Homework 6

Markov Chain Simulation and Hierarchical Model

This notebook is arranged in cells. Texts are usually written in the markdown cells, and here you can use html tags (make it bold, italic, colored, etc). You can double click on this cell to see the formatting.

The ellipsis (...) are provided where you are expected to write your solution but feel free to change the template (not over much) in case this style is not to your taste.

Hit "Shift-Enter" on a code cell to evaluate it. Double click a Markdown cell to edit.

Link Okpy

In [ ]:

```
from client.api.notebook import Notebook
ok = Notebook('hw6.ok')
_ = ok.auth(inline = True)
```

Imports

In [1]:

```
import numpy as np
from scipy.integrate import quad
#For plotting
import matplotlib.pyplot as plt
%matplotlib inline
```

Problem 1 - Simulated Annealing

Reference: Newman, Computational Physics (p. 490-497)

For a physical system in equilibrium at temperature T, the probability that at any moment the system is in a state i is given by the Boltzmann probability. Let us assume our system has single unique ground state and let us choose our energy scale so that $E_i = 0$ in the ground state and $E_i > 0$ for all other states. Now suppose we cool down the system to absolute zero. The system will definitely be in the ground state, and consequently one way to find the ground state of the system is to cool it down to $T = 0$.

This in turn suggests a computational strategy for finding the ground state: let us simulate the system at temperature T, using the Markov chain Monte Carlo method, then lower the temperature to zero and the system should find its way to the ground state. This same approach could be used to find the minimum of any function, not just the energy of a physical system. we can take any mathematical function $f(x, y, z, \dots)$ and treat the independent variables x, y, z as defining a "state" of the system and f as being the energy of that system, then perform a Monte Carlo simulation. Taking the temperature down to zero will again cause the system to fall into its ground state, i.e. the state with the lowest value of f, and hence we find the minimum of the function.

However, if the system is cooled rapidly, it can get stuck in a local energy minimum. On the other hand, an annealed system, one that is cooled sufficiently slowly, can find its way to the ground state. Simulated annleaing applies the same idea in a computational setting. It mimics the slow cooling of a material on the computer by using a Monte Carlo simulation with a temperature parameter that is gradually lowered from an initially high value towards zero. The initial temperature should be chosen so that the system equilibrates quickly. To achieve this, we should choose the thermal energy to be significantly greater than the typical energy change accompanying a single Monte Carlo move.

As for the rate of cooling, one typically specifies a "cooling schedule," a trajectory for the temperature as a function of time, and the most common choise is the exponential one:

$$T = T_0 e^{-t/\tau}$$

where T_0 is the initial temperature, and τ is a time constant. Some trial error may be necessary to find a good value for τ.

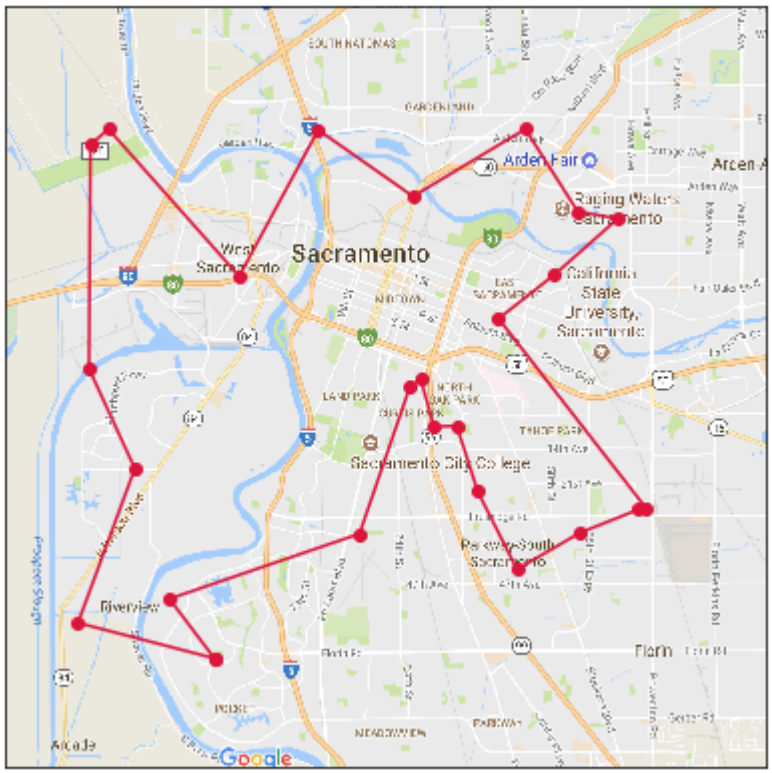
As an example of the use of simulated annealing, we will look at one of the most famous optimization problems, traveling salesman problem, which involves finding the shortest route that visits a given set of locations on a map. A salesman wishes to visit N given cities, and we assume that he can travel in a straight line between any pair of citiies. Given the coordinates of the cities, the problem is to devise the shortest tour. It should start and end at the same city, and all cities must be visited at least once. Let us denote the position of the city i by the two-dimensional vector $r_i = (x_i, y_i)$.

