Readme — MC Stretch

Alex Kaiser
Courant Institute of Mathematical Sciences, New York University
adkaiser@gmail.com

June 20, 2013

This document describes MC Stretch, a software package for Markov chain Monte Carlo sampling. The package is suitable for sampling generic continuous probability distributions. The algorithm for MCMC is called the Stretch Move, and is described in detail in section 2. It can be used for parameter estimation, integration, or any generic problem where sampling is required.

The code includes high performance parallel implementations of this algorithm. The package is written in C with OpenCL, which allows for control of the GPU (graphics processing unit) or other devices. It is designed to offer good parallel performance on difficult sampling problems.

Care has been taken to make the package easy to use. The user provides generic C code to evaluate a PDF, adjusts a few parameters and the code does the rest. To use the package effectively, the user needs nearly zero knowledge of parallel programming, OpenCL or GPU hardware .

1 Quickstart

Follow these steps to get started quickly with the included examples.

1. Place generic C code to evaluate the log of the probability density function to sample in "pdf.h."
2. Pick one of the included examples in "stretch_move_main.c" to run. Check that parameters in the example match your problem, which usually means setting the correct dimension.
3. Type "make" on the command line to build the code with the included makefile.
4. Type "./stretch_move_main" to run the executable. The code will prompt to select an OpenCL implementation and hardware, then run the sampler.

2 The Stretch Move Ensemble Sampler

The MCMC algorithm I investigate is called the stretch move [1]. The algorithm has a similar structure to Metropolis, and still uses a proposal and an accept/reject step. In this algorithm, we use a group or *ensemble* of sequences. Each member of the ensemble is called a *walker*. After burn-in, each walker is distributed according to the invariant distribution. On each iteration the algorithm generates multiple samples. The PDF of each walker is independent of the other walkers. The walker is correlated with its own previous state.

To update one walker, another walker is randomly selected from the *complementary ensemble*. The algorithm requires an auxiliary random variable Z with PDF $g(z)$, which must satisfy $g(1/z) = (1/z)g(z)$. This distribution is usually $g(z) = 1/\sqrt{2z}$ on $(1/a, a)$ for $a > 1$. A move is then proposed alone a line between the current walker k and the complementary walker j according to the distribution

$$Y = X(j) + z(X(k) - X(j))$$

and then accepted or rejected.

The algorithm is efficient and can be parallelized. The computation required to update a single walker is comparable to other MCMC methods. To accomplish this, split the walkers into two groups X_{red}, X_{black}. The walkers in each group can be updated simultaneously. If all the walkers were updated at once, there would be undefined behavior, since the update may or may not have finished on the complementary walker.

The algorithm is *affine invariant*, which means that for A, a linear transformation, and vector b, sampling a PDF $g(x) = Af(x) + b$ is equivalent to sampling f then applying A and b after. Heuristically, this means that the algorithm is not sensitive to skewed distributions. Because of this, the algorithm generally has better autocorrelation time as compared to other MCMC methods.

A single iteration to update one walker is described in algorithm 1. This iteration can be run in parallel for a each group of walkers.

Algorithm 1 Stretch Move Step to update $X_{red}(k, t)$

```
1: Randomly select a walker from the complementary ensemble $X_{black}(j, t)$
2: Sample $z \sim g(Z)$.
3: Compute a proposed move $Y$
       $$Y = X_{black}(j, t) + z(X_{red}(k, t) - X_{black}(j, t))$$
4: Compute the likelihood ratio
       $$q = z^{N-1} \frac{f(Y)}{f(X_{red}(k, t))}$$
5: If $q > 1$, accept. Otherwise, accept with probability $q$.
6: If accept $X_{red}(k, t+1) = Y$, else $X_{red}(k, t+1) = X_{red}(k, t)$.
```

The factor of z^{N-1} in the likelihood ratio ensures the invariant distribution is preserved.

3 A Short Description of GPUs and OpenCL

This package is written in C and OpenCL. OpenCL is a set of extensions to C that allow the user to access the GPU for general computations. The program execution in controlled by a standard CPU program. The CPU is also referred to as the *host*, and the GPU as the *device*. An OpenCL program that runs on the device is called a *kernel*. The device may not be a GPU, as OpenCL also runs on other types of hardware, but I will focus on the GPU here.

A GPU has a lot of parallelism on a small, relatively inexpensive chip. A GPU thread is called a *work-item*. OpenCL automatically manages the parallel execution of work-items on the GPU. The work-items run in a group of threads called a *work-group*. The execution of a work-group is SIMD, or "single instruction, multiple data." The work-items in a work-group execute the same instructions simultaneously, in parallel. Instructions that would not be executed in a serial program (for example, an else clause in an if statement that evaluates to true) are still executed, but their values are thrown away. This means that branches and loops which run for a variable number of iterations makes performance slow. The GPU can also perform latency hiding, which means that it can execute instructions while waiting for memory reads and writes to complete.

