Brief introduction to `emcee`

`emcee` is a popular package for running MCMC. Full documentation is here:

https://emcee.readthedocs.io/en/stable/

In a nutshell, the program requires you to set up a function for the target PDF, which for our purposes, is the posterior PDF.

$$\pi(\theta) = p(\theta|D,I) \propto p(\theta|I)p(D|\theta,I)$$

where θ is the vector of model parameters.

Let's solve a problem we know how to solve, but using `emcee`.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
```

In [2]:
```
# These are data from an old notebook (LinearRegressionMatrix2) that approximat
x     = np.array([1.0, 2.0, 3.0, 4.0, 5.0])
y     = np.array([1.1, 1.8, 3.3, 4.2, 4.9])
sigy  = np.array([0.12, 0.15, 0.11, 0.18, 0.09])
```

Let's fit a straight line

$$y = a + bx$$

through the data points. The model is linear so we can use matrix algebra to solve for the best fit parameters $\hat{\theta}$ and the covariance matrix $\mathbf{\Sigma}$.

In [3]:
```
# data covariance matrix
vars = sigy*sigy
E = np.diag(vars)
Einv = np.linalg.inv(E)
```

In [4]:
```
# design matrix G
G1 = np.ones_like(x)
G2 = x
G = np.vstack([G1, G2]).T
```

In [5]:
```
# response vector
D = y
```

In [6]:
```
PSI      = np.dot(G.T, np.dot(Einv, G))
thetahat = np.dot(np.linalg.inv(PSI), np.dot(G.T, np.dot(Einv, D)))
thetacov = np.linalg.inv(PSI)
```

In [7]:
```
thetahat
```

Out[7]:
```
array([0.15898778, 0.96590654])
```

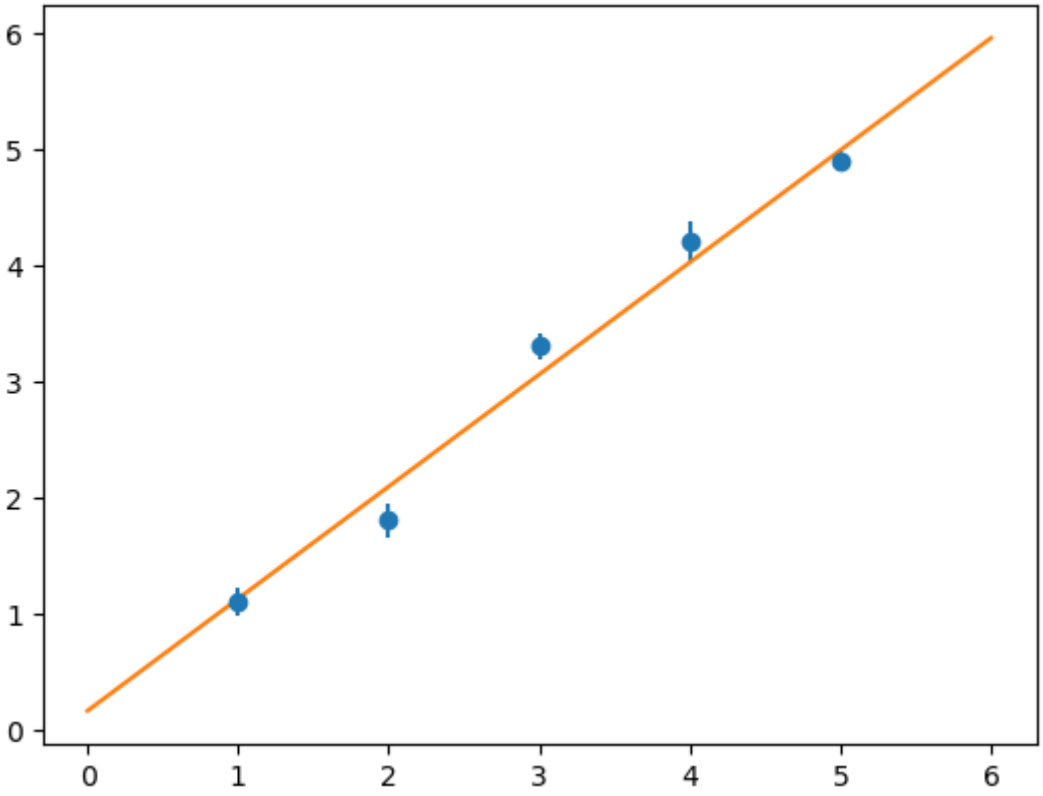
In [8]:
```
thetacov
```

Out[8]:
```
array([[ 0.01586015, -0.00397958],
       [-0.00397958,  0.0012173 ]])
```

In [9]:
```
def model(x, a, b):
    return a + b*x
```

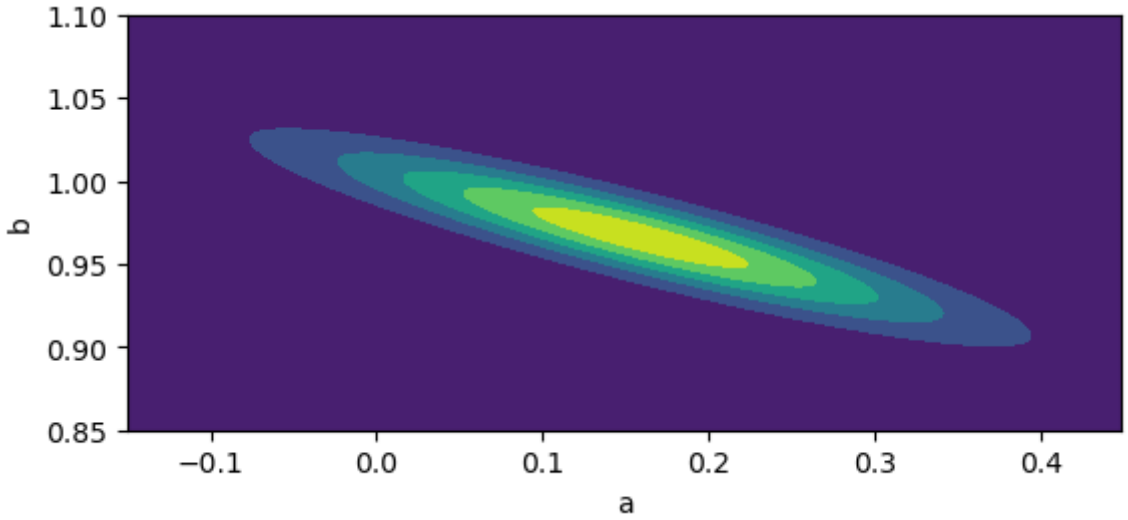
In [10]:
```
plt.errorbar(x, y, sigy, fmt='o')
xgrid = np.linspace(0, 6, 1000)
plt.plot(xgrid, model(xgrid, *thetahat), c='#ff7f0e')
```

Out[10]:
```
[<matplotlib.lines.Line2D at 0x10d0b5db0>]
```



In [11]:
```
# Borrowing this cell from FitLline n.b.

