HW-7-dlhogan

November 22, 2021

1 Homework 7

1.1 CEWA 565

1.2 Daniel Hogan

[1]:
```
import pandas as pd
import numpy as np
import scipy.stats as stats
from scipy import sparse
import matplotlib.pyplot as plt
%matplotlib inline
```

Problem 1:

[2]:
```
df = pd.read_csv('ENSO_to2021.csv', comment='#')
data = df['ENSO Phase']
# np.random.seed(1)
df.head(3)
```

[2]:

	Water Year	ENSO Phase
0	1900	1
1	1901	2
2	1902	2

A. Using the time series of the phase of the El Niño Southern Oscillation (ENSO) from 1900-2021, create a lag-1 Markov model of the ENSO phase.

Observed Phases of ENSO: - 1: warm (El Niño)
- 2: neutral (ENSO neutral)
- 3: cool, (La Niña)

[3]:
```
# count the transitions from each state to the next
# This counts the transitions from each state to the next and marks that count
S = sparse.csr_matrix((np.ones_like(data[:-1]), (data[:-1], data[1:])),␣
 ↪dtype=float)
# convert transition counts to matrix form
# This converts those counts to matrix form
tm = S.todense()
```

```
print(tm)
```

```
[[ 0.  0.  0.  0.]
 [ 0. 11. 12. 17.]
 [ 0. 12. 15. 10.]
 [ 0. 16. 10. 18.]]
```

Normalize the transition matrix to get probabilities. This will create our lag-1 Markov Model.

[4]:
```
tm_norm = tm / tm.sum(axis=1)

print(tm_norm) # This is our normalized transition matrix.
```

```
[[       nan        nan        nan        nan]
 [0.         0.275      0.3        0.425     ]
 [0.         0.32432432 0.40540541 0.27027027]
 [0.         0.36363636 0.22727273 0.40909091]]
```

```
<ipython-input-4-58e3ca30d8c7>:1: RuntimeWarning: invalid value encountered in
true_divide
  tm_norm = tm / tm.sum(axis=1)
```

Compute cumulative sums along the rows, make sure these sum to 1. (We will use this cdf matrix below in a simulation of ENSO phases)

[5]:
```
# We take the above probabilities of transitions, and turn them into discrete␣
 ↪CDF's.
# These will allow us to map random numbers generated from a uniform␣
 ↪distribution into
# transitions that follow these probability rules.
tm_cdf = np.cumsum(tm_norm,1)
```

B. Using this Markov model and a random number generator, simulate 5,000 years of ENSO data.

[6]:
```
# pick the number of years we want to simulate (5000)
n_years = 5000
# use a uniform random number for 5000 years
q = np.random.uniform(0,1,n_years); # uniformly distributed random numbers␣
 ↪n_years long

# start off in state 2, neutral
initialstate = 2; # give it an initial state, doesn't really matter which

Nrand = np.zeros_like(q) # initialize an array of the proper size, with the␣
 ↪initial state
Nrand[0] = initialstate;

# Now, just like we did when we created monte carlo simulations from empirical␣
 ↪CDFs,
```

```
# we use our uniform random numbers to look up the next state in the transition␣
 ↪matrix
for i in range(1,n_years):
    if q[i] <= tm_cdf[int(Nrand[i-1]),1]: #probability of transitioning from␣
 ↪state i to 1
        Nrand[i] = 1;
    elif q[i] <= tm_cdf[int(Nrand[i-1]),2]: #transition from state i to 2
        Nrand[i] = 2;
    else:
        Nrand[i] = 3;
```

[7]:
```
Nrand[-1]
```

[7]: 2.0

C. Using this randomly generated data, answer the following questions.

- According to the model, what is the probability that three warm ENSO years would occur in a row?
- What is the large-sample probability that three cool ENSO years would happen in a row?

(Try refreshing the numbers several times to increase the sample size if the condition never happens.)

[8]:
```
# And how many times did state 1 appear?
Test1 = [Nrand[0:-2], Nrand[1:-1], Nrand[2:]] # stack our data 3 times,␣
 ↪shifting it to the right by 1 each time
Test1 = np.stack(Test1, axis=1)

G2 = np.where((np.max(Test1, axis=1) == 1) & (np.min(Test1, axis=1) == 1))
# if both the maximum and the minimum are 3, then we have 3 ones in our sequence

freqofthree1s = G2[0].size / Test1.shape[0]

print('Frequency of three warm ENSOs in a row = {}%'.format(100*np.
 ↪round(freqofthree1s,2)))
```

```
Frequency of three warm ENSOs in a row = 2.0%
```

[9]:
```
# And how many times did state 1 appear?
Test1 = [Nrand[0:-2], Nrand[1:-1], Nrand[2:]] # stack our data 3 times,␣
 ↪shifting it to the right by 1 each time
Test1 = np.stack(Test1, axis=1)

G2 = np.where((np.max(Test1, axis=1) == 3) & (np.min(Test1, axis=1) == 3))
# if both the maximum and the minimum are 3, then we have 3 ones in our sequence

freqofthree1s = G2[0].size / Test1.shape[0]
```

```
print('Frequency of three cold ENSOs in a row = {}%'.format(100*np.
 ↪round(freqofthree1s,3)))
```

```
Frequency of three cold ENSOs in a row = 5.2%
```

According to the model, what is the probability that three warm ENSO years would occur in a row? The probability of three warm ENSO years in a row is approximately 3.0%

What is the large-sample probability that three cool ENSO years would happen in a row? (Try refreshing the numbers several times to increase the sample size if the condition never happens.) The probability of three cool ENSO years in a row is approximately 7.1%

Problem 2: Following the class discussion and Lab 7-3, explore how the rating curve and the 95% confidence intervals for the Lyell Fork streamflow site change depending on the method you use to determine the rating curve:

- Least squares linear regression fitting (with transformed variables) using b = 0.28 m Make 95% confidence intervals around this regression fit Then, assume that we don't know exactly what b is. Try additional linear regressions using different values of b = 0.10, 0.20, 0.30, 0.40, and 0.50 m (you do not need to calculate 95% confidence intervals for these additional fits) Qualitatively, is the range between these 5 additional lines with different b values larger or smaller than the range between the 95% confidence lines from the original fitted line (the one with b = 0.28 cm)?
- Direct monte carlo parameter estimation
- Bayesian MCMC fitting

[10]:
```
import numpy as np
import pandas as pd
import scipy.stats as st
import matplotlib.pyplot as plt
import scipy.io as sio

%matplotlib inline
```

Load in the provided data:

[11]:
```
data = sio.loadmat('Lyell_h_Q_sorted.mat')
```

We can convert this dictionary into a pandas dataframe and select only the columns of data that we want (ignoring the file metadata): (Even though we know, that in cases outside of the classroom, people only ignore metadata at their own peril.)

[12]:
```
df = pd.DataFrame(np.hstack((data['date_of_obs'], data['h1'], data['Qobs1'])),
                  columns=['date_of_obs','h1', 'Qobs1'])
df.head()
```

[12]:

	date_of_obs	h1	Qobs1
0	[9/26/08 16:30]	0.1805	0.07786
1	[9/19/08 0:02]	0.2197	0.071914
2	[9/10/08 21:35]	0.2406	0.143829
3	[9/10/10 17:52]	0.2407	0.168177
4	[8/20/07 18:22]	0.2565	0.243489

And make sure our date_of_obs is being interpreted as a datetime correctly (see documentation here and here):