A GPU has a large piece of RAM called *global memory*. It has a smaller, faster piece of memory called *local memory* which is shared within a work-group. Local memory must be managed manually. There is a still smaller, very fast piece of memory called *private memory*, or *registers*, that is owned solely by a work-item. This is the default memory location for literals and statically defined arrays. If a work-item uses too much private memory, OpenCL will place the data in global. This usually results in bad performance. The GPU also has an interconnect to CPU memory. On modern chips this is high-bandwidth, but also high latency.

An included package called Ranluxcl, see [2], is included for generating random numbers.

Here's what this means for using the package. The user specifies the PDF function in standard C code, with a few restrictions. There is no access to I/O, so all data must be passed to the function in memory rather than read from a file. There is no dynamic memory allocation, recursion or function pointers.

For performance, use a big enough work-group to get SIMD parallelism, that means hundreds. Use enough walkers to keep the GPU busy, that means thousands or tens of thousands of walkers. Anything that runs for a variable amount of time will run as slow as the slowest if your work group. It may be worth working to make sure a Newton's solve, for example, finishes in a small number of iterations. If performance is still bad, consider more sophisticated memory management or other optimizations. See section 6 for more advice.

4 Using the code

4.1 Building

The code uses generic C with OpenCL extensions and library calls. You must have OpenCL and a C compiler installed to use this package, but there are no other external requirements. A generic makefile is supplied that should work on most Unix or Linux machines, including OSX.

4.2 Examples

The file "stretch_move_main.c" includes two examples of using the sampler. The first shows the simplest possible example of running the sampler. The second uses more features of the code. It uses data, runs burn-in and performs sampling. Is computes basic statistics including autocorrelation time, calculates histograms and outputs that data to a file.

4.3 Setting things up

First, you need to specify the PDF you want to sample. For numerical reasons, the code requires the logarithm of the PDF, rather than the PDF itself. The PDF does not need to be normalized. To do this, place C code to evaluate the PDF in the file "pdf.h."

Suppose one wanted to sample an N dimensional Gaussian with PDF

$$f_X(x) \propto \exp\left(-\frac{1}{2}\sum_{i=0}^{N}(x_{i+1}-x_i)^2\right).$$

where by definition $x_0, x_{N+1} = 0$. Then the log PDF function would be implemented as

```
float log_pdf(PROPOSAL_TYPE *x, __global const data_struct *data_st,
              DATA_ARRAY_TYPE *data){
        float sum = x[0]*x[0] + x[NN-1]*x[NN-1];
        for(int i=0; i<NN-1; i++)
            sum += (x[i+1] - x[i]) * (x[i+1] - x[i]);
        return -0.5f * sum;
}
```

The dimension "NN" is set up at kernel compile time by the initialization routine and is always available in this function. The data structure "data_st" and array "data" are passed even if they are not used. PROPOSAL_TYPE and DATA_ARRAY_TYPE are always float, but using this definition allows easy changes from global to local, as described below. This is for generality, since the package handles all the data movement.

Next, define needed constants in "constants.h." These may define the dimension of the problem or observations, values for the prior, loop bounds, or anything else.

Also, if the PDF requires additional data, add the types to "data_struct.h." Because of OpenCL rules, you must also specify this structure in "pdf.h" for the kernel to use. By default this contains a float, you may add any scalars or small statically defined arrays.

If you wish to include large arrays, use the buffer "data." This is a single float array, and all the arrays must be packed into this array. OpenCL is not flexible to complicated structures involving variable numbers of pointers or pointers to pointers. The best general structure is to insist that the user gets one data array, and should unpack it manually.

Consider sampling a multivariate normal with mean $\vec{\mu}$ and covariance matrix Σ. The PDF is given

$$f(\vec{x}) \propto \exp\left[-\frac{1}{2}(\vec{x}-\vec{\mu})\Sigma^{-1}(\vec{x}-\vec{\mu}) \right].$$

We want to pass a vector "mu" of length NN for the mean, and an NN by NN matrix for the inverse covariance "inv_cov." Set these to be contiguous in memory in the array "data" which is passed to the initialization routine. For example, allocate "data" to be a length NN + NN^2 float array. Suppose one wanted $\mu = (0, 1 \ldots \text{NN-1})$ and Σ^{-1} to be a (-1,2,-1) tridiagonal matrix. Initialize the data as follows:

```
for(int i=0; i < dimension; i++)
    data[i] = (cl_float) i;

for(int i=dimension; i < data_length; i++)
    data[i] = 0.0f;

for(int i=0; i < (dimension-1); i++){
    data[ i   +     i*dimension + dimension ] =  2.0f;
    data[ i+1 +     i*dimension + dimension ] = -1.0f;
    data[ i   + (i+1)*dimension + dimension ] = -1.0f;
}
data[ (dimension-1) + (dimension-1)*dimension + dimension ] = 2.0f;
```

The package handles moving the packed array to the device.

