Markov Chain MonteCarlo

The purpose of this notebook is to provide a very simple explanation of Markov Chain Monte Carlo (MCMC) in the context of parameter fitting.

Parameter selection problem

Suppose we assume a model M to describe data d. For a particular choice x of model parameters, the probability that data d would be measured, $\Pr(d|x, M)$, is called the *likelihood*. Now, the whole point of having a model M is that it gives us a way of simulating, on the basis of parameters x, the whole process by which data are measured. Therefore, my premise is that **given a model M, it is straightforward to calculate the likelihood.**

What we really wish for, though, is the *posterior distribution*, $Pr(x|d, M)$. The usefulness of the posterior distribution should be apparent; we can use it, for example, to put confidence limits on all of the model parameters. The posterior distribution is related to the likelihood as follows according to Bayes' theorem:

$$\Pr(x|d, M) = \frac{\Pr(d|x, M)\ \Pr(x|M)}{\Pr(d|M)}. \quad (1)$$

We refer to $\Pr(x|M)$ as the *prior*, and $\Pr(d|M)$ is the *evidence*. The evidence is just a single number for a given dataset d, yet it's a bit of a chore to calculate. In principle, it involves marginalizing over all possible model parameters x:

$$\Pr(d|M) = \sum_x \Pr(d|x, M).$$

While the evaluation of each term in the sum is straightforward per my first premise, the sum over all possible model parameters makes marginalization potentially daunting. This is especially true for models that have many degrees of freedom. Thus, **calculating the posterior is a pain.** A pain in the posterior, one supposes.

Likelihood ratio

Now, suppose we compare the posterior probability for two different choices of model parameters, x and y:

$$\frac{\Pr(y|d, M)}{\Pr(x|d, M)} = \frac{\Pr(d|y, M)\ \Pr(y|M)}{\Pr(d|x, M)\ \Pr(x|M)} \quad (2)$$

The evidence has canceled out, which is good news if that marginalization sum looked daunting. Now, notice the ratio of priors on the right hand side. This could be useful if we have some reason to expect a particular distribution of model parameters, $P(x|M)$. That could emerge, for example, from previous attempts to measure the parameters of the same system. But if we have no prior reason to favor one set of model parameters over another, then we employ a *flat prior*, that is $\Pr(y|M) = \Pr(x|M)$. The ratio of the posteriors is therefore just the likelihood ratio:

$$\boxed{\text{Assuming a flat prior,} \quad \frac{\Pr(y|d, M)}{\Pr(x|d, M)} = \frac{\Pr(d|y, M)}{\Pr(d|x, M)}} \quad (3)$$

Now, the right hand side is straightforward to calculate, because likelihoods are straightforward. **If only we had a way to back out the posterior distribution itself from such ratios!**

MCMC

In my study of Markov Chains (./MarkovChain.ipynb), I found that the Metropolis-Hastings algorithm makes it possible to draw random samples from probability distribution $P(x)$ by repeatedly evaluating the ratio $\Pr(y)/(\Pr(x)$. It was cool, but it seemed like a solution waiting for a problem. Well, now we have found the problem. The elements of our solution are as follows:

1. A means to evaluate the ratio, $\Pr(y|d, M)/\Pr(x|d, M)$. We use the simple likelihood ratio as in equation (3) for the flat prior, or implement nontrivial priors via equation (2).
2. A routine for generating proposed jumps, satisfying the condition that a proposal to jump $x \to y$ will have the same probability as $y \to x$.
3. A Metropolis-Hastings implementation, such as my `metro()` function (./MarkovChain.ipynb).
4. Generate a reasonable initial guess x_0, and let Metropolis-Hastings have at it.
5. The resulting Markov chain, $x_0, x_1, x_2, x_3, \ldots$ will evolve in such a way as to conform to the posterior distribution $\Pr(x|d, M)$.
6. There exist standard plotting packages in Python for estimating and illustrating the posterior density from the MCMC samples. See Handley (2018) (https://www.theoj.org/joss-papers/joss.00849/10.21105.joss.00849.pdf) and references therein.

Caveats

Typically, the calculation of likelihood ratios is the most intensive part of the MCMC. It is most efficient to have code to evaluate the likelihood (or likelihood times the prior, if we are going with equation (3)) rather than the likelihood ratio. That information can be kept within the Metropolis-Hastings routine, so that there are no unnecessary recalculations of $\Pr(d|y, M)$.

The jump implementation may require strategic thinking. At minimum, this means that each parameter in x should be randomly perturbed with a scale that is roughly comparable to the expected uncertainty in that parameter, but sometimes there is more to it than that, as we found earlier with bimodal distributions. More on this later.

When I first started running the example, I ran into serious problems with floating underflows in the likelihood. I staved off the IEEE NaNs (NaN = Not a Number) by some careful logic, but that wasn't the end of my woes. It turns out that if all the likelihoods are zero, the Markov chain simply wanders around at random! In some remote corner of my mind, I remembered hearing the gravity folks going on about “log likelihood” in the context of MCMC. Suddenly that made a lot of sense!

