MCMC in Action: Applying the Metropolis-Hastings Algorithm to a Two Dimensional Gaussian

Dakshina Scott

November 28, 2013

Abstract

Rejection sampling and the Metropolis-Hastings algorithm were both implemented in the Python programming language and applied to the univariate Gaussian probability distribution for a parameter in a toy problem. Comparing these results the benefits of the latter algorithm were seen, and thus it was used to generate samples from the bivariate distribution for another toy problem with two unknown parameters. The samples were used, via maximum likelihood estimates, to find Monte Carlo estimates for the parameters of the distribution, which were then compared to those found analytically. The results were found to match well, with an analytical solution of $\theta = \begin{bmatrix} -0.012 \\ 1.329 \end{bmatrix}$ and a Monte Carlo estimate $\theta_{MC} = \begin{bmatrix} -0.011 \\ 1.326 \end{bmatrix}$, suggesting that the algorithm performed well.

Contents

1 Background

Markov Chain Monte Carlo (MCMC) can be used to solve problems which would otherwise not be solvable - such as intractable integrations or sampling from complicated multivariate probability distributions.

While the idea of Monte Carlo simulations has been around for much longer, MCMC has flourished since the rise of computers allowed much larger simulations. It was originally developed by Metropolis et al. at Los Alamos in 1953 to investigate the equation of state for substances consisting of individual interacting molecules [1]. Today Markov Chain Monte Carlo methods are used for many applications in physics and particularly statistical mechanics, for example in simulating the Ising model[2].

1.1 Monte Carlo Methods

Monte Carlo methods use random numbers to solve problems. A Monte Carlo method may be more specifically defined as "representing the solution of a problem as a parameter of a hypothetical population, and using a random sequence of numbers to construct a sample of the population, from which statistical estimates of the parameter can be found" [3].

A very simple example is a Monte Carlo estimate for the value of π. Assume we have a circle of radius one, contained exactly within a square ranging [-1, 1]. The probability of random points from a uniform distribution within this range landing in the circle is given by:

$$P(inside) = \frac{\text{area of circle}}{\text{area of square}} = \frac{\pi}{4}. \tag{1}$$

As there are only two possible outcomes for each simulation - a point lands either inside or outside of the circle - a population of N points can be described by a binomial distribution in which a 'success' is a point landing within the circle:

$$I \sim \mathcal{B}(N, \theta) \tag{2}$$

where $\theta = P(\text{inside})$, and I is the number of successes. Thus we have represented the solution of our problem as a parameter of a hypothetical population, that is a binomial population with unknown probability of success. We can estimate θ using its maximum-likelihood estimate [4], based on the number of success we observed in our random sampling:

$$\theta = \frac{I}{N}, \tag{3}$$

So a Monte Carlo estimate for π is given by

$$\pi_{MC} = 4\theta = 4\frac{I}{N}. \tag{4}$$

1.2 Rejection Sampling

Rejection sampling is a particular type of Monte Carlo method which can be used if the target distribution can be evaluated, at least to within a normalization constant.

A random number r is generated from some proposal distribution, $Q(\theta)$. A corresponding random number is generated from a uniform distribution in the range $[0, Q(\theta = r)]$, representing a value on the y-axis. Both the proposal distribution and the target distribution, $P(\theta|x)$, are evaluated at this value. The probability of 'accepting' the sample is given by $\frac{P(\theta=r|x)}{Q(\theta=r)}$ - in practice this is implemented by accepting the sample if $y < P(\theta = r|x)$, and rejecting otherwise. The accepted points are effectively a series of samples from the target distribution. From these samples using the maximum-likelihood estimates for mean and variance gives the Monte Carlo estimates for said quantities. It is important that $Q(\theta) > P(\theta|x)$ for every theta value. For simplicity, a uniform distribution is often used.

1.3 Markov Chains & The Metropolis-Hastings Algorithm

Markov chains describe the probability of transitions between different states in a system. Specifically, for a sequence to be a Markov chain, the probability of transitioning to a state must depend only on the current state and not on any previous states.

In Markov Chain Monte Carlo a Monte Carlo method is used where the sequence of states is a Markov Chain. There are a number of algorithms which achieve this (see, for example, Gibbs sampling). Here we have used the Metropolis-Hastings algorithm, in which the next state is given by a proposal distribution, similar to that described above. However, in this case the proposal distribution is always centred on the current state. This results in a Markov Chain with the target distribution as its equillibrium distribution.

2 Univariate Target Distribution

Here we use a toy problem based on set of measurements of size $N = 10$ shown in appendix G, with sample mean $\bar{x}$ and variance $\sigma^2 = 0.1$. These data are randomly generated from a Gaussian distribution, and are used in place of actual experimental results to update our knowledge about a distribution from the prior to the posterior. We use a Gaussian prior with mean $\mu_{prior} = 0$ and variance $\Sigma^2 = 1.0$.

2.1 Analytical Solution

In this case we have a simple Gaussian prior and likelihood, for which it can be shown that the resulting posterior is also a Gaussian (see appendix B). Because we know the form of the equation for a gaussian distribution, it can be seen that the posterior mean and standard deviation are given by:

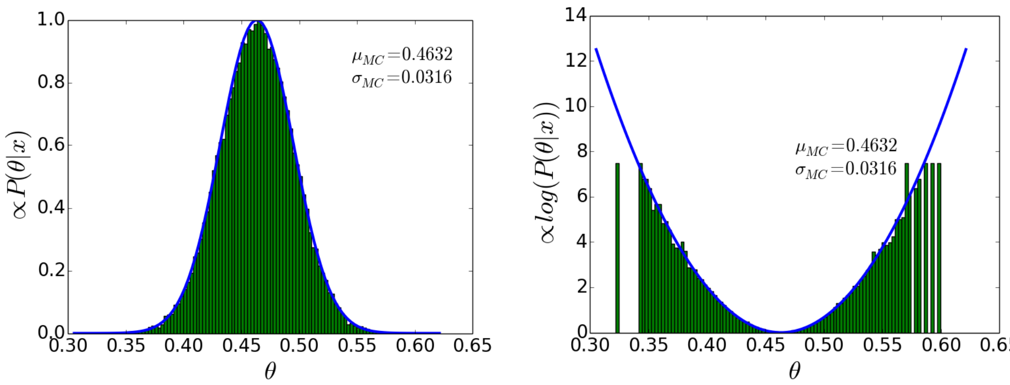
$$\mu_{post} = \frac{\Sigma^2}{\sigma^2/N + \Sigma^2}\bar{x} = 0.463, \tag{5}$$

$$\sigma_{post} = \left(\frac{1}{\Sigma^2} + \frac{N}{\sigma^2}\right)^{-\frac{1}{2}} = 0.032. \tag{6}$$

This analytical solution gives us something to which our MCMC results can be compared. However, in real world applications of MCMC this of course would not be available.

2.2 Rejection Sampling

First a simple Monte Carlo method - rejection sampling - was applied to our toy problem.



(a) The histogram in green represents the distribution found by rejection sampling. The Monte Carlo mean and standard deviation are found to be μ_{MC} = 0.463 and μ_{MC} = 0.032. The blue line is the analytical distribution.

(b) A log-plot makes it easier to see the discrepencies at the extremities of the plot. These are due to the finite domain over which samples are taken using this method.

Figure 1

In figure 1a, the results appear to fit the analytical curve well. However, plotting the logarithm of the results allows us to see clearly differences at the edges of our Monte Carlo sample - this is because we can't take samples over an infinite domain, and in this algorithm we must choose definite cut-off points. The wider the domain the smaller the effect of this will be, but the number of rejected candidate points will be larger. As there always has to be a cut-off somewhere, this is an area where the Metropolis-Hastings algorithm will be more effective.

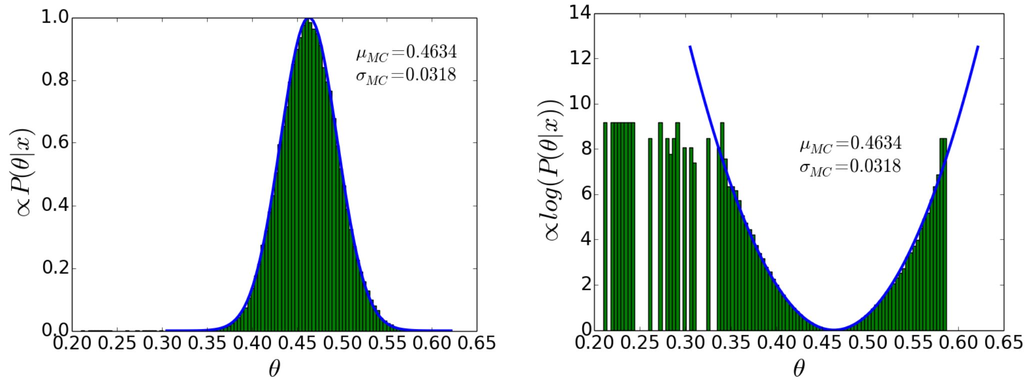
2.3 Metropolis-Hastings Algorithm

The Metropolis-Hastings algorithm was then applied to the same problem, using a Gaussian proposal distribution.

It can be seen from figure 2b that there are some samples which are clear outliers from the distribution. This is due to a poor starting value for the chain. Starting far from the high-probability parts of the distribution results in a disproportionate number of extreme values being accepted in the early iterations. These early samples are sometimes discarded as burn-in. There is some debate over how best to determine the length of the burn-in period, and indeed whether a burn-in should be used at all [7]. A simple qualitative approach for determining the length of the burn-in would be to run multiple chains from very different starting positions. When these chains meet, one would expect the chains to have converged to the target distribution and so all previous points can be discarded.

While this approach provides some evidence that the chains are converging regardless of starting position, it should not be regarded as definitive as not all starting positions can be tested. For some multi-peak distributions it is conceivable that multiple chains with very different initial states may become stuck in the same area for some time.

In figure 3 we see three very different starting θ values which appear to converge after about 200 iterations, so we take 200 iterations as the burn-in period, as is seen in figure 4. In figure 4b it appears that altough closer than the rejection sampling case, there are still some issues at extreme θ values. It can be seen that the histogram slightly overshoots the analytical plot at the edges - as



(a) The histogram in green represents the distribution found by the Metropolis-Hastings algorithm. The Monte Carlo mean and standard deviation are found to be μ_{MC} = 0.463 and σ_{MC} = 0.032. The blue line is the analytical distribution.

(b) A log-plot of the Metropolis-Hastings results shows that the sample is more consistent with the analytical solution, as compared with figure 1b.

