Guide to Ensemble Monte Carlo Fitting

What is a Monte Carlo?

Monte Carlo simulations, named after the famous casino in Monaco, are simulations whereby parameters are not analytically evaluated, but rather statistically estimated by conducting repeated sampling of statistical distributions. These needn't necessicarily be computerized, although modern computation allows for millions of samples in short order.

One classic "analogue" Monte Carlo involves the estimation of the relative areas of two arbitrary 2-D shapes (often done with a circle and a square of equal radius, thus allowing one to derive pi). By drawing these shapes to the same scale upon a piece of paper and then scattering grains of rice across the page, counting the ratio of the number of grains of rice which come to rest within the bounds of each shape will, over repeated trials, converge to the ratio of the areas of the shapes, assuming random and consistent scattering of rice grains.

What is MCMC?

The Markov Chain Monte Carlo (MCMC) is the basic working method of this fitting program. By definition, a Markov chain is a numerical chain where the generation of the n^{th} element is dependent only upon the $(n-1)^{th}$ element, and not any of the other elements in the chain. This is advantageous for us, as it means the numerical computation of each step in our Monte Carlo requires only the information contained in the previous step, drastically reducing computational requirements.

Basics of Bayesian Probability Theory

The most basic element of Bayesian probability theory is Bayes' Theorem:

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}$$

Where:

- P(A|B) is "Probablity of A, Given B to be true"
- P(A|B) is called the "posterior"
- P(A) and P(B) are the "prior" distributions for A and B, respectively. P(B) is often left out (assumed to be 1) and the function is normalized later.
- P(B|A) is called the "likelihood"

Within this fitting routine, our implementation of Bayes' Theorem looks like:

$$P(\text{Model} \mid \text{Data}) = P(\text{Data} \mid \text{Model})P(\text{Model})$$

Here, I have decided to use the reduced Chi Squared statistic,

$$\chi^2_{reduced} = \frac{\chi^2}{\nu - 1}$$

and

$$\chi^2 = \Sigma \frac{(O_i - M_i)^2}{\sigma_i}$$

with O_i M_i and σ_i being the i^{th} observed data point, modeled data point, and standard deviation, respectively.

For our data, since we are in effect "counting" the discrete number of photons which fall upon each CCD pixel in a given time interval, the standard deviation of each pixel is given by the Poisson distribution. For a sufficiently large number of incident photons, *n*, this standard deviation converges towards $\sqrt{n}$. Typically, *n* >25 is sufficiently large to make this approximation. Here, n typically lies in the range $1000<n<15000$.

In this script, we're actually using log(prior), log(likelihood), and log(posterior). This choice is made for two reasons: 1) dealing with small decimal numbers, floating point rounding errors can become non-trivial within our probability space. Moving to log space fixes this. 2) Our Posterior, which in linear space would be the product of our likelihood function and our prior, now becomes the sum of these two functions. This simplifies computation (slightly) and makes our posterior more human-readable.

For our log prior, I chose a uniform distribution which returns 0 {log(1) = 0} for a specified range of values and negative infinity otherwise. Since we are attempting to maximize our log posterior, this allows us to drastically narrow the search space for our simulation by defining a plausible value for each fitting parameter and constrain our posterior to only these values (since the maximization routine will obviously avoid exploring the range of space where the prior is valued at minus infinity).

Introduction: Importing Data

Data importing is handled automatically by the script once the folder directory containing the data. One thing to note: the automatic file naming scheme in CamWare is editable. Users can specify how many leading zeros should be written before the trial number. (i.e. the choice of 'data_005.dat' vs. 'data_0005.dat'). Currently this script is set up to parse data file names with three-digit data numbers. This can be easily changed in the future though.

The variable `binfactor` is a user-specified base two number which bins the data from the CCD camera down to a smaller resolution while preserving the shape of the data. Currently analyzing the effects of binning on computation time and accuracy are on the top of my to-do list.

Ensemble MCMC package within Python

This fitting program makes use of the (ingeniously named) ensemble MCMC Python package "emcee Hammer," as well as some basic Bayesian statistical techniques. The emcee package is not included in the default Anaconda install which most users will have on their machine. To install, simply open the terminal and type "pip install emcee". Other installation methods exist, and are detailed on the dev webpage for emcee, but pip install should work for most every user.