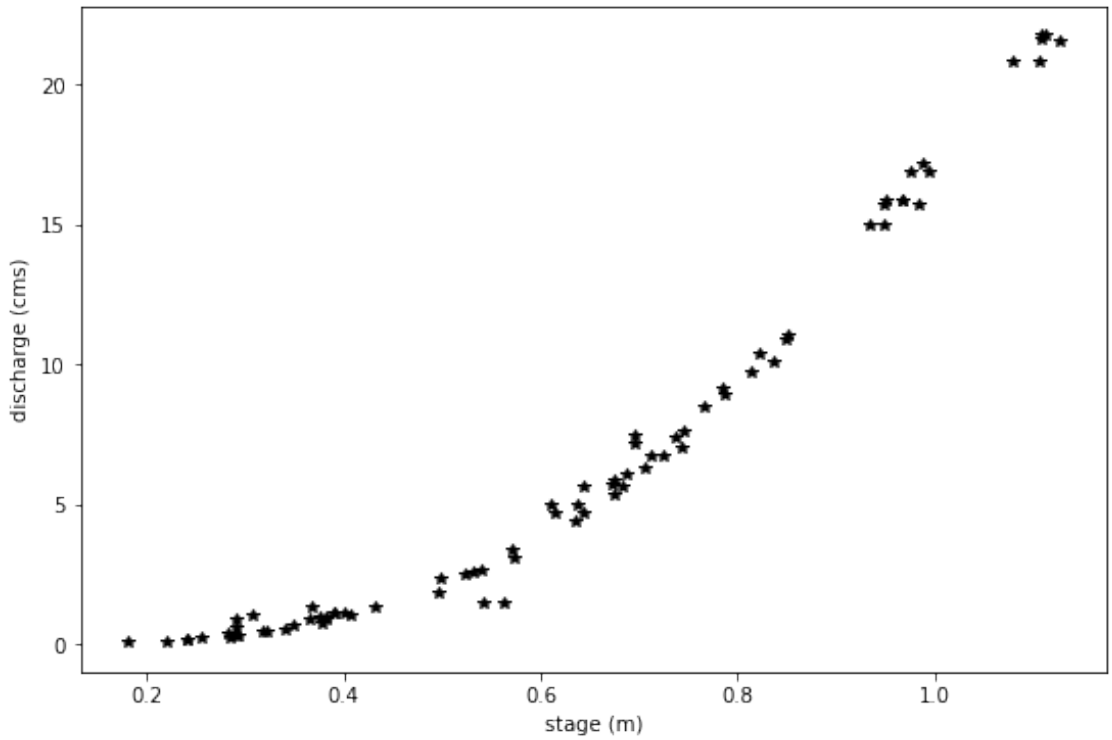
[13]:
```
df['date_of_obs'] = [pd.to_datetime(dt[0], format='%m/%d/%y %H:%M') for dt in␣
↪df['date_of_obs']]
```

Calculating rating curves: At stream gauges all over the world, we measure a timeseries of water height at a fixed point, and we generally develop an empirical rating curve to determine discharge, which is volume of flow per unit time, in units of of 3/ (cubic meters per second or cms) or 3/ (cubic feet per second of cfs).

In developing this rating curve, we are trying to solve for a, b, and c in the equation = (−)

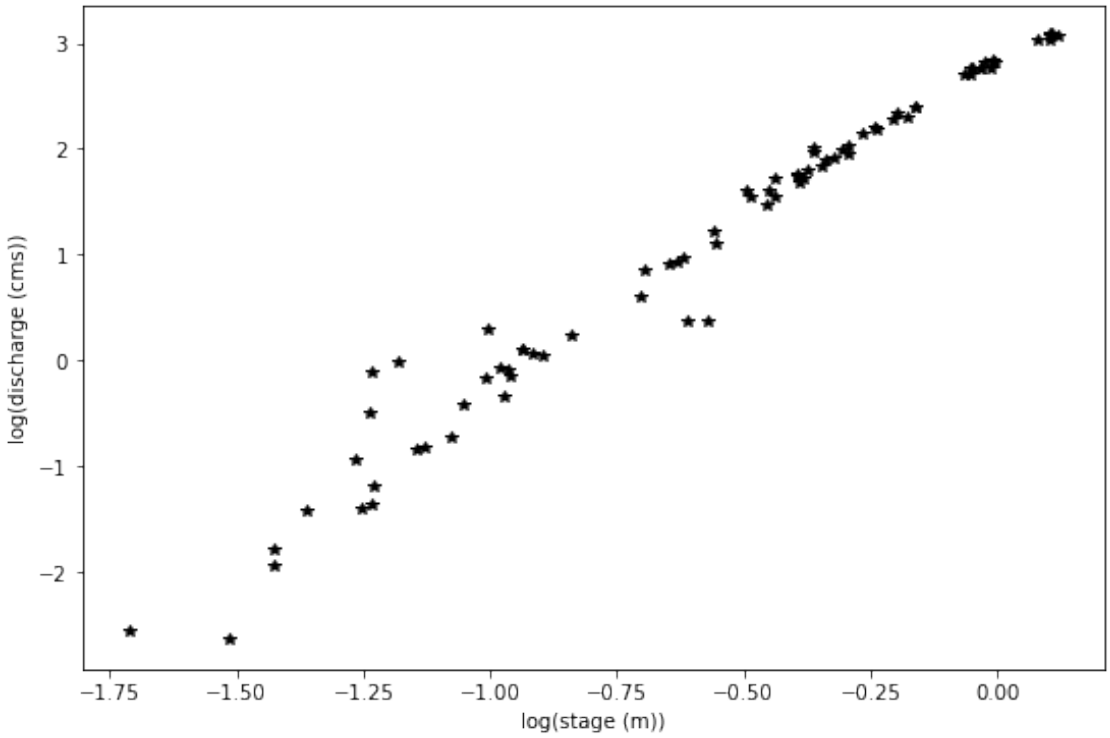
[14]:
```
# First, let's plot all of the data we just read in.
plt.figure(figsize=(9,6))

plt.xlabel('stage (m)')
plt.ylabel('discharge (cms)')
plt.plot(df.h1,df.Qobs1,'k*')
```

[14]:
```
[<matplotlib.lines.Line2D at 0x7fd95e3b1880>]
```



[15]:
```
#And plot the log transform of both variables
loght=np.log(df.h1.astype('float'))
logQ=np.log(df.Qobs1.astype('float'))

plt.figure(figsize=(9,6))

plt.xlabel('log(stage (m))')
plt.ylabel('log(discharge (cms))')
plt.plot(loght,logQ,'k*')
```

[15]: [<matplotlib.lines.Line2D at 0x7fd95e2a8310>]



PART 1:

Linear Regression of Transformed Variables:

As illustrated above, due to the nature of our channel, we would need to separate our variables and fit two lines to it. For simplicity, we will focus here on the upper portion of the rating curve, looking only at higher flows. Work by CEE M.S. student Gwyn Perry, the transition between the two slopes is about 0.54. I also want to ignore two outlier measurements right around this transition, so I choose to look at data above a stage height of 0.59. You can change this cut-off value and see how it changes the results. Ideally, we would have a survey of the location that identifies the exact stage when the bedrock control becomes dominant.

[16]:
```
h11 = 0.59
```

First, we identify the rows in our data frame that correspond to flows with h1 > h11, these correspond to data above the change in channel control.

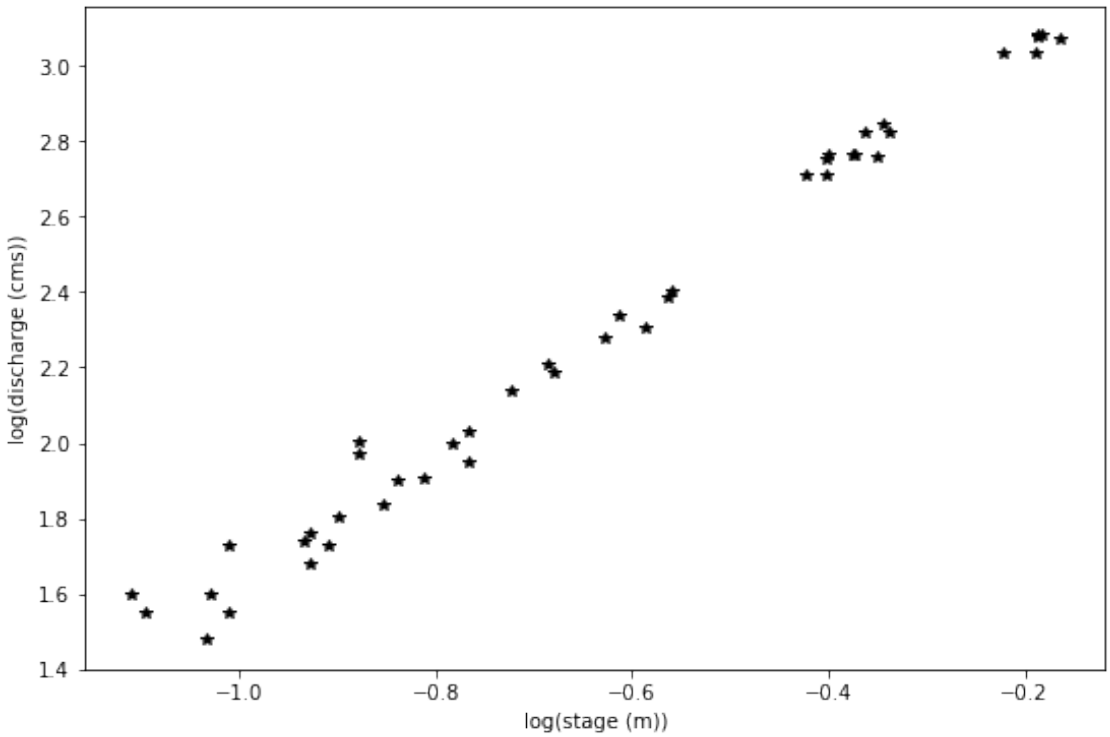
[17]:
```
Qobs_now = df.Qobs1[df.h1 > h11]
h_now = df.h1[df.h1 > h11]
```