Here is the solution:

In [2]:

```
# Traveling salesman (Newman p. 493)
from math import sqrt,exp
from numpy import empty
from random import random,randrange
from scipy.misc import imread

N = 25
R = 0.02
Tmax = 10.0
Tmin = 1e-3
tau = 1e4

# Function to calculate the magnitude of a vector
def mag(x):
    return sqrt(x[0]**2+x[1]**2)

# Function to calculate the total length of the tour
def distance():
    s = 0.0
    for i in range(N):
        s += mag(r[i+1]-r[i])
    return s

# Choose N city locations and calculate the initial distance
r = empty([N+1,2],float)
for i in range(N):
    r[i,0] = random()
    r[i,1] = random()
r[N] = r[0]
D = distance()

# Main loop
t = 0
T = Tmax
while T>Tmin:

    # Cooling
    t += 1
    T = Tmax*exp(-t/tau)

    # Choose two cities to swap and make sure they are distinct
    i,j = randrange(1,N),randrange(1,N)
    while i==j:
        i,j = randrange(1,N),randrange(1,N)

    # Swap them and calculate the change in distance
    oldD = D
    r[i,0],r[j,0] = r[j,0],r[i,0]
    r[i,1],r[j,1] = r[j,1],r[i,1]
    D = distance()
    deltaD = D - oldD

    # If the move is rejected, swap them back again
    if random()>exp(-deltaD/T):
        r[i,0],r[j,0] = r[j,0],r[i,0]
        r[i,1],r[j,1] = r[j,1],r[i,1]
        D = oldD

plt.figure(figsize = (8, 7))
```

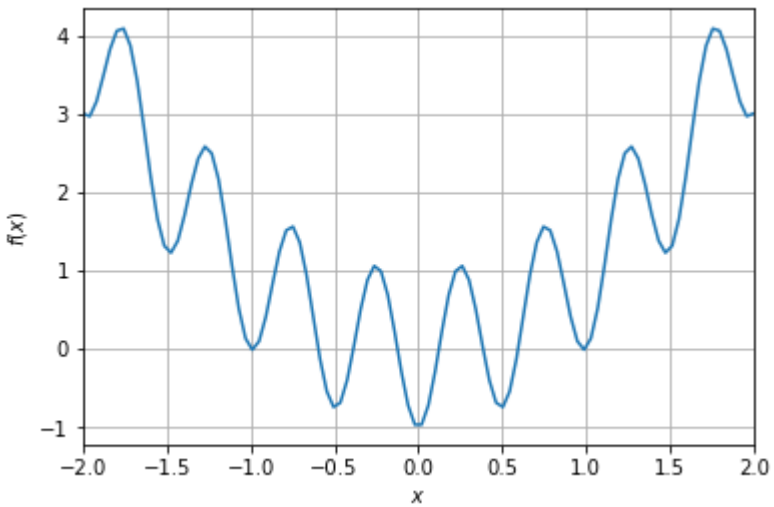
```
img = imread("map_sacramento.png")
plt.plot(r[:,0], r[:,1], 'o-', color = 'crimson', zorder=1)
plt.imshow(img,zorder=0, extent=[-0.1, 1.1, -0.1, 1.1])
plt.xticks([])
plt.yticks([])
plt.show()
```



Now, consider the function $f(x) = x^2 - \cos(4\pi x)$, which looks like this:

In [3]:

```
x = np.linspace(-2, 2, 100)
y = x**2 - np.cos(4*np.pi*x)
plt.plot(x, y)
plt.grid(True); plt.xlim(-2, 2); plt.xlabel('$x$'); plt.ylabel('$f(x)$')
plt.show()
```



Clearly the global minimum of this function is at $x = 0$.