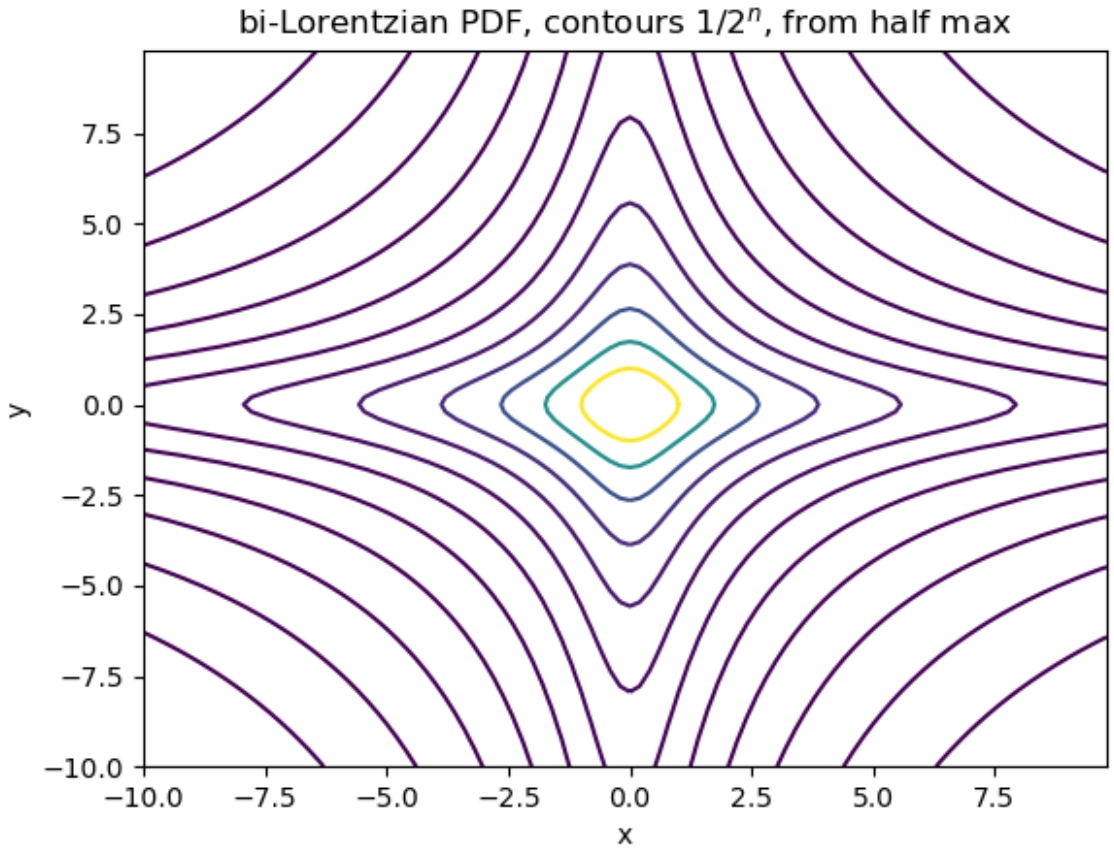
Jump strategy

I like a Lorentzian jump proposal distribution (./MarkovChain.ipynb) so that sometimes long jumps will be proposed, probing for additional modes of the posterior distribution.

It might sound reasonable to apply an independent Lorentzian jump on each parameter, but *the result of such an approach in multidimensions is that the long jumps tend to be in one dimension only.* Below, I have plotted a joint Lorentzian distribution of two parameters, x and y, to illustrate that issue.

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.special as spc
%matplotlib notebook

# Illustration of the shortcoming described above.
# Joint probability distribution of two Lorentzian-distributed, in

domain = 10 # domain size in x and y
xy = np.arange(-domain,domain,0.2)
[x,y] = np.meshgrid(xy,xy)
Pxy = 1/((1+x**2)*(1+y**2)) # Not normalized.
plt.contour(x,y,Pxy, 0.5**np.arange(12,0,-1))
plt.title('bi-Lorentzian PDF, contours $1/2^n$, from half max')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



The illustration above shows what would have been a very selective exploration of our model parameter space, taking jumps along the coordinate axes only. It looks bad in 2D; in higher dimensions it would look even worse. A second mode off at some arbitrary angle would be effectively unreachable. In order to adequately explore the posterior, we must avoid imposing preferred directions.

Isotropized parameter space. The strategy I outline below will work best on a parameter space with modest dimensionality ($\lesssim 10$) and a single, identifiable scale for each parameter. My first step will be to *isotropize* the parameter space. The idea is to represent my model parameters in a vector space with no preferred direction, using dimensionless coordinates that may, in principle, take on any real value.

I need to be careful with Q and B, as both are positive definite parameters. I will represent both parameters by their logarithms. I am then free to think of these parameters as coordinates on an unbounded domain, much like the components of $\mathbf{r}$. This entails that I must accept a flat prior in $\log Q$ and $\log B$.

Assuming positions are in two dimensions ($\mathbf{r} = [x, y]$), the isotropized vector of model parameters is

$$\mathbf{m} = \left[\frac{x}{s_0}, \frac{y}{s_1}, \frac{\log(Q/Q_0)}{s_2}, \frac{\log(B/B_0)}{s_3} \right].$$

With this choice of coordinates for the model, I want the origin ($x = y = 0, Q = Q_0, B = B_0$) to represent a reasonable initial guess. I also choose scales $\mathbf{s} = [s_0, s_1, s_2, s_3]$ to represent an uncertainty in that guess. None of the scales or offsets has to be very accurate; we are after rough orders of magnitude only. If there is concern about being able to explore the posterior adequately, then the scales can be increased, at the expense of lowering the jump acceptance ratio.

Isotropic jumps are chosen as follows:

1. Choose a direction vector for the jump.
   A. Choose a vector of uniform deviates $\mathbf{R}$ on $[0, 1)$, with the same dimensionality as $\mathbf{m}$.
   B. If $|\mathbf{R}|^2 > 1$, go to the previous step.[†]
   C. $\hat{\mathbf{m}} = \frac{\mathbf{R}}{|\mathbf{R}|}$.
2. Choose one more uniform deviate, $\rho \in [0, 1)$. The proposal is:

$$\mathbf{m}^* = \mathbf{m} + \frac{J}{2}\tan\left(\pi\left[\rho - \frac{1}{2}\right]\right)\hat{\mathbf{m}}.$$

   The parameter J is a dimensionless *jump radius*, which can be tuned to get the desired acceptance ratio for the Markov chain. The tangent can result in any real number, positive or negative, distributed as a Lorentzian with FWHM of unity, while the components of $\hat{\mathbf{m}}$ are all positive. Transition probability depends only on the Euclidian distance $|\mathbf{m}^* - \mathbf{m}|$. Thus, the transition probability matrix is symmetric, which is a requirement for jump proposals in Metropolis-Hatings.

[†] Rejecting the "corner" of the hypercube that lies outside the hypersphere is essential to get an isotropic distribution of jump directions. Unfortunately, rejections become increasingly probable as the dimensionality grows. In 5D, the rejection ratio is ~ 0.9. In 11D, it is ~ 0.999. This is because the unit hypersphere occupies a smaller and smaller fraction of the unit hypercube as the dimensionality increases, as we learned when doing Monte Carlo integration of hyperspheres. Fortunately, the iterations are not computationally expensive; they require only random number generation and simple arithmetic, plus the interpreter overhead of the `while` loop.

Simulated Annealing.

The final concept we need for a successful demonstration of MCMC is *burn in*. Unless we have some special insight, the initial guess is often not very good. Even in the relatively modest dimensionality of my example problem $\left(\mathfrak{R}^4\right)$, the Markov chain can get stuck in a local minimum and never find the main peak of the likelihood distribution. Burn in is a strategy based on *simulated annealing* to locate that main peak.

