Dynamic Gauss Newton Metropolis Algorithm

The MCMC Jagger

Mehmet Ugurbil

New York University, Courant Institute of Mathematical Sciences

251 Mercer Street
New York, NY 10012
01/2016

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
Department of Mathematics
New York University
01/2016

Jonathan Goodman

Contents

Abstract

GNM: The MCMC Jagger. A rocking awesome sampler.

This python package is an affine invariant Markov chain Monte Carlo (MCMC) sampler based on the dynamic Gauss-Newton-Metropolis (GNM) algorithm. The GNM algorithm is specialized in sampling highly non-linear posterior probability distribution functions of the form $e^{-||f(x)||^2/2}$, and the package is an implementation of this algorithm.

On top of the back-off strategy in the original GNM algorithm, there is the dynamic hyper-parameter optimization feature added to the algorithm and included in the package to help increase performance of the back-off and therefore the sampling. Also, there are the Jacobian tester, error bars creator and many more features for the ease of use included in the code.

The problem is introduced and a guide to installation is given in the introduction. Then how to use the python package is explained. The algorithm is given and finally there are some examples using exponential time series to show the performance of the algorithm and the back-off strategy.

Section 1: Introduction

1.1 The Problem

The GNM algorithm is for sampling generalizations of probability densities of the form $p(x) \propto e^{-||f(x)||^2/2}$. Here $x \in \mathbb{R}^n$ is a "parameter" vector, $f : \mathbb{R}^n \to \mathbb{R}^m$ is a "model" function, and $||f(x)||^2 = \sum_{k=1}^m f_k^2(x)$. Typically $m > n$. GNM is a powerful algorithm for ill-conditioned problems because it is affine invariant.

This is related to nonlinear least squares $\min_x ||f(x)||^2$ that arises in statistics. For example, if $m(x)$ predicts the outcome of an experiment with parameter x, and y_k are measurements, then the goodness of fit is $\sum_{k=1}^m \frac{(m_k(x)-y_k)^2}{2\sigma_k^2}$. If $f_k(x) = \frac{m_k(x)-y_k}{\sigma_k}$, then this sum has the form $||f(x)||^2/2$.

One generalization to the probability density function $p(x)$ is adding a Gaussian prior probability distribution on the parameters, $\pi : \mathbb{R}^n \to \mathbb{R}$, such that $\pi(x) = \frac{1}{Z} e^{-(x-m)^T H(x-m)/2}$, where m is the mean and H is the precision matrix, the inverse of the covariance. Another generalization is having an indicator function $\chi : \mathbb{R}^n \to \{0, 1\}$ to limit the domain of the model function.

The user of the sampler must write Python code to evaluate $\chi(x)$, $f(x)$ and $\nabla f(x)$, the Jacobian. It is optional to add prior information, mean vector $m \in \mathbb{R}^n$ and precision matrix $H \in \mathbb{R}^{n \times n}$ (inverse of the covariance matrix).

1.2 Installation

The **gnm** is a Python package. The Python package **numpy** is required before the execution of **gnm**. To use the examples and plot the results, you will need **matplotlib**. To use the *acor* feature, you will need the package **acor**.

From the default Python packages, you will need **os**, **setuptools** (or **distutils**), **re**, **sys**, **copy**, and **json**. These packages likely come with your Python installation.

The easiest way to install **gnm** would be to use pip.

```
$ pip install gnm
```

To download manually, use git clone or download as zip from https://github.com/mugurbil/gnm.

```
$ git clone https://github.com/mugurbil/gnm.git
```

Then you can install manually by going into the **gnm** directory, and then running setup.py.

```
$ python setup.py install
```

To clean the repository after installation, one can run clean with setup.py.

```
$ python setup.py clean
```

Section 2: Code

2.1 User Function

The user needs to create Python code that contains the model information. This Python function, which we will call the user function, should have two inputs, and three outputs. This code will be passed to the sampler.

The first input should be the parameter vector, $x \in \mathbb{R}^n$. The second input should be the arguments that the function takes, which may include the data.

The first output is the constraint $\chi(x)$. It is supposed to tell where the model function is defined. Thus, it would have value 1 (or True) when the function and its derivative are defined, and 0 (or False) otherwise. If D is the domain of f and ∇f, then $\chi(x) = \begin{cases} 1 & x \in D \\ 0 & x \notin D \end{cases}$. The second output is an array of size m of f evaluated at x, $f(x) \in \mathbb{R}^m$ (range), that should be convertible to an **np.array**. The third output is an array of size m by n that is the Jacobian matrix, ∇f, evaluated at x, $J(x) \in \mathbb{R}^{m \times n}$. This also needs to be convertible to an **np.array**.

Listing 1: User defined function outline.

```
1 def model(x, args):
2     ...
3     return chi(x), f(x), J(x)
```

2.2 The Sampler

The sampler needs the user function (**model**), arguments the user function takes (**args**), and the initial guess of the sampler (**x_0**). This information is provided to the sampler during initialization. The function will be evaluated at the initial guess to check if it is defined at that point.

Listing 2: Sampler initialization.

```
1 jagger = gnm.sampler(x_0, model, args)
```

The prior information is optional. For a Gaussian prior distribution, mean vector m ($\mathbf{m}$) and precision matrix H ($\mathbf{H}$) need to be provided.

Listing 3: Setting the prior.

```
1 jagger.prior(m, H)
```

2.3 Jacobian Test

The **Jtest** method is to check whether the derivative information and the model function given in the user function match by employing numerical differentiation. It has two required inputs, six optional inputs, and one output.

The domain, D_J, for **Jtest** should be a rectangle given by two vectors, the top right and the bottom left corners of the rectangle, $\mathbf{x}_{\max}$ and $\mathbf{x}_{\min}$ respectively. These vectors are the two required inputs. Note that they should have the same dimensions, and all values of $\mathbf{x}_{\min}$ should be less than the corresponding values of $\mathbf{x}_{\max}$ so that the domain is nonempty.

$$D_J = \{\mathbf{x} : \forall i, (\mathbf{x}_{\min})_i < \mathbf{x}_i < (\mathbf{x}_{\max})_i\}$$

The method has one output, **error**, that tells whether the numerical Jacobian, Df, converged to the Jacobian supplied by the user function, ∇f. The output is 0 if the convergence occurred. Otherwise, it is the norm of the difference of the numeric and provided Jacobians, $||Df - \nabla f||$, at the point it failed to converge.

Listing 4: Jtest usage.

```
1 error = jagger.Jtest(x_min,x_max)
```

```
2 assert error == 0 #Is the Jacobian info correct?
```

2.4 Advanced Jtest

The six optional inputs of **Jtest** and their default values are given below. We then explain how the method works and where these variables come into use.

Table 1: **Jtest** optional inputs.

SYMBOL	IN CODE	DEFAULT VALUE
dx	**dx**	$2 \cdot 10^{-4}$
N	**N**	1000
$\epsilon_{\max}$	**eps_max**	$1 \cdot 10^{-4}$
p	**p**	2
$l_{\max}$	**l_max**	50
r	**r**	0.5

Jtest chooses a i.i.d. uniform random variable, $\mathbf{x}_k$, in its specified domain. Then it calculates the numerical Jacobian at that point by the symmetric difference quotient. This operation is done for all entries in the Jacobian, for $j = 1$ to n, and for each j, $i = 1 \to n$. Note that there is no i in the code since the operations are done in vectors.

$$(D^{(l)}\mathbf{f}(\mathbf{x}_k))_{ij} = \frac{f_i(\mathbf{x}_k + \delta\mathbf{x}_j^{(l)}) - f_i(\mathbf{x}_k - \delta\mathbf{x}_j^{(l)})}{2\delta_j^{(l)}} \approx \frac{\partial f_i}{\partial x_j}(\mathbf{x}_k)$$

Here $\delta\mathbf{x}_j^{(l)} = \delta_j^{(l)} * \mathbf{e}_j$, where $\mathbf{e}_j$ has 0s everywhere but a 1 at the jth entry, and $\delta_j^{(l)}$ is the magnitude of the perturbance in $\mathbf{e}_j$ direction at the lth stage. The initial perturbance magnitude, $\delta_j^{(0)}$, depends on the domain and dx by the equation $\delta_j^{(0)} = (\mathbf{x}_{\max} - \mathbf{x}_{\min})_j * dx$.

The numerical Jacobian is compared under $\mathbf{p}$ norm to the Jacobian given by the user function evaluated at $\mathbf{x}_k$, $\epsilon_k^{(l)} = ||D^{(l)}\mathbf{f}(\mathbf{x}_k) - \nabla\mathbf{f}(\mathbf{x}_k)||_p$. If this error, $\epsilon_k^{(l)}$, is smaller than $\epsilon_{\max}$, this point passes the differentiation test. Otherwise, the magnitudes of the perturbations are reduced by a multiplication with r: $\delta_j^{(l)} = \delta_j^{(l-1)} * r = \delta_j^{(0)} * r^l$, $\forall j$.

This is repeated for the same point a maximum amount of $l_{\max}$ times, so that $l \leq l_{\max}$. If the point is still generating an error greater than $\epsilon_{\max}$, the Jacobian is said to be incorrect. This is because the numerical Jacobian is not convergent to the provided Jacobian as we decrease $\delta\mathbf{x}$ repeatedly, $\delta\mathbf{x} \to \mathbf{0}$. In this case, the program returns $\epsilon_k^{(l_{\max})}$, the final error at this point.

This procedure is repeated for N different points. If all the points pass the differential test, then the method returns 0.

Listing 5: Advanced **Jtest** usage.

```
error = jagger.Jtest(x_min,x_max,N=5,dx=0.001)
```

2.5 Back-Off

There are currently two types of back-off strategies, **static** and **dynamic**. Static way is by fixing the maximum number of back-offs as well as the back-off dilation factor. Dynamic way is by fixing the max number of back-offs but changing the dilation factor based on the current position of the sampler.

Listing 6: Setting back-off.

```
jagger.static(5, 0.2)
# OR
jagger.dynamic(3)
```

2.6 Sampling

Once all the information is given to the sampler, all it requires is to call the **sample** method with the number of samples wanted (**n_samples**).

The sampling process can be divided into sessions. If you run the sample method again, the sampler will remember where it left off and continue the sampling from there. Also, you can divide sampling **n_samples** into **n_divs** by setting the option *divs*=**n_divs**.

If used on terminal with **n_divs**> 1, to see the progress of sampling, *visual* option could be set to **True**. Turning *visual* to **True** shows the percentage of the sampling completed.

The sampling can be done in safe mode by setting *safe*=**True**. This will cause the sampler to save progress at every division, and once the sampling is completed.

Listing 7: Sampling.

```
1 jagger.sample(n_samples, divs=n_divs,
2                visual=True, safe=True) # sample
```

The initial guess might be far away from the region where the chain is supposed to be, causing delay before the chain converges to the desired distribution. After the sampling, these undesired initial samples can be burned (discarded) by calling **burn** and providing the number of samples to be burned (**n_burned**).

Listing 8: Burning samples.

```
1 jagger.burn(n_burned) # burn
```

2.7 Outputs

Outputs of the sampler can be accessed after the sampling as properties. These outputs are provided in table 2. The main output is the **chain**, which contains the Markov chain of the samples. This is an **np.array** of shape (**n_samples**, n). Thus, it has length equal to the number of samples and each sample is a vector of size n containing the position of that sample.

Listing 9: Information access.

```
1 chain = jagger.chain
```

Table 2: Outputs.

CODE	EXPLANATION
chain	The Markov chain
n_samples	Number of elements sampled
n_accepted	Number of samples accepted
accept_rate	Acceptance rate of the algorithm
call_count	Number of function calls
step_count	Array of size **max_steps** indicating how many samples got accepted at each back-off step

There is a method called **error_bars** for ease of use. It evenly divides the space given by a rectangle into bins and sums up the number of samples that fall into each bin. Also, it estimates the error bars for each bin. It requires three inputs: an integer specifying the number of bins in each dimension (**n_dims**), a vector specifying the minimum of each dimension of the rectangle (**d_min**), and a vector specifying the maximum (**d_max**). These vectors need to be castable to **np.array** and have dimension n. The method produces three outputs: the

estimate of the posterior (**p_x**), the estimate of the error bars (**err**), and the mid-point location of each bin (**x**). All three of these outputs are of form **np.array** and of shape $(n, \mathbf{n\_dims})$.

Listing 10: Error bars.

```
1 x,p_x,err = jagger.error_bars(n_dims,d_min,d_max)
```

Section 3: Algorithm

The Gauss-Newton-Metropolis algorithm and the back-off strategy is taken from Zhu [1]. The dynamic back-off is original.

3.1 Gauss-Newton-Metropolis

We cannot sample from the posterior distribution directly as it may be highly non-linear. Instead, we can create an approximation, such as a Gaussian approximation, from which we can sample directly. For this purpose, as in the Gauss-Newton algorithm, we are going to replace the value of the function f by its approximation. That is, if we are sampling z from x, we are going to replace $f(z)$ by its approximation around x: $f(x) + \nabla f(x)(z - x)$. Hence, we get the proposal distribution:

$$K(x, z) = \frac{1}{Z}\pi(z)e^{-||f(x)+\nabla f(x)(z-x)||^2/2}$$

What may not be clear is that this approximation is Gaussian and therefore can be sampled directly. Up to a factor of $-\frac{1}{2}$, the exponential part of this proposal is:

$$(z - m)^T H(z - m) + ||f(x) + \nabla f(x)(z - x)||^2$$

After some algebra, as a function of z, this may be reduced to:

$$z^T(H + ||\nabla f(x)||^2)z - 2z^T(Hm - \nabla f(x)^T f(x) + ||\nabla f(x)||^2 x) + O(1)$$

Therefore, we can complete the square to get $(z - \mu)^T P(z - \mu)$ by setting:

$$P = (H + ||\nabla f(x)||^2) \text{ and } \mu = P^{-1}(Hm - \nabla f(x)^T f(x) + ||\nabla f(x)||^2 x)$$

Thus, we get that $K(x, z) \sim N(\mu, P)$, which is Gaussian and can be sampled directly. Then we use the Metropolis-Hastings algorithm to achieve our original goal of sampling $p(x)$. We need to have an acceptance probability A to satisfy detailed balance:

$$p(x)K(x, z)A(x, z) = p(z)K(z, x)A(z, x)$$

$$\text{Therefore, } A(x, z) = \min\left\{1, \frac{p(z)K(z, x)}{p(x)K(x, z)}\right\}$$

3.2 Back-Off

Algorithms that are not globally convergent may have trouble at points that are away from the optimum. The proposal at these points would not be good approximations of the distribution that is being simulated. This leads to a low acceptance probability, and hence a large autocorrelation time. This situation is undesirable and needs to be overcome. Back-off strategy is a way to overcome this difficulty.

In Markov chain Monte Carlo algorithms, first a proposal is made, then it is accepted according to an acceptance recipe. If it is rejected, the next element in the chain is the same as the previous element. Thus, the chain remains rigid in place. To overcome this, rather than rejecting a sample that is not accepted, the proposal distribution is dilated and a new proposal is made. This is motivated by step size control methods used in line search.

We need to generalize the detailed balance condition and update the acceptance recipe accordingly to accommodate for the back-off. This can be achieved by the "very detailed balance" and the new acceptance recipe:

$$A(x, z_n|z_1, ..., z_{n-1}) = \min\left\{1, \frac{p(z_n)K_1(z_n, z_1)[1 - A(z_n, z_1)] \cdots K_n(z_n, x)}{p(x)K_1(x, z_1)[1 - A(x, z_1)] \cdots K_n(x, z_n)}\right\}$$

The back-off strategy raises the question: how do we choose the dilation factor? The simple way to choose the back-off dilation factor is by setting it to a constant value.

3.3 Dynamic Back-Off

A better way to choose the step size is to note where we want the algorithm to sample most. This would be the area with the highest probability, where $p(x)$ has the highest value. We can approximately achieve this by minimizing $||f(x)||^2$ since $p(x)$ is inversely proportional to it:

$$p(x) = \chi(x)\frac{1}{Z}\pi(x)e^{-||f(x)||^2/2} \propto e^{-||f(x)||^2/2}$$

We can minimize $||f(x)||^2$ in a search direction by polynomial interpolation [2]. If $(z - x)$ is the search direction, then we want to minimize $\phi(t) = ||f(x + t(z - x))||^2$ over t. This gives $\phi'(t) = 2f(x + t(z - x))\nabla f(x + t(z - x)) \cdot (z - x)$. So we know the four values $\phi(0)$, $\phi(1)$, $\phi'(0)$, and $\phi'(1)$ that can be used to employ cubic interpolation and find the approximate minimum. This is used as our dilation factor.

Section 4: Examples

4.1 Quick Start

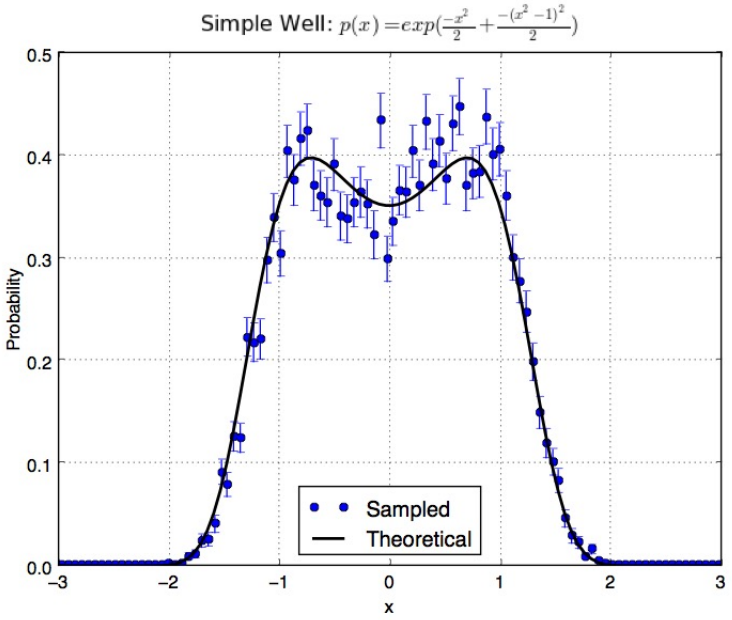
The quickstart.py file in ∼/gnm/examples directory contains a simple example. Running this file should create a plot that looks like Figure 1.

```
$ python quickstart.py
```

In this example, $n = m = 1$, $f(x) = \frac{x^2 - y}{\sigma}$, $f'(x) = \frac{2x}{\sigma}$. The prior has mean $m = 0$ and precision matrix $H = [1]$, giving $\pi(x) = \frac{1}{Z} e^{-x^2/2}$. Therefore, the probability distribution that we want to sample is:

$$p(x) = \frac{1}{Z} \pi(x) e^{-|f(x)|^2/2} = \frac{1}{Z} e^{-x^2/2 - (x^2 - y)^2/(2\sigma^2)}$$

Figure 1: Sampling results for the simple well problem.



4.2 Well

The problem in quick-start can be made harder by deepening the well, that is taking y to be bigger. This example is in the ∼/well folder. The following line will give all the options for the file that you can play around with.

```
$ python well.py -h
```

4.3 Jtest and Simple 2D

In this example we first check the usage of Jtest, then sample a simple 2D example. There is a rotator for visiulization so that we can plot any cut of the posterior probability distribution along the plane.

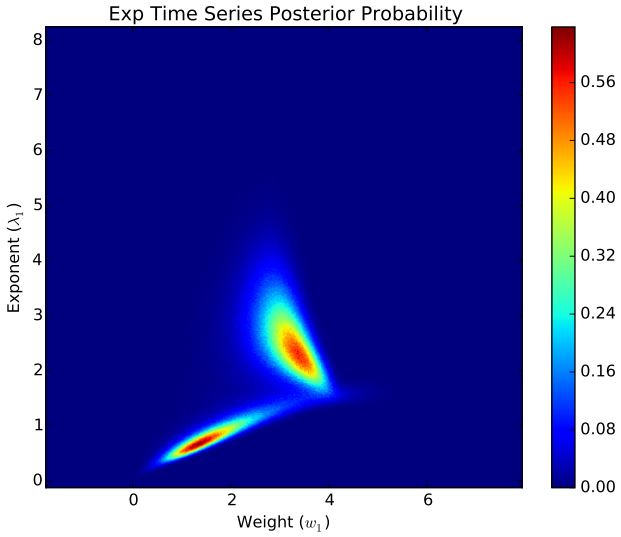
4.4 Exponential Time Series

Suppose we have a process that describes the relationship between time and data by $g(t) = \sum_{i=1}^{n} w_i e^{-\lambda_i t}$. We have measurements $y_k = g(t_k) + \epsilon_k$ for $k = 1, ..., m$, where the noise is independent and identically distributed and $\epsilon_k \sim N(0, \sigma_k^2)$. Then the parameter is $x = (w_1, ..., w_d, \lambda_1, ..., \lambda_d)$, with $n = 2d$. The model function is given by $f_k(x) = \frac{g_k(t,x) - y_k}{\sigma_k}$ for $k = 1, ...m$.

Experiment was done for $n = 4$ and $m = 10$. The sampler was run for 10^5 samples and first 2000 was burned in. The mean of the prior was taken to be $[4., 2., 0.5, 1.]$ and the precision matrix was chosen as the identity multiplied with 0.5. The initial guess is the mean of the prior. Data is created by taking the input as $[1., 2.5, 0.5, 3.1]$ and adding random normal noise with standard deviation 0.1.

A 2D marginal histogram of the probability distribution sampled, λ_1 versus w_1, is shown in Figure 2. Figure 3 presents a 1D marginal histogram of the

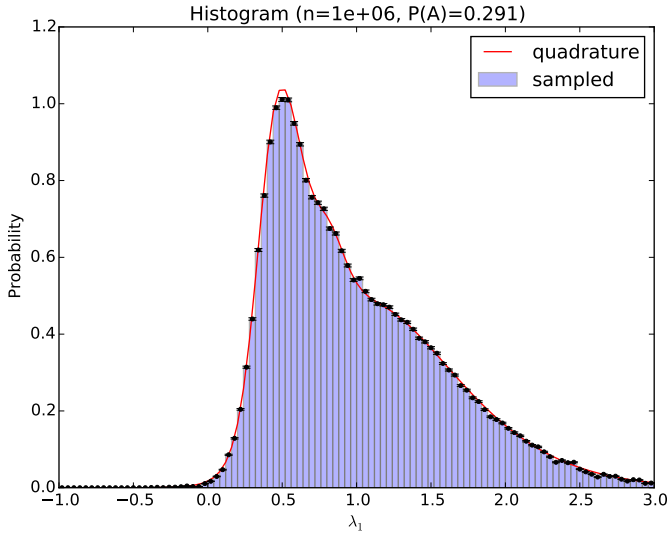
Figure 2: 2D marginal posterior probability function for exponential time series for variables λ_1 versus w_1.



probability distribution sampled, for λ_1. This figure also displays quadrature (theoretical) curve versus the sampled probability function where we can observe the correctness of the algorithm.

In this experiment we see significant gains using back-off both in the acceptance rate and in the autocorrelation time after analyzing Table 4, however this conclusion does not hold for the dynamic back-off. Since lower autocorrelation time implies higher effective sample size, we have for instance 8850 effective samples with 1 back-off step versus 5952 effective samples for no back-off steps. These translate to $\frac{8850}{1.75*10^5} = 0.0506$ versus $\frac{5952}{1.12*10^5} = 0.0531$ effective samples per function call implying that the efficiency increases with back-off. This number drops down significantly together with efficiency, to $\frac{9434}{2.66*10^5} = 0.0355$, for 3 back-off steps. This suggests that using too many back-offs leads to too many function calls to get the same amount of effective samples, making it inefficient.

Figure 3: 1D marginal posterior probability function for exponential time series for the variable λ_1.



In Figure 4, we can observe that the covariance of the chain is stable implying that the chain is stationary. We can see in Figure 5 the percentage of samples accepted at each step (-1 is reject). This example shows that using too many back-off steps does not help significantly, however adding the first back-off step increases the acceptance rate of the algorithm significantly, around 30%.

Table 3: Comparison of sampling strategies for sampling exponential time series with different back-off strategies.

Max Steps	**Dilation Factor**	**Acceptance Rate**	**Acor Time**	**Effective Size**	**Number of Function Calls**
0	-	0.273	2880	3470	$1.00 * 10^7$
1	Dynamic	0.653	1720	5810	$1.73 * 10^7$
1	0.1	0.603	1390	7180	$1.73 * 10^7$
1	0.5	0.411	1760	5680	$1.73 * 10^7$
2	0.1	0.812	1510	6610	$2.12 * 10^7$

Figure 4: Covariance of the first component of the markov chain.

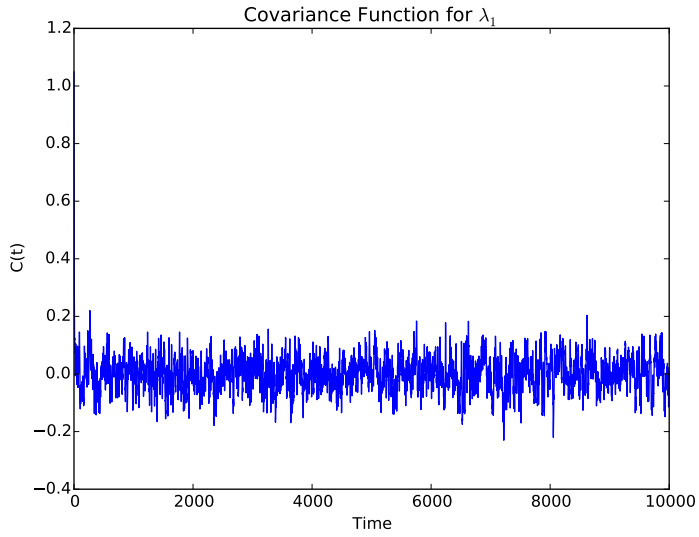
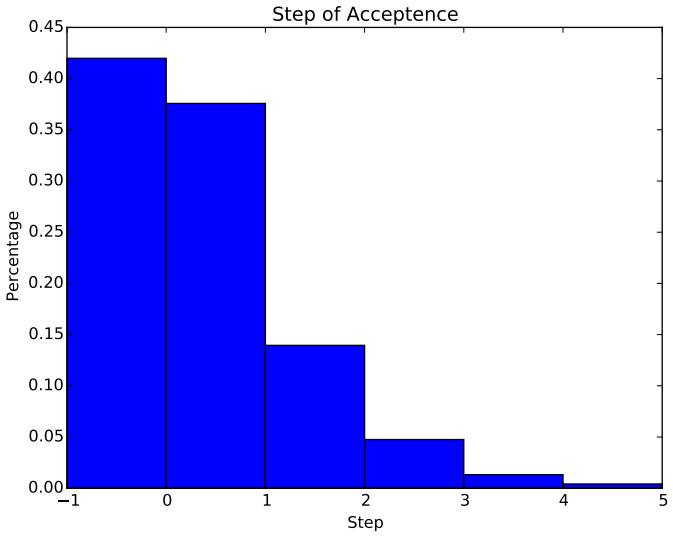


Figure 5: Percentage of samples accepted at each back-off step.



Section 5: References

[1] B. Zhu. *Gauss-Newton-Metropolis with backup strategy.* Courant Institute of Mathematical Sciences, New York University, New York, NY, 2013.

[2] J. Nocedal and S. Wright. *Numerical Optimization.* Springer, New York, 2000.

[3] J. Goodman and J. Weare. *Ensemble samplers with affine invariance.* CAMCS, Vol. 5 (2010), No. 1, pp. 6580.

[4] W. R. Gilks, S. Richardson and D. J. Spiegelhalter. *Markov Chain Monte Carlo in practice.* Chapman, 1996.