Figure 2

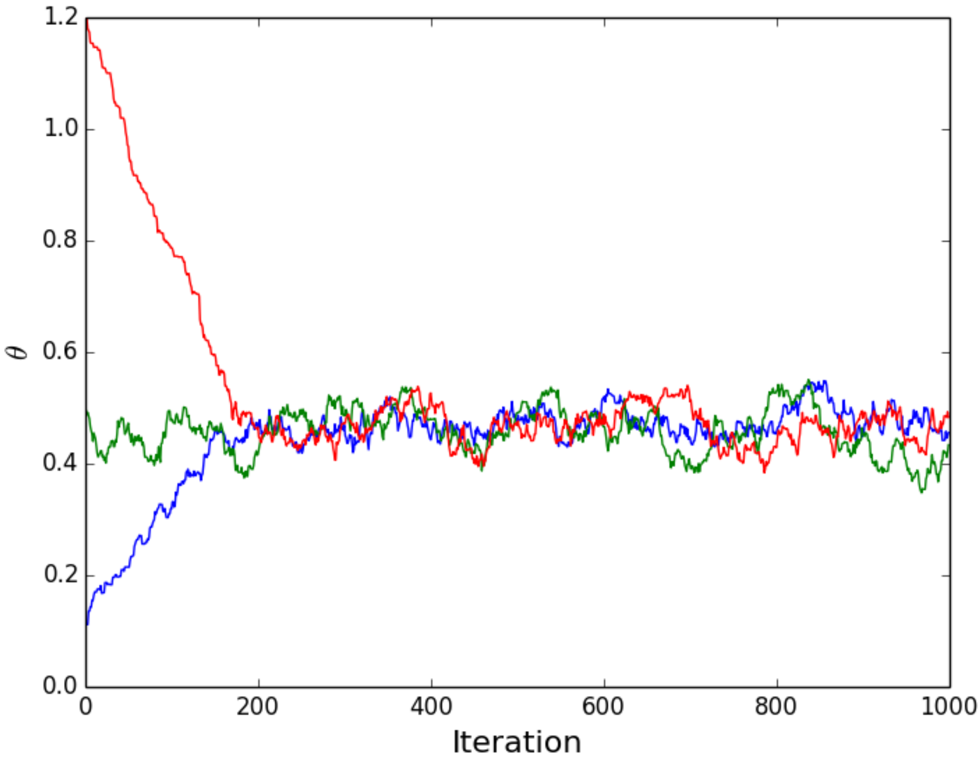
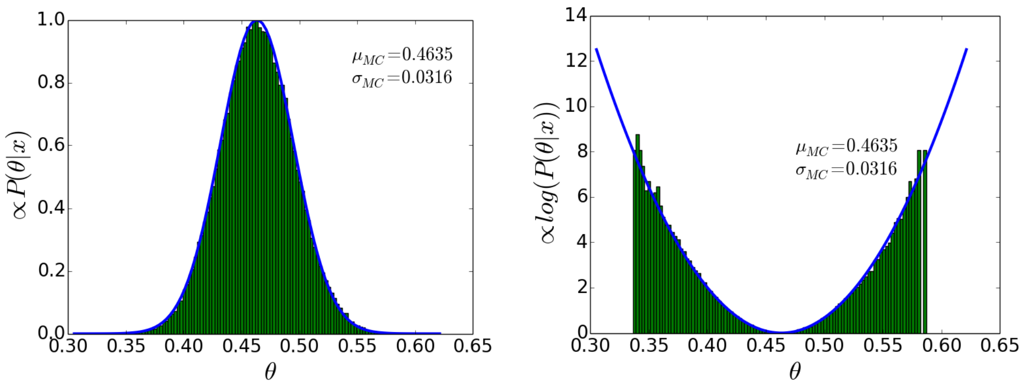


Figure 3: Starting the chain from three very different states gives us an idea of how the chain converges. We see the chains meet after about 200 iterations, suggesting that the chain converges around here.

this is a log plot this suggests that the more extreme values are under-represented in our sample. PROPOSAL SIGMA/ACCEPTANCE RATE ??? MC SIGMA INDEPENDENT OF PROPOSAL SIGMA



(a) The results from the Metropolis-Hastings algorithm with the first 200 iterations removed as burn-in.

(b) A log plot of the Metropolis-Hastings results with burn-in removed. Although the plot appears different, μ remains the same to 3 significant figures, and the standard deviation is also unaffected at this precision.

Figure 4

2.4 Analytical vs Rejection vs MH

3 Bivariate Target Distribution

3.1 Analytical Solution

Consider the linear model

$$y = F\theta + \epsilon \tag{7}$$

where y is a vector containing our data, θ is a vector of unkown parameters, F is the design matrix and ϵ is a vector containing the noise. If we assume that the noise is randomly gaussian distributed with zero mean and zero correlation, then the likelihood function can be shown to take the form

$$p(y|\theta) = \mathcal{L}_0 \exp\left[-\frac{1}{2}(\theta - \theta_0)^t L(\theta - \theta_0)\right], \tag{8}$$

where L is the likelihood fisher matrix, $\mathcal{L}_1$ is a constant, and θ_0 is dependent on L and constants related to the linear model above (see appendix E).

If we then also say we have a Gaussian prior with zero mean and Fisher matrix P, i.e.

$$p(\theta) = \frac{|P|^{1/2}}{(2\pi)^{n/2}} \exp\left[\frac{1}{2}\theta^t P\theta\right], \tag{9}$$

then using Bayes theorem the posterior can be shown to follow

$$p(\theta|y) \propto \exp\left[-\frac{1}{2}(\theta - \bar{\theta})^t \mathcal{F}(\theta - \bar{\theta})\right], \tag{10}$$

where $\mathcal{F} = L + P$ and $\bar{\theta} = \mathcal{F}^{-1}L\theta_0$ (see appendix F). From this it is clear that $\bar{\theta}$ is the posterior mean and $\mathcal{F}$ is the posterior Fisher matrix, but only because both the prior and the likelihood had the same (Gaussian) form - resulting in a Gaussian posterior.

While these results apply to the multivariate case in general, here we specialize to the bivariate case. Specifically we take a prior with Fisher matrix $P = \begin{bmatrix} 10^{-2} & 0 \\ 0 & 10^{-2} \end{bmatrix}$, and a set of simulated data points with noise (see appendix G). From this we find that the posterior mean, $\bar{\theta} = \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} = \begin{bmatrix} -0.012 \\ 1.329 \end{bmatrix}$.

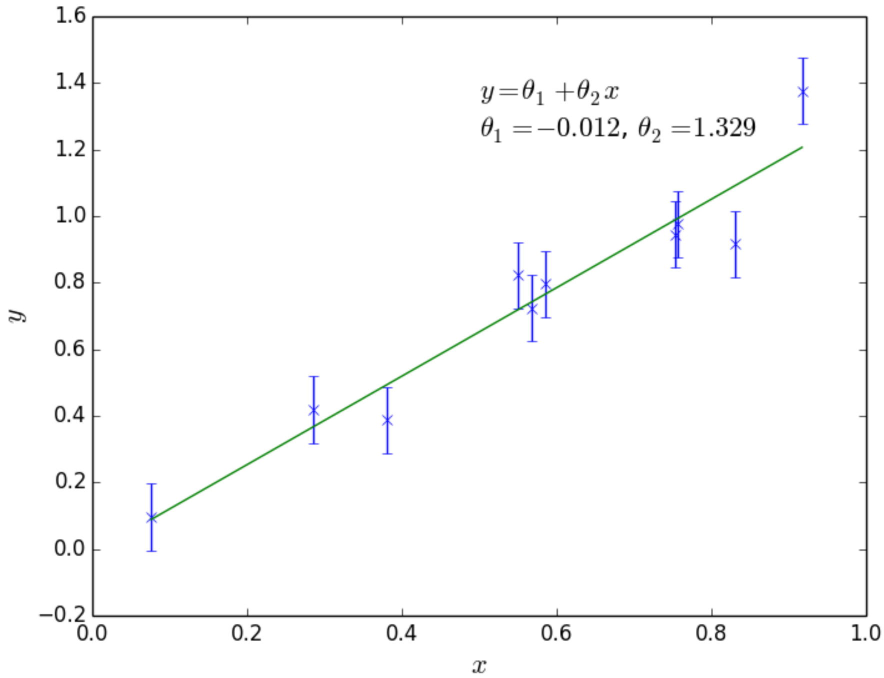
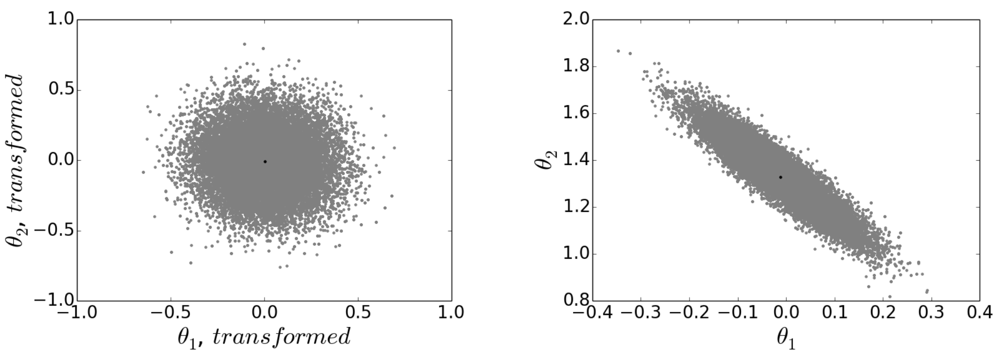


Figure 5: In blue are the simulated data with their associated error bars. The green line is the linear model $y = \theta_1 + \theta_2 x$ with the parameters θ_1 and θ_2 found analytically.

3.2 Metropolis-Hastings Algorithm



(a) Metropolis-Hastings samples as they appear when transformed. The proposal standard deviations can now be parallel to the axes and easily explore the whole target distribution.

(b) Metropolis-Hastings samples transformed back to an ellipse. Proposal standard deviations are hard to optimise for the distribution in this form.

Figure 6

The Metropolis-Hastings algorithm was then applied to this bivariate posterior distribution. The program includes a preliminary run - this is done in order to find a rough outline of the posterior distribution. From this the orientation of the elliptical distribution can be found, which is used in order to transform the distribution to a unit circle. Samples are taken from this using Metropolis-Hastings, and then transformed back to the original distribution, as seen in figure 6. This makes the choice of proposal distribution simpler and the algorithm more efficient - it is easiest to choose the proposal distribution such that the standard deviation is specified in the x and y directions. However if the ellipse of our desired distribution is not aligned with the axes then this will reduce the efficiency of the mixing. Transforming the ellipse not only aligns it to the axes (and thus to the proposal distribution) but also means we can use the same proposal standard devation in along each axis.