Simulated annealing is a Markov chain application in which we are looking to minimize a function $E(\mathbf{r})$. The approach has proved particularly successful at solving difficult combinatoric problems such as the traveling salesman problem (https://en.wikipedia.org/wiki/Travelling_salesman_problem). A variation of the Metropolis-Hastings algorithm is used:

1. Beginning in state $\mathbf{r}$, evaluate the energy $E = E(\mathbf{r})$.
2. Propose a random jump from $\mathbf{r}$ to $\mathbf{r'}$ based on a symmetric transition probability, $T_{\mathbf{r}\mathbf{r}'}$.
3. Evaluate $E' = E(\mathbf{r'})$.
4. If $E'/E \leq 1$, accept the jump.
5. Otherwise, accept the jump with probability $\exp\left(-\frac{E'-E}{kT}\right)$.

Notice that the equilibrium posterior distribution for this Markov chain is the familiar Boltzmann distribution from statistical mechanics,

$$P(\mathbf{r}|T) \propto \exp\left(-\frac{E(\mathbf{r})}{kT}\right).$$

The minimum energy is found by starting at a high temperature, T, and running the Metropolis algorithm while gradually reducing the temperature until $kT \ll \min_{\mathbf{r}} E(\mathbf{r})$. This emulates the tendency of a physical system to find its minimum energy when it is cooled slowly. There are many physics applications of simulated annealing, such as the Ising model of magnetic materials (./ising/).

The idea behind simulated annealing is that a sufficiently high temperature allows the Markov chain to traverse the topography with relative ease, so that it samples all the local minima. As the temperature is slowly lowered, the shallower basins become inaccessible one by one, until only the global minimum is left. Of course, there are never any guarantees of finding an absolute minimum in a high-dimensional space. A lot depends on the *annealing schedule*, by which we mean the sequence of temperatures and Metropolis iterations. The moral of the simulated annealing story is that patience is a virtue.

MCMC Burn In

Comparing MCMC to simulated annealing, we can make the following analogy:

$$\frac{1}{T}\log\left(\frac{P'}{P}\right) \longleftrightarrow -\frac{\Delta E}{kT},$$

where the left hand side is the log-likelihood ratio divided by "temperature". Notice that E is to be minimized; P is to be maximized. In MCMC, if we divide the log-likelihood ratio by the new, dimensionless parameter T, we can run our Metropolis-Hastings algorithm like a simulated annealing scheme. We begin with $T \gg 1$, and slowly reduce the temperature to unity over the course of many MCMC iterations. At that point, the annealed state $\mathbf{r}_\mathrm{A}$ reached by the Markov chain is deemed a good initial guess. We throw away all previous iterations, and with $T = 1$, we run the MCMC through enough iterations to explore the posterior distribution. The simulated annealing run to obtain $\mathbf{r}_\mathrm{A}$ is called *burn in*.

During burn in, I make occasional automatic updates to the jump radius as follows:

$$J' = J\,\frac{A_J}{A_\mathrm{desired}},$$

where A_J is the acceptance ratio observed after many iterations with jump radius J, and A_desired is the desired acceptance ratio. According to conventional wisdom, $A_\mathrm{desired} \approx 0.3$ is optimal for multi-dimensional MCMC. After burn in, I do not adjust J.

My approach to setting up burn in is hands-on: I generate random scenarios with synthetic data, try MCMC on them, and tune my annealing schedule until the MCMC returns reasonably reliable answers.

Example: Locating radioactive contamination

A radiation source of activity Q, in events per second, is located at an unknown point $\mathbf{r}$. N identical detectors with effective area A are placed at locations $\mathbf{r'_n}$. The expected number of counts at detector n over time Δt is:

$$\langle d_n \rangle = \left[\frac{AQ}{4\pi(\mathbf{r}-\mathbf{r'_n})^2} + B\right]\,\Delta t, \quad n = 0, 1, 2, .., N-1;$$

where B is the background rate. Let us assume that A, $\mathbf{r'_n}$, and Δt are known. The model parameters to be determined are then $\mathbf{r}$, Q, and B. What I want to find is the posterior, which is the probability distribusion of these model parameters given the data.

Likelihoods

The measurement d_n is some number of counts, subject to Poisson noise (https://en.wikipedia.org/wiki/Poisson_distribution). The likelihood of measuring d_n counts at detector n is

$$\Pr(d_n|\mathbf{m}) = \frac{\langle d_n \rangle^{d_n} e^{-\langle d_n \rangle}}{d_n!},$$

where $\mathbf{m}$ denotes the array of model parameters. Note that d_n is an integer, but $\langle d_n \rangle$ is not. Taking into account all the detectors, the likelihood of the observed data $\mathbf{d} = [d_0, d_1, \ldots, d_{N-1}]$ is

$$\Pr(\mathbf{d}|\mathbf{m}) = \prod_n \Pr(d_n|\mathbf{m}).$$

Taking a flat prior, all we need is the likelihood ratio $\Pr(\mathbf{d}|\mathbf{r}^*, Q^*, B^*)/\Pr(\mathbf{d}|\mathbf{r}, Q, B)$, where * denotes the model parameters after a proposed jump.

