod, dy):
            """
            function to calculate chi square
            """
            return np.sum((y-y_mod)**2/dy**2)

        chisq_1a = chisq(y_1, x_1 * m_1 + b_1, dy_1)
        chisq_1b = chisq(y, x * m + b, dy)

        chisq_1a0 = chisq_1a/(len(x_1) - 2)
        chisq_1b0 = chisq_1b/(len(x) - 2)

        print 'chi square calculation:'
        print 'For 1a,\n chisq = {0:.2f}\n chisq/(N-2) = {1:.2f}'.format(chisq_1a, chisq_1a0)
        print 'For 1b,,\n chisq = {0:.2f}\n chisq/(N-2) = {1:.2f}'.format(chisq_1b, chisq_1b0)
```

```
chi square calculation:
For 1a,
 chisq = 18.68
 chisq/(N-2) = 1.33
For 1b,,
 chisq = 289.96
 chisq/(N-2) = 16.11
```

The results of χ^2 calculation shows that, 1. for 1a, $\chi^2/(N-2) \sim 1$, the data are well fitted to a linear function. 2. for 1b, $\chi^2/(N-2) \gg 1$, the data are not well defined by a linear function.

2.2 replace σ to get a 'good' χ

suppose

$$\chi_0^2 = \sum \frac{(y_i - y(x_i))^2}{S^2(N-2)} = 1 \tag{6}$$

we can get

$$S = \sqrt{\sum((y_i - y(x_i))^2/(N-2)} \tag{7}$$

In[12]:
```
S_1a = np.sqrt(chisq(y_1, m_1 * x_1 + b_1, 1)/(len(x_1) - 2))
S_1b = np.sqrt(chisq(y, m * x + b, 1)/(len(x) - 2))

print 'For 1a, this S value is S = {0:.2f}'.format(S_1a)
print 'For 1b, this S value is S = {0:.2f}'.format(S_1b)
```

```
For 1a, this S value is S = 32.16
For 1b, this S value is S = 104.01
```

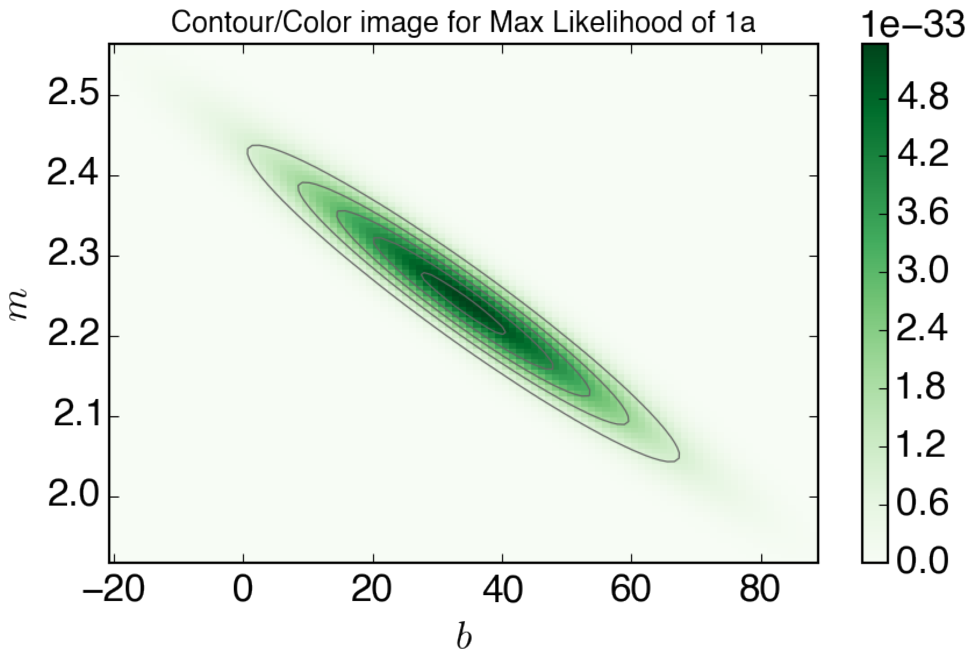
- 1.b needs a much larger S value

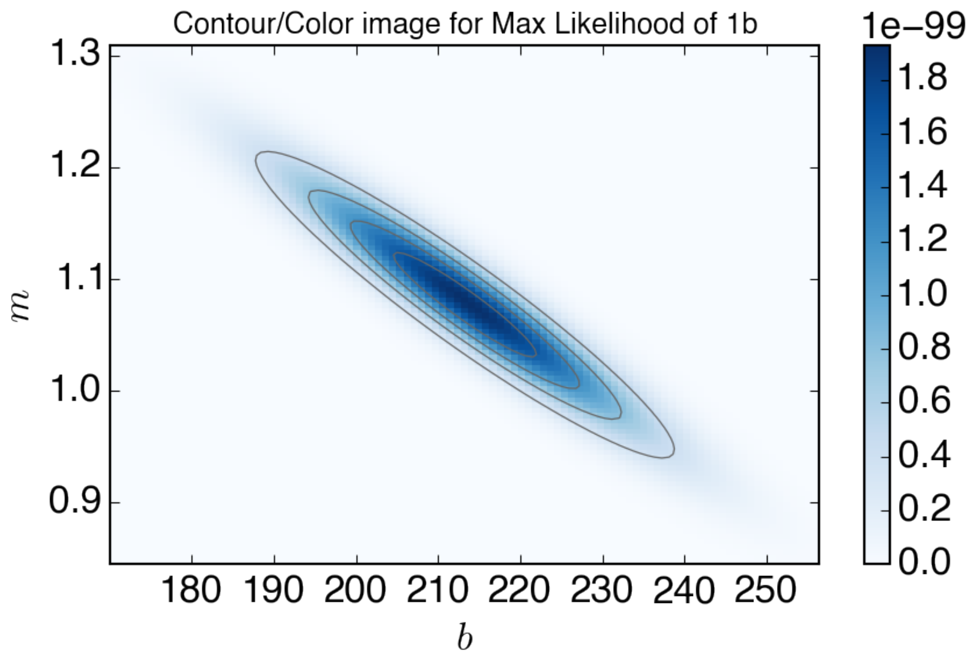
2.3 Maximum likelihood calculation

In[13]:
```
def maxLikelihood(func, x, y, dy):
    """
    maximum likelihood calculation
    assuming that y have a Gaussian distribution
    ignoring the x's uncertaities
    """
    return np.prod(1/np.sqrt(2 * np.pi * dy**2) * np.exp(-(y - func(x))**2/(2 * dy**2)))