[18]:
```
# based on field measurements, we know b must be between 10 and 50 cm,
# which is the same as 0.1 to 0.5 meters.
# We start with a guess.
b=0.28
# and we subtract this value off of the measured stream height
```

```
hobs_minusb=h_now.subtract(b)
```

[19]:
```
b=0.28
# and we subtract this value off of the measured stream height
hobs_minusb=h_now.subtract(b)
#And plot the log transform of both variables
loght=np.log(hobs_minusb.astype('float'))
# note that the above is taking the log of the observed values minus b
logQ=np.log(Qobs_now.astype('float'))

plt.figure(figsize=(9,6))

plt.xlabel('log(stage (m))')
plt.ylabel('log(discharge (cms))')
plt.plot(loght,logQ,'k*')
```

[19]: [<matplotlib.lines.Line2D at 0x7fd95a18a8e0>]



With linear regression, we can only solve for two of the three unknowns in our rating curve equation: $= (-)$ which we have log transformed to be $( ) = ( ) + * (-)$ First, we will assume that b is 0.28 m. In practice, this is often estimated from field surveys as the maximum height of water in the measuring pool when flow stops.

We then use the same code from basic linear regresssion (Lab 4.3), where our calculated slope will

be c, and our calculated intercept will be log(a).

[20]:
```
x=loght
# Note that our x value here includes the log of our measured stage minus b
 ↪(see above)
y=logQ
n = len(x)

B1 = ( n*np.sum(x*y) - np.sum(x)*np.sum(y) ) / ( n*np.sum(x**2) - np.sum(x)**2
 ↪) # B1 parameter, slope
B0 = np.mean(y) - B1*np.mean(x) # B0 parameter, y-intercept

print('B0 : {}'.format(np.round(B0,4)))
print('B1 : {}'.format(np.round(B1,4)))
```

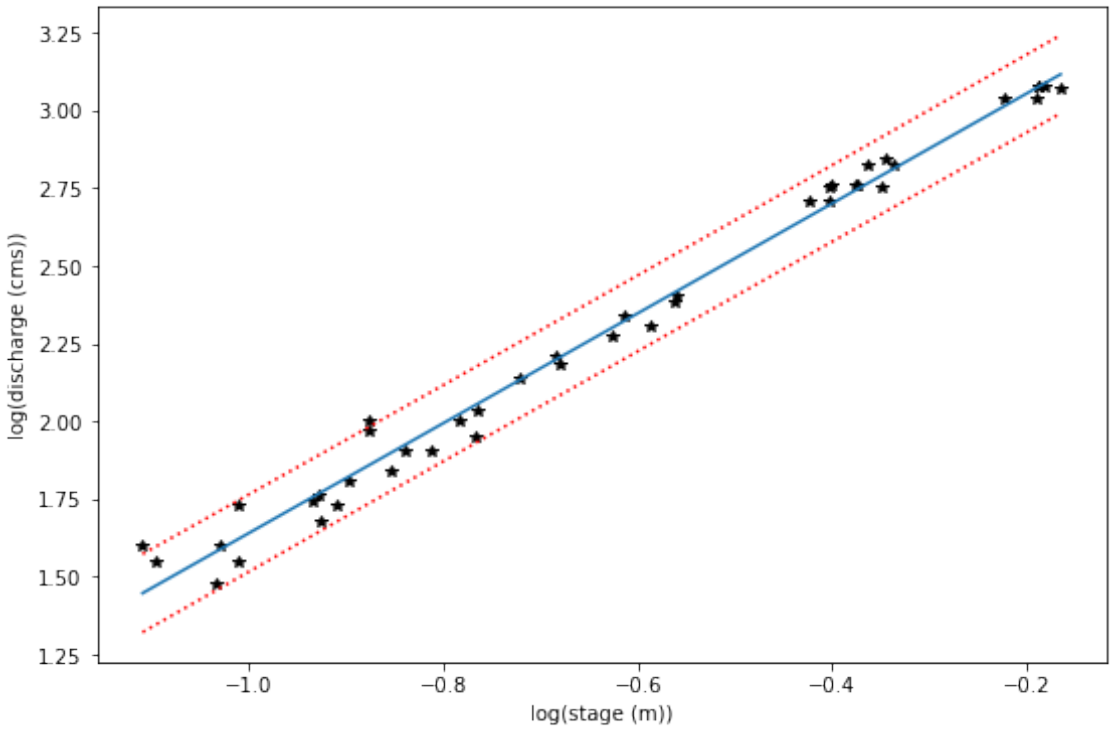
```
B0 : 3.4077
B1 : 1.7662
```

[21]:
```
# Now, how do we find 95% confidence intervals?  We do this for our estimates
 ↪of logQ
# Again, borrowing from Lab 4.3
y_predicted = B0 + B1*x
residuals = (y - y_predicted)
# sum of squared errors
sse = np.sum(residuals**2)
# standard error of regression
s = np.sqrt(sse/(n-2))

# create an array of x values
p_x = np.linspace(x.min(),x.max(),100)

# using our model parameters to predict y values
p_y = B0 + B1*p_x

# calculate the standard error of the predictions
sigma_ep = np.sqrt( s**2 * (1 + 1/n + ( ( n*(p_x-x.mean())**2 ) / ( n*np.
 ↪sum(x**2) - np.sum(x)**2 ) ) ) )

# our chosen alpha
alpha = 0.05

# compute our degrees of freedom with the length of the predicted dataset
n = len(p_x)
dof = n - 2

# get the t-value for our alpha and degrees of freedom
t = st.t.ppf(1-alpha/2, dof)
```

```
# compute the upper and lower limits at each of the p_x values
p_y_lower = p_y - t * sigma_ep
p_y_upper = p_y + t * sigma_ep
```

[22]:
```
# First let's make a plot in the log-transformed space
plt.figure(figsize=(9,6))

plt.xlabel('log(stage (m))')
plt.ylabel('log(discharge (cms))')
plt.plot(loght,logQ,'k*')
plt.plot(p_x,p_y)
plt.plot(p_x,p_y_lower,':r')
plt.plot(p_x,p_y_upper,':r')
```