In [2]:

```
Nmod = 4     # dimensionality of model (x,y,Q,B)
# Detector parameters (known)
Ndet = 6     # number of detectors
Adet = 0.01  # detector effective area
xdet = np.random.normal(size=Ndet)
ydet = np.random.normal(size=Ndet)
# Implicitly let Delta t = 1. It will be omitted from the model.

# Scale factors relating dimensionless model m[] to x,y,Q,B
s = np.array((1,1,1,1))  # scale factor array
Q0 = 1e6    # expected source activity Q
B0 = 100     # expected background B

# Define variables to hold the data and the unknowns.
# I am doing this here so the variables will be within
# the scope of the routines in this cell.
data = np.empty( (Ndet) )
x = 0.0
y = 0.0
Q = Q0
B = B0

def model(xm,ym,Qm,Bm):
    """
    Calculate array of expectation values for the data,
    given the chosen model parameters:
    xm = source x-coordinate
    ym = source y-coordinate
    Qm = source activity
    Bm = background count rate
    """
```

```
    """
    expected = np.empty((Ndet))
    for n in range(Ndet):
        expected[n] = Bm + Qm * Adet / ((xm-xdet[n])**2 + (ym-yde
    return expected

def m2xyQB(m):
    """
    Convert dimensionless model parameters to x, y, Q, B
    m = 4-element numpy array of dimensionless model parameters.
    """
    xm = s[0] * m[0]
    ym = s[1] * m[1]
    Qm = Q0 * np.exp( s[2] * m[2] )
    Bm = B0 * np.exp( s[3] * m[3] )
    return (xm,ym,Qm,Bm)

def xyQB2m(xm,ym,Qm,Bm):
    """
    Convert physical parameters x, y, Q, B to dimensionless model
    """
    m = np.empty((Nmod))
    m[0] = xm / s[0]
    m[1] = ym / s[1]
    m[2] = np.log( Qm/Q0 ) / s[2]
    m[3] = np.log( Bm/B0 ) / s[3]
    return m

def log_factorial(a):
    """
    Returns ln(a!) for integer a.
    """
    return spc.gammaln(a+1)

def log_likelihood(m):
    """
    Evaluate the log-likelihood. Since the likelihood involves pr
    small numbers, we need to use logarithnms to avoid floating u
    problems -- even in double precision.

    Parameters:
    m = array of Nmod dimensionless model parameters

    Returns: ln(Pr(m|data))
    """
    (x,y,Q,B) = m2xyQB(m)
    expected = model(x,y,Q,B)
    log_like = np.sum( log_poisson(expected, data) )
    return log_like

def log_likelihood_with_prior(m):
```

```
    """
    Evaluate the log-likelihood. Since the likelihood involves pr
    small numbers, we need to use logarithnms to avoid floating u
    problems -- even in double precision.

    This version incorporates prior information about location ba

    Parameters:
    m = array of Nmod dimensionless model parameters

    Returns: ln(Pr(m|data))
    """
    (x,y,Q,B) = m2xyQB(m)
    expected = model(x,y,Q,B)
    log_like = np.sum( log_poisson(expected, data) )
    log_like -= np.sum( (m*s)**2 / 2 ) # prior based on scenario
    return log_like


def log_poisson(expected, data):
    """
    For an arbitrary array of expectation values, and
    a data array of the same size, find the log-probability
    using the analytic form of the Poisson distribution.
    The reason for the logarithm

    Parameters:
    data = data array (integers)
    expected = array of expectation values (floats, same shape as

    Returns: Pr(data|expected)
    """
    return ( data*np.log(expected) - expected - log_factorial(dat

def proposal_gaussian(mi, dilate=1):
    """
    Propose a jump in a random direction in N-dimensional space,
    with the displacement drawn from a Gaussian distributon.
    By default, the Lorentzian has unit standard deviation.

    Parameters:
    mi = initial state, an N-dimensional numpy array.
    dilate = factor by which to increase standard deviation of ju

    Returns: final state, mf
    """
    return mi + dilate * np.random.normal(size=Nmod)

def proposal_lorentzian(mi, dilate=1):
    """
```

```
    Propose a jump in a random direction in N-dimensional space,
    with the displacement drawn from a Lorentzian distributon.
    By default, the Lorentzian has unit FWHM.

    Parameters:
    mi = initial state, an N-dimensional numpy array.
    dilate = factor by which to increase FWHM of jump distributio

    Returns: final state, mf
    """
    r2=2.
    while(r2>1):
        R = np.random.random(size=Nmod)
        r2 = np.sum(R**2)
    mhat = R/np.sqrt(r2)
    return mi + ( dilate * 0.5 * np.tan( np.pi*(np.random.random(

def metro(m0, jump_func=proposal_lorentzian, loglike_func=log_lik
          N=10000, T=1, dilate=1):
    mc = np.empty((N,Nmod))
    loglike = np.empty((N)) # can't hurt to track the likelihoods
    mc[0,:] = m0
    loglike[0] = loglike_func(m0)
    i=0
    misses=0
    for i in range(N-1):
        mc[i+1,:] = jump_func(mc[i,:], dilate=dilate) # propose a
        loglike[i+1] = loglike_func(mc[i+1,:])
        logratio = loglike[i+1] - loglike[i]
        if not(np.isfinite(loglike[i+1])):
            logratio = -1e100  # ensure rejection
        if logratio < 0:
            if (np.log(np.random.rand()) > logratio/T): # reject
                mc[i+1,:] = mc[i,:]
                loglike[i+1] = loglike[i]
                misses+=1
    return mc, (N-misses)/N, loglike
```

In [3]:
```
def confplot(x, y, LL, conf_lo=50, conf_hi=95):
    N = np.size(LL)
    n_outer = int(round((1-conf_hi/100)*N))
    n_inner = int(round((1-conf_lo/100)*N))
    ssrank = np.argsort(LL, axis=None)
    band1 = ssrank[0:n_outer]
    band2 = ssrank[n_outer:n_inner]
    band3 = ssrank[n_inner:-1]
    plt.plot(x[band1], y[band1], 'b.', label = '>'+str(conf_hi)+'%
    plt.plot(x[band2], y[band2], 'g.', label = str(conf_hi)+'-'+st
    plt.plot(x[band3], y[band3], 'c.', label = '<'+str(conf_lo)+'%


```

In [4]:
```
# Generate a random scenario (unknowns and data)
mtrue = np.random.normal(size=Nmod) / s
(x,y,Q,B) = m2xyQB(mtrue)
data = np.random.poisson(model(x,y,Q,B))
```

In [5]:
```
# Have a look at the scenario and the data.
print('source:', Q)
print('background:', B)
print('data set (counts):',data)
print('data minus background:',data-B)
```

```
source: 870078.2822752994
background: 17.622464154388375
data set (counts): [258 156 337 201 443 975]
data minus background: [240.37753585 138.37753585 319.37753585 183.37
753585 425.37753585
 957.37753585]
```

In [6]:
```
# Burn in by simulated annealing
m0=np.zeros((Nmod))
target_acceptance = 0.3
jump_radius = 3.0

for T in (10**np.arange(2,0,-0.02)):
        Niter = 2000
        (mc, accept_ratio, loglike) = metro(m0, N=Niter, T=T, dila
                    loglike_func = log_likelihood_with_prior)
        print(Niter,' iterations at T = ', T, '; jump factor = ',
              '; acceptance ratio = ', accept_ratio)
        jump_radius *= (accept_ratio / target_acceptance)
        m0 = mc[-1]