def linearMLMat(m_min, m_max, b_min, b_max, x, y, dy, ngrid = 100):
    """
    return a 2d array that contains the ML calculation for each grid point
    """
    m_range = np.linspace(m_min, m_max, ngrid)
    b_range = np.linspace(b_min, b_max, ngrid)

    mb_mesh = [(mi, bj) for mi in m_range for bj in b_range]# a list of the grid coordinates

    def ml_func(m, b):
        return maxLikelihood(lambda xi: m * xi + b, x, y, dy)

    # use 'map' to avoid for-loop to accelerate the program
    return m_range, b_range, np.array(map(ml_func, mb_mesh)).reshape((len(m_range), len(b_range)))

mrange_1a, brange_1a, mlmat_1a = linearMLMat(m_1 - 3 * dm_1, m_1 + 3 * dm_1,
                                              b_1 - 3 * db_1, b_1 + 3 * db_1, x_1, y_1, dy_1)
mrange_1b, brange_1b, mlmat_1b = linearMLMat(m - 3 * dm, m + 3 * dm, b - 3 * db,
                                              b + 3 * db, x, y, dy)
```

In[13]:
```
### plot the contour/color image
fig, ax = plt.subplots()
cax_1a = ax.pcolormesh(brange_1a, mrange_1a, mlmat_1a, cmap = 'Greens')
ax.contour(brange_1a, mrange_1a, mlmat_1a, 5, linewidths = 0.6, colors = '0.4')
ax.set_xlabel('$b$')
ax.set_ylabel('$m$')
ax.set_title('Contour/Color image for Max Likelihood of 1a')
ax.autoscale(tight = True)
```