# For m = 2 dimensions, you need to create a grid of (x,y) values,
# which you can compute the PDF on.

x, y = np.mgrid[-0.15:0.45:0.001, 0.85:1.1:0.001]
pos  = np.dstack((x, y))

pdf  = stats.multivariate_normal.pdf(pos, mean=thetahat, cov=thetacov)

fig = plt.figure()
ax  = fig.add_subplot()
ax.contourf(x, y, pdf)
ax.set_xlabel('a')
```

```
ax.set_ylabel('b')
ax.set_aspect('equal')  # plot with equal axes
```



Let's now solve this problem using the `emcee` package.

In [12]:
```
import emcee
```

In [13]:
```
def log_prior(theta):
    return 1.0
```

In [14]:
```
def log_like(theta, x, y, sigy):
    a, b = theta
    y_model = a + b*x
    return -0.5*np.sum(((y-y_model)/sigy)**2) - 0.5*np.sum(np.log(2*np.pi*sigy*
```

In [15]:
```
def log_posterior(theta, x, y, sigy):
    return log_prior(theta) + log_like(theta, x, y, sigy)
```

In [16]:
```
# Reload data (just in case it got messed up)
x     = np.array([1.0, 2.0, 3.0, 4.0, 5.0])
y     = np.array([1.1, 1.8, 3.3, 4.2, 4.9])
sigy  = np.array([0.12, 0.15, 0.11, 0.18, 0.09])
```

In [17]:
```
nparams  = 2      # number of parameters theta=(a,b)
nwalkers = 50     # number of independent walkers
nburn    = 1000   # number of burn-in steps to throw out (later)
nsteps   = 2000   # number of total steps including burn in
starting_guesses = np.random.rand(nwalkers, nparams)  # starting guesses for th
```

In [18]:
```
#starting_guesses
```

This command sets up the sampler with the desired parameters and log_posterior.

In [19]:
```
sampler = emcee.EnsembleSampler(nwalkers, nparams, log_posterior, args=[x, y, s
```

In [20]:
```
# You can watch the progress if you install the `tqdm` package.
sampler.run_mcmc(starting_guesses, nsteps) #, progress=True)
```