[22]:
```
[<matplotlib.lines.Line2D at 0x7fd95a109df0>]
```



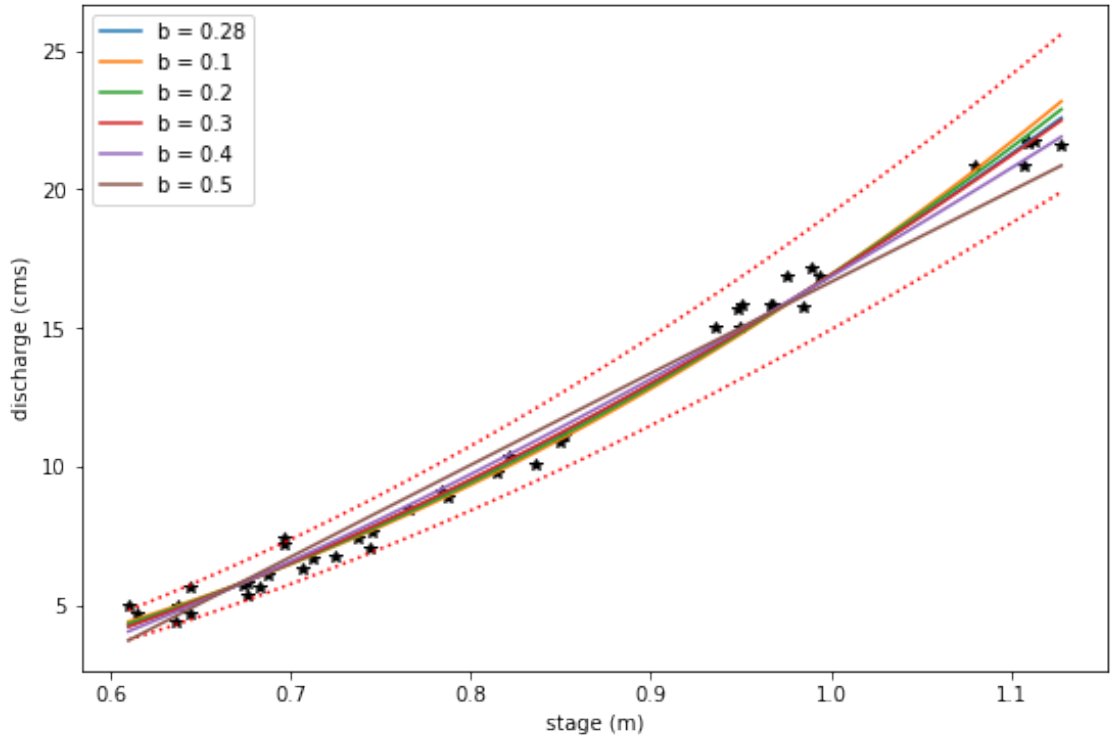
[23]:
```
# Now we transform each piece back into the original form
Q_predict=np.exp(p_y)
Q_predict_upper=np.exp(p_y_upper)
Q_predict_lower=np.exp(p_y_lower)
x_topredict=np.exp(p_x) + b
# Plot the original data and then the prediction lines
```

```
plt.figure(figsize=(9,6))

plt.xlabel('stage (m)')
plt.ylabel('discharge (cms)')
plt.plot(h_now,Qobs_now,'k*')
plt.plot(x_topredict,Q_predict,label='b = {}'.format(b))
plt.plot(x_topredict,Q_predict_lower,':r')
plt.plot(x_topredict,Q_predict_upper,':r')
for i in [0.1,0.2,0.3,0.4,0.5]:
    b=i
    print('b parameter = {}'.format(i))
    # and we subtract this value off of the measured stream height
    hobs_minusb=h_now.subtract(b)
    # And plot the log transform of both variables
    loght=np.log(hobs_minusb.astype('float'))
    # note that the above is taking the log of the observed values minus b
    logQ=np.log(Qobs_now.astype('float'))
    x=loght
    # Note that our x value here includes the log of our measured stage minus b␣
 ↪(see above)
    y=logQ
    n = len(x)

    B1 = ( n*np.sum(x*y) - np.sum(x)*np.sum(y) ) / ( n*np.sum(x**2) - np.
 ↪sum(x)**2 ) # B1 parameter, slope
    print('c parameter = {}'.format(B1))
    B0 = np.mean(y) - B1*np.mean(x) # B0 parameter, y-intercept
    print('a parameter = {}'.format(B0))
    # create an array of x values
    p_x = np.linspace(x.min(),x.max(),100)
    # using our model parameters to predict y values
    p_y = B0 + B1*p_x
    # remove log transform
    Q_predict=np.exp(p_y)
    x_topredict=np.exp(p_x) + b
    plt.plot(x_topredict,Q_predict, label='b = {}'.format(i))
plt.legend()
```

```
b parameter = 0.1
c parameter = 2.3635759709347663
a parameter = 3.0770917973279666
b parameter = 0.2
c parameter = 2.0337254298995338
a parameter = 3.281869841754415
b parameter = 0.3
c parameter = 1.6985918980552672
a parameter = 3.4330245688765815
```

```
b parameter = 0.4
c parameter = 1.3536993758545448
a parameter = 3.515431256260974
b parameter = 0.5
c parameter = 0.9860500676674817
a parameter = 3.495950195366812
```

[23]:
```
<matplotlib.legend.Legend at 0x7fd95a0ce0d0>
```



Part 2: Brute force parameter estimation

For this example, my priors are uniform. As shown in the lecture slides, I will sample the space and not use a random number generator for the first go around.

Recall that we are trying to solve for a, b, and c in the equation $= (-)$ I'll just take 10 values out of each uniform distribution.

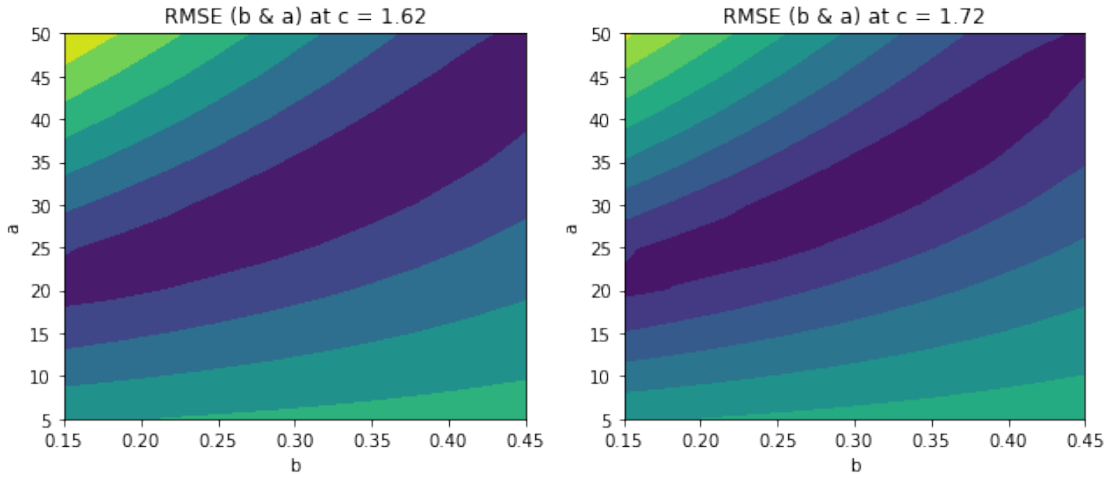
[24]:
```
nx=10
# for c, we know that at higher flows (which we've restricted our sample to), c␣
 ↪= 5/3 or 1.67
# We allow c to vary slightly around this theoretical value
c = np.linspace(1.67-0.05,1.67+0.05,nx)
# b is an empirical constant of where the upper level equation intersects 0
# b only represents the true 0 level for the lower portion of the rating curve␣
 ↪equation
```

```
# here, we guess a range of values based on visual inspection of our data
b = np.linspace(0.15,0.45,nx)
# a can be estimated as a function of channel slope and roughness (see paper by LeCoz et al.)
# for now, we will just guess a range of values that seem reasonable by visual inspection
a = np.linspace(5,50,nx)
```

[25]:
```
# Set up the arrays we'll populate with data below
Qest = np.ones((10,10,10,h_now.size)) # for estimating Q with each parameter set
Qfit = np.ones((10,10,10)) # for RMSE values
```

[26]:
```
# Iterate through all combinations of a, b, and c parameters
# Then calculate Qest, and Qfit (RMSE)

for ic in range(nx):
    for ib in range(nx):
        for ia in range(nx):
            Qest[ia,ib,ic,:] = a[ia] * (h_now-b[ib])**c[ic]
            temp = np.reshape(Qest[ia,ib,ic,:],Qobs_now.size)
            Qfit[ia,ib,ic]=np.sqrt( np.mean( (temp-Qobs_now)**2 ) ) # calculate RMSE for this parameter set
```

[27]:
```
f, ax = plt.subplots(1,2,figsize=(9,4))
ic = 0 # select the first value of c
Qfit_temp = np.reshape(Qfit[:,:,ic],[10,10]);
ax[0].contourf(b,a,Qfit_temp)
ax[0].set_xlabel('b')
ax[0].set_ylabel('a')
ax[0].set_title('RMSE (b & a) at c = {}'.format(np.round(c[ic],2)))

ic = -1 # the last value of c (using the index "-1" to represent the last value in the array)
Qfit_temp2 = np.reshape(Qfit[:,:,ic],[10,10]);
ax[1].contourf(b,a,Qfit_temp2)
ax[1].set_xlabel('b')
ax[1].set_ylabel('a')
ax[1].set_title('RMSE (b & a) at c = {}'.format(np.round(c[ic],2)))

f.tight_layout()
```