```

```
/Users/kankel/anaconda3/lib/python3.7/site-packages/ipykernel_launche
```

```
r.py:44: RuntimeWarning: overflow encountered in exp
/Users/kankel/anaconda3/lib/python3.7/site-packages/ipykernel_launche
r.py:114: RuntimeWarning: invalid value encountered in subtract

2000  iterations at T =  100.0 ; jump factor =  3.0 ; acceptance rati
o =  0.383

/Users/kankel/anaconda3/lib/python3.7/site-packages/ipykernel_launche
r.py:114: RuntimeWarning: divide by zero encountered in log
/Users/kankel/anaconda3/lib/python3.7/site-packages/ipykernel_launche
r.py:45: RuntimeWarning: overflow encountered in exp

2000  iterations at T =  95.49925860214358 ; jump factor =  3.8300000
000000005 ; acceptance ratio =  0.2905
2000  iterations at T =  91.20108393559097 ; jump factor =  3.7087166
66666667 ; acceptance ratio =  0.2965
2000  iterations at T =  87.09635899560806 ; jump factor =  3.6654483
05555556 ; acceptance ratio =  0.2955
2000  iterations at T =  83.17637711026708 ; jump factor =  3.6104665
809722225 ; acceptance ratio =  0.321
2000  iterations at T =  79.43282347242814 ; jump factor =  3.8631992
416402783 ; acceptance ratio =  0.283
2000  iterations at T =  75.85775750291836 ; jump factor =  3.6442846
179473287 ; acceptance ratio =  0.275
2000  iterations at T =  72.44359600749898 ; jump factor =  3.3405942
33118385 ; acceptance ratio =  0.288
2000  iterations at T =  69.18309709189363 ; jump factor =  3.2069704
637936494 ; acceptance ratio =  0.34
2000  iterations at T =  66.06934480075958 ; jump factor =  3.6345665
25632803 ; acceptance ratio =  0.306
2000  iterations at T =  63.0957344480193 ; jump factor =  3.70725785
6145459 ; acceptance ratio =  0.291
2000  iterations at T =  60.255958607435744 ; jump factor =  3.596040
1204610952 ; acceptance ratio =  0.273
2000  iterations at T =  57.543993733715666 ; jump factor =  3.272396
5096195973 ; acceptance ratio =  0.3275
2000  iterations at T =  54.95408738576243 ; jump factor =  3.5723661
896680605 ; acceptance ratio =  0.221
2000  iterations at T =  52.48074602497723 ; jump factor =  2.6316430
930554713 ; acceptance ratio =  0.123
2000  iterations at T =  50.118723362727195 ; jump factor =  1.078973
6681527433 ; acceptance ratio =  0.246
2000  iterations at T =  47.863009232263806 ; jump factor =  0.884758
4078852496 ; acceptance ratio =  0.2765
2000  iterations at T =  45.708818961487474 ; jump factor =  0.815452
3326009051 ; acceptance ratio =  0.2965
2000  iterations at T =  43.65158322401656 ; jump factor =  0.8059387
220538945 ; acceptance ratio =  0.277
2000  iterations at T =  41.68693834703351 ; jump factor =  0.7441500
866964293 ; acceptance ratio =  0.292
2000  iterations at T =  39.81071705534969 ; jump factor =  0.7243060
```

```
843845245 ; acceptance ratio =  0.3605
2000  iterations at T =  38.01893963205609 ; jump factor =  0.8703744
780687369 ; acceptance ratio =  0.2855
2000  iterations at T =  36.3078054770101 ; jump factor =  0.82830637
82954146 ; acceptance ratio =  0.268
2000  iterations at T =  34.67368504525313 ; jump factor =  0.7399536
979439039 ; acceptance ratio =  0.2825
2000  iterations at T =  33.11311214825908 ; jump factor =  0.6967897
322305094 ; acceptance ratio =  0.308
2000  iterations at T =  31.62277660168376 ; jump factor =  0.7153707
917566563 ; acceptance ratio =  0.293
2000  iterations at T =  30.19951720402013 ; jump factor =  0.6986788
066156676 ; acceptance ratio =  0.305
2000  iterations at T =  28.840315031266027 ; jump factor =  0.710323
4533925954 ; acceptance ratio =  0.28
2000  iterations at T =  27.542287033381633 ; jump factor =  0.662968
5564997558 ; acceptance ratio =  0.2655
2000  iterations at T =  26.30267991895379 ; jump factor =  0.5867271
72502284 ; acceptance ratio =  0.303
2000  iterations at T =  25.11886431509577 ; jump factor =  0.5925944
442273069 ; acceptance ratio =  0.3
2000  iterations at T =  23.988329190194875 ; jump factor =  0.592594
4442273069 ; acceptance ratio =  0.3025
2000  iterations at T =  22.908676527677702 ; jump factor =  0.597532
7312625345 ; acceptance ratio =  0.293
2000  iterations at T =  21.877616239495495 ; jump factor =  0.583590
3008664087 ; acceptance ratio =  0.294
2000  iterations at T =  20.892961308540364 ; jump factor =  0.571918
4948490805 ; acceptance ratio =  0.291
2000  iterations at T =  19.95262314968877 ; jump factor =  0.5547609
400036081 ; acceptance ratio =  0.284
2000  iterations at T =  19.054607179632445 ; jump factor =  0.525173
6898700823 ; acceptance ratio =  0.2905
2000  iterations at T =  18.197008586099805 ; jump factor =  0.508543
189690863 ; acceptance ratio =  0.2935
2000  iterations at T =  17.378008287493728 ; jump factor =  0.497524
75391422757 ; acceptance ratio =  0.306
2000  iterations at T =  16.595869074375578 ; jump factor =  0.507475
2489925122 ; acceptance ratio =  0.328
2000  iterations at T =  15.84893192461111 ; jump factor =  0.5548396
055651468 ; acceptance ratio =  0.271
2000  iterations at T =  15.135612484362056 ; jump factor =  0.501205
110360516 ; acceptance ratio =  0.298
2000  iterations at T =  14.45439770745925 ; jump factor =  0.4978637
429581125 ; acceptance ratio =  0.2855
2000  iterations at T =  13.803842646028825 ; jump factor =  0.473800
32871513705 ; acceptance ratio =  0.2835
2000  iterations at T =  13.182567385564047 ; jump factor =  0.447741
31063580447 ; acceptance ratio =  0.299
2000  iterations at T =  12.58925411794165 ; jump factor =  0.4462488
```

```
396003518 ; acceptance ratio =  0.277
2000  iterations at T =  12.022644346174106 ; jump factor =  0.412036
4285643249 ; acceptance ratio =  0.3095
2000  iterations at T =  11.481536214968806 ; jump factor =  0.425084
2488021952 ; acceptance ratio =  0.288
2000  iterations at T =  10.964781961431829 ; jump factor =  0.408080
87885010735 ; acceptance ratio =  0.286
2000  iterations at T =  10.471285480508975 ; jump factor =  0.389037
104503769 ; acceptance ratio =  0.3075
2000  iterations at T =  9.999999999999979 ; jump factor =  0.3987630
3211636327 ; acceptance ratio =  0.2895
2000  iterations at T =  9.549925860214339 ; jump factor =  0.3848063
2599229053 ; acceptance ratio =  0.3315
2000  iterations at T =  9.120108393559079 ; jump factor =  0.4252109
902214811 ; acceptance ratio =  0.303
2000  iterations at T =  8.709635899560787 ; jump factor =  0.4294631
0012369593 ; acceptance ratio =  0.266
2000  iterations at T =  8.317637711026691 ; jump factor =  0.3807906
154430104 ; acceptance ratio =  0.2845
2000  iterations at T =  7.943282347242797 ; jump factor =  0.3611164
336451215 ; acceptance ratio =  0.304
2000  iterations at T =  7.585775750291821 ; jump factor =  0.3659313
1942705653 ; acceptance ratio =  0.291
2000  iterations at T =  7.244359600749884 ; jump factor =  0.3549533
7984424485 ; acceptance ratio =  0.3
2000  iterations at T =  6.918309709189349 ; jump factor =  0.3549533
7984424485 ; acceptance ratio =  0.288
2000  iterations at T =  6.606934480075944 ; jump factor =  0.3407552
44650475 ; acceptance ratio =  0.294
2000  iterations at T =  6.309573444801917 ; jump factor =  0.3339401
397574655 ; acceptance ratio =  0.281
2000  iterations at T =  6.025595860743563 ; jump factor =  0.3127905
9757282607 ; acceptance ratio =  0.2715
2000  iterations at T =  5.754399373371554 ; jump factor =  0.2830754
9080340766 ; acceptance ratio =  0.313
2000  iterations at T =  5.495408738576232 ; jump factor =  0.2953420
954048887 ; acceptance ratio =  0.273
2000  iterations at T =  5.248074602497712 ; jump factor =  0.2687613
068184488 ; acceptance ratio =  0.3
2000  iterations at T =  5.01187233627271 ; jump factor =  0.26876130
68184488 ; acceptance ratio =  0.3045
2000  iterations at T =  4.786300923226371 ; jump factor =  0.2727927
2642072555 ; acceptance ratio =  0.291
2000  iterations at T =  4.570881896148737 ; jump factor =  0.2646089
446281038 ; acceptance ratio =  0.2995
2000  iterations at T =  4.365158322401648 ; jump factor =  0.2641679
2972039027 ; acceptance ratio =  0.2865
2000  iterations at T =  4.168693834703342 ; jump factor =  0.2522803
728829727 ; acceptance ratio =  0.283
2000  iterations at T =  3.981071705534961 ; jump factor =  0.2379844
```

```
850862709 ; acceptance ratio =  0.298
2000  iterations at T =  3.801893963205601 ; jump factor =  0.2363979
218523624 ; acceptance ratio =  0.3085
2000  iterations at T =  3.630780547701003 ; jump factor =  0.2430958
6297151267 ; acceptance ratio =  0.2895
2000  iterations at T =  3.467368504525306 ; jump factor =  0.2345875
0776750972 ; acceptance ratio =  0.2805
2000  iterations at T =  3.311311214825901 ; jump factor =  0.2193393
1976262164 ; acceptance ratio =  0.2965
2000  iterations at T =  3.1622776601683698 ; jump factor =  0.216780
3610320577 ; acceptance ratio =  0.2985
2000  iterations at T =  3.0199517204020068 ; jump factor =  0.215696
4592268974 ; acceptance ratio =  0.2755
2000  iterations at T =  2.884031503126597 ; jump factor =  0.1980812
4839003413 ; acceptance ratio =  0.3385

2000  iterations at T =  2.754228703338158 ; jump factor =  0.223501
7526675518 ; acceptance ratio =  0.268
2000  iterations at T =  2.6302679918953733 ; jump factor =  0.19966
49657163465 ; acceptance ratio =  0.3105
2000  iterations at T =  2.5118864315095717 ; jump factor =  0.20664
6489516419 ; acceptance ratio =  0.2795
2000  iterations at T =  2.3988329190194824 ; jump factor =  0.19252
58960661304 ; acceptance ratio =  0.302
2000  iterations at T =  2.290867652767765 ; jump factor =  0.193812
1353732378 ; acceptance ratio =  0.297
2000  iterations at T =  2.187761623949545 ; jump factor =  0.191873
9240195055 ; acceptance ratio =  0.289
2000  iterations at T =  2.089296130854032 ; jump factor =  0.184838
126805457 ; acceptance ratio =  0.296
2000  iterations at T =  1.9952623149688726 ; jump factor =  0.18237
09784480507 ; acceptance ratio =  0.2825

2000  iterations at T =  1.9054607179632406 ; jump factor =  0.17173
6088038581 ; acceptance ratio =  0.31
2000  iterations at T =  1.819700858609977 ; jump factor =  0.177460
2909732006 ; acceptance ratio =  0.271
2000  iterations at T =  1.7378008287493691 ; jump factor =  0.16030
6499512458 ; acceptance ratio =  0.303
2000  iterations at T =  1.6595869074375547 ; jump factor =  0.16190
70645075825 ; acceptance ratio =  0.3095
2000  iterations at T =  1.5848931924611076 ; jump factor =  0.16703
8154883656 ; acceptance ratio =  0.286
2000  iterations at T =  1.5135612484362024 ; jump factor =  0.15924
8107655752 ; acceptance ratio =  0.2625
2000  iterations at T =  1.4454397707459221 ; jump factor =  0.13933
70941987833 ; acceptance ratio =  0.326
2000  iterations at T =  1.3803842646028797 ; jump factor =  0.15141
47090293444 ; acceptance ratio =  0.2895
2000  iterations at T =  1.3182567385564021 ; jump factor =  0.14611
06942133173 ; acceptance ratio =  0.291
```

```
2000  iterations at T =  1.2589254117941624 ; jump factor =  0.14172
70733869178 ; acceptance ratio =  0.2885
2000  iterations at T =  1.2022644346174083 ; jump factor =  0.13629
77355737527 ; acceptance ratio =  0.318
2000  iterations at T =  1.1481536214968782 ; jump factor =  0.14447
51997081778 ; acceptance ratio =  0.2785
2000  iterations at T =  1.0964781961431807 ; jump factor =  0.13411
5843729092 ; acceptance ratio =  0.286
2000  iterations at T =  1.0471285480508954 ; jump factor =  0.12786
67043550675 ; acceptance ratio =  0.3
```

