(https://colab.research.google.com/github/christianhbye/bayesian-analysis/blob/main/homeworks/HW5_288.ipynb)

Homework 5

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.

Imports

In [1]:
```
import numpy as np
from scipy.integrate import quad
#For plotting
import matplotlib.pyplot as plt
%matplotlib inline
```

Mounting Google Drive locally

Mount your Google Drive on your runtime using an authorization code.

Note: When using the 'Mount Drive' button in the file browser, no authentication codes are necessary for notebooks that have only been edited by the current user.

In [2]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

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 $r_i = (x_i, y_i)$.

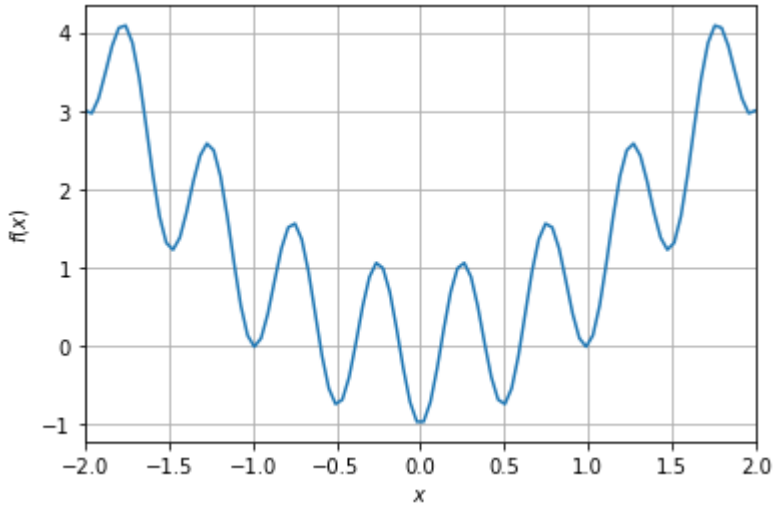
Here is the solution:

In [3]:
```
# Traveling salesman (Newman p. 493)
from math import sqrt,exp
from numpy import empty
from random import random,randrange
from imageio 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("/content/drive/My Drive/P188_288/P188_288_HW5/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 [4]:
```
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.

The necessary number of steps, N, given $T_{\min}$, $T_{\max}$ and τ can be computed as:

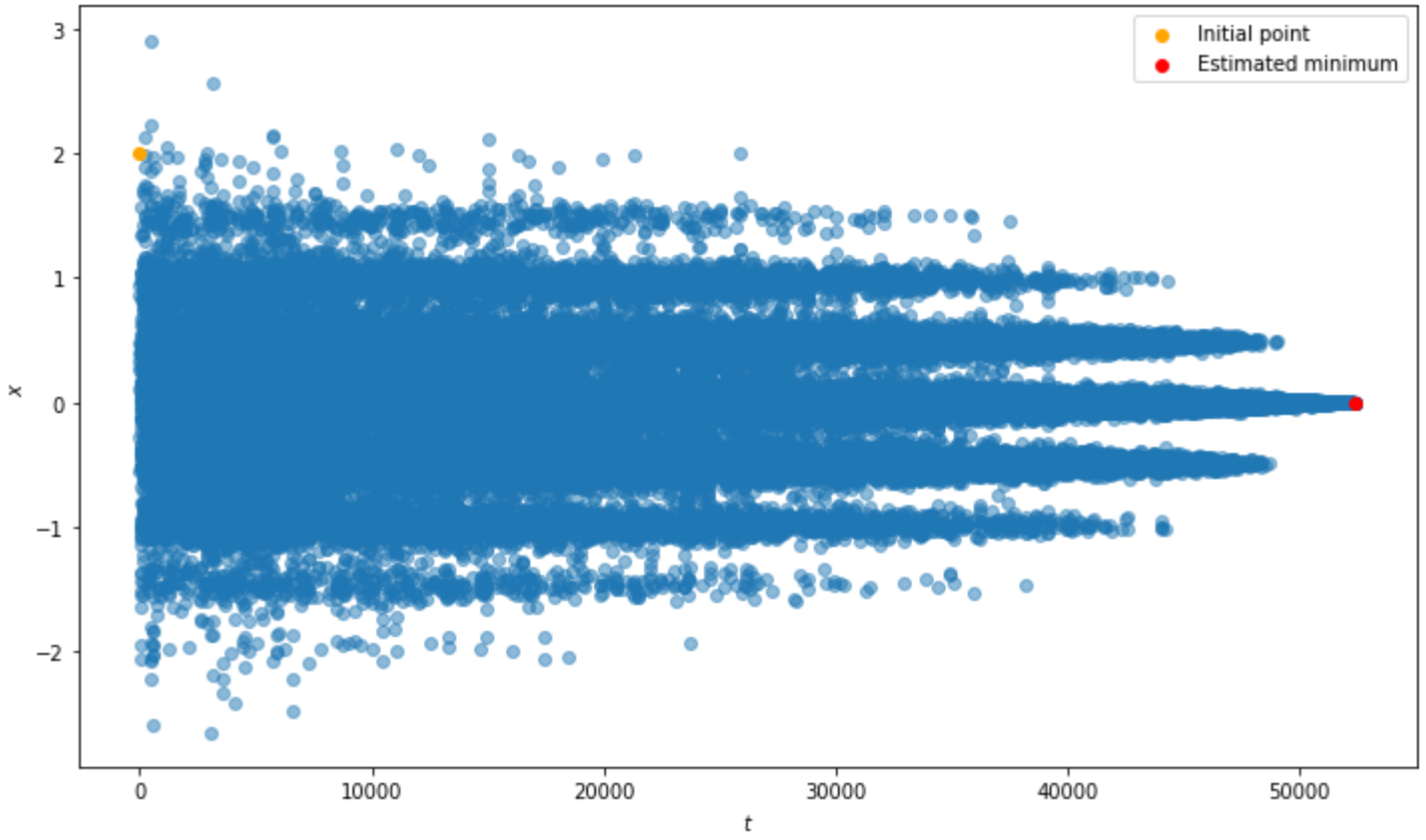
$$T_{\min} \geq T_{\max} e^{-N/\tau}$$
$$\frac{N}{\tau} \geq \log \frac{T_{\max}}{T_{\min}}$$
$$N \geq \tau \log \frac{T_{\max}}{T_{\min}}$$

In [14]:
```
def sim_anneal(f, x0, Tmax, Tmin, tau):
  """
  Simulate annealing of function f given an initial point x0, maximum and
  minimum temepratures, and the time constant tau
  """
  nsteps = np.ceil(tau * np.log(Tmax/Tmin)).astype(int)
  print(f"Number of steps: {nsteps}.")
  # initialize temeprature and time
  T = Tmax
  t = 0
  x = x0
  xvals = [x0]
  y = f(x0)
  for t in range(1, nsteps):
    if t % (nsteps//10) == 0:
      print(f"{t}/{nsteps}")
    T = Tmax * np.exp(-t/tau)
    delta = np.random.normal()
    xnew = x + delta
    ynew = f(xnew)
    dy = ynew - y
    if np.random.uniform() <= np.exp(-dy/T):  # accept step
      y = ynew
      x = xnew
      xvals.append(x)

  print("\nEstimated minimum:")
  print(f"x = {x:.3f}, y = {y:.3f}")
  return np.array(xvals)
```