In this program, I've decided on using uniform priors, which do not weight the exploration of probability space other than excluding values. These can be produced by quickly looking at a 2D plot of the data. As I will illustrate later, this simulation is quite robust for even very broad priors, so getting hyper-accurate initial guesses is unnecessary.

Multiplying this prior by the likelihood function (here, adding, since we are working with logarithms), we now not only have a "goodness of fit" statistic, but a range of values (constrained by our prior) across which we want to "search" for the optimal parameters. This is where ensemble MCMC is most effective; it offers the ability to quickly and efficiently search through a parameter space without having to spend the computational resources to fully evaluate the space at every point. This Monte Carlo program can evaluate a series of our data in approximately one tenth the time that the orthogonal data regression package `scipy.odr` can do the same.

In this program, we initialize a number of walkers, given by `NWALKERS`. These walkers are given initial position values within our n-dimensional space, defined by our initial human-generated guesses.

From there, we add a small random perturbation in each dimension to each walker, to ensure that the walkers quickly spread out and fully explore the parameter space.

At each step, each walker randomly chooses a point within an n-sphere of radius *r* of its current location. It then computes the LogPosterior at that location. If the Posterior at the new location is better than at its current location, it then jumps to the new location. If it is not, the walker then chooses a random number twice. If the first number is lower than the second, the walker jumps to the new location as well. This is to prevent the walker from getting stuck within small local minima in the

probability space. Emcee hammer also includes a biasing function so that the jump directions of individual walkers are weighted towards the positions of other walkers which have significantly better LogPosteriors. This is, in effect, the walkers "talking" to one another as they explore the parameter space.

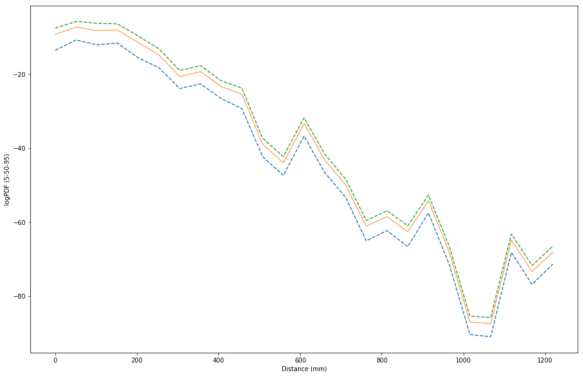
We then execute the MCMC. I've set the number of iterations to 500 by default, but this can be changed in the execution of `sampler.run_mcmc` by changing the `NUMSIMS` argument.

Notes on the General Structure of this Jupyter Notebook:

Under each sub-heading, there are a variable number of cells. If there are user-changeable values within that sub-section, such as changing the number of walkers or changing the number of simulations, these values will be instantiated in the first cell in the sub-section. This decision was made in order to make the notebook more user-friendly and help keep track of variables.

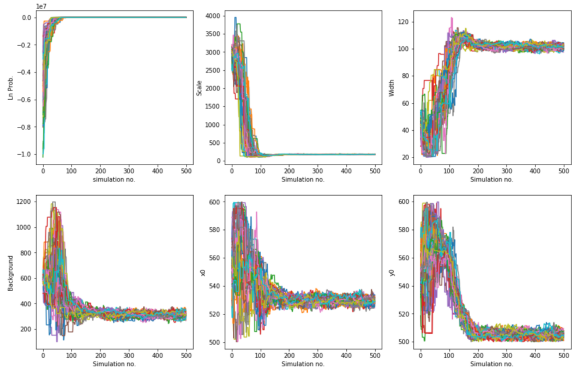
Analyzing the Results

The Monte Carlo routine has cells for printing graphs of beam width and log posterior as a function of propagation distance. By default, they print a 95% interval for the log posterior in dotted lines, with the median as a solid line:

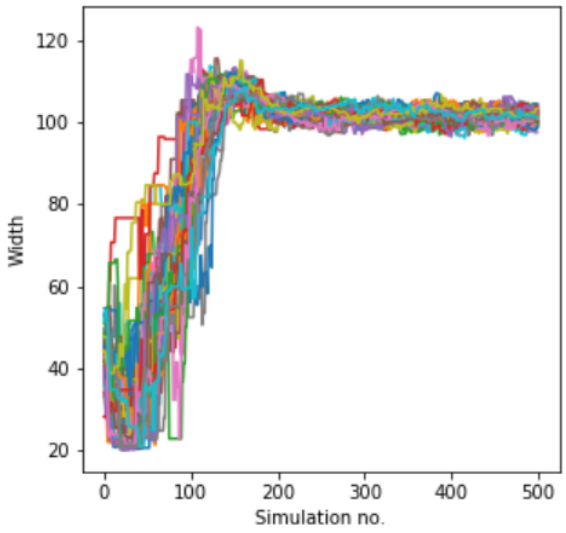


This allows one to quickly look at the entire data run, albeit on a superficial level. However, if one wishes to look in depth at any individual data points, there is a series of cells which allow one to do exactly that. Each cell is labeled with a header describing its function. First, one must choose the index of the image one wishes to examine in detail. From there, everything else is automatic.