Another benefit of using a preliminary run is that the resulting mean can be used as the initial θ in the main run. This may reduce mixing time, and remove the need for a burn-in[7].

The standard deviation was chosen such the the acceptance rate $a \approx 0.35$ as according to Gelman et. al this is the optimal value for a normal target distribution in two dimensions [6]. The results are shown below in figure 7 with confidence regions indicating those regions containing 67%, 95% and 99% of the samples.

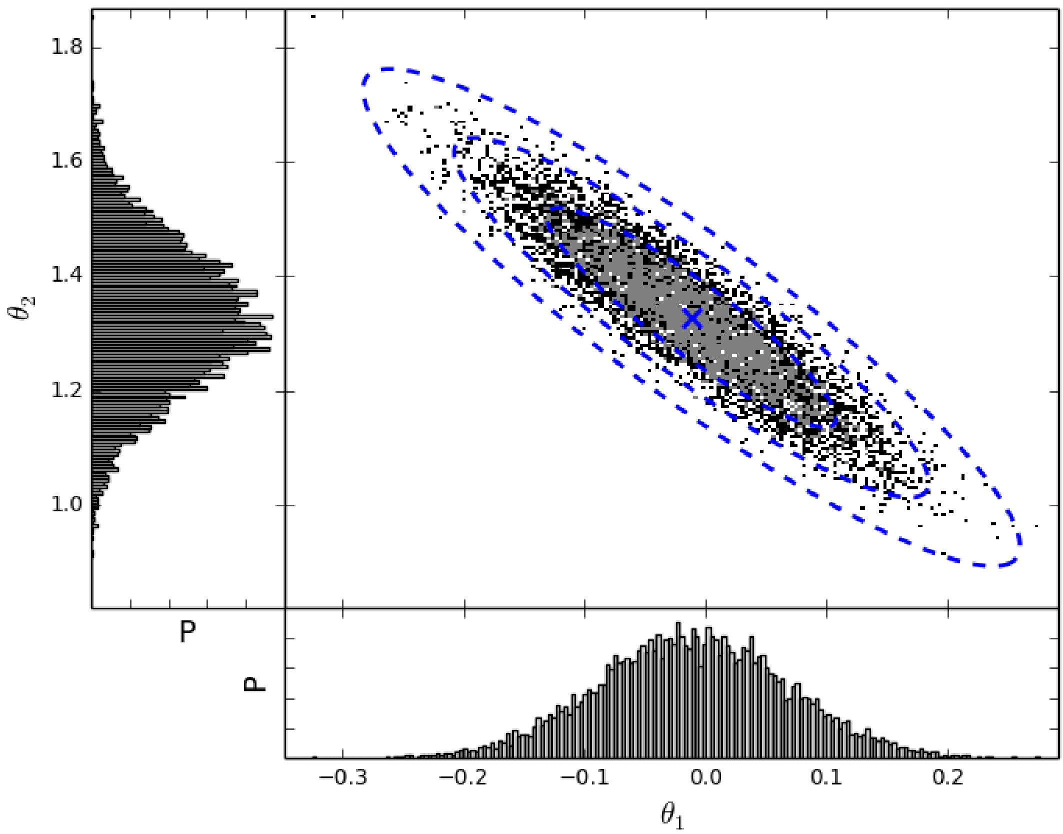


Figure 7: On the top right is a plot of Metropolis-Hastings samples, shaded to represent 1−, 2− and $3-\sigma$ confidence regions as approximated numerically. This is overlaid with the analytically calculated confidence regions outlined in blue. Alongside are the one-dimensional marginalized posterior distributions for each parameter.

Figure 7 shows both the numerically approximated and analytically calculated confidence regions, as well as marginalized distributions for each parameter. The numerical approximation was made under the assumption that the population always decreases radially out from the mean. As we have a finite sample size this in practice is not always the case, and so the confidence regions are not as clear as they would be if we had not used such an assumption. However they serve to visually give us a clearer idea of of how well our MCMC samples fit the target distribution.

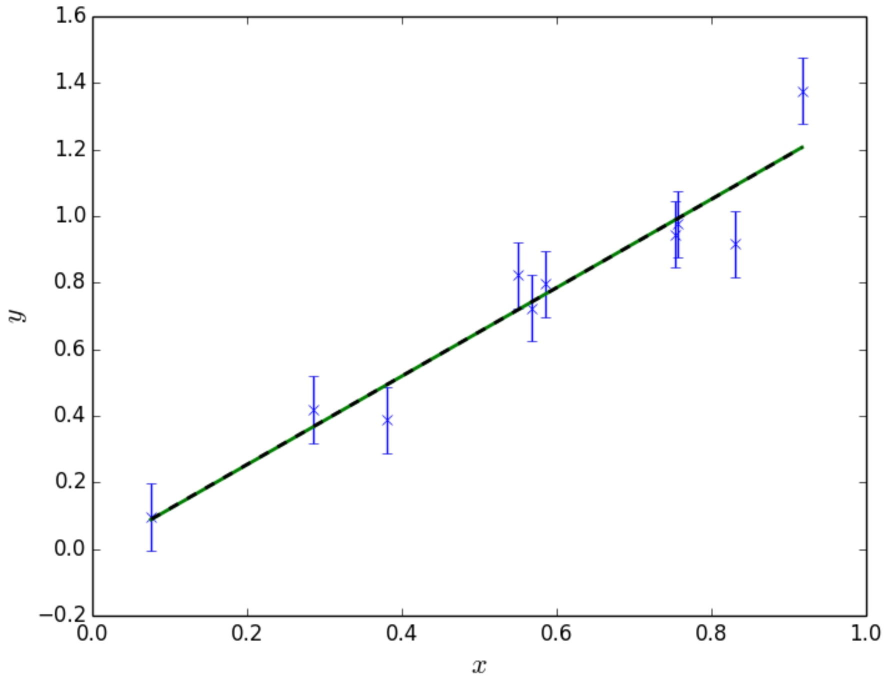


Figure 8: As above, the green line is the analytical solution. The dashed black line shows the model found using the Metropolis-Hastings method.

3.3 Analytical vs MH

4 Conclusion

We have seen that the Metropolis-Hastings algorithm produces better results in the one-dimensional case than rejection sampling. On applying this to a two-dimensional toy problem we found that it produced results which match the analytical solution well, suggesting that it would be a good choice of MCMC method for future problems. However, we used a preliminary run to improve the results of the main run, which takes extra time and may not always be convenient.

A One-Dimensional Likelihood Function

The likelihood distribution for a set of N measurements is given by the product of the likelihood for each measurement:

$$\begin{aligned}\mathcal{L}(\theta) &= \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{1}{2}\frac{(\theta - \hat{x}_i)^2}{\sigma^2}\right) \\ &= \left(\frac{1}{\sqrt{2\pi}\sigma}\right)^N \exp\left(-\frac{1}{2}\sum_{i=1}^{N}(\frac{\theta - \hat{x}_i)^2}{\sigma^2}\right).\end{aligned}$$

Taking, for now, just the exponent, and using that $\bar{x} = \frac{1}{N}\sum_i \bar{x}_i \Rightarrow \sum_i \hat{x}_i = N\bar{x}$:

$$
\begin{aligned}
-\frac{1}{2\sigma^2}\sum_i^N(\theta-\hat{x})^2 &= -\frac{1}{2\sigma^2}(N\theta^2 - 2\theta\sum_i^N \hat{x}_i + \sum_i^N \hat{x}_i^2) \\
&= -\frac{1}{2\sigma^2}(N\theta^2 - 2\theta N\bar{x} + \sum_i^N \hat{x}_i^2) \\
&= -\frac{N}{2\sigma^2}(\theta^2 - 2\theta\bar{x}[+\bar{x}^2 - \bar{x}^2] + \frac{1}{N}\sum_i^N \hat{x}_i^2) \\
&= -\frac{N}{2\sigma^2}(\theta-\bar{x})^2 - \frac{N}{2\sigma^2}(\frac{1}{N}\sum_{i=1}^N \hat{x}_i^2 - \bar{x}^2).
\end{aligned}
$$

So

$$
\begin{aligned}
\mathcal{L}(\theta) &= \left(\frac{1}{\sqrt{2\pi}}\right)^N \exp\left(-\frac{N}{2\sigma^2}(\frac{1}{N}\sum_{i=1}^N \hat{x}_i^2 - \bar{x}^2)\right)\exp\left(-\frac{N}{2\sigma^2}(\theta-\bar{x})^2\right) \\
&= L_0 \exp\left(-\frac{N}{2\sigma^2}(\theta-\bar{x})^2\right),
\end{aligned}
$$

where $L_0 = \left(\frac{1}{\sqrt{2\pi}}\right)^N \exp\left(-\frac{N}{2\sigma^2}(\frac{1}{N}\sum_i \hat{x}_i^2 - \bar{x}^2)\right)$.

B Computing the Posterior Probability for Theta

Bayes theorem is given by:

$$
p(\theta|x) = \frac{p(x|\theta)p(\theta)}{p(x)}.
$$

Ignoring the normalization constant (as it is independent of θ):

$$
p(\theta|x) \propto p(x|\theta)p(\theta).
$$

We have that

$$
p(x|\theta) = L_0 \exp\left(-\frac{N}{2\sigma^2}(\theta-\bar{x})^2\right),
$$

and

$$
p(\theta) = \frac{1}{\sqrt{2\pi}}\exp\left(-\frac{1}{2}\frac{\theta^2}{\Sigma^2}\right).
$$

From these,

$$
p(\theta|x) \propto \exp\left[-\frac{1}{2}\left(\frac{(\theta-\bar{x})^2}{\sigma^2/N} + \frac{\theta^2}{\Sigma^2}\right)\right].
$$

Taking just the exponent,

$$
\begin{aligned}
\frac{(\theta-\bar{x})^2}{\sigma^2/N} + \frac{\theta^2}{\Sigma^2} &= \frac{N}{\Sigma^2\sigma^2}\left[\Sigma^2(\theta-\bar{x})^2 + \frac{\sigma^2}{N}\theta^2\right] \\
&= \left(\frac{1}{\Sigma^2} + \frac{N}{\sigma^2}\right)\left(\frac{1}{\sigma^2/N + \Sigma^2}\right)\left[(\Sigma^2 + \frac{\sigma^2}{N})\theta^2 - 2\bar{x}\Sigma^2\theta + \Sigma^2\bar{x}^2\right] \\
&= \left(\frac{1}{\Sigma^2} + \frac{N}{\sigma^2}\right)\left(\theta^2 - 2\frac{\Sigma^2}{\sigma^2/N + \Sigma^2}\theta + \frac{\Sigma^2}{\sigma^2/N + \Sigma^2}\bar{x}^2\right).
\end{aligned}
$$

The final term above is independent of θ. Thus, as we are dealing with an exponent, we can subtract this term and add its square without losing proportionality. This allows us to complete the square so we have:

$$\left(\frac{1}{\Sigma^2} + \frac{N}{\sigma^2}\right)\left(\theta - \frac{\Sigma^2}{\sigma^2/N + \Sigma^2}\bar{x}\right)^2.$$

Putting this back inside the exponential we are left with:

$$p(\theta|x) \propto \exp\left[-\frac{1}{2}\left(\frac{1}{\Sigma^2} + \frac{N}{\sigma^2}\right)\left(\theta - \frac{\Sigma^2}{\sigma^2/N + \Sigma^2}\bar{x}\right)^2\right],$$

i.e. the posterior follows a Gaussian distribution with mean $\frac{\Sigma^2}{\sigma^2/N+\Sigma^2}\bar{x}$ and standard deviation $\left(\frac{1}{\Sigma^2} + \frac{N}{\sigma^2}\right)^{-\frac{1}{2}}$.