```
        330829849, 1312866445, 2669184139, 2628880780,   43345786,
       2569686590,  780049836,  401789372,   64131103, 1175484377,
       4044378447,  363289648,  376830729, 3129987516,  827166550,
       1242912457, 1875787001, 3124848058, 2113918828, 2566807173,
       2920034991,  905674012,  883817782, 3277729560,  246225537,
       1791581121, 2950321640, 4033826008,  155127235, 3092835712,
       1052378036, 4027634852, 3648974411,  235586976, 1852047891,
       1197093614, 1640303734, 1427461778,  418694260, 3384363097,
       1398114099, 2775899393, 1336849515, 1666716880, 3419109589,
       3971261125, 2146131563, 2312968859, 1019998335, 3071129896,
       3046215477,  584712739, 4021614341,  311631034, 1960526953,
       2034241294, 2594316273, 3868251305, 3907634981, 1884566108,
       2637330530, 1916568509, 1486408979, 3323270196], dtype=uint32), 427, 0,
0.0))
```

`sampler.chain` is an array of dimensions `(nwalkers, nsteps, nparams)`. Let's cut out the burn-in steps and join the the walkers into an array of dimensions `(nwalkers*(nsteps-nburn), nparams)`.

In [21]:
```
samples = np.array(sampler.chain[:, nburn:, :])
print(samples.shape)
samples = samples.reshape(-1, nparams)
print(samples.shape)
```

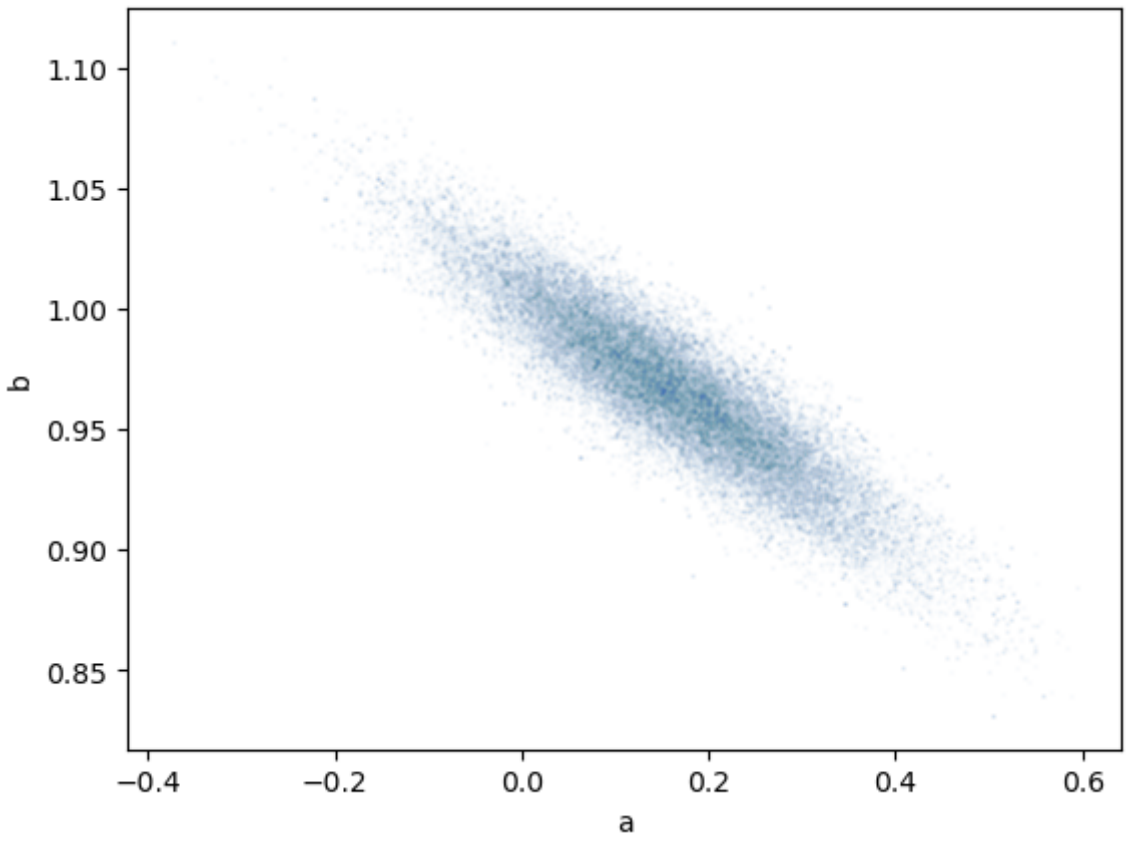
```
(50, 1000, 2)
(50000, 2)
```

In [22]:
```
print(samples)
```