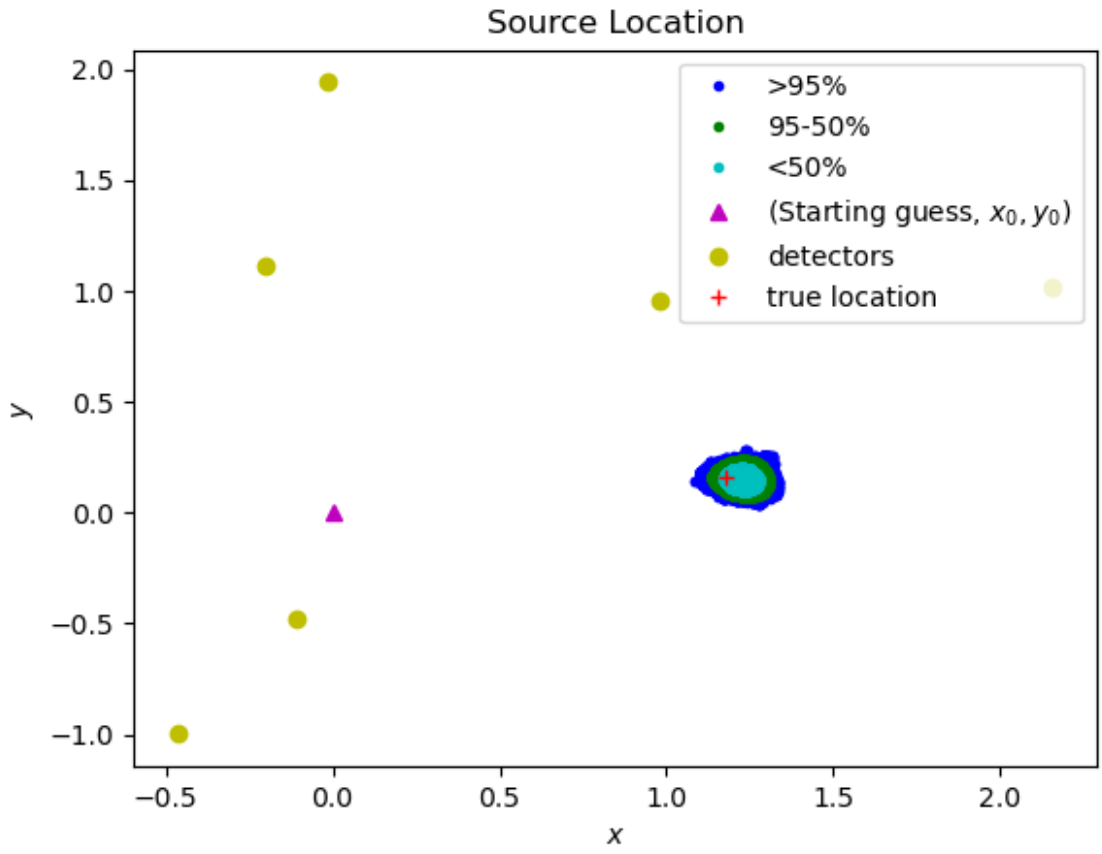
In [7]:

```
# Run MCMC from where we left off after burn-in.
m0 = mc[-1]
# m0 = xyQB2m(.73,1.24,7.1e6,23.4)
(mc, accept_ratio, loglike) = metro(m0, N=810000, dilate=jump_radi
                loglike_func = log_likelihood_with_prior)

# Convert mc back to physical variables
Nmc = mc.shape[0]
xmc = np.empty((Nmc))
ymc = np.empty((Nmc))
Qmc = np.empty((Nmc))
Bmc = np.empty((Nmc))
for i in range(Nmc):
    (xmc[i],ymc[i],Qmc[i],Bmc[i]) = m2xyQB(mc[i])
```

```
/Users/kankel/anaconda3/lib/python3.7/site-packages/ipykernel_launche
r.py:44: RuntimeWarning: overflow encountered in exp
/Users/kankel/anaconda3/lib/python3.7/site-packages/ipykernel_launche
r.py:45: RuntimeWarning: overflow encountered in exp
/Users/kankel/anaconda3/lib/python3.7/site-packages/ipykernel_launche
r.py:114: RuntimeWarning: invalid value encountered in subtract
/Users/kankel/anaconda3/lib/python3.7/site-packages/ipykernel_launche
r.py:114: RuntimeWarning: divide by zero encountered in log
```