In [19]:
```
def f(x):
  return x**2 - np.cos(4*np.pi*x)

xvals = sim_anneal(f, 2, 1, 1e-4, 1e5)
```

```
Number of steps: 921035.
92103/921035
184206/921035
276309/921035
368412/921035
460515/921035
552618/921035
644721/921035
736824/921035
828927/921035
921030/921035

Estimated minimum:
x = 0.000, y = -1.000
```

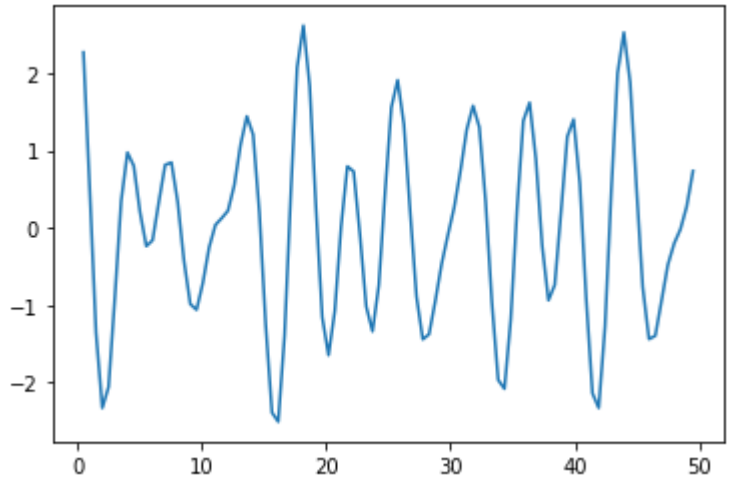
In [20]:
```
tvals = np.arange(xvals.size)
plt.figure(figsize=(12, 7))
plt.scatter(tvals[0], xvals[0], c="orange", label="Initial point", zorder=10)
plt.scatter(tvals[1:-1], xvals[1:-1], alpha=.5)
plt.scatter(tvals[-1], xvals[-1], c="red", label="Estimated minimum")
plt.legend()
plt.xlabel("$t$")
plt.ylabel("$x$")
plt.show()
```



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.)

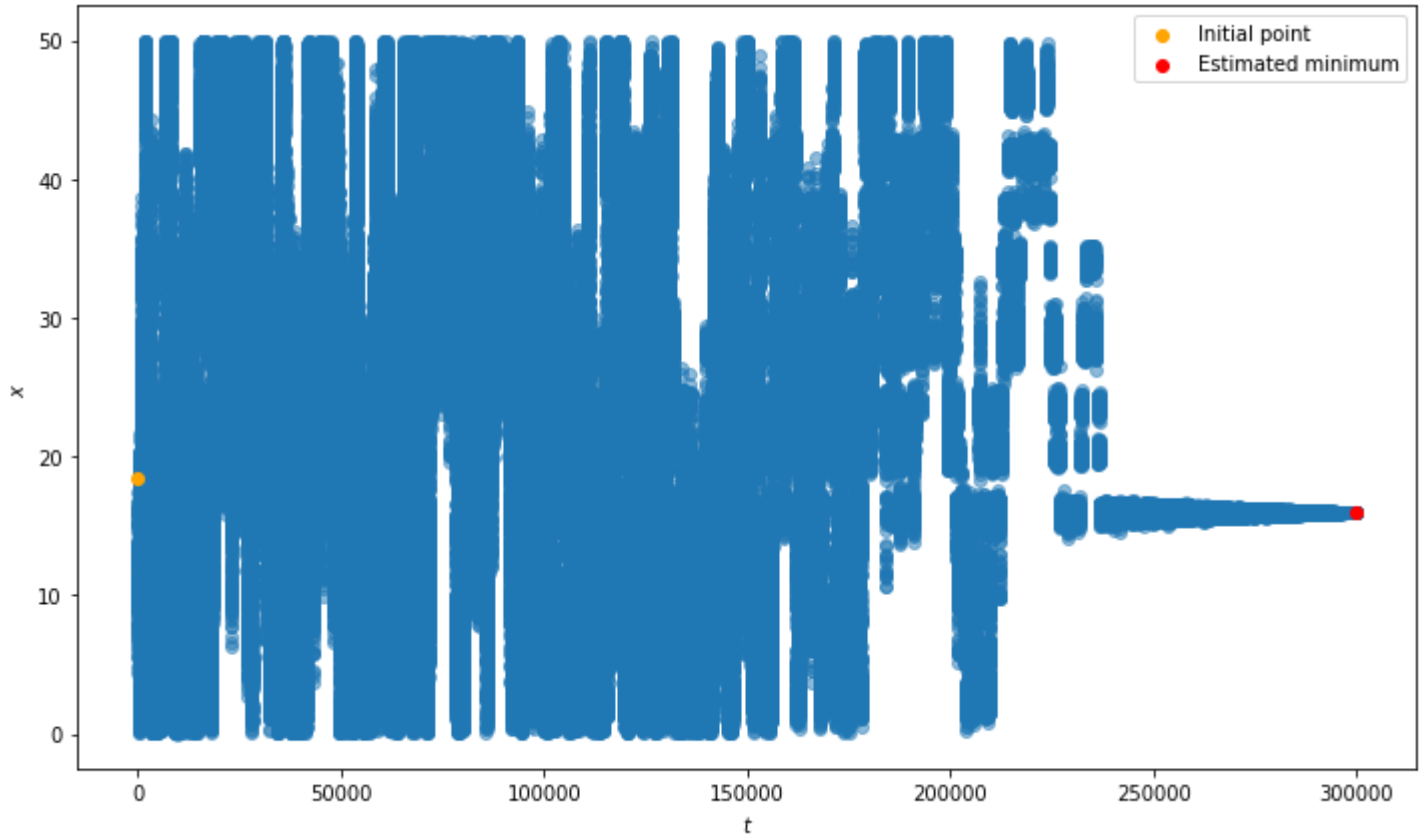
In [21]:
```
def f(x):
  if x <= 0 or x >= 50:
    return np.inf  # guarantees that it is not accepted
  return np.cos(x) + np.cos(np.sqrt(2)*x) + np.cos(np.sqrt(3)*x)

xgrid = np.linspace(0, 50, num=100)[1:-1]
ygrid = np.cos(xgrid) + np.cos(np.sqrt(2)*xgrid) + np.cos(np.sqrt(3)*xgrid)
plt.figure()
plt.plot(xgrid, ygrid)
plt.show()
```



In [30]:
```
x0 = np.random.uniform(0, 50)
print(f"Starting point: x = {x0}")
xvals = sim_anneal(f, x0, 10, 1e-4, 1e5)
```

```
Starting point: x = 18.364739230854767
Number of steps: 1151293.
115129/1151293
230258/1151293
345387/1151293
460516/1151293
575645/1151293
690774/1151293
805903/1151293
921032/1151293
1036161/1151293
1151290/1151293

Estimated minimum:
x = 15.963, y = -2.613
```

In [31]:
```
tvals = np.arange(xvals.size)
plt.figure(figsize=(12, 7))
plt.scatter(tvals[0], xvals[0], c="orange", label="Initial point", zorder=10)
plt.scatter(tvals[1:-1], xvals[1:-1], alpha=.5)
plt.scatter(tvals[-1], xvals[-1], c="red", label="Estimated minimum")
plt.legend()
plt.xlabel("$t$")
plt.ylabel("$x$")
plt.show()
```