1. Write a program to confirm this fact using simulated annealing starting at, say, $x = 2$, with Monte Carlo moves of the form $x \to x + \delta$ where δ is a random number drawn from a Gaussian distribution with mean zero and standard deviation one. Use an exponential cooling schedule and adjust the start and end temperatures, as well as the exponential constant, until you find values that give good answers in reasonable time. Have your program make a plot of the values of x as a function of time during the run and have it print out the final value of x at the end. You will find the plot easier to interpret if you make it using dots rather than lines, with a statement of the form plot(x,".") or similar.

In [ ]:

```
...
```

2. Now adapt your program to find the minimum of the more complicated function $f(x) = \cos(x) + \cos(\sqrt{2}x) + \cos(\sqrt{3}x)$ in the range $0 < x < 50$.

(Hint: The correct answer is around $x = 16$, but there are also competing minima around $x = 2$ and $x = 42$ that your program might find. In real-world situations, it is often good enough to find any reasonable solution to a problem, not necessarily the absolute best, so the fact that the program sometimes settles on these other solutions is not necessarily a bad thing.)

In [ ]:

```
...
```

Problem 2 - Hierarchial Normal Model

Reference: Gelman et al., Bayesian Data Analysis (p. 288-290)

Diet	Measurements
A	62, 60, 63, 59
B	63, 67, 71, 64, 65, 66
C	68, 66, 71, 67, 68, 68
D	56, 62, 60, 61, 63, 64, 63, 59

Table 1. Coagulation time in seconds for blood drawn from 24 animals randomly allocated to four different diets. Different treatments have different numbers of observations because the randomization was unrestricted.

Under the hierarchical normal model, data y_{ij}, for $i = 1, \ldots, n_j$ and $j = 1, \ldots, J$, are independently normally distributed within each of J groups, with means θ_j and common variance σ^2. The data is presented in Table 1. (In this case, there are $J = 4$ groups (or 4 sets of experiments - A, B, C, and D), and for each group j, we have a data vector y_j with the mean θ_j; $y_j = [y_{1j}, \ldots, y_{n_j j}]$ (there have been n_j observations made.) (e.g. j = 1 represents the diet A group. So $y_{i1} = [y_{11}, y_{21}, y_{31}, y_{41}] = [62, 60, 63, 59]$ with $n_1 = 4$.

The total number of observations is $n = \sum_{j=1}^{J} n_j$. The group means (θ_j) are assumed to follow a normal distribution with unknown mean μ and variance τ^2, and a uniform prior distribution is assumed for $(\mu, \log\sigma, \tau)$, with $\sigma > 0$ and $\tau > 0$; equivalently, $p(\mu, \log\sigma, \log\tau) \propto \tau$.

The joint posterior density of all the parameters is

$$p(\theta, \mu, \log\sigma, \log\tau \mid y) \propto p(\mu, \log\sigma, \log\tau) \prod_{j=1}^{J} \text{Normal}(\theta_j \mid \mu, \tau^2) \prod_{j=1}^{J} \prod_{i=1}^{n_j} \text{Normal}(y_{ij} \mid \theta_j, \sigma^2)$$

where $\text{Normal}(\theta_j \mid \mu, \tau^2) = \frac{1}{\sqrt{2\pi\tau^2}} \exp(-\frac{(\theta_j - \mu)^2}{2\tau^2})$.

1. Now, find the MAP (Maximum A Posteriori) solution to this (find the solution to MAP for all these parameters). In other words, find $\theta_j, \mu, \sigma, \tau$ which maximizes the likelihood.

(Hint: The likelihood is given as $\prod_{j=1}^{J} \text{Normal}(\theta_j \mid \mu, \tau^2) \prod_{j=1}^{J} \prod_{i=1}^{n_j} \text{Normal}(y_{ij} \mid \theta_j, \sigma^2)$. Take the log of the likelihood and maximize it using scipy.optimize.fmin (https://docs.scipy.org/doc/scipy-0.19.1/reference/generated/scipy.optimize.fmin.html (https://docs.scipy.org/doc/scipy-0.19.1/reference/generated/scipy.optimize.fmin.html)). Note that you need to make initial guesses on the parameters in order to use fmin. Make a reasonable guess! You can use a different in-built function to maximize the likelihood function.
Caveat: "fmin" minimizes a given function, so you should multiply the log-likelihood by -1 in order to maximize it using fmin.)