```
fig.colorbar(cax_1a)
fig.tight_layout()

figb, axb = plt.subplots()
cax_1b = axb.pcolormesh(brange_1b, mrange_1b, mlmat_1b, cmap = 'Blues')
axb.contour(brange_1b, mrange_1b, mlmat_1b, 5, linewidths = 0.6, colors = '0.4')
axb.set_xlabel('$b$')
axb.set_ylabel('$m$')
axb.set_title('Contour/Color image for Max Likelihood of 1b')
axb.autoscale(tight = True)
figb.colorbar(cax_1b)
figb.tight_layout()
```





3 Problem 3, Bayesian analysis

3.1 a, Maximum likelihood with assumption of 'bad point' probability

- Following function `linearML_ba_Mat` calcualtes the maximum likelihood grids.

In[14]:
```
def bayesianML(func, x, y, dy):
    """
    calculate Max likelihood using bayesian analysis
    """
    Pb_range = np.linspace(0, 1, 11)
    Vb_range = np.linspace(0, 4000, 11)
    Yb_range = np.linspace(0, 700, 11)
    PVY_list = [(pi, vj, yk) for pi in Pb_range for vj in Vb_range for yk in Yb_range]

    def bayesian_func(pb, vb, yb):
        np.prod((1 - pb)/np.sqrt(2 * np.pi * dy**2) * np.exp(-(y - func(x))**2/(2 * dy**2))
                + pb/np.sqrt(2 * np.pi * (dy**2 + vb)) * np.exp(-(y - yb)**2/(2 * dy**2)))

    return np.sum(map(bayesian_func, PVY_list))

def linearML_ba_Mat(m_min, m_max, b_min, b_max, x, y, dy, ngrid = 20):
    """
    return a 2d array that contains the ML-Batesian Analysis calculation for each grid point
    """
    m_range = np.linspace(m_min, m_max, ngrid)
    b_range = np.linspace(b_min, b_max, ngrid)
    mb_mesh = [(mi, bj) for mi in m_range for bj in b_range]# a list of the grid coordinates
    ba_ml_func = lambda (m, b): bayesianML(lambda xi: m * xi + b, x, y, dy)
    ba_ml_mat = np.array(map(ba_ml_func, mb_mesh)).reshape((len(m_range), len(b_range)))
    return m_range, b_range, ba_ml_mat
```