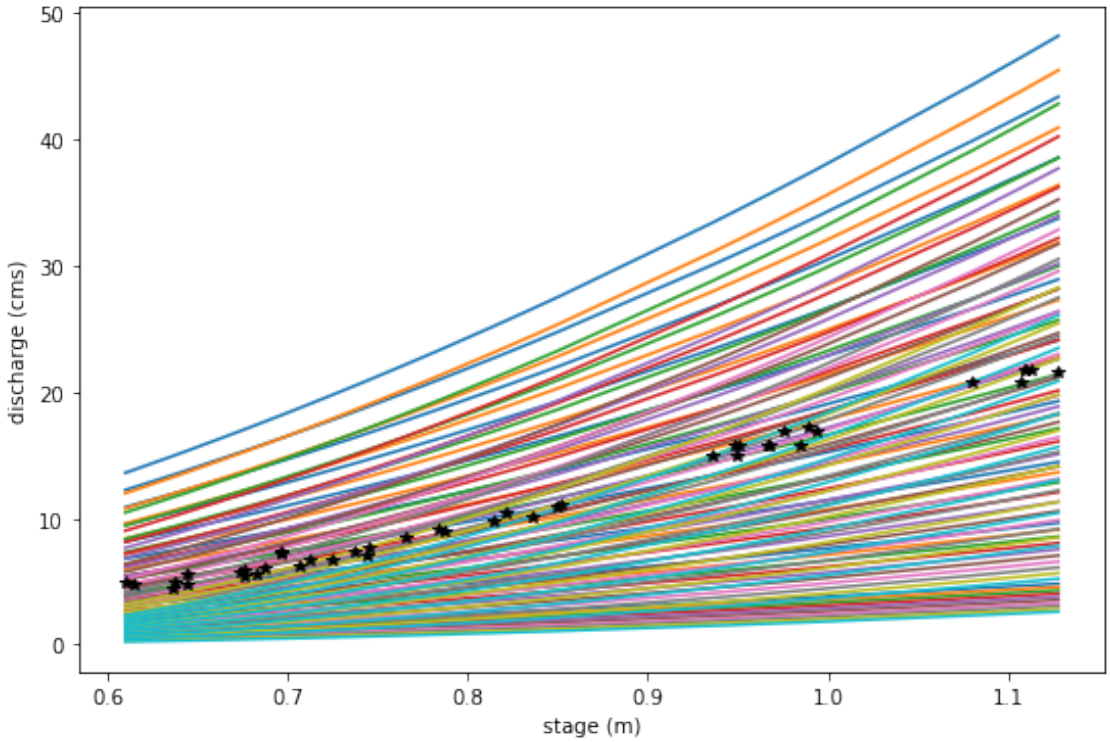


From the above, we can see that c makes a relatively slight difference, but a and b vary with each other quite a bit.

And then plot this as rating curves to see how our different parameter choices impact the spread c in estimated discharge as a function of h

[28]:
```
plt.figure(figsize=(9,6))
# Note, because c doesn't matter much, we will just choose a c value from the␣
 ↪middle

for ia in range(nx):
    for ib in range(nx):
        plt.plot(h_now,np.reshape(Qest[ia,ib,5,:],h_now.size))

plt.xlabel('stage (m)')
plt.ylabel('discharge (cms)')
plt.plot(h_now,Qobs_now,'k*')
```

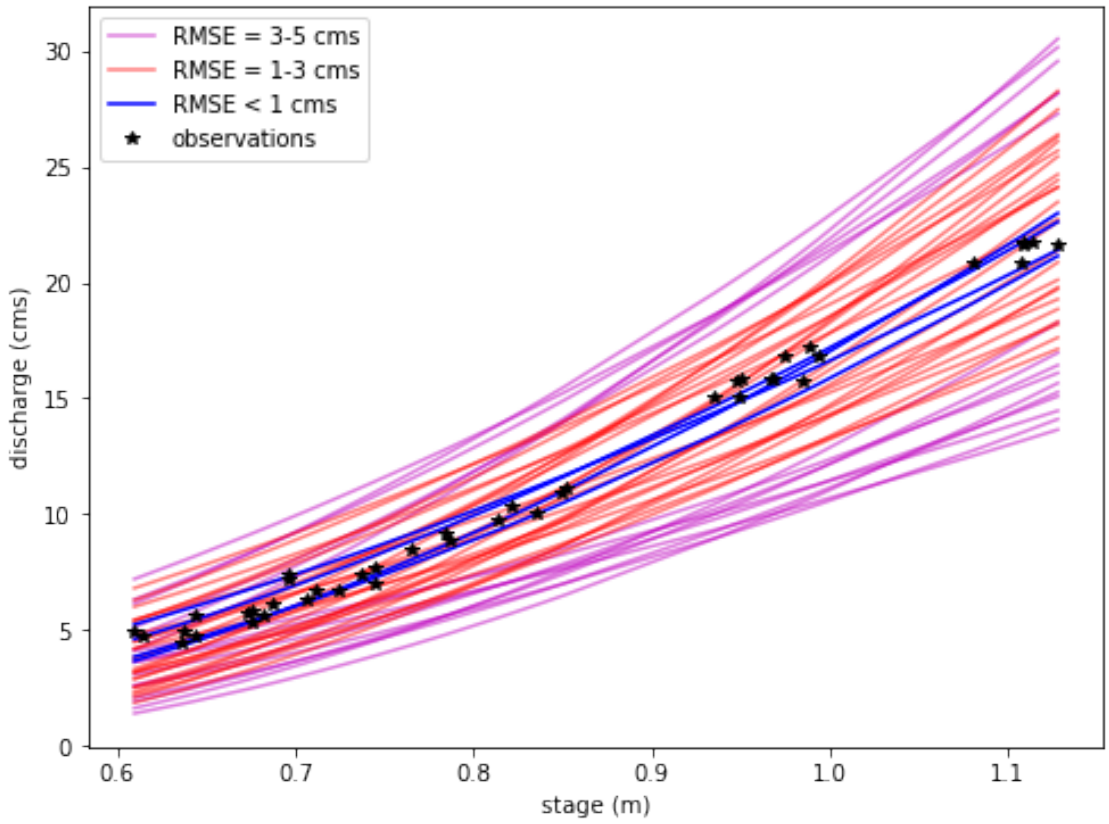
[28]:
```
[<matplotlib.lines.Line2D at 0x7fd959e20070>]
```



[29]:
```
# Reshape Qfit and Qest for a single c value to only look at variations of a␣
↪and b
ic = 5
Qfit2 = np.reshape(Qfit[:,:,ic],[10,10])
Qest2 = np.reshape(Qest[:,:,ic,:],[10,10,h_now.size])

# Find the corresponding Qest values for different ranges of RMSE
Qest_rmse1 = Qest2[Qfit2<1]                 # RMSE < 1
Qest_rmse3 = Qest2[(Qfit2>=1) & (Qfit2<3)] # 1 <= RMSE < 3
Qest_rmse5 = Qest2[(Qfit2>=3) & (Qfit2<5)] # 3 <= RMSE < 5
```

[30]:
```
plt.figure(figsize=(8,6))

# Plot the rating curves with RMSE between 3 and 5 cms
for i in range(Qest_rmse5.shape[0]):
    if i ==0:
        label='RMSE = 3-5 cms'
    else:
        label=None
    plt.plot(h_now, Qest_rmse5[i],'m',alpha=0.5,label=label)

# Plot the rating curves with RMSE between 1 and 3 cms
for i in range(Qest_rmse3.shape[0]):
```

```
    if i ==0:
        label='RMSE = 1-3 cms'
    else:
        label=None
    plt.plot(h_now, Qest_rmse3[i],'r',alpha=0.5,label=label)

# Finally plot the rating curves with RMSE < 1 cms
for i in range(Qest_rmse1.shape[0]):
    if i ==0:
        label='RMSE < 1 cms'
    else:
        label=None
    plt.plot(h_now, Qest_rmse1[i],'b',alpha=1,label=label)

plt.xlabel('stage (m)')
plt.ylabel('discharge (cms)')
plt.plot(h_now,Qobs_now,'k*',label='observations')
plt.legend();
```



Note that what we're starting to do here is to sort our rating curve parameter space into more likely and less likely parameter sets, based on how well they fit the data. However, this is imprecise, and

we cannot quantify 95% uncertainty limits in any discharge estimation.

PART 3: Use MCMC to sample the parameter space.

With uniform priors, this is very similar to the part 1 parameter estimation problem.

So, the above gives us an idea of Monte-Carlo parameter estimation but does not illustrate Bayesian or MCMC routines

I'm going to simplify to a two parameter problem, where I say that c=1.66 and that I know that (okay, not exactly true but it seemed to matter the least in this particular problem), and I'm going to start with the expected values from my priors for my initial guess.