```
[[0.05638749 0.9754506 ]
 [0.08958178 0.97303423]
 [0.11313854 0.96837137]
 ...
 [0.10552153 0.95603283]
 [0.10552153 0.95603283]
 [0.107564   0.95456785]]
```

In [23]:
```
# plot the chains as a scatter plot
plt.scatter(samples[:, 0], samples[:, 1], alpha=0.01, s=1)
plt.xlabel('a')
plt.ylabel('b')
```

Out[23]:
```
Text(0, 0.5, 'b')
```



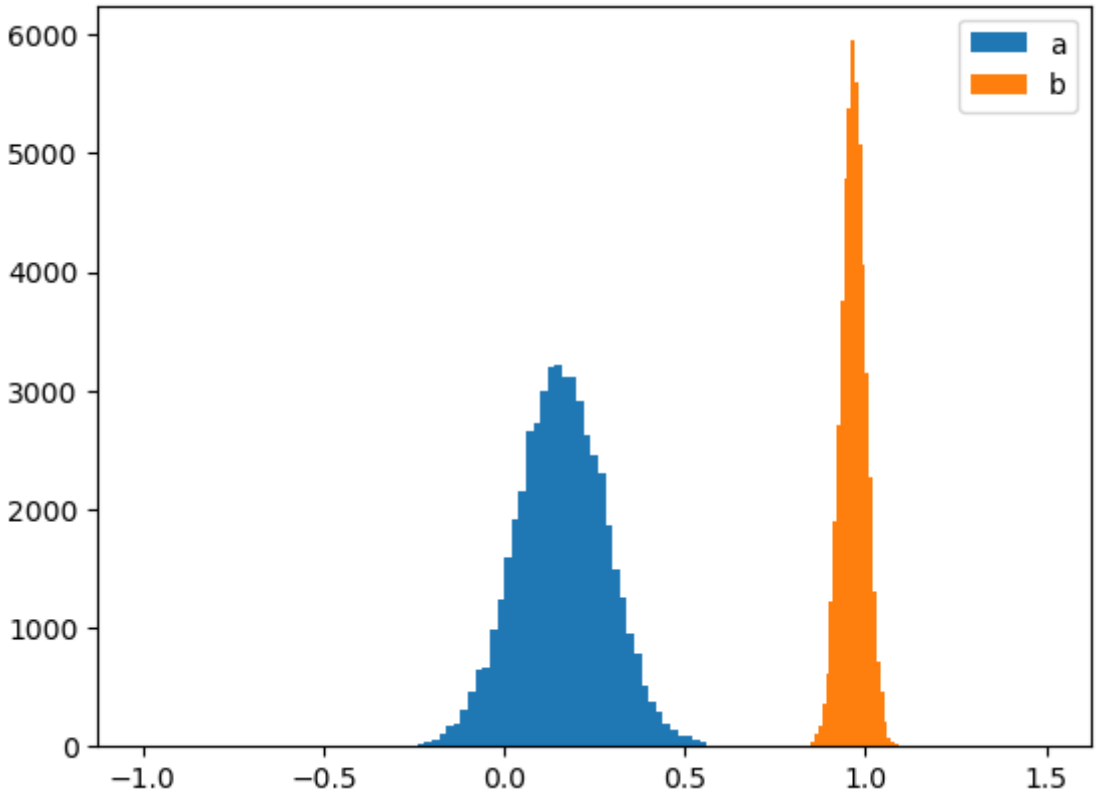
In [24]:
```
# acceptance fractions of each walker
sampler.acceptance_fraction
```

Out[24]:
```
array([0.7095, 0.703 , 0.714 , 0.719 , 0.708 , 0.7235, 0.725 , 0.7345,
       0.7475, 0.7125, 0.7055, 0.736 , 0.7215, 0.7355, 0.7015, 0.703 ,
       0.7225, 0.7   , 0.7225, 0.703 , 0.7105, 0.6975, 0.7005, 0.708 ,
       0.7325, 0.724 , 0.7245, 0.7045, 0.725 , 0.7335, 0.707 , 0.7065,
       0.7425, 0.7055, 0.709 , 0.7035, 0.6985, 0.699 , 0.7045, 0.7305,
       0.712 , 0.7105, 0.7165, 0.695 , 0.6975, 0.6895, 0.703 , 0.709 ,
       0.71  , 0.728 ])
```

In [25]:
```
asample = samples[:,0]
bsample = samples[:,1]
```

In [26]:
```
ahist = plt.hist(asample, range=[-1,1], bins=100, label='a')
bhist = plt.hist(bsample, range=[0.5,1.5], bins=100, label='b')
plt.legend()
```

Out[26]:
```
<matplotlib.legend.Legend at 0x10d311780>
```



In [27]:
```
samples[:,0].shape
```

Out[27]:
```
(50000,)
```

In [28]:
```
print('a = %8.4f +/- %8.4f' % (np.mean(asample), np.std(asample)))
print('b = %8.4f +/- %8.4f' % (np.mean(bsample), np.std(bsample)))
# full covariance matrix
print(np.cov(samples[:,0], samples[:,1]))
```

```
a =   0.1587 +/-   0.1236
b =   0.9665 +/-   0.0343
[[ 0.01528559 -0.00382214]
 [-0.00382214  0.00117799]]
```

In [29]:
```
# from the matrix solution
print('a = %8.4f +/- %8.4f' % (thetahat[0], np.sqrt(thetacov[0,0])))
print('b = %8.4f +/- %8.4f' % (thetahat[1], np.sqrt(thetacov[1,1])))
print('cov_ab = %8.5f' % thetacov[0,1])
```

```
a =   0.1590 +/-   0.1259
b =   0.9659 +/-   0.0349
cov_ab = -0.00398
```

In [ ]:

Note - the target posterior, in principle, can be any PDF and it doesn't have to contain a likelihood (i.e., comparisons of model with data). Here we take one that consists of two multivariate gaussians with means and covariance matrices given by

$$\mu_1 = \begin{pmatrix} -1 \\ -1 \end{pmatrix}, \Sigma_1 = \begin{pmatrix} 1 & -0.8 \\ -0.8 & 1 \end{pmatrix}$$

$$\mu_2 = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \Sigma_2 = \begin{pmatrix} 1.5 & 0.6 \\ 0.6 & 0.8 \end{pmatrix}$$

In [30]:
```
def log_posterior2(theta):
    x, y = theta
    g1 = stats.multivariate_normal.pdf((x,y), [-1,-1], [[1,-0.8],[-0.8,1]])
    g2 = stats.multivariate_normal.pdf((x,y), [1,2], [[1.5,0.6],[0.6,0.8]])
    return np.log(g1+g2)
```

In [31]:
```
nparams   = 2    # number of parameters
nwalkers = 50
nburn    = 1000
nsteps   = 2000  # 2000 will result in error when you call get_autocorr_time()
starting_guesses = np.random.rand(nwalkers, nparams)
```

In [32]:
```
sampler = emcee.EnsembleSampler(nwalkers, nparams, log_posterior2) #, args=[x,
```

In [33]:
```
sampler.run_mcmc(starting_guesses, nsteps, progress=True)
```