C Asymptotic Independence of Posterior on Prior

For the posterior

$$p(\theta|x) \propto \exp\left[-\frac{1}{2}\frac{\left(\theta - \frac{\Sigma^2}{\sigma^2/N+\Sigma^2}\bar{x}\right)^2}{\left(\frac{1}{\Sigma^2} + \frac{N}{\sigma^2}\right)^{-1}}\right],$$

as $N \to \infty$,

$$\begin{array}{lll} \left(\frac{1}{\Sigma^2} + \frac{N}{\sigma^2}\right)^{-1} & \to \frac{\sigma^2}{N} & \left(\frac{N}{\sigma^2} \gg \frac{1}{\Sigma^2}\right), \text{ and the posterior becomes} \\ \frac{\Sigma^2}{\sigma^2/N+\Sigma^2}\bar{x} & \to \bar{x} & \left(\frac{\sigma^2}{N} \to 0\right), \end{array}$$

$$p(\theta|x) \propto \exp\left[-\frac{1}{2}\frac{(\theta - \bar{x})^2}{\sigma^2/N}\right],$$

i.e. the likelihood.

D Asymptotic Convergence of the Posterior Mean to the MLE Mean for Theta

The posterior mean is given by:

$$\langle\theta\rangle = \int_{-\infty}^{+\infty} \theta p(\theta|x) \mathrm{d}\,\theta.$$

Inserting our equation for the posterior for $N \to \infty$, we have

$$\langle\theta\rangle = \int_{-\infty}^{+\infty} \theta \exp\left[-\frac{1}{2}\frac{(\theta - \bar{x})^2}{\sigma^2/N}\right] \mathrm{d}\,\theta.$$

Making the substitution $y = \theta - \bar{x}$:

$$\begin{aligned} \langle\theta\rangle &= \int_{-\infty}^{+\infty} (y + \bar{x}) \exp\left[-\frac{1}{2}\frac{y^2}{\sigma^2/N}\right] \mathrm{d}\,y \\ &= \int_{-\infty}^{+\infty} \bar{x} \exp\left(-\frac{1}{2}\frac{N}{\sigma^2}y^2\right) \mathrm{d}\,y, \end{aligned}$$

as $y \exp\left(-\frac{1}{2}\frac{N}{\sigma^2}y^2\right)$ is an odd function.

Substituting back:

$$\int_{-\infty}^{+\infty} \bar{x} \exp\left[-\frac{N}{\sigma^2}(\theta - \bar{x})^2\right] \mathrm{d}\,\theta = \bar{x} \int_{-\infty}^{+\infty} p(\theta|x) \mathrm{d}\,\theta$$
$$= \bar{x},$$

as the integral of a pdf between infinite limits is equal to one.

E 2-D Likelihood in Gaussian Form

For the linear model

$$y = F\theta + \epsilon$$

where ϵ is uncorrelated, the likelihood function is given by

$$p(y|\theta) = \frac{1}{(2\pi)^{\frac{d}{2}} \Pi_j \tau_j} \exp\left[-\frac{1}{2}(b - A\theta)^t (b - A\theta)\right],$$

where $A_{ij} = {}^{F_{ij}}/_{\tau_i}$ and $b_i = {}^{y_i}/_{\tau_i}$.

Taking just the variable part of the exponent,

$$\begin{aligned}
(b - A\theta)^t (b - A\theta) &= (A\theta - b)^t (A\theta - b) \\
&= (\theta - A^{-1}b)^t A^t A (\theta - A^{-1}b) \\
&= (b - AA^{-1}(A^t)^{-1}A^t b)^t (b - AA^{-1}(A^t)^{-1}A^t b) + \\
&\quad\ (\theta - A^{-1}(A^t)^{-1}A^t b)^t A^t A (\theta - A^{-1}(A^t)^{-1}A^t b) \\
&= (b - AL^{-1}A^t b)^t (b - AL^{-1}A^t b) + (\theta - L^{-1}A^t b)^t L (\theta - L^{-1}A^t b) \\
&= (b - A\theta_0)^t (b - A\theta_0) + (\theta - \theta_0)^t L (\theta - \theta_0).
\end{aligned}$$

Putting this back into the exponent above we have

$$\begin{aligned}
p(y|\theta) &= \frac{1}{(2\pi)^{\frac{d}{2}} \Pi_j \tau_j} \exp\left[-\frac{1}{2}(b - A\theta_0)^t (b - A\theta_0) - \frac{1}{2}(\theta - \theta_0)^t L (\theta - \theta_0)\right], \\
&= \frac{1}{(2\pi)^{\frac{d}{2}} \Pi_j \tau_j} \exp\left[-\frac{1}{2}(b - A\theta_0)^t (b - A\theta_0)\right] \exp\left[-\frac{1}{2}(\theta - \theta_0)^t L (\theta - \theta_0)\right]. \\
&= \mathcal{L}_0 \exp\left[-\frac{1}{2}(\theta - \theta_0)^t L (\theta - \theta_0)\right].
\end{aligned}$$

Where $\mathcal{L}_0 = \frac{1}{(2\pi)^{\frac{d}{2}} \Pi_j \tau_j} \exp\left[-\frac{1}{2}(b - A\theta_0)^t (b - A\theta_0)\right]$, $\theta_0 = L^{-1}A^t b$ and $L \equiv A^t A$.

F 2-D Posterior in Gaussian Form

If the prior probability distribution function goes as

$$p(\theta) \propto \exp\left[-\frac{1}{2}\theta^t P \theta\right],$$

where P is the prior Fisher information matrix, and the likelihood function goes as

$$p(y|\theta) \propto \exp\left[-\frac{1}{2}(\theta - \theta_0)^t L (\theta - \theta_0)\right],$$

then according to Bayes theorem the posterior follows

$$\begin{aligned} p(\theta|y) &\propto p(\theta)p(y|\theta) \\ &\propto \exp\left[-\frac{1}{2}\theta^t P\theta\right]\exp\left[-\frac{1}{2}(\theta-\theta_0)^t L(\theta-\theta_0)\right]. \end{aligned}$$

Combining the exponents and taking the variable part of the result,

$$\begin{aligned} \theta^t P\theta + (\theta-\theta_0)^t L(\theta-\theta_0) &= \theta^t P\theta + \theta^t L\theta - \theta_0^t L\theta - \theta^t L\theta_0 + \theta_0^t L\theta_0 \\ &= \theta^t (L+P)\theta - \theta_0^t L\theta - \theta^t L\theta_0 + \theta_0^t L\theta_0. \end{aligned}$$

Constants can be added and subtracted from the exponent without affecting the proportionality, so we subtract $\theta_0^t L\theta_0$ and add $\theta_0^t L(L+P)^{-1}L\theta_0$:

$$\begin{aligned} &\theta^t (L+P)\theta - \theta_0^t L\theta - \theta^t L\theta_0 + \theta_0^t L(L+P)^{-1}L\theta_0 \\ &\quad = \theta^t (L+P)\theta - \theta_0^t L\theta - \theta^t (L+P)(L+P)^{-1}L\theta_0 + \theta_0^t L(L+P)^{-1}L\theta_0 \\ &\quad = (\theta^t (L+P) - \theta_0^t L)(\theta - (L+P)^{-1}L\theta_0) \\ &\quad = (\theta^t - \theta_0^t L(L+P)^{-1})(L+P)(\theta - (L+P)^{-1}L\theta_0) \end{aligned}$$

P and L are both fisher matrices, i.e. inverse covariance matrices. As covariance matrices are symmetric, it follows that the inverse of the sum of these two matrices is symmetric, and so $(L+P)^{-1} = ((L+P)^{-1})^t$. Thus we have

$$\begin{aligned} &(\theta^t - \theta_0^t L^t((L+P)^{-1})^t)(L+P)(\theta - (L+P)^{-1}L\theta_0) \\ &\quad = (\theta - (L+P)^{-1}L\theta_0)^t (L+P)(\theta - (L+P)^{-1}L\theta_0) \\ &\quad = (\theta - \bar{\theta})^t \mathcal{F}(\theta - \bar{\theta}), \end{aligned}$$

where $\mathcal{F} = L + P$ and $\bar{\theta} = \mathcal{F}^{-1}L\theta_0$. This, when put back into the exponent, gives the form of a multivariate Gaussian:

$$p(\theta|y) \propto \exp\left[-\frac{1}{2}(\theta-\bar{\theta})^t \mathcal{F}(\theta-\bar{\theta})\right],$$

with mean $\bar{\theta}$ and Fisher matrix $\mathcal{F}$.