In [ ]:

```
# Load data
A = np.array([62, 60, 63, 59])
B = np.array([63, 67, 71, 64, 65, 66])
C = np.array([68, 66, 71, 67, 68, 68])
D = np.array([56, 62, 60, 61, 63, 64, 63, 59])

data = []
data.append(A)
data.append(B)
data.append(C)
data.append(D)

data = np.array(data)
```

In [ ]:

```
from scipy import optimize

def minus_log_likelihood(param, y_i1=data[0], y_i2=data[1], y_i3=data[2], y_i4=d
ata[3]):
    theta1, theta2, theta3, theta4, mu, sigma, tau = param
    return ...
```

In [ ]:

```
...
```

You should find that the MAP solution is dependent on your initial guesses. The point is that the maximal likelihood estimator is biased, even though we have all the parameters. Hence, it is better to use the Monte Carlo simulation for the parameter estimation; we can also determine posterior quantiles with the Monte Carlo method. First, we will try the **Gibbs sampler**.

Starting points:
In this example, we can choose overdispersed starting points for each parameter θ_j by simply taking random points from the data y_{ij} from group j. We obtain 10 starting points for the simulations by drawing θ_j independently in this way for each group. We also need starting points for μ, which can be taken as the average of the starting θ_j values. No starting values are needed for τ or σ as they can be drawn as the first steps in the Gibbs sampler.

Conditional posterior distribution of σ^2:
The conditional posterior density for σ^2 has the form corresponding to a normal variance with known mean; there are n observations y_{ij} with means θ_j. The conditional posterior distribution is

$$\sigma^2|\theta,\mu,\tau,y \sim \text{Inv-}\chi^2(n,\hat{\sigma}^2)$$

where

$$\text{Inv-}\chi^2(x|n,\hat{\sigma}^2) = \text{Inv-gamma}\left(\alpha=\frac{n}{2},\beta=\frac{n}{2}\hat{\sigma}^2\right) = \frac{\beta^\alpha}{\Gamma(\alpha)}x^{-(\alpha+1)}\exp(-\beta/x)$$

$$\hat{\sigma}^2 = \frac{1}{n}\sum_{j=1}^{J}\sum_{i=1}^{n_j}(y_{ij}-\theta_j)^2$$

(Hint: You can take random samples from the inverse gamma function using scipy.stats.invgamma - https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.stats.invgamma.html (https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.stats.invgamma.html). invgamma.rvs(alpha, scale = beta, size=1) will take one random sample from $\text{Inv-gamma}(\alpha,\beta)$.)

Conditional posterior distribution of τ^2:
Conditional on y and the other parameters in the model, μ has a normal distribution determined by the J values θ_j:

$$\tau^2|\theta,\mu,\sigma,y \sim \text{Inv-}\chi^2(J-1,\hat{\tau}^2)$$

with

$$\hat{\tau}^2 = \frac{1}{J-1}\sum_{j=1}^{J}(\theta_j-\mu)^2.$$

Conditional posterior distribution of each θ_j:
The factors in the joint posterior density that involve θ_j are the $N(\mu,\tau^2)$ prior distribution and the normal likelihood from the data in the jth group, y_{ij} , $i=1,\ldots,n_j$. The conditional posterior distribution of each θ_j given the other parameters in the model is

$$\theta_j|\mu,\sigma,\tau,y \sim \text{Normal}(\hat{\theta_j},V_{\theta_j})$$

where the parameters of the conditional posterior distribution depend on μ,σ,τ as well as y:

$$\hat{\theta_j} = \frac{\frac{1}{\tau^2}\mu + \frac{n_j}{\sigma^2}(\frac{1}{n_j}\sum_{i=1}^{n_j} y_{ij})}{\frac{1}{\tau^2} + \frac{n_j}{\sigma^2}}$$

$$V_{\theta_j} = \frac{1}{\frac{1}{\tau^2} + \frac{n_j}{\sigma^2}}$$

These conditional distributions are independent; thus drawing the θ_j's one at a time is equivalent to drawing the vector θ all at once from its conditional posterior distribution.