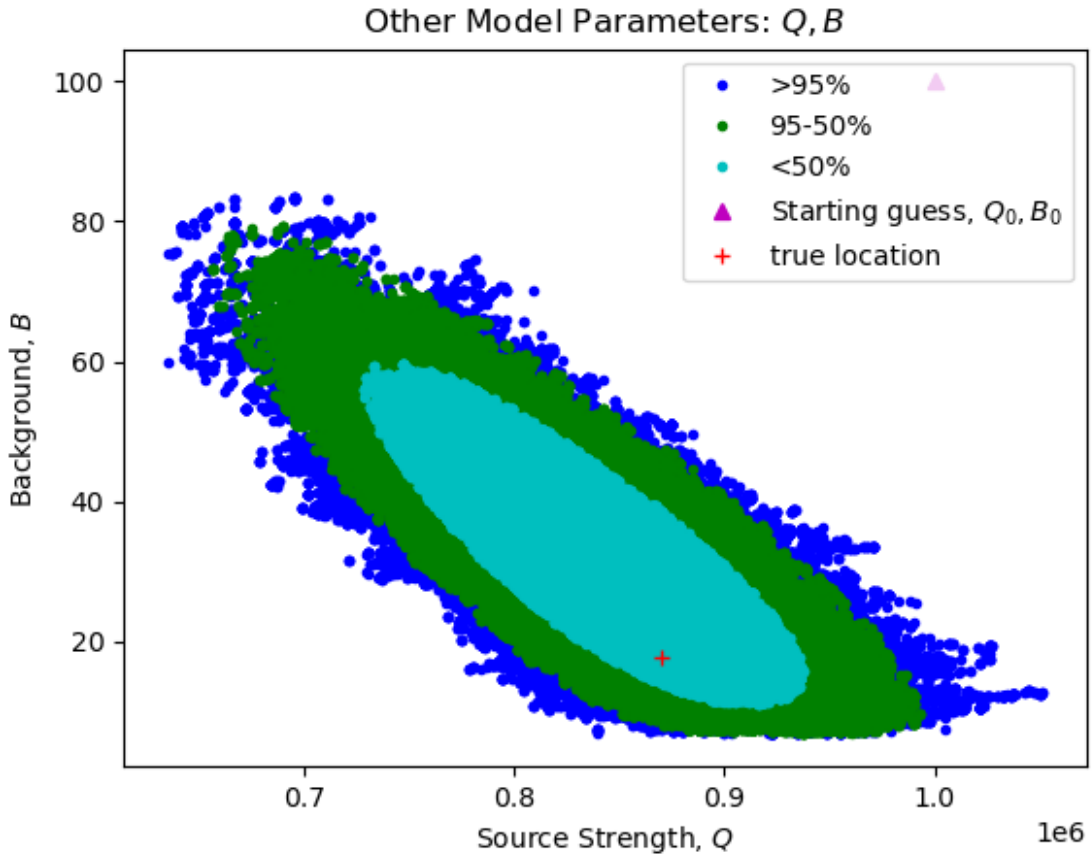
In [8]:

```
plt.figure()
confplot(xmc,ymc,loglike)
plt.plot(0,0,'m^', label='(Starting guess, $x_0, y_0$)')
plt.plot(xdet,ydet,'yo', label='detectors')
plt.plot(x,y, 'r+', label='true location')
plt.title('Source Location')
plt.xlabel('$x$')
plt.ylabel('$y$')
plt.legend()
plt.show();