```
m_ba_range_1b, b_ba_range_1b , ba_ml_mat_1b = linearML_ba_Mat(1.6, 3.2, -120, 120,
                                                              x, y, dy, ngrid = 20)
```

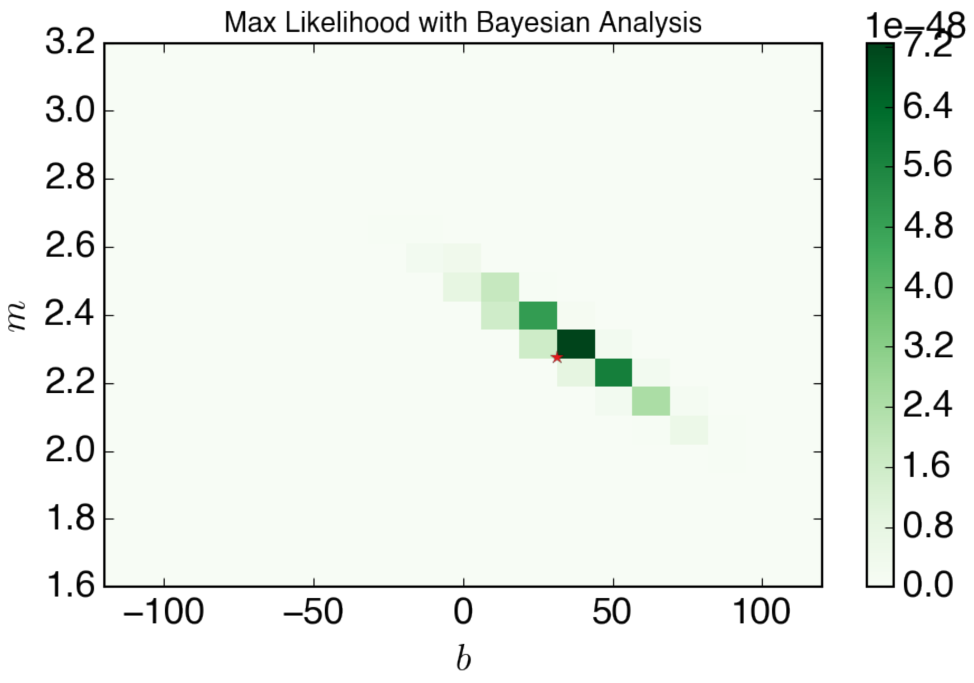
- Considering the large time consuming of function `linearML_ba_Mat`, the parameters' ranges need to be constrained step by step. For every step, the result is plotted with function `plotML_ba` defined as below.

In[15]:
```
def plotML_ba(m_lim, b_lim, x, y, dy, ngrid = 20):
    """
    calculate max likelihood matrix with bayesian analysis
    plot the result and return the coordinate (m, b) that gives the max likelihood
    """
    m_range, b_range, ba_ml_mat = linearML_ba_Mat(m_lim[0], m_lim[1], b_lim[0], b_lim[1],
                                                  x, y, dy, ngrid = ngrid)
    fig, ax = plt.subplots()
    cax = ax.pcolormesh(b_range, m_range, ba_ml_mat, cmap = "Greens")
    fig.colorbar(cax)
    fig.tight_layout()
    ax.set_title('Max Likelihood with Bayesian Analysis')
    ax.set_xlabel('$b$')
    ax.set_ylabel('$m$')
    ax.autoscale(tight = True)
    i,j = np.unravel_index(ba_ml_mat.argmax(), ba_ml_mat.shape) # get the position of the maximum
    ax.plot(b_range[j], m_range[i], '*')
    return m_range[i], b_range[j]
```

3.1.1 First step, roughly define the position of the maximum

- $m$ from 1.6 to 3.2, $b$ from -120 to 120, 20 number of grid points on each dimension
- The resolution for $m$ is 0.08, and for $b$ is 12

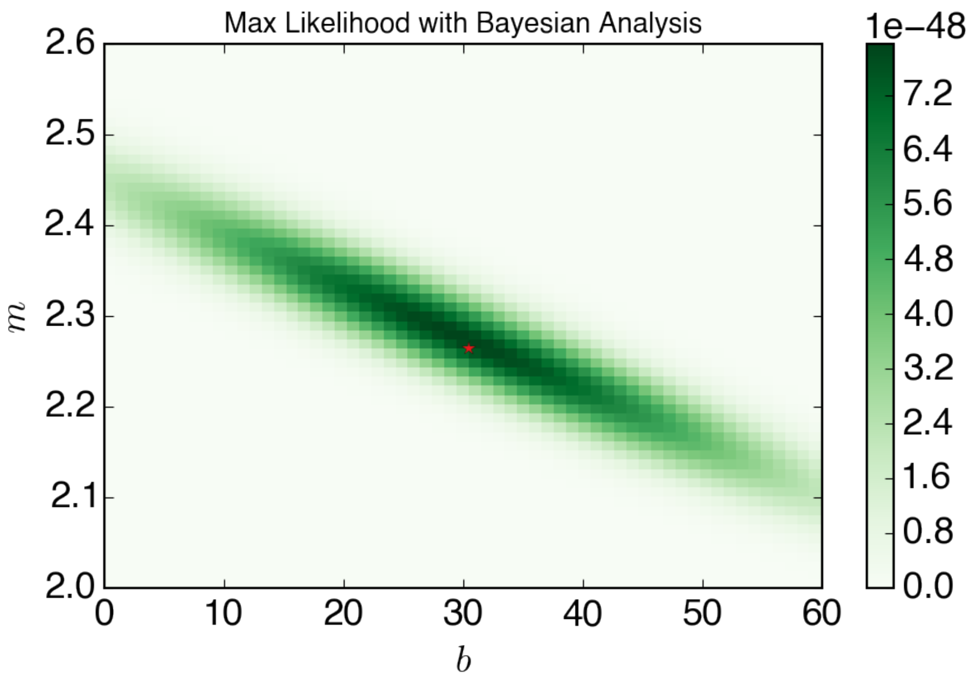
In[16]:
```
m_max, b_max = plotML_ba([1.6, 3.2], [-120, 120], x, y, dy, ngrid = 20)
```



- According to the result of first step, we can find the maximum point is roughly in the range of $m \sim (2, 2.6)$, $b \sim (0, 60)$ ### Second step, precisely define the maximum position
- `m_min = 2, m_max = 2.6, b_min = -0, b_max = 60, ngrid = 60`

In[97]:
```
m_max, b_max = plotML_ba([2, 2.6], [0, 60], x, y, dy, ngrid = 60)

print 'The maximum likelihood locates at:'
print 'm = {0:.2f}\n b = {1:.2f}'.format(m_max, b_max)
```