The first cell is for graphing the log posterior of the walkers, as well as graphically representing the walkers' exploration of the five dimensional fitting space. Each dimension is graphed separately versus the simulation index number. The varied colors represent different walkers.

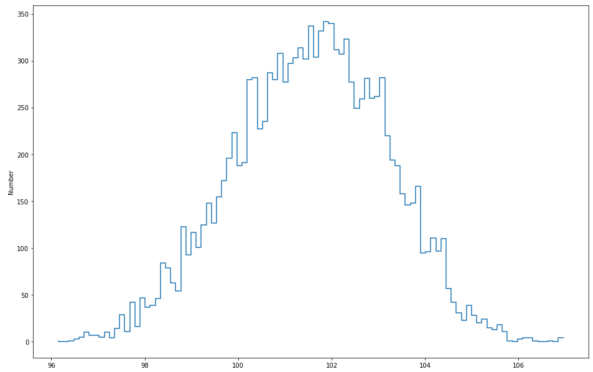


Let's focus on a single one of these graphs to better understand what's going on:



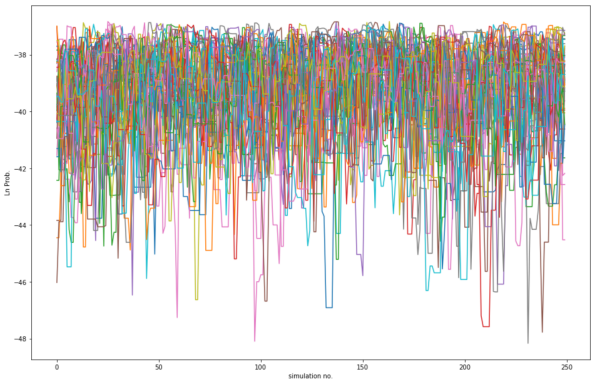
Here we see the 50 walkers are instantiated randomly across a range of 20-60. However, they converge after approximately 200 simulations to a value of approximately 105. Note that despite nearly 300 further simulations after the walkers do not converge to a single value. This results in a

"width" in the y-axis which corresponds to the uncertainty of our simulation. We can actually graph this as a histogram to better visualize it. It is common practice in Monte Carlo simulations to have a certain number of simulations as a "burn-in" interval. Within this interval, the simulation has not yet converged. By removing the "burn-in" range and evaluating only the simulations where the walkers have converged, we can gather meaningful information about our system and our model. Here, I have set the default burn-in to be 250 simulations. Again, this is editable via the `BURNIN` argument.



This histogram illustrates the strong convergence of the walkers upon a central value. This histogram was created on data binned by a factor of 4. For more highly binned data, the standard deviations of these histograms grows proportional to the square of the binning factor.

Here is a zoomed in image of the log posterior after excluding the burn-in values. By taking the median of these values, we can get our purity statistic. This statistic is graphed as a function of distance on the previous page.



Planned Future Additions:

Test Binning Factor – run trials with multiple modes at multiple binfactor settings to confirm that binning data is allowable and isn't corrupting the results.

Fitting multiple modes to fit systematic errors (a la 2002 paper) – architecture written, still waiting to talk to Dr. Abraham to discuss exactly how it should be implemented (which modes? Cap the contribution of non-primary mode? Etc.)

Addition of Radial asymmetry term (isn't it L nodes?) – Unwritten, sort of unsure how to go about that? Maybe add an A*sin(L*pi*theta) term to the model?

Save graphs/processed data to file – not written, but relatively simple. Need to decide exactly which data features are important to save and which are just of use diagnostically for the routine itself. (maybe save it all? Honestly we're only talking about maybe 2Mb for an entire data run?)

User prompt for folder file tree for data instead of string written into cell: Wouldn't take that long to do, but it's low priority right now.