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 [32]:
```
# 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)
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:13: VisibleDeprecationWarning: C
  del sys.path[0]
```

In [33]:
```
from scipy import optimize

def minus_log_normal(x, mean, sigma):
  """
  Negative log of normal distribution
  """
  log_amplitude = 1/2 * (np.log(2) + np.log(np.pi)) + np.log(sigma)
  log_exp = (x-mean)**2 / (2 * sigma**2)
  return log_amplitude + log_exp

def minus_log_likelihood(param, y_i1=data[0], y_i2=data[1], y_i3=data[2], y_i4=data[3]):
    theta1, theta2, theta3, theta4, mu, sigma, tau = param
    # likelihood of means
    log_lh = minus_log_normal(theta1, mu, tau)
    log_lh += minus_log_normal(theta2, mu, tau)
    log_lh += minus_log_normal(theta3, mu, tau)
    log_lh += minus_log_normal(theta4, mu, tau)
    # likelihood of datasets
    log_lh += minus_log_normal(y_i1, theta1, sigma).sum()
    log_lh += minus_log_normal(y_i2, theta2, sigma).sum()
    log_lh += minus_log_normal(y_i3, theta3, sigma).sum()
    log_lh += minus_log_normal(y_i4, theta4, sigma).sum()
    return log_lh
```

In [34]:
```
# set initial parameters based on the sample mean and variances
theta_init = [d.mean() for d in data]
sigma_init = np.mean([np.std(d) for d in data])
x0 = [*theta_init, np.mean(theta_init), sigma_init, np.std(theta_init)]
map = optimize.fmin(minus_log_likelihood, x0=x0)
```

```
Optimization terminated successfully.
         Current function value: 62.526371
         Iterations: 238
         Function evaluations: 385
```

In [35]:
```
params = [f"theta_{i+1}" for i in range(4)] + ["mu", "sigma", "tau"]

print("MAP:")
for i in range(len(params)):
  print(f"{params[i]} = {map[i]:.4f}")
```

```
MAP:
theta_1 = 61.4008
theta_2 = 65.8159
theta_3 = 67.6305
theta_4 = 61.2147
mu = 64.0155
sigma = 2.1798
tau = 2.7835
```

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 [36]:
```
from scipy.stats import invgamma

class Gibbs:

  def __init__(
      self, data=data, nparams=7, nchains=10, nsamples=100, burn_in=0.5
  ):
    self.data = data
    self.J = len(self.data)  # number of datasets
    self.nj = [d.size for d in data]  # observations per dataset
    self.n = np.sum(self.nj)  # total number of observations
    self.nparams = nparams
    self.nchains = nchains
    self.nsamples = nsamples
    self.burn_in = burn_in

  def _init_params(self):
    """
    Draw initial paramaters
    """
    theta_init = [np.random.choice(d) for d in self.data]
    mu_init = np.mean(theta_init)
    return theta_init, mu_init

  def _update_params(self, theta, mu):
    """
    Use the conditional posteriors to update each parameter once
    """
    # update sigma
    sigma_hat_sq = 0
    for j in range(self.J):
      sigma_hat_sq += np.sum((self.data[j] - theta[j])**2)
    sigma_hat_sq /= self.n
    alpha = self.n / 2
    beta = self.n / 2 * sigma_hat_sq
    sigma_sq = invgamma.rvs(alpha, scale=beta)

    # update tau
    tau_hat_sq = 1 / (self.J - 1) * np.sum((theta - mu)**2)
    alpha = (self.J - 1) / 2
    beta = (self.J - 1) / 2 * tau_hat_sq
    tau_sq = invgamma.rvs(alpha, scale=beta)

    # update theta
    V_theta = 1 / (1 / tau_sq + self.nj / sigma_sq)
    theta_hat = mu / tau_sq + np.array([d.sum() for d in self.data]) / sigma_sq
```

```
    theta_hat *= V_theta
    theta = np.random.normal(loc=theta_hat, scale=np.sqrt(V_theta))

    # update mu
    mu_hat = theta.mean()
    mu = np.random.normal(loc=mu_hat, scale=np.sqrt(tau_sq/self.J))

    return theta, mu, sigma_sq, tau_sq

  def run_sampler(self):
    """
    Sample a number of Gibbs chains
    """
    chain = np.empty((self.nparams, self.nchains, self.nsamples))
    for i in range(self.nchains):
      theta, mu = self._init_params()
      for j in range(self.nsamples):
        new_params = self._update_params(theta, mu)
        theta = new_params[0]
        mu = new_params[1]
        sigma = np.sqrt(new_params[2])
        tau = np.sqrt(new_params[3])
        chain[:, i, j] = [*theta.ravel(), mu, sigma, tau]

    N_burn = int(self.burn_in * self.nsamples)
    return chain[:, :, N_burn:]
```

In [37]:
```
gibbs_sampler = Gibbs()
gibbs_chains = gibbs_sampler.run_sampler()
```

3. Estimate posterior quantiles. Find 2.5%, 25%, 50%, 75%, 97.5% posterior percentiles of all parameters. Print results. (suggestion - you may find pandas.DataFrame useful.)

(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 [38]:
```
percentiles = [2.5, 25, 50, 75, 97.5]
quant = np.percentile(gibbs_chains, percentiles, axis=(1, 2))

print("Percentiles")
for i, par in enumerate(params):
  print(f"\n{par}:")
  for j, p in enumerate(percentiles):
    print(f"{p}%: {quant[j, i]:.4f}")
```

```
Percentiles

theta_1:
2.5%: 59.1238
25%: 60.4999
50%: 61.2305
75%: 62.0027
97.5%: 63.6583

theta_2:
2.5%: 63.8590
25%: 65.1836
50%: 65.8879
75%: 66.5050
97.5%: 67.7486

theta_3:
2.5%: 65.7885
25%: 67.1349
50%: 67.7870
75%: 68.4651
97.5%: 69.8415

theta_4:
2.5%: 59.4046
25%: 60.6334
50%: 61.2671
75%: 61.7845
97.5%: 62.9731

mu:
2.5%: 54.5212
25%: 62.0961
50%: 63.9700
75%: 65.7909
97.5%: 72.4000
```

```
sigma:
2.5%: 1.8354
25%: 2.1682
50%: 2.4065
75%: 2.6470
97.5%: 3.4315

tau:
2.5%: 2.0788
25%: 3.4339
50%: 5.0987
75%: 8.3615
97.5%: 23.3618
```

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 [39]:
```
# code taken from project2

npar, m, n = gibbs_chains.shape
print("R:")
for i in range(npar):
  par_samples = gibbs_chains[i]
  chain_avg = par_samples.mean(axis=-1)
  global_avg = chain_avg.mean()
  B = n / (m-1) * np.sum((chain_avg - global_avg)**2)
  chain_var = np.var(par_samples, axis=-1)
  W = chain_var.mean()
  V = (n-1)/n * W + 1/n * B
  R = np.sqrt(V/W)
  print(f"{params[i]}: {R:.5f}")
```

```
R:
theta_1: 0.99843
theta_2: 0.99617
theta_3: 1.01575
theta_4: 0.99738
mu: 0.99462
sigma: 0.99890
tau: 1.03427
```

The condition R < 1.1 is satisfied for all 7 parameters.