[31]:
```
b0 = 0.2833
a0 = 25
c0 = 1.66 # and this one I won't vary
```

Now, we will ask the computer to navigate the parameter space. I will determine where to go according to how well my modeled Q fits my observed Q at all of my measurement points, with the understanding that the parameters that give me the minimum error are most likely to be true.

initial guess = likelihood = P(Q|theta), that's our model compared to our obs – only need relative sense of error, so we look at the SSE's

[32]:
```
Qest0 = a0 * (h_now-b0)**c0
SSE0 = np.sum((Qest0-Qobs_now)**2)
SSE0
```

[32]:
```
158.11032268871054
```

Now, in the bayesian sense, we have to multiply this by the liklihood of our parameters P(theta) – if I ignore normalization for now, I currently have uniform priors, so if a and b fall within my range, P0 is 1, and if either falls out of my range, P0 is 0 (not possible). (Note that this is the simplest case. If we had a normal distribution for the prior, we would multiply that by the liklihood as we did in lab 5.)

[34]:
```
# So, I set my limits of my uniform distribution
amin = 5
amax = 50
bmin = 0.15
bmax = 0.45

# for first step, I know I picked values in with the range, so
P0 = 1
```

[36]:
```
Lmc = 10000 #number of steps to run MCMC routine for.
burn_in = 1000 #estimate the number of steps before Markov Chain becomes␣
 ↪stationary
# basically, the burn-in steps will not be used in calculating final␣
 ↪probabilities because
```

```
# they depend on the initial parameter estimates.

# Initialize arrays to store results in
PthetaQ = np.ones((Lmc,1))
amc = np.zeros((Lmc,1))
bmc = np.zeros((Lmc,1))
```

And we can calculate P(theta|Q). Note that we are defining this by our goodness-of-fit statistic. Here, we use the sum of squared errors, where if it is smaller, P(theta|Q) is greater, and the choice is a better fit.

[37]:
```
PthetaQ[0] = SSE0*P0 # this is our starting point.
    # Note that this calculation is not a true probability of P(theta/Q) but a␣
 ↪relative metric
amc[0] = a0
bmc[0] = b0
# We save all the values at each step.
```

Now we start marching in MCMC space...

I need to randomly pick a new location. This is art, so I'm going to use two gaussians around my current location with sigma equal to the half-width of my prior distribution to pick a test jump location. I will use a Gibbs approach and jump in one variable followed by the other each time. (Note that a lot of the active research in MCMC code involves both how you pick your initial position and in how you pick your jumping strategy.)

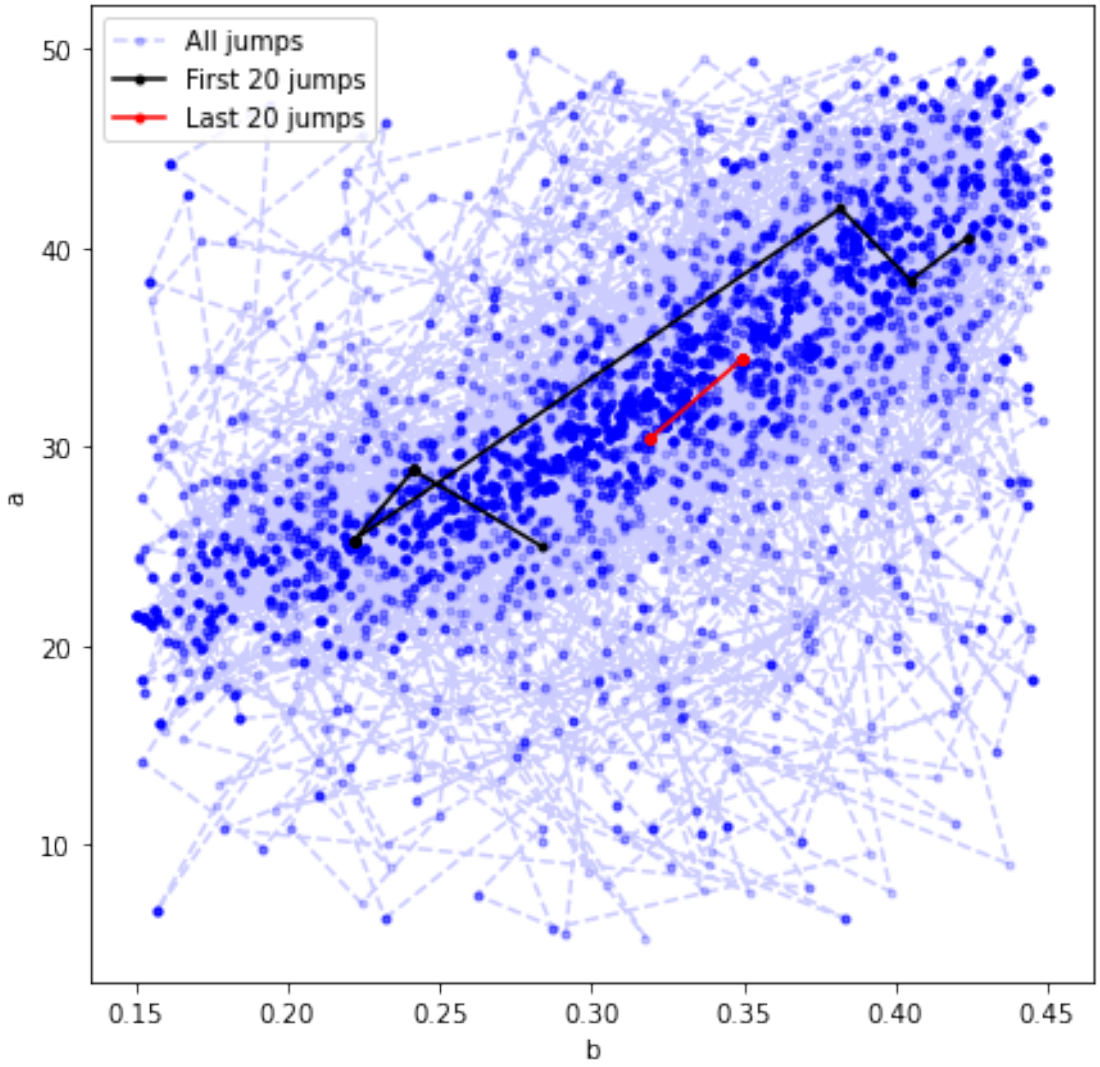
I'm going to assume that my two parameters are totally independent (probably not true – a next step would be to try a bivariate gaussian, but we stick with the simple case).

[38]:
```
for imc in range(1,Lmc):

        # first we jump in a
    newjumpa = amc[imc-1] + (amax-amin)/(10*np.random.normal())

        # Now we repeat with a jump in b, assuming it's totally independent of a
        # and require both to be within their uniform distributions.

    newjumpb = bmc[imc-1] + (bmax-bmin)/(10*np.random.normal())

    if (newjumpa >= amin) and (newjumpa <= amax) and (newjumpb >= bmin) and␣
 ↪(newjumpb <= bmax):
        #then the prior is okay, and we can proceed
        # (Note that prior makes any choice outside of our set range impossible)
        Qest1 = newjumpa*(h_now-newjumpb)**c0

        # calculate how well the parameters at our new location lead to a model␣
 ↪that matches the data
        SSE1=np.sum((Qest1-Qobs_now)**2)
```