G Data

Univariate Sample Data
0.594
0.360
0.432
0.537
0.398
0.492
0.517
0.416
0.369
0.519

Bivariate Sample Data	
x	y
0.8308	0.9160
0.5853	0.7958
0.5497	0.8219
0.9172	1.3757
0.2858	0.4191
0.7572	0.9759
0.7537	0.9455
0.3804	0.3871
0.5678	0.7239
0.0759	0.0964

H Python Code for Univariate Model

```python
import math
import random
from argparse import ArgumentParser
import numpy
import matplotlib.pyplot as plt


def simulate_experiment(mean, stdev, N):
        """ Generate values for N independently Gaussian distributed 'measurements' """
        random.seed(0)
        data = [random.gauss(mean, stdev) for i in range(N)]

        return data

def sigma_range(mean, stdev, width):
        """ Find range of x values to be included, based on number of standard deviations to be included """
        lower = mean - width * stdev
        upper = mean + width * stdev
        bounds = {'min': lower, 'max': upper}

        return bounds

def posterior_info(sample_mean, prior_stdev, sample_size, population_stdev):
        """ Calculate the posterior mean and standard deviation """
        posterior_stdev = (1/(prior_stdev**2) + sample_size/(population_stdev**2))**(-1/2)
        posterior_mean = sample_mean * prior_stdev**2 / (prior_stdev**2 + population_stdev**2/sample_size)
        posterior_stats = {'mean': posterior_mean, 'stdev': posterior_stdev}

        return posterior_stats

def setup(population_mean, population_stdev, sample_size, prior_stdev, width):
        data = simulate_experiment(population_mean, population_stdev, sample_size)
        sample_mean = numpy.mean(data)
        posterior_stats = posterior_info(sample_mean, prior_stdev, sample_size, population_stdev)
        theta_range = sigma_range(posterior_stats['mean'], posterior_stats['stdev'], width)

        return posterior_stats, theta_range

def analytical(posterior_stats, theta_range, data_points):
        """ Generate theta values within desired range and
```

```
        calculate corresponding posteriors """
        thetas = numpy.linspace(theta_range['min'], theta_range['max'], data_points)
        posteriors = numpy.exp(-0.5 * ((thetas - posterior_stats['mean'])/posterior_stats['stdev'])**2)

        return thetas, posteriors

def rejection_sampling(posterior_stats, theta_range, iterations):
        """ Numerical solution (Rejection sampling) """
        posterior_min = 0
        posterior_max = 1

        # Generate random uniformly distributed x and y coordinates in appropriate ranges
        x = [random.uniform(theta_range['min'], theta_range['max']) for _ in range(iterations)]
        y = [random.uniform(posterior_min, posterior_max) for _ in range(iterations)]

        # Calculate posterior at each x value
        comparison_posterior = [calculate_posterior(i, posterior_stats) for i in x]
        x_accepts = []
        for i,j,k in zip(x,y,comparison_posterior):
                if j <= k:
                        x_accepts.append(i)

        return x_accepts

def generate_candidate(theta_current, proposal_stdev):
        """ Generate a candidate theta value """
        theta_proposed = random.gauss(theta_current, proposal_stdev)

        return theta_proposed

def calculate_posterior(theta, posterior_stats):
        """ Calculate the value of the posterior for a given theta and posterior mean and standard deviation """
        posterior = math.exp(-0.5 * ((theta - posterior_stats['mean'])/posterior_stats['stdev'])**2)

        return posterior

def calculate_acceptance_probability(posterior_current, posterior_proposed):
        """ Calculate 'probability' of accepting proposed theta """
        acceptance_probability = posterior_proposed / posterior_current

        return acceptance_probability
```

```
def metropolis_hastings(posterior_stats, theta_initial, proposal_stdev, iterations):
    """ Numerical solution (Metropolis-Hastings algorithm) """

    thetas_mh = []
    # For burn-in plot
    posteriors_mh = []
    theta_current = theta_initial
    posterior_current = calculate_posterior(theta_current, posterior_stats)
    # For acceptance rate plot
    accepts = 0
    for i in range(iterations):
        theta_proposed = generate_candidate(theta_current, proposal_stdev)
        posterior_proposed = calculate_posterior(theta_proposed, posterior_stats)
        acceptance_probability = calculate_acceptance_probability(posterior_current, posterior_proposed)

        # Always accept proposed value if it is more likely than current value
        # If proposed value less likely than current value, accept with probability 'acceptance_proability'
        if (acceptance_probability >= 1) or (random.uniform(0,1) <= acceptance_probability):
            theta_current = theta_proposed
            posterior_current = posterior_proposed
            accepts += 1

        thetas_mh.append(theta_current)
        posteriors_mh.append(posterior_current)

    return thetas_mh, posteriors_mh, accepts

def plot_rejection_sampling(thetas, posteriors, x_accepts, bins):
    """ Plot analytical solution and rejection sampling solution on same graph """
    fig, ax = plt.subplots()
    plt.plot(thetas, posteriors, linewidth=3)

    #Rejection sampling plot
    hist, bin_edges = numpy.histogram(x_accepts, bins)
    bin_width = bin_edges[1] - bin_edges[0]
    hist = hist / max(hist)
    ax.bar(bin_edges[:-1], hist, bin_width, color='green')
    ax.tick_params(axis='both', which='major', labelsize=20)
```

```python

    # Create strings to show numerical mean and standard deviation on graphs
    mean = numpy.mean(x_accepts)
    stdev = numpy.std(x_accepts)
    display_string = ('$\mu_{{MC}} = {0:.4f} $\n$\sigma_{{MC}} = {1:.4f}$').format(mean, stdev)

    plt.xlabel(r'$\theta$', fontsize=28)
    plt.ylabel(r'$\propto P(\theta|x)$', fontsize=28)
    plt.text(0.7, 0.8, display_string, transform=ax.transAxes, fontsize=20)
    plt.savefig('rejection.png', bbox_inches='tight')

    # Plot log
    fig, ax = plt.subplots()
    plt.plot(thetas, -numpy.log(posteriors), linewidth=3)
    ax.bar(bin_edges[:-1], -numpy.log(hist), bin_width, color='green')
    ax.tick_params(axis='both', which='major', labelsize=20)

    plt.xlabel(r'$\theta$', fontsize=28)
    plt.ylabel(r'$\propto log(P(\theta|x))$', fontsize=28)
    plt.text(0.5, 0.5, display_string, transform=ax.transAxes, fontsize=20)

    plt.savefig('rejlog.png', bbox_inches='tight')

def plot_metropolis_hastings(thetas, posteriors, thetas_mh, bins):
    """ Plot analytical solution and Metropolis-Hastinga solution on same graph """
    fig, ax = plt.subplots()
    plt.plot(thetas, posteriors, linewidth=3)

    # Metropolis-Hastings plot
    hist, bin_edges = numpy.histogram(thetas_mh, bins)
    bin_width = bin_edges[1] - bin_edges[0]
    hist = hist / max(hist)
    ax.bar(bin_edges[:-1], hist, bin_width, color='green')
    ax.tick_params(axis='both', which='major', labelsize=20)

    # Create strings to show numerical mean and standard deviation on graphs
    mean = numpy.mean(thetas_mh)
    stdev = numpy.std(thetas_mh)
    display_string = ('$\mu_{{MC}} = {0:.4f} $\n$\sigma_{{MC}} = {1:.4f}$').format(mean, stdev)

    plt.xlabel(r'$\theta$', fontsize=28)
    plt.ylabel(r'$\propto P(\theta|x)$', fontsize=28)
    plt.text(0.7, 0.8, display_string, transform=ax.transAxes, fontsize=20)
    plt.savefig('metropolishastings.png', bbox_inches='tight')
```

```
        # Plot log
        fig, ax = plt.subplots()
        plt.plot(thetas, -numpy.log(posteriors), linewidth=3)
        ax.bar(bin_edges[:-1], -numpy.log(hist), bin_width, color='green')
        ax.tick_params(axis='both', which='major', labelsize=20)

        plt.xlabel(r'$\theta$', fontsize=28)
        plt.ylabel(r'$\propto log(P(\theta|x))$', fontsize=28)
        plt.text(0.5, 0.5, display_string, transform=ax.transAxes, fontsize=20)

        plt.savefig('mhlog.png', bbox_inches='tight')

        # Plot with burn-in removed
        hist, bin_edges = numpy.histogram(thetas_mh[200:], bins)
        bin_width = bin_edges[1] - bin_edges[0]
        hist = hist / max(hist)

        fig, ax = plt.subplots()
        plt.plot(thetas, posteriors, linewidth=3)

        # Metropolis-Hastings
        ax.bar(bin_edges[:-1], hist, bin_width, color='green')
        ax.tick_params(axis='both', which='major', labelsize=20)

        # Create strings to show numerical mean and standard deviation on graphs
        mean = numpy.mean(thetas_mh[200:])
        stdev = numpy.std(thetas_mh[200:])
        display_string = ('$\mu_{{MC}} = {0:.4f} $\n$\sigma_{{MC}} = {1:.4f}$').format(mean, stdev)

        plt.xlabel(r'$\theta$', fontsize=28)
        plt.ylabel(r'$\propto P(\theta|x)$', fontsize=28)
        plt.text(0.7, 0.8, display_string, transform=ax.transAxes, fontsize=20)
        plt.savefig('metropolishastings-burnin.png', bbox_inches='tight')

        # Plot log
        fig, ax = plt.subplots()
        plt.plot(thetas, -numpy.log(posteriors), linewidth=3)

        ax.bar(bin_edges[:-1], -numpy.log(hist), bin_width, color='green')
        ax.tick_params(axis='both', which='major', labelsize=20)

        plt.xlabel(r'$\theta$', fontsize=28)
        plt.ylabel(r'$\propto log(P(\theta|x))$', fontsize=28)
        plt.text(0.5, 0.5, display_string, transform=ax.transAxes, fontsize=20)
```

```

    plt.savefig('mhlog-burnin.png', bbox_inches='tight')

def plot_convergence(posterior_stats, proposal_stdev, start1=0.1, start2=0.5, start3=0.9, iterations=2000):
    """ Burn-in plot for Metropolis-Hastings method """
    thetas_mh = {'1':1, '2':1, '3':1}
    thetas_mh['1'], posteriors_mh, accepts = metropolis_hastings(posterior_stats, start1, proposal_stdev, iterations)
    thetas_mh['2'], posteriors_mh, accepts = metropolis_hastings(posterior_stats, start2, proposal_stdev, iterations)
    thetas_mh['3'], posteriors_mh, accepts = metropolis_hastings(posterior_stats, start3, proposal_stdev, iterations)
    plt.plot(range(iterations), thetas_mh['1'][0:iterations])
    plt.plot(range(iterations), thetas_mh['2'][0:iterations])
    plt.plot(range(iterations), thetas_mh['3'][0:iterations])

    plt.xlabel('Iteration', fontsize=16)
    plt.ylabel(r'$\theta$', fontsize=16)
    plt.savefig('convergence.png', bbox_inches='tight')

def plot_burn_in(iterations, thetas_mh, posteriors_mh):
    """ Burn-in plot for Metropolis-Hastings method """

    plt.plot(range(20000), thetas_mh[1:20001])
    plt.xlim(-100)
    plt.ylim(0.05, 0.6)

    plt.xlabel('Iteration', fontsize=16)
    plt.ylabel(r'$\theta$', fontsize=16)
    plt.savefig('burnin.png', bbox_inches='tight')

def proposal_stdev_effects(posterior_stats, theta_initial, iterations, proposal_stdev_min = 0.06, proposal_stdev_max = 0.26, data_points = 20):
    """ Returns data showing effects of changing the standard deviation of the proposal distribution """
    mh_stdevs = []
    acceptance_rates = []
    proposal_stdevs = []
    proposal_stdev_interval = (proposal_stdev_max - proposal_stdev_min) / data_points
    proposal_stdev = proposal_stdev_min
    for i in range(data_points):
        thetas_mh, posteriors_mh, accepts = metropolis_hastings(posterior_stats, theta_initial, proposal_stdev, iterations)

        acceptance_rates.append(accepts / iterations)
        proposal_stdevs.append(proposal_stdev)
```

```
            mh_stdevs.append(numpy.std(posteriors_mh))

            proposal_stdev = proposal_stdev + proposal_stdev_interval

    return(proposal_stdevs, acceptance_rates, mh_stdevs)

def plot_proposal(proposal_stdevs, acceptance_rates, mh_stdevs):
    """ Plots showing effect of changing te standard deviation of the proposal distribution """
    plt.figure()
    plt.subplot(1, 2, 1)
    # Plot acceptance rate for different standard deviations of the proposal distribution
    plt.plot(proposal_stdevs, acceptance_rates, marker='x', linestyle='none')
    plt.xlabel('Proposal Standard Deviation')
    plt.ylabel('Acceptance Rate')

    plt.subplot(1, 2, 2)
    # Plot standard devation of posterior from MH method against that of proposal distribution
    plt.plot(proposal_stdevs, mh_stdevs, marker='x', linestyle='none')
    plt.xlabel('Proposal Standard Deviation')
    plt.ylabel('Posterior Standard Deviation')
    plt.savefig('proposalstdev.png', bbox_inches='tight')


def main():

    # Use command line arguments to determine which parts of code to run
    modes = ['convergence', 'rejection', 'metropolis_hastings', 'all', 'proposal']
    parser = ArgumentParser(description='One dimensional MCMC')
    parser.add_argument('--mode', type=str, default='all', choices=modes, help='Specify which section of the program to run.')
    args = parser.parse_args()

    population_mean = 0.5
    population_stdev = 0.1
    sample_size = 10
    prior_stdev = 1

    bins = 100
    # Number of stdevs from the mean over which analytical and rejection sampling results will be found
    width = 5
    iterations = 200000