```
You must install the tqdm library to use progress indicators with emcee
```

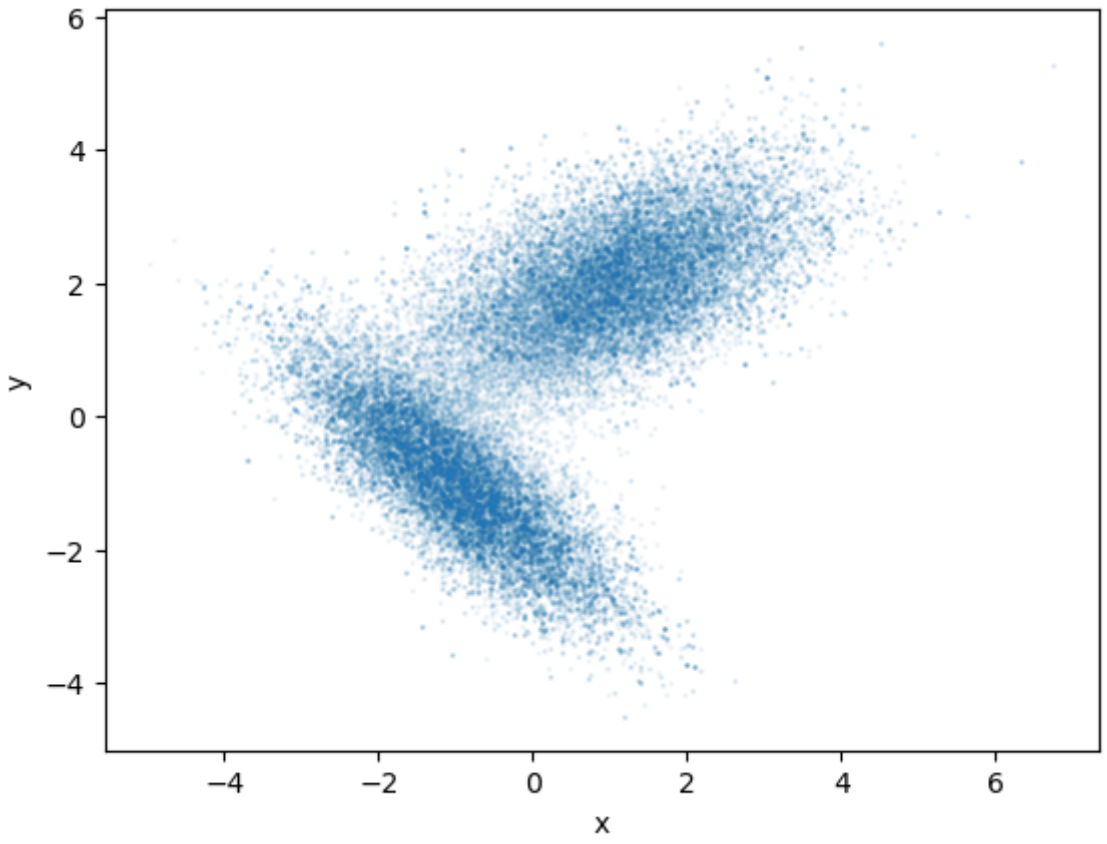
```
        330829849, 1312866445, 2669184139, 2628880780,   43345786,
       2569686590,  780049836,  401789372,   64131103, 1175484377,
       4044378447,  363289648,  376830729, 3129987516,  827166550,
       1242912457, 1875787001, 3124848058, 2113918828, 2566807173,
       2920034991,  905674012,  883817782, 3277729560,  246225537,
       1791581121, 2950321640, 4033826008,  155127235, 3092835712,
       1052378036, 4027634852, 3648974411,  235586976, 1852047891,
       1197093614, 1640303734, 1427461778,  418694260, 3384363097,
       1398114099, 2775899393, 1336849515, 1666716880, 3419109589,
       3971261125, 2146131563, 2312968859, 1019998335, 3071129896,
       3046215477,  584712739, 4021614341,  311631034, 1960526953,
       2034241294, 2594316273, 3868251305, 3907634981, 1884566108,
       2637330530, 1916568509, 1486408979, 3323270196], dtype=uint32), 427, 0,
0.0))
```

In [34]:
```
samples = np.array(sampler.chain[:, nburn:, :])
print(samples.shape)
samples = samples.reshape(-1, nparams)
print(samples.shape)
```