Conditional posterior distribution of μ:
Conditional on y and the other parameters in the model, μ has a normal distribution determined by the J values θ_j:

$$\mu|\theta, \sigma, \tau, y \sim \text{Normal}(\hat{\mu}, \tau^2/J)$$

where $\hat{\mu} = \frac{1}{J}\sum_{j=1}^{J} \theta_j$.

2. Define a function which does the Gibbs sampling. Take 100 samples. Remove the first 50 sequences and store the latter half. Repeat this 10 times so that you get ten Gibbs sampler sequences, each of length 50. We have 7 parameters ($\theta_1, \ldots, \theta_4, \mu, \sigma, \tau$), and for each parameter, you created 10 chains, each of length 50.

In [ ]:

```
...
```

3. Estimate posterior quantiles. Find 2.5%, 25%, 50%, 75%, 97.5% posterior percentiles of all parameters. (Hint: You can use np.percentile - https://docs.scipy.org/doc/numpy-dev/reference/generated/numpy.percentile.html (https://docs.scipy.org/doc/numpy-dev/reference/generated/numpy.percentile.html).)

In [ ]:

```
...
```

4. Now, test for convergence using "Gelman-Rubin statistic." For all seven parameters, compute R and determine if the condition $R < 1.1$ is satisfied.

For a given parameter θ, the R statistic compares the variance across chains with the variance within a chain.
Given chains $J = 1, \ldots, m$, each of length n,
Let $B = \frac{n}{m-1} \sum_j \left(\bar{\theta}_j - \bar{\theta}\right)^2$, where $\bar{\theta}_j$ is the average θ for chain j and $\bar{\theta}$ is the global average. This is proportional to the variance of the individual-chain averages for θ.
Let $W = \frac{1}{m} \sum_j s_j^2$, where s_j^2 is the estimated variance of θ within chain j. This is the average of the individual-chain variances for θ.
Let $V = \frac{n-1}{n} W + \frac{1}{n} B$. This is an estimate for the overall variance of θ.

Finally, $R = \sqrt{\frac{V}{W}}$. We'd like to see $R \approx 1$ (e.g. $R < 1.1$ is often used). Note that this calculation can also be used to track convergence of combinations of parameters, or anything else derived from them.

In [ ]:

```
...
```

Now, try the **Metropolis algorithm**.

5. Run ten parallel sequences of Metropolis algorithm simulations using the package "emcee" (http://dfm.io/emcee/current/). First, define the log of prior (already given to you), likelihood, and posterior (Hint: http://dfm.io/emcee/current/user/line/)

In [ ]:

```
import emcee
```

In [ ]:

```
def log_prior(param):
    theta1, theta2, theta3, theta4, mu, sigma, tau = param
    if sigma > 0 and tau > 0:
        return 0.0
    return -np.inf

def log_likelihood(param, data0, data1, data2, data3):
    theta1, theta2, theta3, theta4, mu, sigma, tau = param
    return ...

def log_posterior(param, data0, data1, data2, data3):
    return ...
```

6. Now, try different number of MCMC walkers and burn-in period, and number of MCMC steps. At which point do you obtain similar results to those obtained using Gibbs sampling? Run the MCMC chain and estimate posterior quantiles as in Part 3.

In [ ]:

```
emcee_trace = []
for i in range(10):
    # Here we'll set up the computation. emcee combines multiple "walkers",
    # each of which is its own MCMC chain. The number of trace results will
    # be nwalkers * nsteps

    ndim = 7  # number of parameters in the model
    nwalkers = 50  # number of MCMC walkers
    nburn = 500 # "burn-in" period to let chains stabilize
    nsteps = 1000  # number of MCMC steps to take

    # set theta near the maximum likelihood, with
    np.random.seed(0)
    starting_guesses = np.random.random((nwalkers, ndim))

    # Here's the function call where all the work happens:
    # we'll time it using IPython's %time magic

    sampler = emcee.EnsembleSampler(nwalkers, ndim, log_posterior, args=[data[0
], data[1], data[2], data[3]])
    sampler.run_mcmc(starting_guesses, nsteps)

    emcee_trace.append(sampler.chain[:, nburn:, :].reshape(-1, ndim).T)

emcee_trace = np.array(emcee_trace)
```

In [ ]:

```
np.shape(emcee_trace)
```

In [ ]:

```
...
```

Using the package "corner," you can also easily plot the 1-d and 2-d posterior (looks familiar?). Make a plot for one chain. Plots along the diagonal correspond to 1-d constraints. The dotted lines show 16%, 50%, and 84% percentile ranges.