```



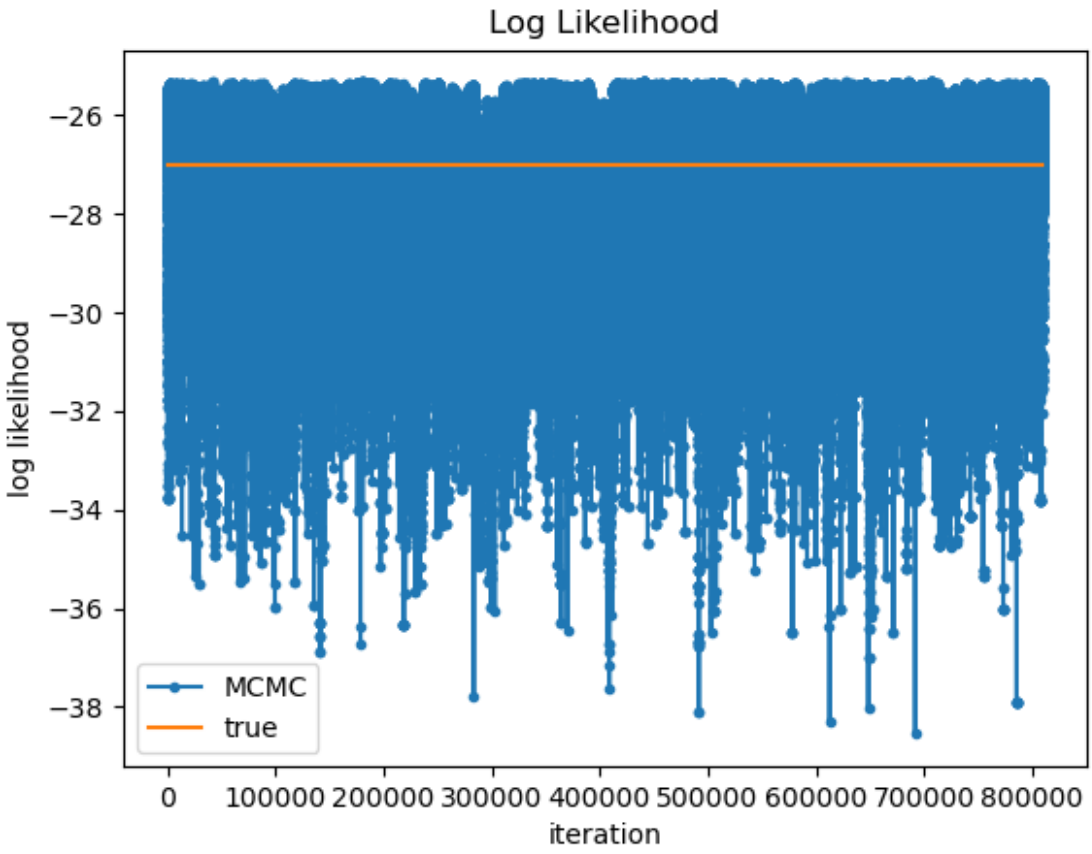
In [9]:

```
plt.figure()
confplot(Qmc,Bmc, loglike)
plt.plot(Q0,B0,'m^', label='Starting guess, $Q_0, B_0$')
plt.plot(Q,B, 'r+', label='true location')
plt.title('Other Model Parameters: $Q,B$')
plt.xlabel('Source Strength, $Q$')
plt.ylabel('Background, $B$')
plt.legend()
plt.show();


```



In [10]:
```
print('acceptance ratio = ',accept_ratio)
plt.figure()
plt.plot(loglike,'.-',label='MCMC')
plt.plot(np.array((0,loglike.size)),log_likelihood(mtrue)*np.array
plt.title('Log Likelihood')
plt.xlabel('iteration')
plt.ylabel('log likelihood')
plt.legend()
plt.show()

```

```
acceptance ratio =  0.2931493827160494
```



Comment

I have put considerable effort into trying to ensure the MCMC fully explores the posterior distribution. To this end, I have developed:

- Lorentzian jumps
- “isotropized”, dimensionless coordinates[†]
- burn in by simulated annealing

Essentially, I've tried to give MCMC a leg up on exploring the posterior while letting it be *Markovian*. Recall that a Markov chain $\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3, \ldots$ has the defining property:

$$\Pr(\mathbf{r}_n | \mathbf{r}_{n-1}, \mathbf{r}_{n-2}, \mathbf{r}_{n-3}, \ldots) = \Pr(\mathbf{r}_n | \mathbf{r}_{n-1}).$$

In words, the Markov chain should have no memory of its past. This property was closely related to the concept of detailed balance, which allows Metropolis-Hastings to sample from the posterior distribution, which in turn is the goal of MCMC. Notice that simulated annealing and my jump radius updates during burn in do not satisfy this condition; that is why I discard everything from burn in except the initial state $\mathbf{R}_\mathrm{A}$ before restarting MCMC.

The state of the art is a strategem called *parallel tempering* (https://en.wikipedia.org/wiki/Parallel_tempering), in which a collection of MCMC simulations run in parallel at different temperatures. Here, temperature is defined in the same way as it was for simulated annealing. These parallel chains trade information and work together to explore the posterior intelligently. The sequence of collective states for this system is Markovian because those information exchanges are triggered and implemented solely on the basis of present conditions. Parallel tempering is an efficient, automatic approach to exploring the posterior without restarts.

[†] I use scare quotes because, as it turns out, strong correlations between parameters often emerge. These constitute important localized anisotropies (prominent diagonal ridges) in the posterior distribution, which are not anticipated by my dimensionless coordinates. MCMC does not explore these narrow ridges optimally.