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 [6]:
```
!pip install emcee
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/sir
Requirement already satisfied: emcee in /usr/local/lib/python3.7/dist-packages (3.1.3)
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages (from emcee) (1
```

In [7]:
```
import emcee
```

In [42]:
```
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):
    """
    This function was defined above when computing the MAP
    """
    mlh = minus_log_likelihood(
        param, y_i1=data0, y_i2=data1, y_i3=data2, y_i4=data3
    )
    return -1 * mlh

def log_posterior(param, data0, data1, data2, data3):
  prior = log_prior(param)
  if prior == -np.inf:
    return -np.inf
  like = log_likelihood(param, data0, data1, data2, data3)
  return prior + like
```

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 [46]:
```
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 = 40  # number of MCMC walkers
    nburn = 400 # "burn-in" period to let chains stabilize
    nsteps = 1200  # 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[
    sampler.run_mcmc(starting_guesses, nsteps)

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

emcee_trace = np.array(emcee_trace)
```

In [47]:
```
np.shape(emcee_trace)
```

Out[47]:
```
(10, 7, 32000)
```

In [48]:
```
quant_emcee = np.percentile(emcee_trace, percentiles, axis=(0, 2))

print("Percentiles")
for i, par in enumerate(params):
  print(f"\n{par}:")
  for j, p in enumerate(percentiles):
    print(f"{p}%: {quant_emcee[j, i]:.4f}")
```

```
Percentiles

theta_1:
2.5%: 58.7418
25%: 60.3976
50%: 61.2105
75%: 62.0256
97.5%: 63.7128

theta_2:
2.5%: 63.8205
25%: 65.1975
50%: 65.8862
75%: 66.5718
97.5%: 67.9338

theta_3:
2.5%: 65.6481
25%: 67.1186
50%: 67.8244
75%: 68.5185
97.5%: 69.8707

theta_4:
2.5%: 59.3257
25%: 60.5097
50%: 61.1060
75%: 61.7140
97.5%: 62.9335

mu:
2.5%: 48.5387
25%: 61.9083
50%: 63.9494
75%: 65.9490
97.5%: 78.0074

sigma:
```

```
2.5%: 1.8672
25%: 2.2456
50%: 2.5014
75%: 2.8093
97.5%: 3.6222

tau:
2.5%: 2.0020
25%: 3.6971
50%: 5.5560
75%: 9.6317
97.5%: 59.3689
```

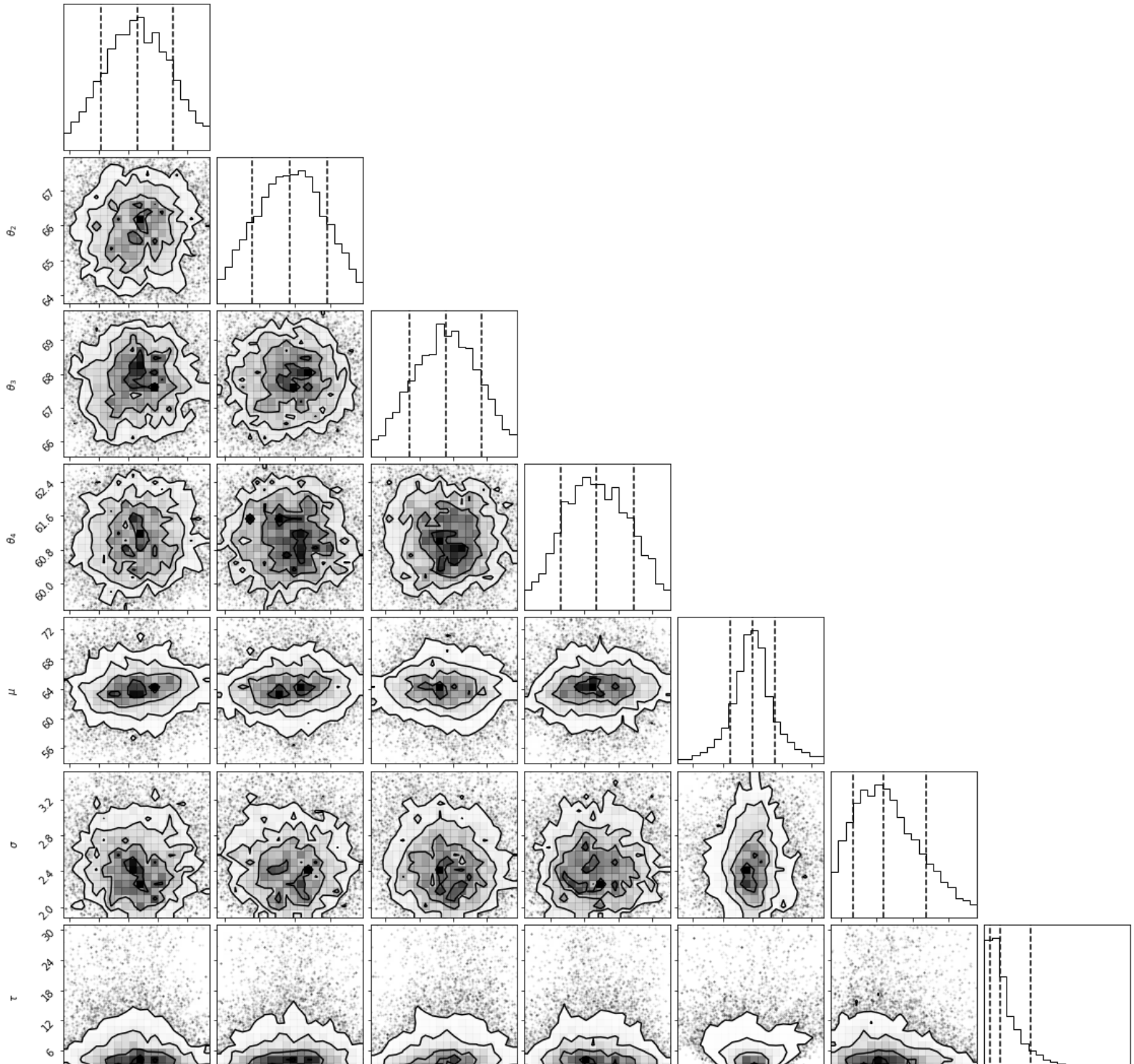
These quantiles match the earlier result quite well, but with a siginificant discrepancy in the mu and especially tau 97.5 percentile values.

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 [49]:
```
!pip install corner
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/sir
Collecting corner
  Downloading corner-2.2.1-py3-none-any.whl (15 kB)
Requirement already satisfied: matplotlib>=2.1 in /usr/local/lib/python3.7/dist-packages (fror
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in /usr/local/lib/pyth
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.7/dist-packages (from ma
Requirement already satisfied: numpy>=1.11 in /usr/local/lib/python3.7/dist-packages (from mat
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.7/dist-packages (fr
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.7/dist-packages (fr
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-packages (from pythor
Installing collected packages: corner
Successfully installed corner-2.2.1
```

In [50]:
```
import corner
fig = corner.corner(emcee_trace[0, :, :].T, labels=["$\\theta_1$", "$\\theta_2$", "$\\theta_3$",
```



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

In [51]:
```
m, npar, n = emcee_trace.shape
print("R:")
for i in range(npar):
  par_samples = emcee_trace[:, i]
  chain_avg = par_samples.mean(axis=-1)
  global_avg = chain_avg.mean()
  B = n / (m-1) * np.sum((chain_avg - global_avg)**2)
  chain_var = np.var(par_samples, axis=-1)
  W = chain_var.mean()
  V = (n-1)/n * W + 1/n * B
  R = np.sqrt(V/W)
  print(f"{params[i]}: {R:.5f}")
```

```
R:
theta_1: 1.00103
theta_2: 1.00154
theta_3: 1.00167
theta_4: 1.00278
mu: 1.00080
sigma: 1.00113
tau: 1.01332
```

The parameters pass the convergence test in this sample too, except for tau (barely).

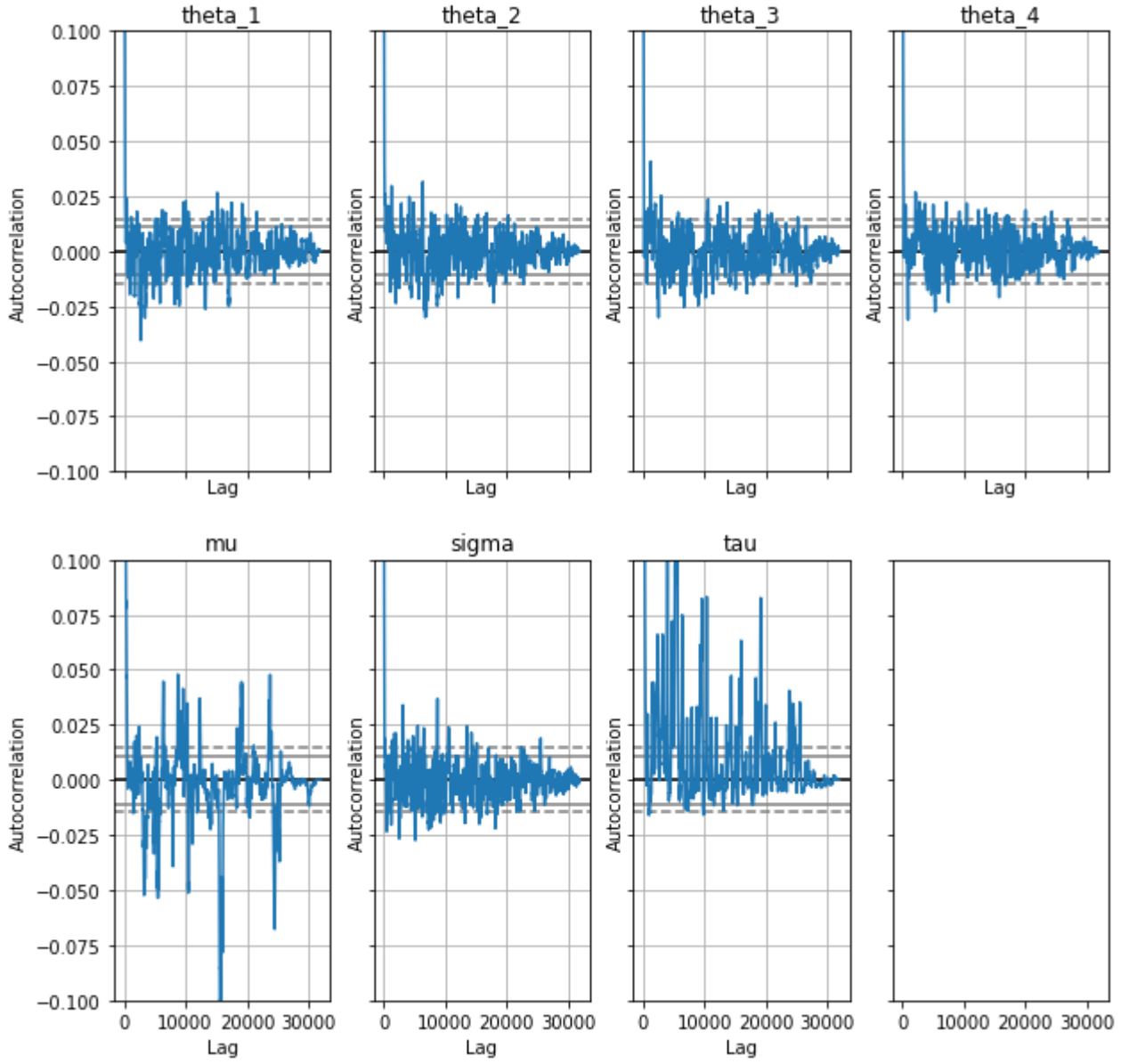
7. Using autocorrelation_plot from pandas (https://pandas.pydata.org/pandas-docs/stable/visualization.html#visualization-autocorrelation), plot the auto-correlation of six parameters and determine that it gets small for large lag.

In [52]:
```
from pandas.plotting import autocorrelation_plot
```

In [58]:
```
emcee_trace.shape
```

Out[58]:
```
(10, 7, 32000)
```

In [60]:
```
fig, axs = plt.subplots(figsize=(10,10), ncols=4, nrows=2, sharex=True, sharey=True)
for i in range(7):
  ax = axs.ravel()[i]
  autocorrelation_plot(emcee_trace[:, i].T, ax=ax)
  ax.set_title(params[i])
plt.setp(axs, ylim=(-0.1, 0.1))
plt.show()
```



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.

In [61]:
```
!pip install daft
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/sir
Requirement already satisfied: daft in /usr/local/lib/python3.7/dist-packages (0.0.4)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-packages (from daf
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages (from daft) (1.
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.7/dist-packages (fr
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.7/dist-packages (from ma
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in /usr/local/lib/pyth
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.7/dist-packages (fr
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-packages (from python
```

In [62]:
```
import daft
```

The below cell sets up latex in matplotlib. This will take few minutes, and you don't need to do it again once they are installed.

In [63]:
```
import matplotlib
from matplotlib import rc
rc('text', usetex=True)
matplotlib.rcParams['text.latex.preamble'] = [r'\usepackage{amsmath}']
!apt install texlive-fonts-recommended texlive-fonts-extra cm-super dvipng
```

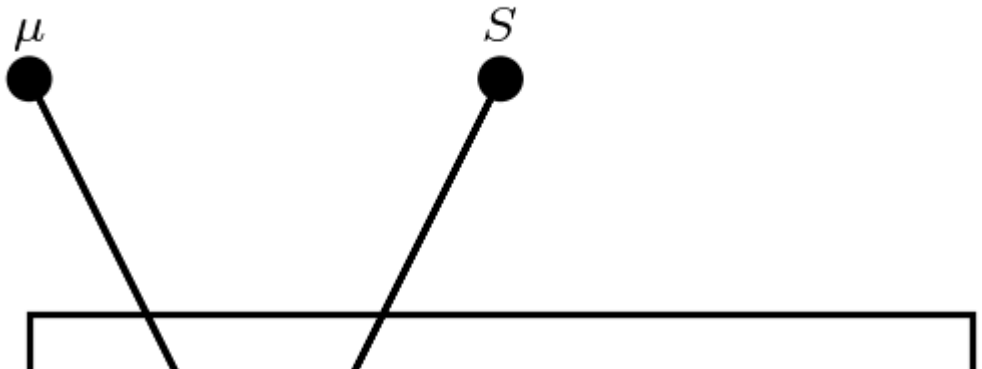
```
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following package was automatically installed and is no longer required:
  libnvidia-common-460
Use 'apt autoremove' to remove it.
The following additional packages will be installed:
  cm-super-minimal fonts-adf-accanthis fonts-adf-berenis fonts-adf-gillius
  fonts-adf-universalis fonts-cabin fonts-comfortaa fonts-croscore
  fonts-crosextra-caladea fonts-crosextra-carlito fonts-dejavu-core
  fonts-dejavu-extra fonts-droid-fallback fonts-ebgaramond
  fonts-ebgaramond-extra fonts-font-awesome fonts-freefont-otf
  fonts-freefont-ttf fonts-gfs-artemisia fonts-gfs-complutum fonts-gfs-didot
  fonts-gfs-neohellenic fonts-gfs-olga fonts-gfs-solomos fonts-go
  fonts-junicode fonts-lato fonts-linuxlibertine fonts-lmodern fonts-lobster
  fonts-lobstertwo fonts-noto-hinted fonts-noto-mono fonts-oflb-asana-math
  fonts-open-sans fonts-roboto-hinted fonts-sil-gentium
  fonts-sil-gentium-basic fonts-sil-gentiumplus fonts-sil-gentiumplus-compact
  fonts-stix fonts-texgyre ghostscript gsfonts javascript-common
  libcupsfilters1 libcupsimage2 libgs9 libgs9-common libijs-0.35 libjbig2dec0
  libjs-jquery libkpathsea6 libpotrace0 libptexenc1 libruby2.5 libsynctex1
  libtexlua52 libtexluajit2 libzzip-0-13 lmodern pfb2t1c2pfb poppler-data
  preview-latex-style rake ruby ruby-did-you-mean ruby-minitest
  ruby-net-telnet ruby-power-assert ruby-test-unit ruby2.5
  rubygems-integration t1utils tex-common tex-gyre texlive-base
  texlive-binaries texlive-fonts-extra-links texlive-latex-base
  texlive-latex-extra texlive-latex-recommended texlive-pictures
  texlive-plain-generic tipa
Suggested packages:
  fonts-noto fontforge ghostscript-x apache2 | lighttpd | httpd poppler-utils
  fonts-japanese-mincho | fonts-ipafont-mincho fonts-japanese-gothic
  | fonts-ipafont-gothic fonts-arphic-ukai fonts-arphic-uming fonts-nanum ri
  ruby-dev bundler debhelper perl-tk xpdf-reader | pdf-viewer
  texlive-fonts-extra-doc texlive-fonts-recommended-doc texlive-latex-base-doc
  python-pygments icc-profiles libfile-which-perl
  libspreadsheet-parseexcel-perl texlive-latex-extra-doc
  texlive-latex-recommended-doc texlive-pstricks dot2tex prerex ruby-tcltk
  | libtcltk-ruby texlive-pictures-doc vprerex
The following NEW packages will be installed:
  cm-super cm-super-minimal dvipng fonts-adf-accanthis fonts-adf-berenis
```

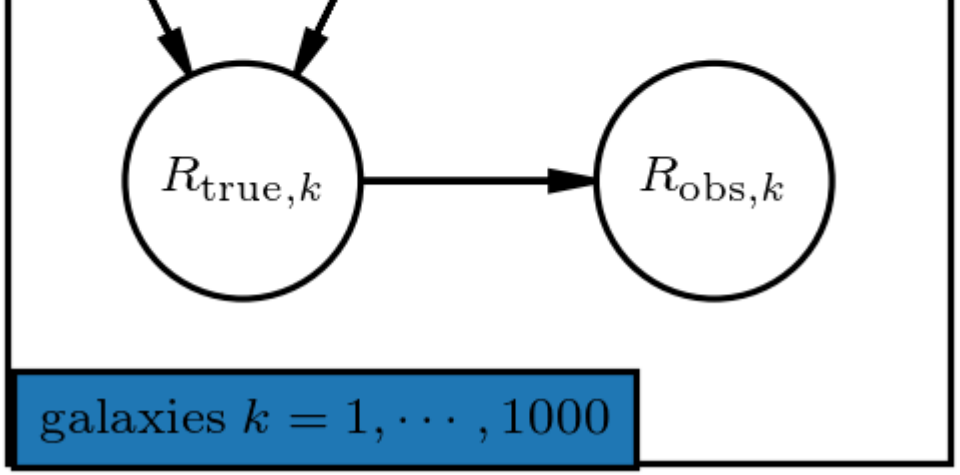
In [64]:
```
!sudo apt install cm-super dvipng texlive-latex-extra texlive-latex-recommended
```

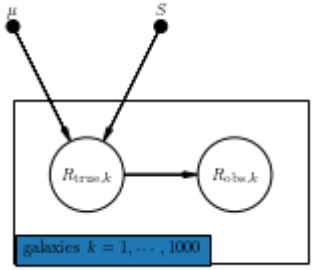
```
Reading package lists... Done
Building dependency tree
Reading state information... Done
texlive-latex-recommended is already the newest version (2017.20180305-1).
texlive-latex-recommended set to manually installed.
cm-super is already the newest version (0.3.4-11).
dvipng is already the newest version (1.15-1).
texlive-latex-extra is already the newest version (2017.20180305-2).
texlive-latex-extra set to manually installed.
The following package was automatically installed and is no longer required:
  libnvidia-common-460
Use 'sudo apt autoremove' to remove it.
0 upgraded, 0 newly installed, 0 to remove and 22 not upgraded.
```

In [72]:
```
rc("font", family="serif", size=8)
rc("text", usetex=True)


# Instantiate a PGM.
pgm = daft.PGM([2.3, 2.05], origin=[0.3, 0.3], grid_unit=2.6, node_unit=1.3, observed_style="inn

# Hierarchical parameters.
pgm.add_node(daft.Node("mu", r"$\mu$", 0.5, 2, fixed=True))
pgm.add_node(daft.Node("S", r"$S$", 1.5, 2, fixed=True))

# Latent variable:
pgm.add_node(daft.Node("Rtrue", r"$R_{{\rm true},k}$", 1, 1))

# Data:
pgm.add_node(daft.Node("Robs", r"$R_{{\rm obs},k}$", 2, 1))

# Add in the edges.
pgm.add_edge("mu", "Rtrue")
pgm.add_edge("S", "Rtrue")
pgm.add_edge("Rtrue", "Robs")

# And a plate.
pgm.add_plate(daft.Plate([0.5, 0.5, 2, 1], label=r"galaxies $k = 1, \cdots, 1000$",
    shift=-0.1))

# Render and save.
pgm.render()
pgm.figure.savefig("samplingdistributions.png", dpi=230)

from IPython.display import Image
Image(filename="samplingdistributions.png")
```

Out[72]:

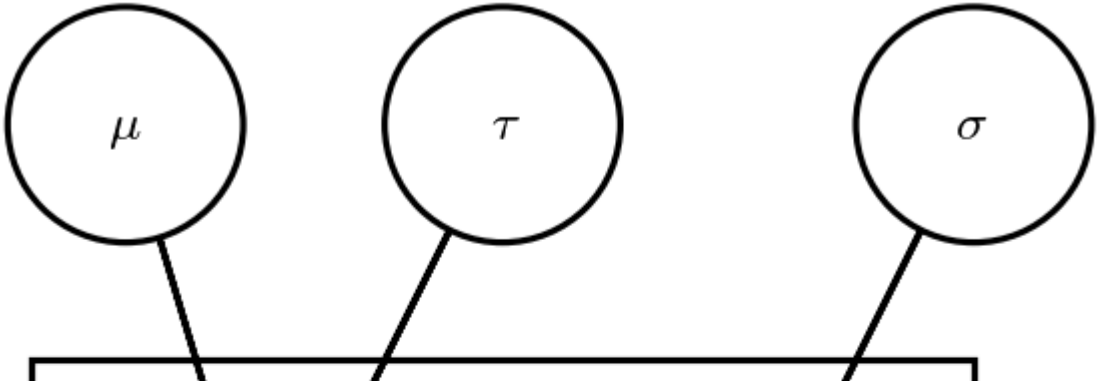


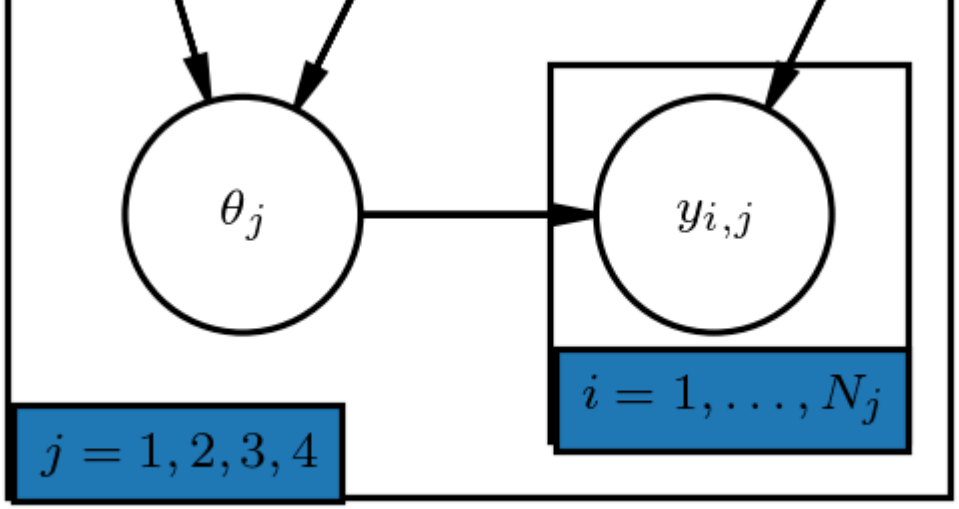

$R_{\text{true},k}$
$R_{\text{obs},k}$
galaxies $k = 1, \cdots, 1000$

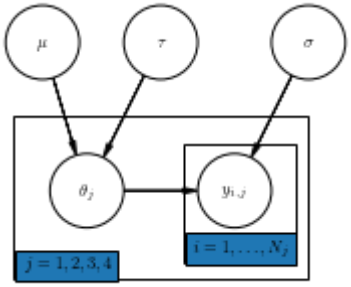

μ
S
$R_{\text{true},k}$
$R_{\text{obs},k}$
galaxies $k = 1, \cdots, 1000$

In [100]:
```
# Instantiate a PGM.
pgm = daft.PGM([2.7, 2.05], origin=[0.3, 0.3], grid_unit=2.6, node_unit=1.3, observed_style="inn

# Hierarchical parameters.
pgm.add_node(daft.Node("mu", r"$\mu$", 0.7, 2, fixed=False))
pgm.add_node(daft.Node("tau", r"$\tau$", 1.5, 2, fixed=False))
pgm.add_node(daft.Node("sigma", r"$\sigma$", 2.5, 2, fixed=False))

# Latent variable:
pgm.add_node(daft.Node("theta", r"$\theta_{j}$", 1, 1))

# Data:
pgm.add_node(daft.Node("y", r"$y_{i, j}$", 2, 1))

# Add in the edges.
pgm.add_edge("mu", "theta")
pgm.add_edge("tau", "theta")
pgm.add_edge("sigma", "y")
pgm.add_edge("theta", "y")

# And a plate.
pgm.add_plate(daft.Plate([0.5, 0.5, 2, 1], label=r"$j = 1, 2, 3, 4$",
    shift=-0.1))

pgm.add_plate(daft.Plate([1.65, 0.62, 0.76, 0.7], label=r"$i = 1, \dots, N_j$",
    shift=-0.1))

# Render and save.
pgm.render()
pgm.figure.savefig("hnm.png", dpi=230)

from IPython.display import Image
Image(filename="hnm.png")
```

Out[100]:







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:

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

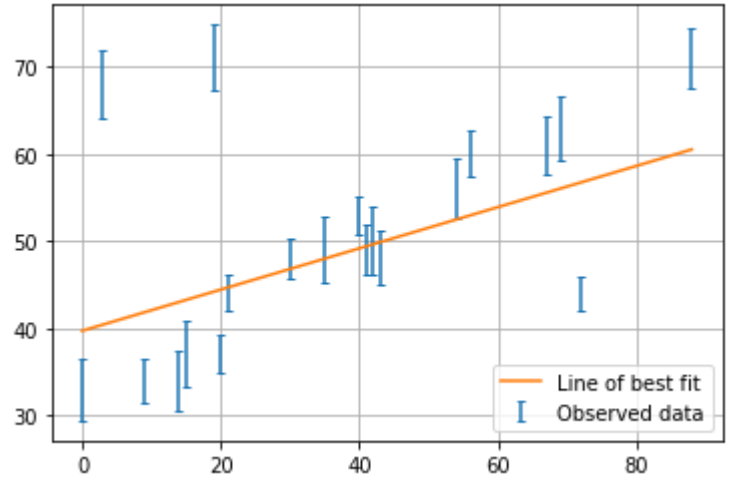
This loss expression is known as a squared loss; here we've simply shown that the squared loss can be derived from the Gaussian log likelihood.

In [2]:
```
# 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. Make sure to show errorbars.

In [3]:
```
from scipy import optimize

def loss(theta, x=x, y=y, e=e):
  yhat = theta[0] * x + theta[1]
  s = (y - yhat)**2/e**2  # summand
  return s.sum()

# initial params
slope_init = (y[-1] - y[0]) / (x[-1] - x[0])
int_init = y[0]
# minimize
mle = optimize.fmin(loss, x0=[slope_init, int_init])
print(f"Max likelihood parameters: theta = {mle}")

xgrid = np.linspace(x.min(), x.max(), num=100)
ygrid = mle[0] * xgrid + mle[1]

plt.figure()
plt.errorbar(x, y, yerr=e, fmt="none", capsize=2, label="Observed data")
plt.plot(xgrid, ygrid, label="Line of best fit")
plt.legend()
plt.grid()
plt.show()
```

```
Optimization terminated successfully.
         Current function value: 201.417425
         Iterations: 39
         Function evaluations: 78
Max likelihood parameters: theta = [ 0.23621166 39.69977582]
```



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.

We want a likelihood of the form:

$$p\left(\{y_i\}|\theta\right) = \prod_i \left(\frac{g_i}{\sqrt{2\pi e_i^2}} \exp\left(-\frac{\left(y_i - (\theta_0 x_i + \theta_1)\right)^2}{2e_i^2} \right) + \frac{1-g_i}{\sqrt{2\pi\sigma_B^2}} \exp\left(-\frac{\left(y_i - (\theta_0 x_i + \theta_1)\right)^2}{2\sigma_B^2} \right) \right)$$

The log-likelihood becomes:

$$\log p\left(\{y_i\}|\theta\right) = \sum_i \log \left(\frac{g_i}{\sqrt{2\pi e_i^2}} \exp\left(-\frac{\left(y_i - (\theta_0 x_i + \theta_1)\right)^2}{2e_i^2} \right) + \frac{1-g_i}{\sqrt{2\pi\sigma_B^2}} \exp\left(-\frac{\left(y_i - (\theta_0 x_i + \theta_1)\right)^2}{2\sigma_B^2} \right) \right)$$

In [4]:
```
def normal(x, mean, sigma):
  """
  A normal distribution
  """
  amplitude = 1 / np.sqrt(2*np.pi*sigma)
  exp = - (x - mean)**2 / (2 * sigma**2)
  return amplitude * np.exp(exp)