In [ ]:

```
import corner
fig = corner.corner(emcee_trace[0, :, :].T, labels=["$\\theta_1$", "$\\theta_2$"
, "$\\theta_3$", "$\\theta_4$", "$\mu$", "$\sigma$", "$\\tau$"], quantiles=[0.16
, 0.5, 0.84], range = 0.95*np.ones(7))
```

6. Test for convergence using Gelman-Rubin statistic as in Part 4.

In [ ]:

```
...
```

7. Using autocorrelation_plot from pandas (https://pandas.pydata.org/pandas-docs/stable/visualization.html#visualization-autocorrelation), plot the auto-correlation of six parameters and determine that it gets small for large lag.

In [ ]:

```
from pandas.tools.plotting import autocorrelation_plot
```

In [ ]:

```
...
```

7. Using the package "daft", plot a graphical model in this problem.
Note that we have J experiments each with n_j data, each its own mean θ_j, but common variance σ. The mean θ_j has a hyperprior, generated as a gaussian with some mean μ and variance τ.
(Hint: https://github.com/KIPAC/StatisticalMethods/blob/8232a7b7e870b82088fe3589b8a796430e9076d6/examples

In [ ]:

```
import daft
from matplotlib import rc
```

In [ ]:

```
...
```

Problem 3 - Mixture Model for Outliers

Suppose we have data that can be fit to a linear regression, apart from a few outlier points. It is always better to understand the underlying generative model of outliers.

Consider the following dataset, relating the observed variables x and y, and the error of y stored in σ_y.

We'll propose a simple linear model, which has a slope and an intercept encoded in a parameter vector θ. The model is defined as follows:

$$\hat{y}(x \mid \theta) = \theta_0 + \theta_1 x$$

Given this model, we can compute a Gaussian likelihood for each point:

$$p(x_i, y_i, e_i \mid \theta) \propto \exp\left[-\frac{1}{2e_i^2}\left(y_i - \hat{y}(x_i \mid \theta)\right)^2\right]$$

The total likelihood is the product of all the individual likelihoods. Computing this and taking the log, we have:

$$\log \mathcal{L}(D \mid \theta) = \text{const} - \sum_i \frac{1}{2e_i^2}\left(y_i - \hat{y}(x_i \mid \theta)\right)^2$$

This should all look pretty familiar if you read through the previous post. This final expression is the log-likelihood of the data given the model, which can be maximized to find the θ corresponding to the maximum-likelihood model. Equivalently, we can minimize the summation term, which is known as the *loss*:

$$\text{loss} = \sum_i \frac{1}{2e_i^2}\left(y_i - \hat{y}(x_i \mid \theta)\right)^2$$

This loss expression is known as a *squared loss*; here we've simply shown that the squared loss can be derived from the Gaussian log likelihood.

In [ ]:

```
# Load the data
x = np.array([ 0,  3,  9, 14, 15, 19, 20, 21, 30, 35,
              40, 41, 42, 43, 54, 56, 67, 69, 72, 88])
y = np.array([33, 68, 34, 34, 37, 71, 37, 44, 48, 49,
              53, 49, 50, 48, 56, 60, 61, 63, 44, 71])
e = np.array([ 3.6, 3.9, 2.6, 3.4, 3.8, 3.8, 2.2, 2.1, 2.3, 3.8,
               2.2, 2.8, 3.9, 3.1, 3.4, 2.6, 3.4, 3.7, 2.0, 3.5])
```

1. Determine $\theta = [\theta_0, \theta_1]$ which maximize the likelihood (or, equivalently, minimize the loss). As in Problem 2-1, you can use scipy.optimize.fmin. Plot the best-fit line (on top of data points) using θ from the MAP solution.