```
(50, 1000, 2)
(50000, 2)
```

In [35]:
```
plt.scatter(samples[:, 0], samples[:, 1], alpha=0.05, s=1)
plt.xlabel('x')
plt.ylabel('y')
```

Out[35]:
```
Text(0, 0.5, 'y')
```



In [36]:
```
sampler.acceptance_fraction
```

Out[36]:
```
array([0.5965, 0.6025, 0.5945, 0.6125, 0.5935, 0.6165, 0.578 , 0.61  ,
       0.6005, 0.609 , 0.6075, 0.59  , 0.615 , 0.6   , 0.585 , 0.579 ,
       0.5745, 0.5975, 0.589 , 0.595 , 0.586 , 0.594 , 0.603 , 0.562 ,
       0.594 , 0.6015, 0.593 , 0.628 , 0.5715, 0.579 , 0.579 , 0.5865,
       0.5925, 0.5935, 0.591 , 0.6055, 0.5905, 0.5925, 0.586 , 0.5965,
       0.561 , 0.606 , 0.5795, 0.592 , 0.5955, 0.592 , 0.5735, 0.617 ,
       0.576 , 0.5805])
```

Some advanced features and usage

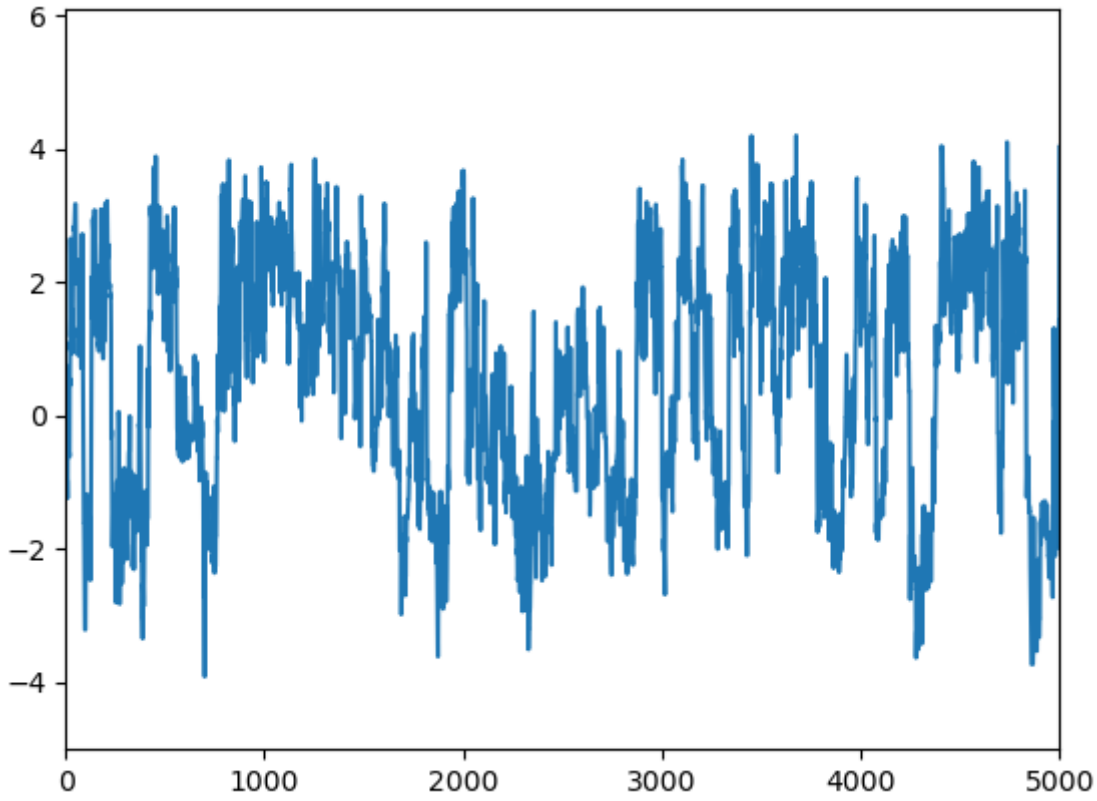
`emcee` has several implementations of "moves" - i.e., how the MCMC decides how to take steps:

https://emcee.readthedocs.io/en/stable/user/moves/

The default is the so-called "stretch move" method from Goodman & Weare (2010). There are other implementations that work better in certain situations. Metropolis-Hastings is also one of the options.

In [37]:
```
plt.plot(samples[:,1])
plt.xlim([0,5000])
```

Out[37]:
```
(0.0, 5000.0)
```



You can see that the chain jumps between the two gaussians with a long-ish time scale (i.e., it spends some time in one before it jumps to the other). Although it is not too important in this case since we can trivially run longer chains, but in other applications where runtime comes with a premium, you want to avoid such behavior as much as possible.

This can be quantified by what's called an autocorrelation time - the average number of steps between independent samples. We want to reduce this as much as possible.

In [38]:
```
# these are the average numbers of steps in (a,b) between independent samples.
print(sampler.get_autocorr_time())
```

```
---------------------------------------------------------------------------
AutocorrError                             Traceback (most recent call last)
Input In [38], in <cell line: 2>()
      1 # these are the average numbers of steps in (a,b) between independent
samples.
----> 2 print(sampler.get_autocorr_time())

File ~/opt/miniconda3/envs/PHYS3358/lib/python3.10/site-packages/emcee/ensembl
e.py:605, in EnsembleSampler.get_autocorr_time(self, **kwargs)
    604 def get_autocorr_time(self, **kwargs):
--> 605     return self.backend.get_autocorr_time(**kwargs)

File ~/opt/miniconda3/envs/PHYS3358/lib/python3.10/site-packages/emcee/backend
s/backend.py:150, in Backend.get_autocorr_time(self, discard, thin, **kwargs)
    131 """Compute an estimate of the autocorrelation time for each parameter
    132
    133 Args:
   (...)
    147
    148 """
    149 x = self.get_chain(discard=discard, thin=thin)
--> 150 return thin * autocorr.integrated_time(x, **kwargs)