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):
  g = theta[2:]  # g_i
  yhat = theta[0] * x + theta[1]
  # gaussian part
  gauss = g * normal(y, yhat, e)
  # outlier part
  out = (1-g) * normal(y, yhat, sigma_B)
  return np.sum(np.log(gauss + out))

def log_posterior(theta, x, y, e, sigma_B):
  prior = log_prior(theta)
  if prior == -np.inf:
    return -np.inf
  like = log_likelihood(theta, x, y, e, sigma_B)
  return prior + like
```

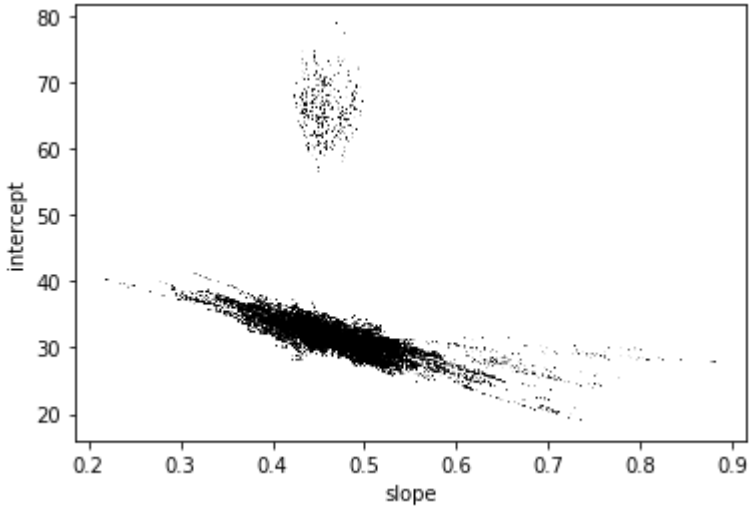
Now, run the MCMC samples.

In [8]:
```
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(mle, 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 [9]:
```
plt.plot(sample[:, 0], sample[:, 1], ',k', alpha=0.1)
plt.xlabel('slope')
plt.ylabel('intercept')
plt.show()
```

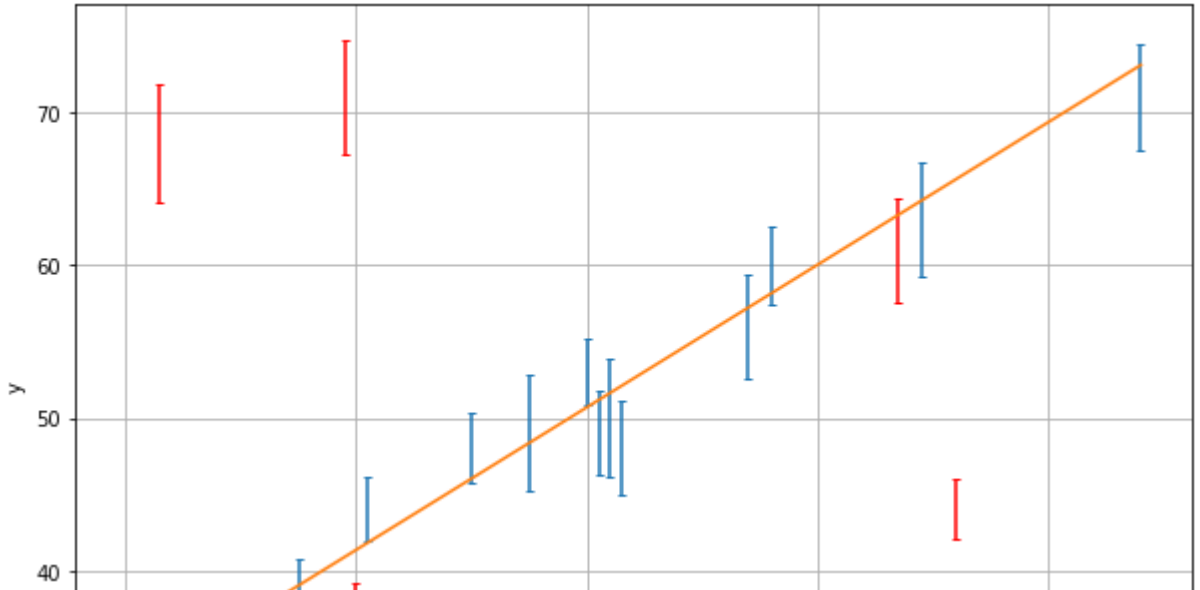


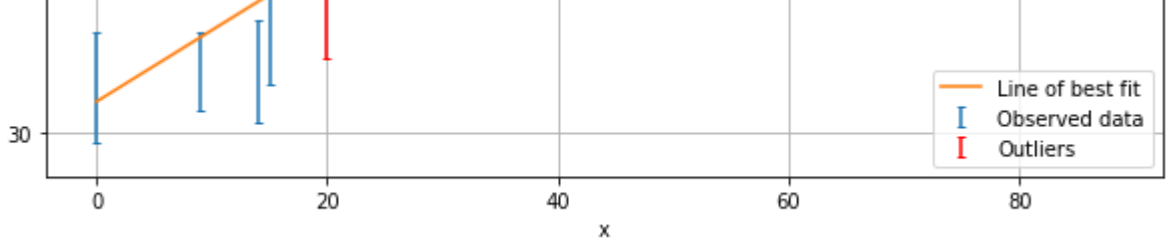
We allowed the model to have a nuisance parameter $0 < g_i < 1$ for each data point: $g_i = 0$ indicates an outlier. We can also allow sb to be a nuisance parameter to marginalize over (or just make it a large number). Now, let us define an outlier whenever posterior $E(g_i) < 0.5$.

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 (with errorbar).

In [10]:
```
expected_vals = sample.mean(axis=0)
theta_best = expected_vals[:2]
print(theta_best)
g_expect = expected_vals[2:]

ygrid = theta_best[0] * xgrid + theta_best[1]

out_args = np.where(g_expect < 0.5)[0]
x_out = x[out_args]
y_out = y[out_args]
e_out = e[out_args]

x_in = np.delete(x, out_args)
y_in = np.delete(y, out_args)
e_in = np.delete(e, out_args)


plt.figure(figsize=(10, 7))
plt.errorbar(x_in, y_in, yerr=e_in, fmt="none", capsize=2, label="Observed data")
plt.errorbar(x_out, y_out, yerr=e_out, fmt="none", capsize=2, c="red", label="Outliers")
plt.plot(xgrid, ygrid, label="Line of best fit")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.grid()
plt.show()
```

```
[ 0.46706476 32.01135784]
```




30
0
20
40
60
80
x
Line of best fit
Observed data
Outliers