```

```
        posterior_stats, theta_range = setup(population_mean, population_stdev, sample_size, prior_stdev, width)

        # Analytical
        print('analytical mean')
        print(posterior_stats['mean'])
        print('analytical standard deviation')
        print(posterior_stats['stdev'])

        data_points = 100
        thetas, posteriors = analytical(posterior_stats, theta_range, data_points)

        if (args.mode == 'convergence') or (args.mode == 'all'):
                plot_convergence(posterior_stats, 0.01, 0.1, 0.5, 1.2, 1000)


        if (args.mode == 'rejection') or (args.mode == 'all'):
                # Rejection sampling
                x_accepts = rejection_sampling(posterior_stats, theta_range, iterations)
                plot_rejection_sampling(thetas, posteriors, x_accepts, bins)

        if (args.mode == 'metropolis_hastings') or (args.mode == 'proposal') or (args.mode == 'all'):
                # Variables required by metropolis and proposal
                theta_initial = 0.2

        if (args.mode == 'metropolis_hastings') or (args.mode == 'all'):
                # Metropolis-Hastings
                proposal_stdev = 0.01
                thetas_mh, posteriors_mh, accepts = metropolis_hastings(posterior_stats, theta_initial, proposal_stdev, iterations)
                plot_metropolis_hastings(thetas, posteriors, thetas_mh, bins)

                plot_burn_in(iterations, thetas_mh, posteriors_mh)

        if (args.mode == 'proposal'):
                # Effects of changing the proposal distributions standard deviation
                proposal_stdevs, acceptance_rates, mh_stdevs = proposal_stdev_effects(posterior_stats, theta_initial, iterations)
                plot_proposal(proposal_stdevs, acceptance_rates, mh_stdevs)

if __name__ == '__main__':
        main()
```

I Python Code for Bivariate Model

```python
import numpy as np
import random
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
import math
from matplotlib.patches import Ellipse


def import_data(textfile, uncertainty):
        """Import measurements from file. Each x and y pair on its own line, delimited by ', '
        i.e. 'x, y\n'. Also specify uncertainty for measurements."""
        f = open(textfile, 'r')
        data = f.readlines()
        x = []
        y = []
        for line in data:
                coords = line.strip()
                coords = coords.split(', ')
                x.append(coords[0])
                y.append(coords[1])

        x = [float(i) for i in x]
        x = np.array(x).reshape((10,1))
        y = [float(i) for i in y]
        y = np.array(y).reshape((10,1))
        data = {'x': x, 'y': y, 'var': uncertainty}

        return data

def get_design_matrix(x):
        """Find design matrix for our specialized case (observations are fitted with linear model)."""
        F = np.ones((10, 1))
        F = np.hstack((F, x))

        return F

def get_likelihood_fisher_matrix(A):
        likelihood_fisher = np.dot(A.transpose(), A)

        return likelihood_fisher

def get_prior_fisher_matrix():
        """Prior fisher matrix for this case. """
        prior_fisher = 0.1 * np.eye(2)

        return prior_fisher

def get_posterior_fisher_matrix(likelihood_fisher, P):
```

```
        posterior_fisher = likelihood_fisher + P

        return posterior_fisher

def get_mle(likelihood_fisher, A, b):
        mle = np.dot(A.transpose(), b)
        mle = np.dot(np.linalg.inv(likelihood_fisher), mle)

        return mle

def get_posterior_mean(likelihood_fisher, posterior_fisher, mle):
        posterior_mean = np.dot(likelihood_fisher, mle)
        posterior_mean = np.dot(np.linalg.inv(posterior_fisher), posterior_mean)

        return posterior_mean

def setup(measurement_uncertainty):
        """Find posterior mean, posterior fisher and covariance matrix"""
        data = import_data('dataset.txt', measurement_uncertainty)
        design = get_design_matrix(data['x'])
        A = design / measurement_uncertainty
        likelihood_fisher = get_likelihood_fisher_matrix(A)
        prior_fisher = get_prior_fisher_matrix()
        posterior_fisher = get_posterior_fisher_matrix(likelihood_fisher, prior_fisher)
        b = data['y'] / measurement_uncertainty
        mle = get_mle(likelihood_fisher, A, b)
        posterior_mean = get_posterior_mean(likelihood_fisher, posterior_fisher, mle)

        covariance = np.linalg.inv(posterior_fisher)

        posterior_stats = {'fisher': posterior_fisher, 'mean': posterior_mean, 'covar': covariance}

        return data, posterior_stats

def calculate_ln_posterior(thetas, posterior_stats):
        """Calculate the natural logarithm of the posterior for given theta values."""
        ln_posterior = np.dot(posterior_stats['fisher'], (thetas - posterior_stats['mean']))
        ln_posterior = - np.dot((thetas - posterior_stats['mean']).transpose(), ln_posterior) /2

        return ln_posterior

def generate_candidates(thetas, proposal_stdev):
        """Generate candidate theta values using proposal 
```

```
    distribution."""
    thetas_proposed = np.zeros((2, 1))
    thetas_proposed[0, 0] = random.gauss(thetas[0][0], proposal_stdev[0][0])
    thetas_proposed[1, 0] = random.gauss(thetas[1][0], proposal_stdev[1][0])

    return thetas_proposed

def calculate_hastings_ratio(ln_proposed, ln_current):
    ln_hastings = ln_proposed - ln_current
    hastings = np.exp(ln_hastings)

    return hastings

def metropolis_hastings(posterior_stats):
    """Sample from posterior distribution using Metropolis-Hastings algorithm."""
    iterations = 5000
    theta = np.array([[-0.05], [0.5]])
    proposal_stdev = np.array([[0.1], [0.1]])
    ln_posterior = calculate_ln_posterior(theta, posterior_stats)
    accepts = 0
    mcmc_samples = theta

    for i in range(iterations):
        theta_proposed = generate_candidates(theta, proposal_stdev)
        ln_posterior_proposed = calculate_ln_posterior(theta_proposed, posterior_stats)

        hastings_ratio = calculate_hastings_ratio(ln_posterior_proposed, ln_posterior)

        acceptance_probability = min([1, hastings_ratio])

        if (random.uniform(0,1) < acceptance_probability):
            #Then accept proposed theta
            theta = theta_proposed
            ln_posterior = ln_posterior_proposed
            accepts += 1
        mcmc_samples = np.hstack((mcmc_samples, theta))

    mcmc_mean = np.array([ [np.mean(mcmc_samples[0])], [np.mean(mcmc_samples[1])] ])
    covariance = np.cov(mcmc_samples)
    mcmc = {'samples': mcmc_samples.transpose(), 'mean': mcmc_mean, 'covar': covariance}
    print('acceptance ratio init')
    acceptance_ratio = accepts / iterations
```

```python
    print(acceptance_ratio)

    return mcmc

def metropolis_hastings_rot(posterior_stats, sample_mean, axis1, axis2):
    """Sample from posterior distribution using Metropolis-Hastings algorithm."""
    iterations = 50000
    theta = sample_mean
    proposal_stdev = np.array([[0.35], [0.35]])
    ln_posterior = calculate_ln_posterior(theta, posterior_stats)
    accepts = 0
    mcmc_samples = theta
    samples_rot = ellipse_to_circle(theta, sample_mean, axis1, axis2)

    for i in range(iterations):
        theta_rot = ellipse_to_circle(theta, sample_mean, axis1, axis2)
        theta_proposed_rot = generate_candidates(theta_rot, proposal_stdev)
        theta_proposed = circle_to_ellipse(theta_proposed_rot, sample_mean, axis1, axis2)
        ln_posterior_proposed = calculate_ln_posterior(theta_proposed, posterior_stats)

        hastings_ratio = calculate_hastings_ratio(ln_posterior_proposed, ln_posterior)

        acceptance_probability = min([1, hastings_ratio])

        if (random.uniform(0,1) < acceptance_probability):
            #Then accept proposed theta
            theta = theta_proposed
            theta_rot = theta_proposed_rot
            ln_posterior = ln_posterior_proposed
            accepts += 1
        mcmc_samples = np.hstack((mcmc_samples, theta))
        samples_rot = np.hstack((samples_rot, theta_rot))

    mcmc_mean = np.array([ [np.mean(mcmc_samples[0])], [np.mean(mcmc_samples[1])] ])
    covariance = np.cov(mcmc_samples)
    mcmc = {'samples': mcmc_samples.transpose(), 'mean': mcmc_mean, 'covar': covariance, 'proposal_stdev': proposal_stdev}
    mcmc_rot = samples_rot.transpose()