File ~/opt/miniconda3/envs/PHYS3358/lib/python3.10/site-packages/emcee/autocor
r.py:112, in integrated_time(x, c, tol, quiet)
    110     msg += "N/{0} = {1:.0f};\ntau: {2}".format(tol, n_t / tol, tau_es
t)
    111     if not quiet:
--> 112         raise AutocorrError(tau_est, msg)
    113     logger.warning(msg)
    115 return tau_est

AutocorrError: The chain is shorter than 50 times the integrated autocorrelati
on time for 2 parameter(s). Use this estimate with caution and run a longer ch
ain!
N/50 = 40;
tau: [48.89315347 62.76975249]
```

Here, `emcee` is giving you and error to tell you that your chains are, in fact, not long enough given our long autocorrelation time.

Let's implement a different set of moves a suggested by in the `emcee` documentation:

https://emcee.readthedocs.io/en/stable/tutorials/moves/

a mixture of `DEMove` (differential evolution) and `DESnookerMove` (another variant). `emcee` allows you to specify what fraction to use for each mover.

In [39]:
```
sampler = emcee.EnsembleSampler(nwalkers, nparams, log_posterior2,
                                moves=[
                                   (emcee.moves.DEMove(), 0.8),        # 80% DE
                                   (emcee.moves.DESnookerMove(), 0.2)  # 20% DE
                                ])
```

In [40]:
```
sampler.run_mcmc(starting_guesses, nsteps, progress=True)
```

```
You must install the tqdm library to use progress indicators with emcee
/var/folders/d2/c2qgnrsx7pq61vp0fh3966zh0000gr/T/ipykernel_35441/4286752579.p
y:5: RuntimeWarning: divide by zero encountered in log
  return np.log(g1+g2)
```

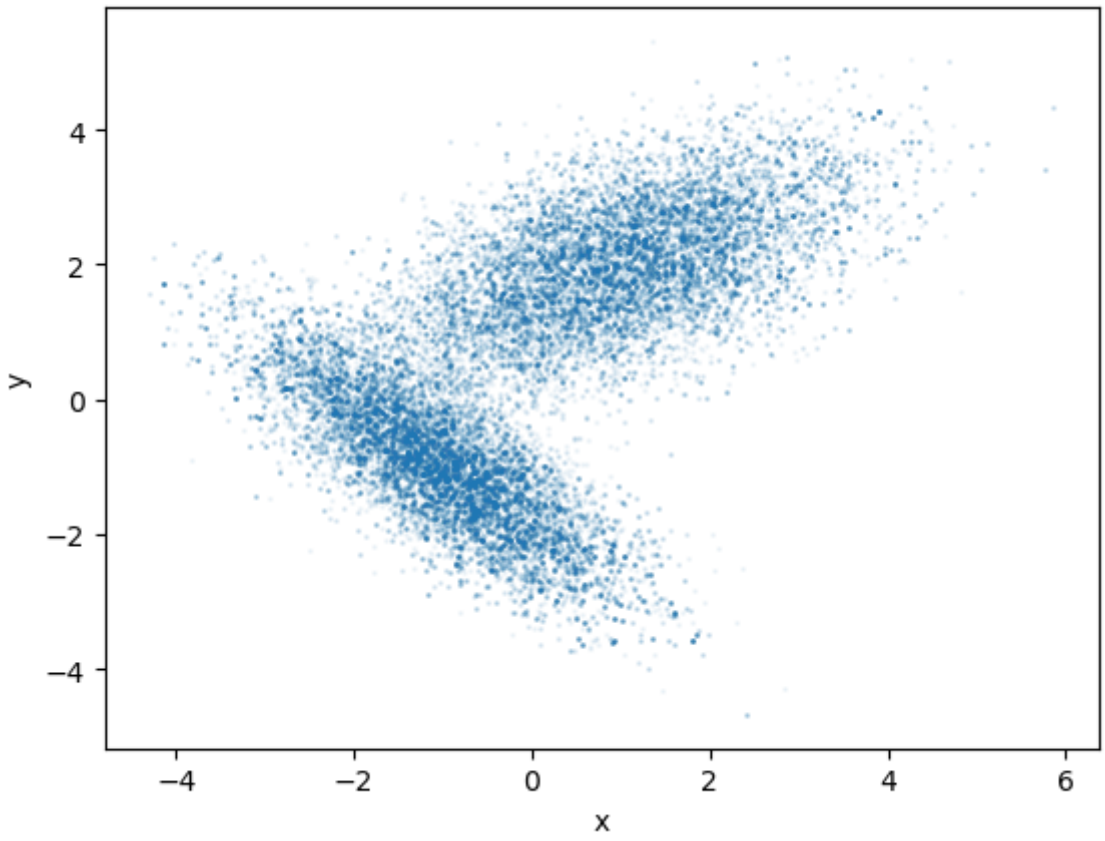
```
 891391195, 2078437973,  291005975, 1222566068, 1986300974,
       3798547085, 3956572551,  369800148, 2289080431, 1374598144,
       3893070298, 2444151122,  729430155,  289405064, 2534189884,
        314790734, 3072728832, 4210223750, 2286972091, 3297883674,
       2072704721, 1916026976, 3328471914, 2974627869, 3695619256,
       3345872068, 1613522274, 1660711734, 3929087114, 1844108947,
       4261779761, 3221499260, 2227194593, 3340642933, 2610320693,
       1683673756, 3315694935, 2863682097, 4272197715,  364876306,
       2396895544, 3785700979, 1241671642,  562848653, 1268613905,
       1027315267,  838131133, 3024301464, 1549955817, 1832416320,
        385736222, 3115419282, 4167866308, 1546047941, 2523896122,
       2215065281,  165352927, 3934509519, 1689823581,  302162718,
       2400430040, 1542597905,   50914886, 1265969546], dtype=uint32), 14, 0,
0.0))
```

In [41]:
```
samples = np.array(sampler.chain[:, nburn:, :])
print(samples.shape)
samples = samples.reshape(-1, nparams)
print(samples.shape)
```