```
The maximum likelihood locates at:
m = 2.26
 b = 30.51
```



3.2 the comparison

- The result of 3.a is very close to the result of 1.a, and different from that of 1.b
- With Bayesian analysis, the 4 points that were ignored in 1.a have larger probibilities to be bad points. Thus these points cannot skew the fitted line very much
- **3.a is the best**. In 1.a, we brutally excludes 4 points that are not belong the linear relation as we expected. This will introduce a bias. In 1.b, the fitting is largely skewed by several points, and the linear relation in 1.b is not prominent. Bayesian analysis reduces the influence of outlier with prior assumption. The better the probability distribution of being a bad point is known, the more accurate the fitting result will be.

4 Problem 4, The uncertainties of fitting parameter, *bootstrap*

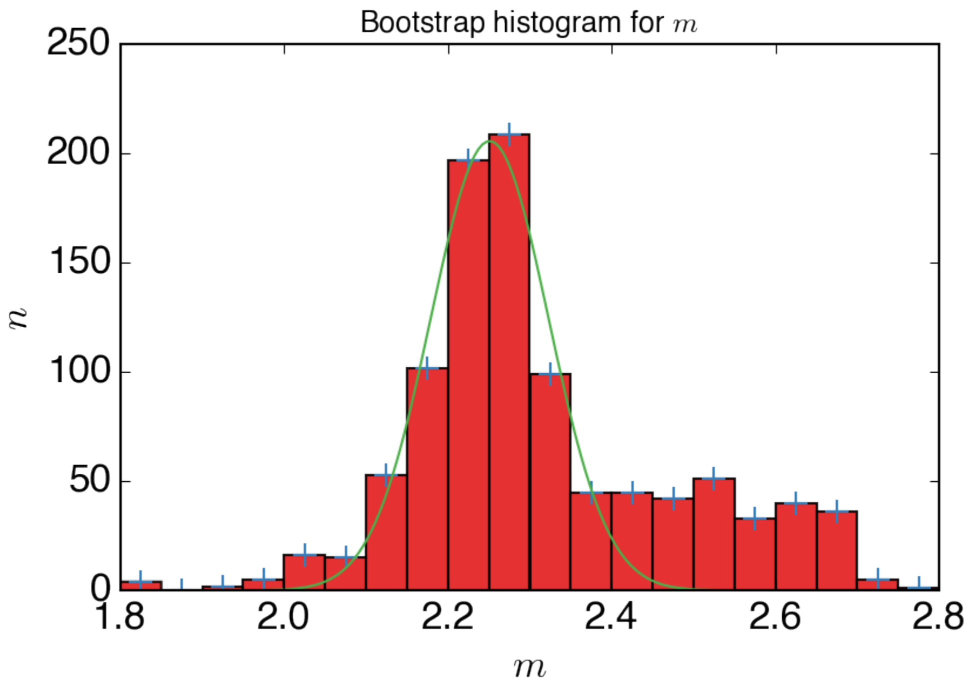
- Following function `bootstrap` caculates the values and uncertainties of m and b with bootstrap method.

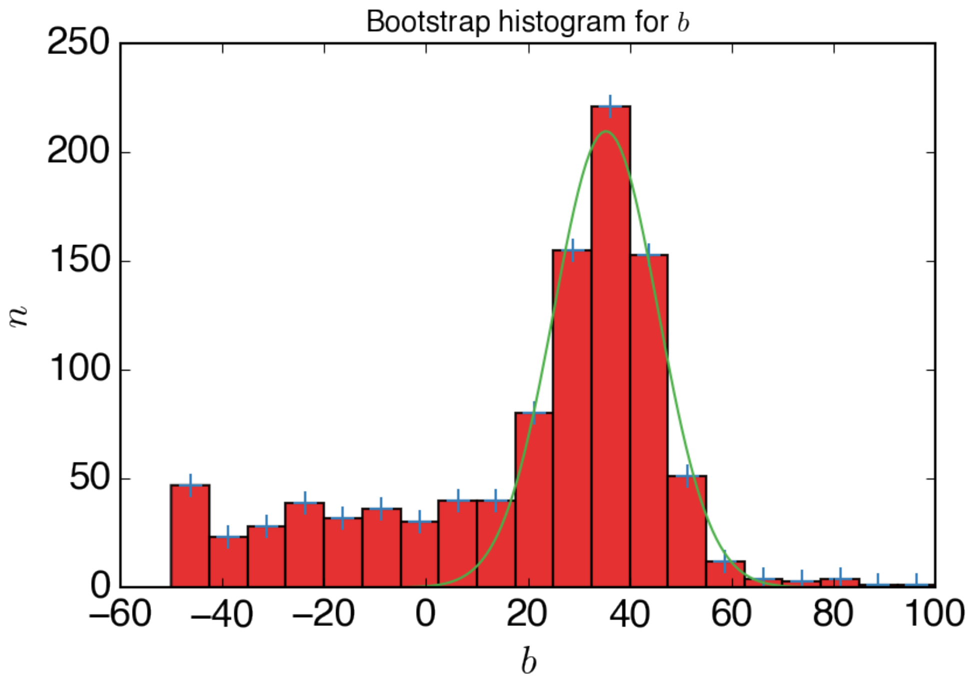
In[]:
```
def bootstrap(niter, x, y, dy):
    """
    calculate the fitting result using Bayesian max likelihood
    input parameter:
        iteration time
        x, y, dy
    output parameter:
        fitting result list
    """
    result_list = []
    for i0 in range(niter):
        # numpy provides random module that can generate random integers in a given interval
        # this function uses these random numbers to index the data
        index = np.random.random_integers(0, len(x) - 1, len(x) - 1)
        xi0 = x[index]
        yi0 = y[index]
        dyi0 = dy[index]
        m_range, b_range, ba_ml_mat = linearML_ba_Mat(1.8, 2.8, -50, 100, xi0, yi0, dyi0, ngrid = 20)
        i,j = np.unravel_index(ba_ml_mat.argmax(), ba_ml_mat.shape)
        result_list.append((m_range[i], b_range[j]))
        print '{0} step finished'.format(i0)