```

```
        print('acceptance ratio rotated')
        acceptance_ratio = accepts / iterations
        print(acceptance_ratio)

        return mcmc, mcmc_rot, acceptance_ratio

def transform_matrix(mean, angle, width, height):
        translate = np.array([ [1, 0, -mean[0]], [0, 1, -mean[1]], [0, 0, 1] ])
        rotate = np.array([ [math.cos(angle), math.sin(angle), 0], [-math.sin(angle), math.cos(angle), 0], [0, 0, 1] ])
        scale = np.array([ [1/width, 0, 0], [0, 1/height, 0], [0, 0, 1] ])

        transform = scale.dot(rotate.dot(translate))

        return transform

def ellipse_to_circle(xy, mean, axis1, axis2):
        transform = transform_matrix(mean, axis2['xangle'], axis1['length'], axis2['length'])
        xy = np.vstack((xy, 1))
        xy = xy.reshape((3, 1))
        xy_rot = transform.dot(xy)

        return xy_rot[:-1,:]

def circle_to_ellipse(xy_rot, mean, axis1, axis2):
        transform = transform_matrix(mean, axis2['xangle'], axis1['length'], axis2['length'])
        inv_transform = np.linalg.inv(transform)
        xy_rot = np.vstack((xy_rot, 1))
        xy_rot = xy_rot.reshape((3,1))
        xy = inv_transform.dot(xy_rot)

        return xy[:-1,:]

# Do I uyse this ???
def edges_to_centers(x_edges, y_edges, res):
        """Given edges and width of bins, find centres."""
        dx = (max(x_edges) - min(x_edges)) / res
        dy = (max(y_edges) - min(y_edges)) / res

        x = x_edges + dx /2
        y = y_edges + dy /2
        x = x[:-1]
        y = y[:-1]

        return x, y

def equal_weight(counts, res):
        """Find equal weight samples."""
```

```python
        multiplicity = counts / counts.max()
        randoms = np.random.random((res, res))

        equal_weighted_samples = multiplicity < randoms

        return equal_weighted_samples

def sigma_boundary(counts, percentage):
        """Find boundary values for each sigma-level."""
        # Sort counts in descending order
        counts_desc = sorted(counts.flatten(), reverse=True)
        # Find cumulative sum of sorted counts
        cumulative_counts = np.cumsum(counts_desc)
        # Create a mask for counts outside of percentage boundary
        sum_mask = cumulative_counts < (percentage /100) * np.sum(counts)
        sigma_sorted = sum_mask * counts_desc
        # Assume that density is ellipse equivalent of radially symmetric
        sigma_min = min(sigma_sorted[sigma_sorted.nonzero()])

        return sigma_min

def find_numerical_contours(counts):
        """Returns array of 3s, 2s, 1s, and 0s, representing one two and three sigma regions respectively."""
        one_sigma_boundary = sigma_boundary(counts, 68)
        one_sigma = counts > one_sigma_boundary
        two_sigma_boundary = sigma_boundary(counts, 95)
        two_sigma = (counts > two_sigma_boundary) & (counts < one_sigma_boundary)
        three_sigma_boundary = sigma_boundary(counts, 98)
        three_sigma = (counts > three_sigma_boundary) & (counts < two_sigma_boundary)

        # Check method: Output actual percentages in each region
        print('total no. samples:')
        print(np.sum(counts))
        print('included in 1st sigma region:')
        print(np.sum(one_sigma * counts) / np.sum(counts))
        print('included in 2 sigma region:')
        print((np.sum(one_sigma * counts) + np.sum(two_sigma * counts)) / np.sum(counts))
        print('included in 3 sigma region:')
        print((np.sum(one_sigma * counts) + np.sum(two_sigma * counts) + np.sum(three_sigma * counts)) / np.sum(counts))

        filled_numerical_contours = one_sigma * 1 + two_sigma * 2 + three_sigma * 3

        return filled_numerical_contours