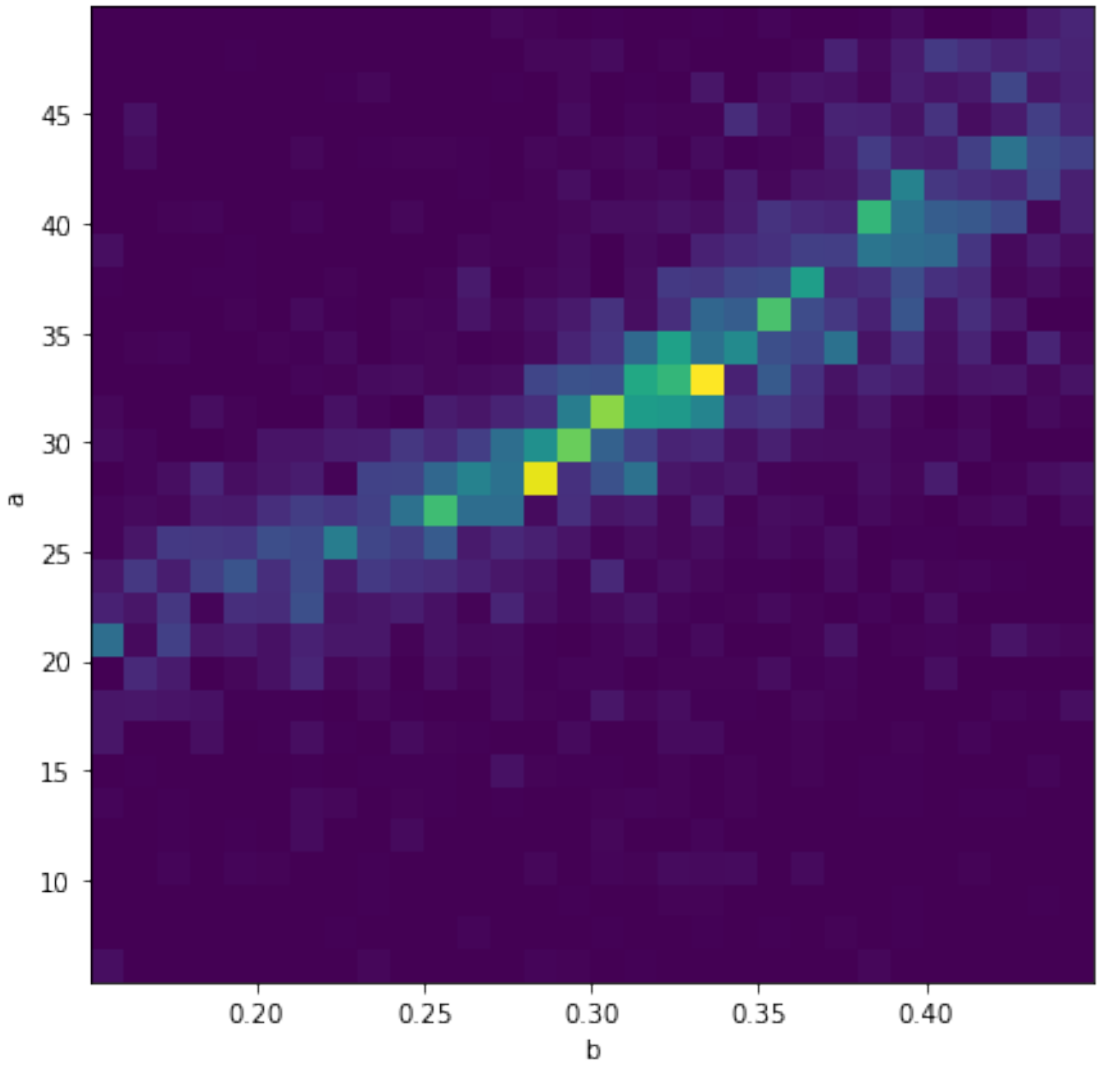
```
        # posterior is SSE1*1 (because we're within the prior uniform domain)
        # the if statement essentially says everything else is multiplied by 0

        if SSE1 < PthetaQ[imc-1]:
            # then the error is less, and we found a better place, and we␣
↪update a and b
            amc[imc] = newjumpa
            bmc[imc] = newjumpb
            PthetaQ[imc] = SSE1 # this becomes the one to beat
        else:
            jumpscore = PthetaQ[imc-1]/SSE1 # this gives a number between 0 and␣
↪1

            # This is essentially a rating of how much worse this new location␣
↪is than
            # where you are right now.  The sum of squared errors is larger,␣
↪but by how much

            # Generate random number from 0 to 1, so that we will jump with a␣
↪probability of
            # the jumpscore
            # if our new SSE is 90% as good as our current one, we jump there␣
↪90% of the time;
            # if it's 10% as good, we jump there 10% of the time
            if np.random.uniform() <= jumpscore:
                # then we go there
                amc[imc] = newjumpa
                bmc[imc] = newjumpb
                PthetaQ[imc] = SSE1 # this becomes the one to beat
            else:
                amc[imc]=amc[imc-1]
                bmc[imc]=bmc[imc-1]
                PthetaQ[imc]=PthetaQ[imc-1] # we stay where we are for another␣
↪timestep

    else:
    # you are outside of the bounds of the uniform prior, so posterior would be␣
↪0 and we don't go there
        amc[imc]=amc[imc-1]
        bmc[imc]=bmc[imc-1]
        PthetaQ[imc]=PthetaQ[imc-1] # we stay where we are
```

[39]:
```
# Visualize all jumps across the a vs b parameter space
plt.figure(figsize=(7,7))
plt.plot(bmc,amc,'b.--',alpha=0.2,label='All jumps')
```

```
# Visualize the first 20 jumps
plt.plot(bmc[:20],amc[:20],'k.-',alpha=1,label='First 20 jumps')
# Visualize the last 20 jumps
plt.plot(bmc[-20:],amc[-20:],'r.-',alpha=1,label='Last 20 jumps')
plt.xlabel('b')
plt.ylabel('a')
plt.legend();
```



[40]:
```
# "Heatmap" of all jumps across the a vs b parameter space
plt.figure(figsize=(7,7))
plt.hist2d(bmc.ravel(),amc.ravel(),30);
plt.xlabel('b')
plt.ylabel('a');
```

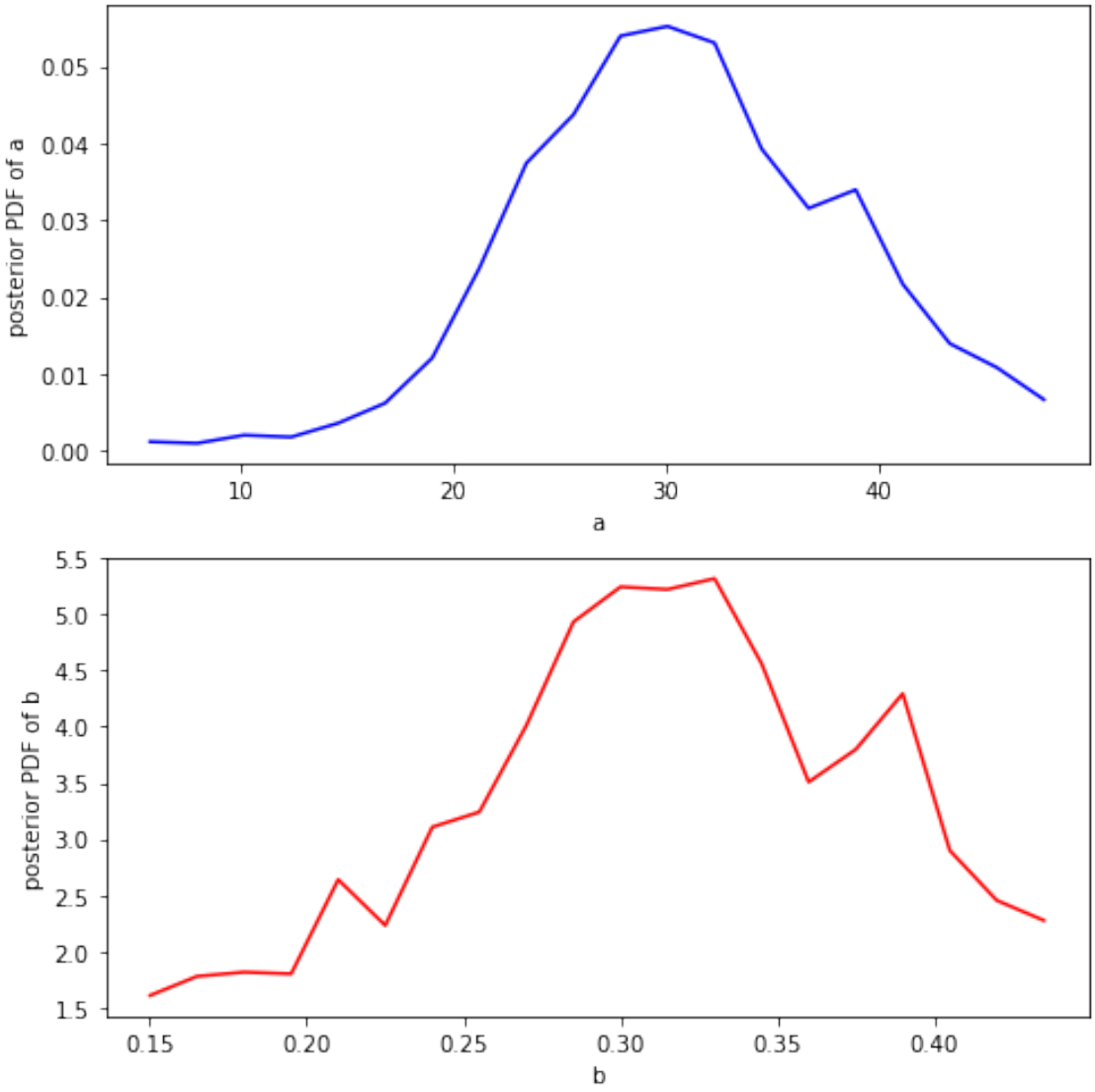


Make some PDFs

Remember that there is a burn-in time, so only calculate stats for steps after the burn_in value. (This is to avoid infuencing final statistics from the choice of the initial state.)

[41]:
```
# What are the marginal PDFs for a and b?
a = plt.hist(amc[burn_in:],20);
b = plt.hist(bmc[burn_in:],20);
plt.close();

f, ax = plt.subplots(2,1,figsize=(7,7))
# A
ax[0].plot(a[1][:-1],a[0]/(np.sum(a[0])*(a[1][1]-a[1][0])),'b')
ax[0].set_xlabel('a')
ax[0].set_ylabel('posterior PDF of a')
```

```
# B
ax[1].plot(b[1][:-1],b[0]/(np.sum(b[0])*(b[1][1]-b[1][0])),'r')
ax[1].set_xlabel('b')
ax[1].set_ylabel('posterior PDF of b')
f.tight_layout()
```



What are the 95% confidence intervals for the predicted Q?

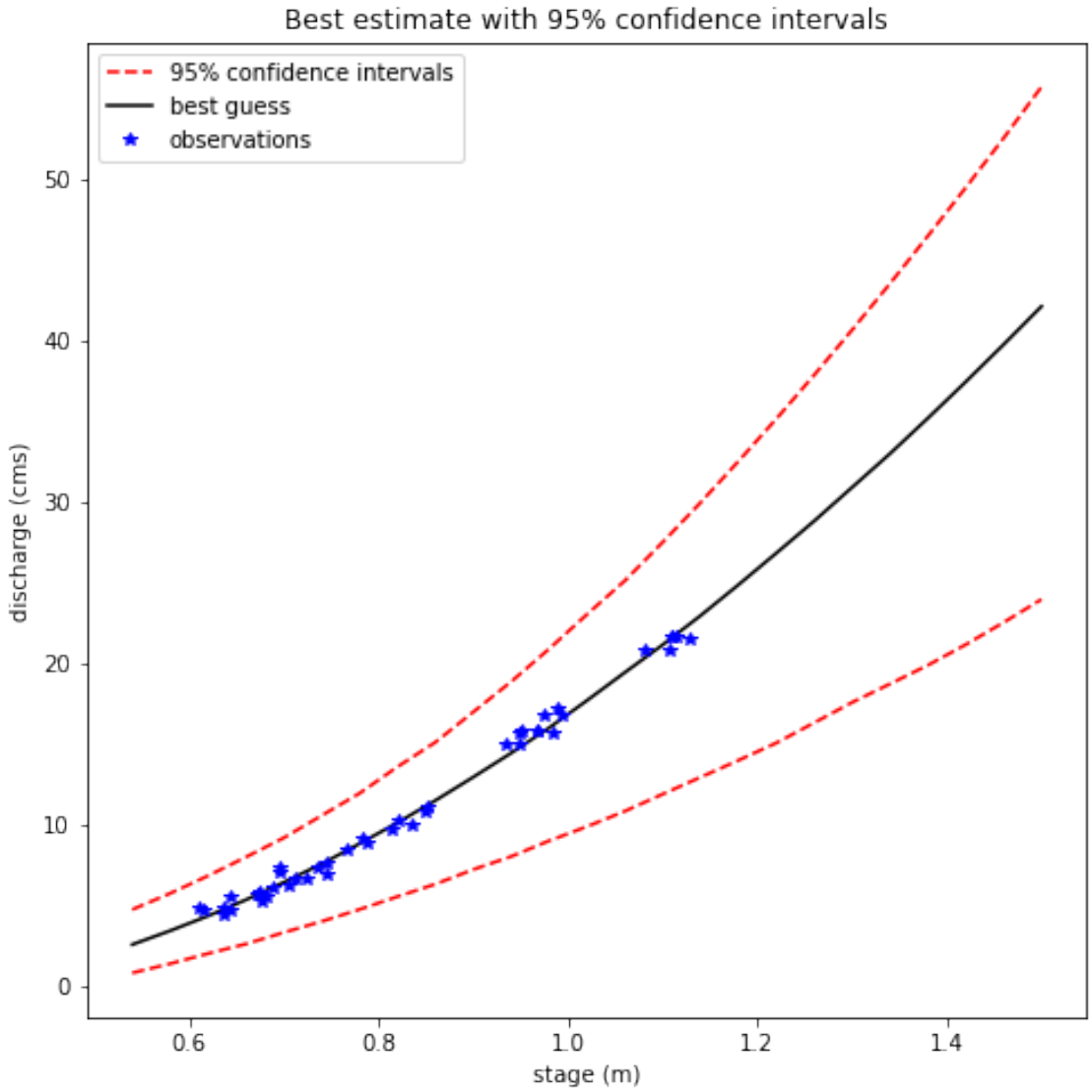
Note that by virtue of how we set up the jumping rules, the "jumper" spent more time in the good-fitting parameter space than in more poorly-fitting parameter space. Even better, it spent exactly as much time in the more poorly-fitting space as measured by how much worse it performed. By this structure, all calculated values of Q after the burn-in time are distributed by their liklihood of being right.

[42]:
```
# evenly space 25 stage values (could have more if doesn't look smooth)
hh=np.linspace(0.54,1.5,25)

# Make an empirical CDF for all calculated values of Q using a and b after the␣
 ↪burn-in time
# and plot the median and 95% confidence values from the Markov Chain generated␣
 ↪probabilities
NN=amc[burn_in:].size
```

[45]:
```
Q025 = np.ones((hh.shape))
Q50 = np.ones((hh.shape))
Q975 = np.ones((hh.shape))

for ih in range (hh.size):
    Qhh = amc[burn_in:]*(hh[ih]-bmc[burn_in:])**c0
    xsort = np.sort(Qhh[:,0])
    ranks = np.array(range(NN))+1
    x_CDF = 1 - (ranks - 0.4)/(NN + 0.2)

    Q025[ih] = np.interp(0.025,1-x_CDF,xsort)
    Q975[ih] = np.interp(0.975,1-x_CDF,xsort)
    Q50[ih] = np.interp(0.5,1-x_CDF,xsort)
```

[46]:
```
plt.figure(figsize=(8,8))
plt.plot(hh,Q025,'r--',label='95% confidence intervals')

plt.plot(hh,Q975,'r--')
plt.plot(hh,Q50,'k-',label='best guess')
plt.xlabel('stage (m)')
plt.ylabel('discharge (cms)')
plt.plot(h_now,Qobs_now,'b*',label='observations')
plt.legend()
plt.title('Best estimate with 95% confidence intervals')
```

[46]:
```
Text(0.5, 1.0, 'Best estimate with 95% confidence intervals')
```



1.2.1 Response:

The results of these three plots really communicate three different things. I will go through what each method shows and highlight the differences between each: - Linear Regression: This linear regression technique assigns a mix of b-values (also known as h_o) to calculate stage and computes the values of a and c to use as parameters within the linear regression. The regression line, and thus the parameters, changes with each b-value. The 95% confidence intervals would also be different for each value of b that is chosen. This makes these plots very busy and difficult to read if this method were to be used. - Brute Force Parameter Estimation: The brute force method assumes we know one parameter rather well (c), but the other two parameters are provided as a distribution (in this case it is uniform) and estimates of flow rate are computed for each parameter. However, there is no inherent intelligence to this process, thus the same amount of computing power is spent on calculating these values for impossible, or near impossible, results. This can make this an extremely

expensive method to use when large amounts of data need to be analyzed. Howeverm in this case, there is not enough data to make it problematic. Additionally, this method can provide many results, but does not provide meaningful statistical results (like confidence intervals). Thus, in order to find what results are best, user specified thresholds need to be set to separate from "good" and "bad" results, as is done in the above example with <1 cfs RMSE being deemed as a good result. - MCMC Approach: The MCMC approach is similar to the combination of the two above approaches: it provides results over the entire parameter space and also can provide meaningful confidence intervals. This method, like brute force, samples the parameter space, but does so intelligently. Given certain "jump rules", the MCMC approach will move through the parameter space but do so in a way that preferentially samples more probabilistically favorable parameter locations, meaning that obtaining results is much less computationally expensive and statistically meaningful. Thus, a single set of 95% confidence intervals can be presented to reperesent the confidence in the calculation of the entire parameter space, which is quite powerful in a data visualization sense.