In [ ]:

```
from scipy import optimize

...
```

Clearly, we get a poor fit to the data because the squared loss is overly sensitive to outliers.

> The Bayesian approach to accounting for outliers generally involves *modifying the model* so that the outliers are accounted for. For this data, it is abundantly clear that a simple straight line is not a good fit to our data. So let's propose a more complicated model that has the flexibility to account for outliers. One option is to choose a mixture between a signal and a background:
>
> $$p(\{x_i\}, \{y_i\}, \{e_i\} \mid \theta, \{g_i\}, \sigma, \sigma_b) = \frac{g_i}{\sqrt{2\pi e_i^2}} \exp\left[\frac{-(\hat{y}(x_i \mid \theta) - y_i)^2}{2e_i^2}\right] + \frac{1-g_i}{\sqrt{2\pi\sigma_B^2}} \exp\left[\frac{-(\hat{y}(x_i \mid \theta) - y_i)^2}{2\sigma_B^2}\right]$$
>
> What we've done is expanded our model with some nuisance parameters: $\{g_i\}$ is a series of weights which range from 0 to 1 and encode for each point i the degree to which it fits the model. $g_i = 0$ indicates an outlier, in which case a Gaussian of width σ_B is used in the computation of the likelihood. This σ_B can also be a nuisance parameter, or its value can be set at a sufficiently high number, say 50.
>
> Our model is much more complicated now: it has 22 parameters rather than 2, but the majority of these can be considered nuisance parameters, which can be marginalized-out in the end, just as we marginalized (integrated) over p in the Billiard example. Let's construct a function which implements this likelihood. As in the previous post, we'll use the emcee package to explore the parameter space.

2. As in Problem2-Part5, define log-prior (already given to you), log-likelihood and log-posterior.

In [ ]:

```
def log_prior(theta):
    #g_i needs to be between 0 and 1
    if (all(theta[2:] > 0) and all(theta[2:] < 1)):
        return 0
    else:
        return -np.inf  # recall log(0) = -inf

def log_likelihood(theta, x, y, e, sigma_B):
    ...

def log_posterior(theta, x, y, e, sigma_B):
    return ...
```

Now, run the MCMC samples.

In [ ]:

```
ndim = 2 + len(x)  # number of parameters in the model
nwalkers = 50  # number of MCMC walkers
nburn = 10000  # "burn-in" period to let chains stabilize
nsteps = 15000  # number of MCMC steps to take

# set theta near the maximum likelihood, with
np.random.seed(0)
starting_guesses = np.zeros((nwalkers, ndim))
starting_guesses[:, :2] = np.random.normal(theta1, 1, (nwalkers, 2))
starting_guesses[:, 2:] = np.random.normal(0.5, 0.1, (nwalkers, ndim - 2))

sampler = emcee.EnsembleSampler(nwalkers, ndim, log_posterior, args=[x, y, e, 50
])
sampler.run_mcmc(starting_guesses, nsteps)

sample = sampler.chain  # shape = (nwalkers, nsteps, ndim)
sample = sampler.chain[:, nburn:, :].reshape(-1, ndim)
```

Once we have these samples, we can exploit a very nice property of the Markov chains. Because their distribution models the posterior, we can integrate out (i.e. marginalize) over nuisance parameters simply by ignoring them!

We can look at the (marginalized) distribution of slopes and intercepts by examining the first two columns of the sample:

In [ ]:

```
plt.plot(sample[:, 0], sample[:, 1], ',k', alpha=0.1)
plt.xlabel('intercept')
plt.ylabel('slope')
plt.show()
```

We allowed the model to have a nuisance parameter $0 < g_i < 1$ for each data point: $g_i = 0$ indicates an outlier. We can also allow sb to be a nuisance parameter to marginalize over (or just make it a large number). Now, let us define an outlier whenever posterior $E(g_i) < 0.5$.

3. Using such cutoff at $g = 0.5$, identify an outlier and mark them on the plot. Also, plot the marginalized best model over the original data.

In [ ]:

```
...
```

To Submit

Execute the following cell to submit. If you make changes, execute the cell again to resubmit the final copy of the notebook, they do not get updated automatically.
We recommend that all the above cells should be executed (their output visible) in the notebook at the time of submission.
Only the final submission before the deadline will be graded.

In [ ]:

```
_ = ok.submit()
```