    return result_list

mb_bootstrap = bootstrap(1000, x, y, dy)
```

In[90]:
```
from scipy.optimize import curve_fit
m_bstrp = map(lambda item: item[0], mb_bootstrap)
b_bstrp = map(lambda item: item[1], mb_bootstrap)

def plot_bootstrap(bstrp, nbins = 20, p0 = [1., 0., 1.]):
    # p0 is the intial values for gaussian fitting
    """
    plot the bootstrap result as a histogram,
    fit it with a gaussian curve, and return the fitting result
    """
    fig = plt.figure()
    ax = fig.add_subplot(111)
    n, bins, patches = ax.hist(bstrp, bins = nbins, alpha = 0.9)
    bin_centers = (bins[:-1] + bins[1:])/2.
    ax.plot(bin_centers, n, '+', mew = '1', ms = 10)

    def gauss(x, *p):# define a gaussian function for fitting
        A, mu, sigma = p
        return A*numpy.exp(-(x-mu)**2/(2.*sigma**2))

    para, var_matrix = curve_fit(gauss, bin_centers, n, p0 = p0)
    x_sample = np.linspace(np.min(bins),np.max(bins), 200)
    n_fit = gauss(x_sample, *para)

    ax.plot(x_sample, n_fit, '-') # plot the gaussian fitting result
    return para, fig

m_bstrp_fit, m_fig = plot_bootstrap(m_bstrp, p0 = [200, 2.2, 0.2])
m_ax = m_fig.axes[0]
m_ax.set_title('Bootstrap histogram for $m$')
m_ax.set_xlabel('$m$')
m_ax.set_ylabel('$n$')

b_bstrp_fit, b_fig = plot_bootstrap(b_bstrp, p0 = [200, 30, 10])
b_ax = b_fig.axes[0]
b_ax.set_title('Bootstrap histogram for $b$')
b_ax.set_xlabel('$b$')
b_ax.set_ylabel('$n$')
```

```
print 'According to bootstraping and the fitting result:'
print 'm = {0:.2f}, sigma_m = {1:.2f}'.format(m_bstrp_fit[1], m_bstrp_fit[2])
print 'b = {0:.2f}, sigma_b = {1:.2f}'.format(b_bstrp_fit[1], b_bstrp_fit[2])
```

```
According to bootstraping and the fitting result:
m = 2.25, sigma_m = 0.07
b = 35.34, sigma_b = 10.24
```





```
In[]:
```