```
(50, 1000, 2)
(50000, 2)
```

In [42]:
```
plt.scatter(samples[:, 0], samples[:, 1], alpha=0.05, s=1)
plt.xlabel('x')
plt.ylabel('y')
```

Out[42]:
```
Text(0, 0.5, 'y')
```

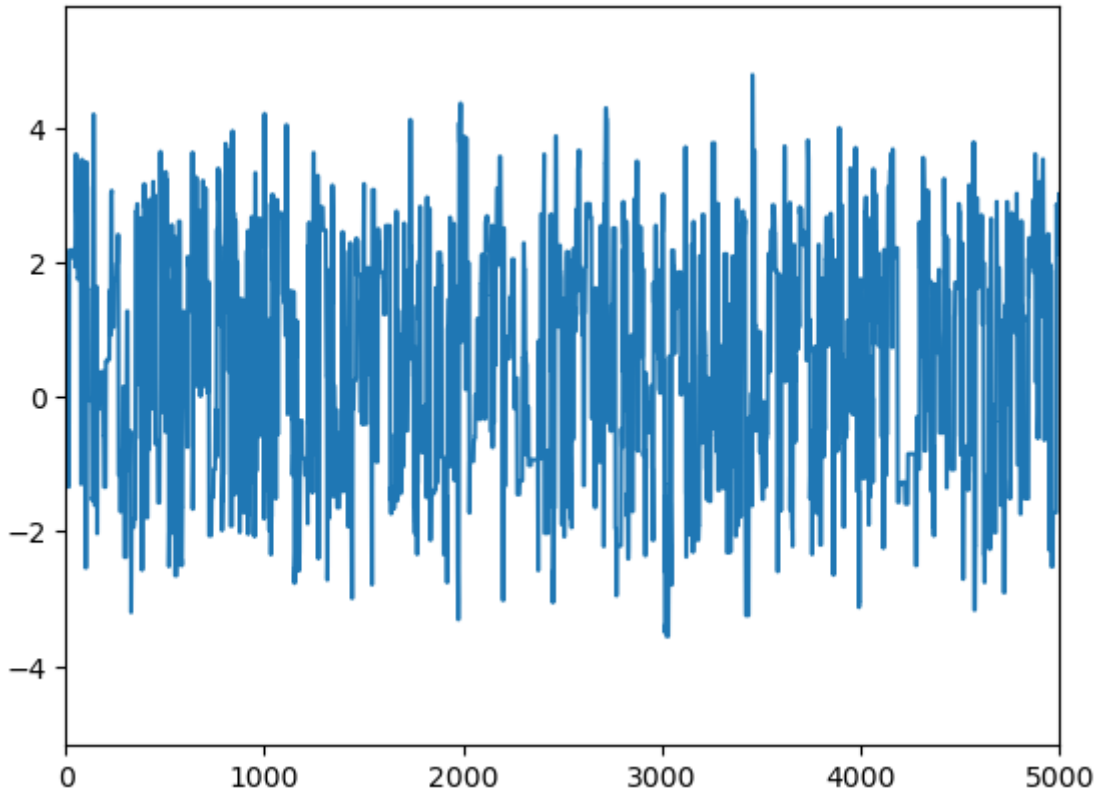


In [43]:
```
sampler.acceptance_fraction
```

Out[43]:
```
array([0.251 , 0.246 , 0.233 , 0.2475, 0.231 , 0.254 , 0.232 , 0.249 ,
       0.243 , 0.2455, 0.2435, 0.254 , 0.242 , 0.228 , 0.2445, 0.2435,
       0.243 , 0.2665, 0.231 , 0.2435, 0.2535, 0.233 , 0.2285, 0.2445,
       0.237 , 0.2255, 0.2385, 0.2635, 0.2375, 0.249 , 0.2415, 0.232 ,
       0.2225, 0.228 , 0.2325, 0.2345, 0.224 , 0.237 , 0.252 , 0.26  ,
       0.2615, 0.256 , 0.239 , 0.234 , 0.231 , 0.249 , 0.2245, 0.2315,
       0.26  , 0.2525])
```

In [44]:
```
plt.plot(samples[:,1])
plt.xlim([0,5000])
```

Out[44]:
```
(0.0, 5000.0)
```



This looks a little better. The walker doesn't spend too much time in one gaussian before jumping to the next.

In [45]:
```
# these are the average numbers of steps in (a,b) between independent samples.
print(sampler.get_autocorr_time())
```

```
[12.87188837 15.76815156]
```

In [ ]: