GNM

The MCMC Jagger

Mehmet Ugurbil

mu388@cims.nyu.edu

Contents

Section 1: Introduction

GNM: The MCMC Jagger. A rocking awesome sampler.

This python package is an affine invariant Markov chain Monte Carlo (MCMC) sampler based on the Gauss-Newton-Metropolis (GNM) algorithm. This algorithm is specialized in sampling highly non-linear posterior distributions. There are also the back-off strategy and the dynamic hyperparameter optimization features included in the package to help increase performance.

Section 2: Installation

To use the **gnm** package you need to have the package **numpy** installed. To use the examples and plot the results, you will need **matplotlib**. To use the *acor* feature, you will need **acor**.

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

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

```
$ pip install gnm
```

If you want to download manually, you can find the package from the website http://cims.nyu.edu/~mu388/gnm. Then you can install manually by going into the **gnm** directory, and then runing setup.py.

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

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

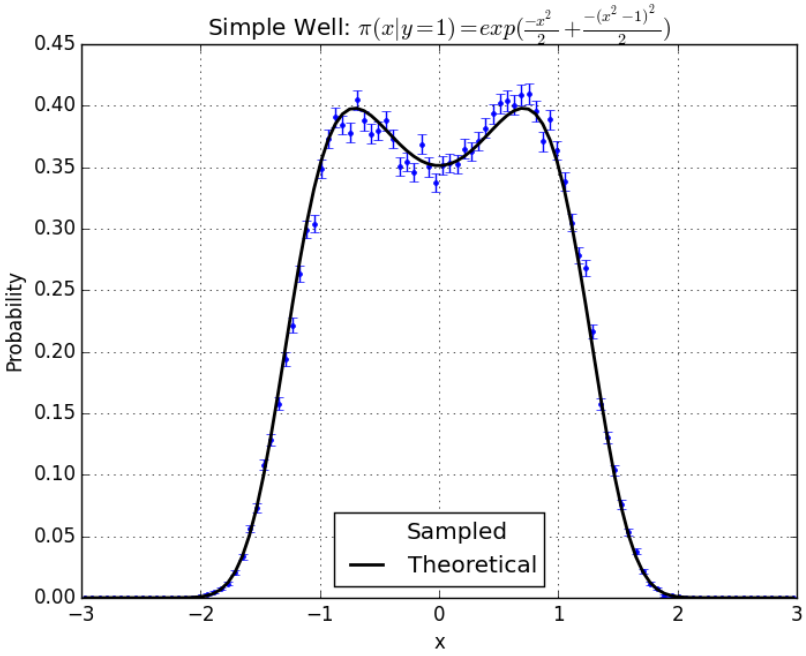
```
$ python setup.py clean
```

Section 3: Quick Start

The quickstart.py file in ∼/gnm/examples directory contains a simple example. Running this file will create a plot that looks like the following graph.

```
$ python quickstart.py
```

Figure 1: Quickstart



To understand what is going on in this file, please read sections 4.1, 4.2, and 5.1. First you will learn how to write your own function, then how to use the developer function and then how to use the sampler.

Section 4: Function

The user has a model function $f : \mathbb{R}^n \to \mathbb{R}^m$ with its derivative information $J_f : \mathbb{R}^n \to \mathbb{R}^{n \times m}$. We need to turn this into python code so that we can use it in the sampler.

4.1 User Function

The user needs to create a python function that contains the model function information. This python function should have two inputs and three outputs. This user function will be fed into the developer function to create an instance of the function.

The first input should be the input vector from the domain, $\mathbf{x} \in \mathbb{R}^n$, and the second input should be a dictionary of the arguments that the function takes. The argument will also be fed into the developer function to create an instance of the user function.

The first output is a Boolean that is supposed to tell when the function is defined, so that it would have value False (or 0) when the function (or its derivative) is not defined. The second output is the vector of f evaluated at $\mathbf{x}$, $\mathbf{f}(\mathbf{x}) \in \mathbb{R}^m$ (range). The third and last output is the Jacobian matrix of f evaluated at $\mathbf{x}$, $\mathbf{J_f}(\mathbf{x}) \in \mathbb{R}^{m \times n}$.

Listing 1: User defined function outline

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

4.2 Developer Function

The developer function is a python class contained in the file func.py, and it is also named **func**. It has three methods, the standard __init__ and __call__ methods and **Jtest**.

When initializing the developer function, the user is creating a function instance of the user function. To do this the user needs to supply two inputs, the user function, **f**, and the specific arguments that create the instance, **args**. These inputs are regarded as properties.

A call to the developer function calls the underlying user function with the specific arguments. So the call has one input, the input vector $\mathbf{x}$, and three outputs of the user function.

The **Jtest** method is to check the derivative information and the model function match by employing a numerical differentiation. This method is more complicated than the previous two. It has two required inputs, six optional inputs, and one output.

The domain, D, for **Jtest** should be a rectangle given by two vectors, the top right and the bottom left corners of the rectangle, $D = \{\mathbf{x} : \forall i, (\mathbf{x}_{\min})_i < \mathbf{x}_i < (\mathbf{x}_{\max})_i\}$. 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.

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

Listing 2: Developer function usage

```
1 f_0 = gnm.func(f, args_0)
```

```
error = f_0.Jtest(x_min,x_max)
assert error == 0 #Is the Jacobian info correct?
```

4.3 *Jtest in Detail

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.

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

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 $\mathbf{x}_k$, $\epsilon_k^{(l)} = ||\mathbf{J_f}(\mathbf{x}_k) - \nabla^{(l)}\mathbf{f}(\mathbf{x}_k)||_p$. 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, a constant: $\delta_j^{(l)} = \delta_j^{(l-1)} * r = \delta_j^{(0)} * r^l$, $\forall j$.

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 3: Sample **Jtest** usage

```
1 error = f_0.Jtest(x_min,x_max,N=5,dx=0.001)
```

Section 5: Sampler

The introduction covers a quick start and the other subsections further the flexibility of use.

5.1 Introduction

The sampler needs a Gaussian prior distribution with mean vector **m** and precision matrix **H**, which is the inverse of the covariance matrix. It also requires the data, **y**, with observational error that is assumed to be i.i.d. $N(0, \sigma^2)$. That is, $y_i \sim N(y_i^{real}, \sigma^2)$ for all i. This information is provided to the sampler during initialization together with the function instance that is created using the developer function (See section 4).

If you have no information on the prior, just set **H to be very small.**

Listing 4: Sampler Initialization

```
f_0 = ... # function instance, see section 4
m = [_, ...]   # prior mean - vector
H = [[_, ...], ...] # prior precision - matrix
y = [_, ...]   # data - vector
sigma = _ # data standart deviation - float
gnm_sampler = gnm.sampler(m, H, y, sigma, f_0)
```

The initial guess of the sampler will be the mean of the prior. If this needs to be changed, the **guess** method needs to be called with the initial guess. This method resets the initial guess and makes sure the function is defined at that point.

Listing 5: Setting initial guess

```
gnm_sampler.guess(x_0) # set initial guess
```

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

Listing 6: Sampler Usage

```
1 n_samples = _ # number of samples
2 gnm_sampler.sample(n_samples) # sample
```

If used on terminal, to see the progress of sampling, **vsample** can be used instead of **sample** method. The visiual sample method, **vsample**, shows the percentage of the sampling completed.

After the sampling, one can burn the first few samples by calling **burn** and providing the number of samples to be burned.

Listing 7: Burning samples

```
1 n_burned = _ # number of samples to be burned
2 gnm_sampler.burn(n_burned) # burn
```

5.2 Statistics