```

```
def plot_samples(mcmc, res):
        """Plot equal-weight samples."""
        fig = plt.figure()
        ax = fig.add_subplot(111)

        counts, x_edges, y_edges = np.histogram2d(mcmc['samples'][:,0], mcmc['samples'][:,1], bins=res)
        counts = np.flipud(np.rot90(counts))
        equal_weighted_samples = equal_weight(counts, res)

        ax.pcolormesh(x_edges, y_edges, equal_weighted_samples, cmap=plt.cm.gray)

        # Labels
        ax.set_xlabel(r'$\theta_1$', fontsize=16)
        ax.set_ylabel(r'$\theta_2$', fontsize=16)
        ax.tick_params(axis='both', which='major', labelsize=14)
        fig.subplots_adjust(bottom=0.15)

        fig.savefig('equalweight.png')

def marginalize(counts):
        """Find marginalized distribution for each parameter."""
        # Sum columns
        x_counts = np.sum(counts, axis=0)
        # Sum rows
        y_counts = np.sum(counts, axis=1)

        marginalized = {'theta_1': x_counts, 'theta_2': y_counts}

        return marginalized

def plot_marginalized(mcmc, res):
        fig = plt.figure(1, figsize=(7,7))
        fig.subplots_adjust(hspace=0.001, wspace=0.001)
        gs = gridspec.GridSpec(2, 2, width_ratios=[1,4], height_ratios=[4,1])

        counts, x_edges, y_edges = np.histogram2d(mcmc['samples'][:,0], mcmc['samples'][:,1], bins=res)
        counts = np.flipud(np.rot90(counts))

        ax1 = plt.subplot(gs[1])

        filled_numerical_contours = find_numerical_contours(counts)
        ax1.pcolormesh(x_edges, y_edges, filled_numerical_contours, cmap=plt.cm.binary)

        # String to display theta on plot
        theta_1 = np.mean(mcmc['samples'][:,0])
```

```
        theta_2 = np.mean(mcmc['samples'][:,1])
        # ??? display_string = (r'$\bar{\theta}_1 = {0:.4f} $''\n'r'$\bar{\theta}_2 = {1:.4f} $').format(theta_1, theta_2)

        #ax1.pcolormesh(x_edges, y_edges, counts, cmap=plt.cm.gray)
        ax1.set_ylim(min(y_edges), max(y_edges))
        ax1.set_xlim(min(x_edges), max(x_edges))
        contours(mcmc, 'blue', 'dashed', 'x')
        # ??? plt.text(0.6, 0.8, display_string, transform=ax1.transAxes, fontsize=14)
        ax1.tick_params(axis='both', labelleft='off', labelbottom='off')


        marginalized = marginalize(counts)

        ax3 = plt.subplot(gs[3], sharex=ax1)
        ax3.bar(x_edges[:-1], marginalized['theta_1'], x_edges[1]-x_edges[0], color='white')
        ax3.tick_params(axis='both', labelsize=10)
        ax3.tick_params(axis='y', labelleft='off', labelsize=10)
        ax3.set_xlabel(r'$\theta_1$', fontsize=14)
        ax3.set_ylabel(r'P', fontsize=14)
        ax3.set_xlim(min(x_edges), max(x_edges))

        ax0 = plt.subplot(gs[0], sharey=ax1)
        ax0.barh(y_edges[:-1], marginalized['theta_2'], y_edges[1]-y_edges[0], color='white')
        ax0.tick_params(axis='both', labelsize=10)
        ax0.tick_params(axis='x', labelbottom='off')
        ax0.set_ylabel(r'$\theta_2$', fontsize=14)
        ax0.set_xlabel(r'P', fontsize=14)
        ax0.set_ylim(min(y_edges), max(y_edges))

        fig.savefig('marginalized.png')

def ellipse_coords(mean, eigenval, eigenvec, level):
        chi_square = {'1': 2.30, '2': 6.18, '3': 11.83}
        level = str(level)

        axis1 = []
        axis1.append(mean + (np.sqrt(chi_square[level] * eigenval[0]) * eigenvec[:,0]))
        axis1.append(mean - (np.sqrt(chi_square[level] * eigenval[0]) * eigenvec[:,0]))

        axis2 = []
        axis2.append(mean + (np.sqrt(chi_square[level] * eigenval[1]) * eigenvec[:,1]))
        axis2.append(mean - (np.sqrt(chi_square[level] * eigenval[1]) * eigenvec[:,1]))
```

```python

    return axis1, axis2

def ellipse_lengths(a1, a2):
    dx1 = a1[1][0] - a1[0][0]
    dy1 = a1[0][1] - a1[1][1]
    length1 = math.sqrt(dx1**2 + dy1**2)

    dx2 = a2[1][0] - a2[0][0]
    dy2 = a2[0][1] - a2[1][1]
    length2 = math.sqrt(dx2**2 + dy2**2)

    axis1 = {'length': length1, 'coords': a1, 'dx': dx1, 'dy': dy1}
    axis2 = {'length': length2, 'coords': a2, 'dx' : dx2, 'dy': dy2}

    return axis1, axis2

def ellipse_angle(dx, dy):
    angle = math.atan(dx/dy)

    return angle

def find_ellipse_info(mean, eigenval, eigenvec, level):
    a1, a2 = ellipse_coords(mean, eigenval, eigenvec, level)
    axis1, axis2 = ellipse_lengths(a1, a2)

    axis1['xangle'] = ellipse_angle(axis1['dx'], axis1['dy'])
    axis2['xangle'] = ellipse_angle(axis2['dx'], axis2['dy'])

    return axis1, axis2

def contours(info, color, line, mean_marker):
    """Add contour lines and mean to current axes."""
    eigenval, eigenvec = np.linalg.eigh(info['covar'])

    axis11, axis12 = find_ellipse_info(info['mean'].flatten(), eigenval, eigenvec, 1)
    axis21, axis22 = find_ellipse_info(info['mean'].flatten(), eigenval, eigenvec, 2)
    axis31, axis32 = find_ellipse_info(info['mean'].flatten(), eigenval, eigenvec, 3)
    angle = axis12['xangle']
    angle = angle * 180 / math.pi

    ellipse1 = Ellipse(xy=info['mean'], width=axis11['length'], height=axis12['length'], angle=angle, visible=True, facecolor='none', edgecolor=color, linestyle=line, linewidth=2)
```

```
    ellipse2 = Ellipse(xy=info['mean'], width=axis21['length'], height=axis22['length'], angle=angle, visible=True, facecolor='none', edgecolor=color, linestyle=line, linewidth=2)
    ellipse3 = Ellipse(xy=info['mean'], width=axis31['length'], height=axis32['length'], angle=angle, visible=True, facecolor='none', edgecolor=color, linestyle=line, linewidth=2)

    ax = plt.gca()
    ax.add_patch(ellipse3)
    ax.add_patch(ellipse2)
    ax.add_patch(ellipse1)
    ax.set_xlim(-0.4, 0.4)
    ax.set_ylim(0.5, 2.0)
    plt.plot(info['mean'][0], info['mean'][1], marker=mean_marker, mfc='none', mec=color, markersize=8, mew=2)
    sigma1 = {'ax1':axis11['length'], 'ax2':axis12['length'], 'xangle1':axis11['xangle'], 'xangle2':axis12['xangle']}
    sigma2= {'ax1':axis21['length'], 'ax2':axis22['length'], 'xangle1':axis21['xangle'], 'xangle2':axis22['xangle']}
    sigma3 = {'ax1':axis31['length'], 'ax2':axis32['length'], 'xangle1':axis31['xangle'], 'xangle2':axis32['xangle']}

    return sigma1, sigma2, sigma3

def ellipse_boundary(axis, coords, mean):
    angle = axis['xangle2']
    minor = axis['ax1']
    major = axis['ax2']
    meanx = mean[0]
    meany = mean[1]
    x = coords[0]
    y = coords[1]

    boundary = ((math.cos(angle)*(x - meanx) + math.sin(angle) * (y - meany) )**2 /minor**2) + ((math.sin(angle) * (x - meanx) - math.cos(angle) * (y - meany))**2 /major**2)

    return boundary


def check_confidence_regions(sigma1, sigma2, sigma3, samples, mean):
    """Count number of points within each confidence region."""
    sigma1_count = 0
    sigma2_count = 0
```

```
    sigma3_count = 0

    for sample in samples[1000:,:]:
        test1 = ellipse_boundary(sigma1, sample, mean)
        test2 = ellipse_boundary(sigma2, sample, mean)
        test3 = ellipse_boundary(sigma3, sample, mean)

        if test1 < 1:
            sigma1_count += 1
            sigma2_count += 1
            sigma3_count += 1
        elif test2 < 1:
            sigma2_count += 1
            sigma3_count += 1
        elif test3 < 1:
            sigma3_count += 1

    region_count = {'1': sigma1_count, '2': sigma2_count, '3': sigma3_count}
    print('region count')
    print(region_count)
    print('sigma1')
    print(sigma1)
    print('sigma2')
    print(sigma2)
    print('sigma3')
    print(sigma3)

    return region_count


def plot_data(data, posterior_stats, mh, theta):
    """Plot simulated data and analytical result"""
    fig, ax = plt.subplots()
    #Plot data
    err = [0.1 for y in data['y']]
    plt.errorbar(data['x'].flatten(), data['y'].flatten(), yerr=err, marker='x', ls='none')

    # Plot model
    x = np.arange(min(data['x']), (max(data['x']) + (max(data['x'] - min(data['x']))/10)), (max(data['x'] - min(data['x']))/10) )
    ax.plot(x, x*posterior_stats['mean'][1] + posterior_stats['mean'][0])
    plt.xlabel('$x$', fontsize=16)
    plt.ylabel('$y$', fontsize=16)

    if (mh == 0):
        # Display analytical theta values
        theta_1 = posterior_stats['mean'][0][0]
        theta_2 = posterior_stats['mean'][1][0]
        print(posterior_stats['mean'])
```

```
            display_string = (r'$y = \theta_1 + \theta_2 x$' '\n' r'$\theta_1 = {0:.4f}$, $\theta_2 = {1:.4f}$').format(theta_1, theta_2)
            text_x = 0.5
            text_y = 0.8
            plt.text(text_x, text_y, display_string, transform=ax.transAxes, fontsize=16)

            plt.savefig('2ddata.png')

        elif (mh == 1):
            # Plot model
            ax.plot(x, x*posterior_stats['mean'][1] + posterior_stats['mean'][0], linestyle='solid', color='green', linewidth=2, antialiased=True)

            # Plot MCMC result
            x_mc = np.arange(min(data['x']), (max(data['x']) + (max(data['x'] - min(data['x']))/10)), (max(data['x'] - min(data['x']))/10) )
            ax.plot(x_mc, x_mc*theta[1] + theta[0], linestyle='dashed', color='black', linewidth=2,  antialiased=True)
            plt.xlabel('$x$', fontsize=16)
            plt.ylabel('$y$', fontsize=16)

            plt.savefig('2ddata-mcmc.png')


def plot_rotation(mcmc_rot, mcmc, sample_mean, axis1, axis2):
        mcmc_unrot = mcmc['samples']
        # Plot rotated
        fig = plt.figure()
        ax = fig.add_subplot(111)
        ax.plot(mcmc_rot[:,0], mcmc_rot[:,1], '.', c='grey')
        # Plot mean
        mean_x = np.mean(mcmc_rot[:,0])
        mean_y = np.mean(mcmc_rot[:,1])
        ax.plot(mean_x, mean_y, '.k')
        ##  ??? Plot proposal standard deviation
        #x_stdev = (mcmc['proposal_stdev'][0][0])
        #x_stdev = [(mean_x + x_stdev), (mean_x - x_stdev)]
        #y_stdev = (mcmc['proposal_stdev'][1][0])
        #y_stdev = [(mean_y + y_stdev), (mean_y - y_stdev)]
        #x = [mean_x, mean_x]
        #y = [mean_y, mean_y]
        #ax.plot(x, y_stdev, 'k', linewidth=2)
        #ax.plot(x_stdev, y, 'k', linewidth=2)
        # Label axes
        ax.set_xlim(-1.0, 1.0)
        ax.set_ylim(-1.0, 1.0)
```

```
        ax.set_xlabel(r'$\theta_{1}$'' ''$transformed$', fontsize=28)
        ax.set_ylabel(r'$\theta_{2}$'' ''$transformed$', fontsize=28)
        ax.tick_params(axis='both', which='major', labelsize=20)
        fig.subplots_adjust(bottom=0.15, left=0.15)

        fig.savefig('rot.png')

        # Plot unrotated for comparison
        fig = plt.figure()
        ax = fig.add_subplot(111)
        ax.plot(mcmc_unrot[:,0], mcmc_unrot[:,1], '.', c='grey', zorder=-10)
        # Plot mean
        ax.plot(np.mean(mcmc_unrot[:,0]), np.mean(mcmc_unrot[:,1]), '.k')
        ## ??? Plot proposal standard deviation
        ## Transform
        #stdev_y_unrot_plus = np.array([[mean_x], [y_stdev[0]]])
        #stdev_y_unrot_minus = np.array([[mean_x], [y_stdev[1]]])
        #stdev_x_unrot_plus = np.array([[x_stdev[0]], [mean_y]])
        #stdev_x_unrot_minus = np.array([[x_stdev[1]], [mean_y]])

        #stdev_y_unrot_plus = circle_to_ellipse(stdev_y_unrot_plus, sample_mean, axis1, axis2)
        #stdev_y_unrot_minus = circle_to_ellipse(stdev_y_unrot_minus, sample_mean, axis1, axis2)
        #stdev_x_unrot_plus = circle_to_ellipse(stdev_x_unrot_plus, sample_mean, axis1, axis2)
        #stdev_x_unrot_minus = circle_to_ellipse(stdev_x_unrot_minus, sample_mean, axis1, axis2)
        #
        #ax.plot([stdev_x_unrot_plus[0], stdev_x_unrot_minus[0]], [stdev_x_unrot_plus[1], stdev_x_unrot_minus[1]], 'k', linewidth=2)
        #ax.plot([stdev_y_unrot_plus[0], stdev_y_unrot_minus[0]], [stdev_y_unrot_plus[1], stdev_y_unrot_minus[1]], 'k', linewidth=2)

        # Label axes
        ax.set_xlabel(r'$\theta_{1}$', fontsize=28)
        ax.set_ylabel(r'$\theta_{2}$', fontsize=28)
        ax.set_xlim(-0.4, 0.4)
        ax.set_ylim(0.8, 2.0)
        ax.tick_params(axis='both', which='major', labelsize=20)
        fig.subplots_adjust(bottom=0.15)

        fig.savefig('unrot.png')
```

```python

def main():
        measurement_uncertainty = 0.1
        data, posterior_stats = setup(measurement_uncertainty)
        print('analytical mean:')
        print(posterior_stats['mean'])

        mcmc_init = metropolis_hastings(posterior_stats)

        eigenval, eigenvec = np.linalg.eigh(mcmc_init['covar'])
        axis1, axis2 = find_ellipse_info(mcmc_init['mean'].flatten(), eigenval, eigenvec, 2)

        mcmc, mcmc_rot, acceptance_ratio = metropolis_hastings_rot(posterior_stats, mcmc_init['mean'], axis1, axis2)
        # mh = 0 for plot without mcmc line, 1 for with it
        plot_data(data, posterior_stats, 0, mcmc['mean'])
        plot_data(data, posterior_stats, 1, mcmc['mean'])
        plot_rotation(mcmc_rot, mcmc, mcmc_init['mean'], axis1, axis2)
        print('mcmc mean:')
        print(mcmc['mean'])
        plot_samples(mcmc, 200)
        plot_marginalized(mcmc, 200)

if __name__ == '__main__':

        main()
```

References

[1] Metropolis et al. (1953) Equation of State Calculations by Fast Computing Machines. *The Journal of Chemical Physics*. [Online] 21 (6), 1087-1092. Available from: doi: 10.1063/1.1699114 [Accessed 21 October 2013].

[2] Newman, M., E., J., Barkema, G., T. (1999) *Monte Carlo Methods in Statistical Physics*. New York, Oxford Universit Press.

[3] Halton, J., H. (1970) A Retrospective and Prospective Survey of the Monte Method. *SIAM Rev.*, Vol. 12, No. 1, pp. 1-63

[4] Trotta, R. (2012) Statistics of Measurement: Summary Handout. London, Imperial College London, p. 14

[5] Gilks, W. R., Richardson, S., Spiegelhalter, D., J. (1996) *Markov Chain Monte Carlo in Practice*. London, Chapman and Hall.

[6] Gelman, A., Roberts, G. O., Gilks, W. R., (1996) Efficient Metropolis Jumping Rules. *Bayesian Statistics 5*. [Online] Oxford University Press. Available from: http://www.stat.columbia.edu/ gelman/research/published/baystat5.pdf [Accessed 24 November 2013]

[7] C. J. Geyer (2011) Introduction to Markov Chain Monte Carlo. In: Brooks, S., Gelman, A., Jones, G., Meng, X. (eds.) *Handbook of Markov Chain Monte Carlo*. Chapman and Hall/CRC Handbooks of Modern Statistical Methods. Florida, Chapman and Hall/CRC, pp. 3-47.