In your code for the PDF, set pointers or otherwise read the data. For example:

```
// The first elements of the data array are the means
DATA_ARRAY_TYPE *mu       = data;

// NN elements later is the inverse covariance matrix
DATA_ARRAY_TYPE *inv_cov  = data + NN;
```

Now these pointers can be used as normal arrays to evaluate the PDF. These pointers must be in the same address space as the data array. Use pointers rather than reallocate and copy to avoid wasting time on memory operations. This is illustrated in full in the second included example.

The default code places the data and proposal in local memory for speed. If your data is too large, then it needs to be kept in global. Similarly, if the number of walkers or dimension is large, the proposals take too much memory to fit in local. To set the data to be in global, remove the definition

```
#define USE_LOCAL_DATA
```

from "pdf.h." To set the proposals to be in global, remove the definition

```
#define USE_LOCAL_PROPOSAL
```

4.4 Sampling

Now that the PDF and definitions are are in place, initialize a sampler object. Set up some parameters for your run, start with the chain length. This is the number of ensemble samples that will be run when you start the sampler.

```
cl_int chain_length       = 100000;
```

Next is the dimension of the problem:

```
cl_int dimension          = 10;
```

Set the size of each half of the ensemble, which also corresponds to the total parallelism available:

```
cl_int walkers_per_group  = 1024;
```

Set the work group size, which must divide the number of walkers per group.

```
size_t work_group_size    = 128;
```

Tell the sampler which components to save

```
cl_int num_to_save        = 2;
cl_int *indices_to_save   = (cl_int *) malloc(num_to_save * sizeof(cl_int));
indices_to_save[0]        = 0;
indices_to_save[1]        = 3;
```

Set the length parameter

```
double a = 2.0;
```

Call the initialization routine to allocate all the necessary arrays and compile the OpenCL kernels.

```
sampler *samp = initialize_sampler(
                    chain_length,          dimension,
                    walkers_per_group,     work_group_size,
                    a,                     0,
                    0,                     NULL,
                    num_to_save,           indices_to_save,
                    CHOOSE_INTERACTIVELY,  CHOOSE_INTERACTIVELY);
```

Use these default values for the other arguments.

Then run the burn-in for 10000 steps:

```
run_burn_in(samp, 10000);
```

The sampler is now ready. Run it:

```
run_sampler(samp);
```

The array `samp->samples_host` is now filled with samples ready for use. Samples are stored in "component major" order, so to access component `i` of sample `j`, use

```
samp->samples_host[i + j*samp->N];
```

The included examples contain also illustrate more features of the package, including generating histograms and basic statistics.

5 Using the Sampler for Inference

The code also contains an example of Bayesian posterior sampling. This is contained in the branch "sde_inference." To see the example, run

```
git checkout sde_inference
```

5.1 Model

The setup is as follows. Let N denote the total dimension of the sample, X the state variable and N_X the number of X components. Let Y denote the observation and N_Y its dimension. Also, denote the number of model parameters as N_θ and the number of steps in the path as N_{steps}.

Each sample also keeps a path $X(1)\dots X(N_{steps})$ for the solution of the dynamics. The total dimension of the state variable is $N = N_\theta + N_{steps}$, which includes all the parameters and the state at each discrete time. The initial condition $X(0)$ is fixed and known throughout. Let f describe the deterministic part of the dynamics and g the deterministic part of the observation.

The dynamics evolves according to the SDE

$$dX = -aXdt + \sigma dW$$

where dW is Brownian motion.

The observations are given

$$Y = bX + \zeta Z$$

where $Z \sim N(0,1)$. All observation noise is assumed to be independent.

The prior is

$$P(\theta) \propto 1$$

if θ is in a particular range. The range is problem specific, with defaults shown in figure 1.

	min	max
a	-10	10
σ	0	10
b	-10	10
ζ	0	10
X	-100	100

Figure 1: Default boundaries for the uniform prior

Changes to these boundaries can make a big difference in convergence and accuracy.

Define $X_{j,pred}$ to be the deterministic model update by integrating the deterministic part of the dynamics. The initial condition for this timestep is the previous sample value. Define $X_j(t)$ to be the current value in the sample. Define $Y_{j,pred}(t) = g(X_j(t))$. This is the no-noise prediction for the given dynamics. It is dependent on the X in the current sample according to the observation model. Define $Y_{j,obs}(t)$ is the true noisy observation at the same time. Finally, Define h to be the timestep for the dynamics.

The distribution is given

$$P(D|\theta) \propto \frac{1}{(\sigma_1 \dots \sigma_{NX})^{N_{steps}}} \frac{1}{(\zeta_1 \dots \zeta_{NY})^{N_{obs}}} \\ \exp\left[-\frac{1}{2} \left(\sum_{t=1}^{N_{steps}} \sum_{j=1}^{NX} \frac{(X_{j,pred}(t) - X_j(t))^2}{h\sigma_j^2} + \sum_{t_{obs}=1}^{N_{obs}} \sum_{j=1}^{NY} \frac{(Y_{j,pred}(t_{obs}) - Y_{j,obs}(t_{obs}))^2}{\zeta_j^2} \right) \right]$$

The double sums are for the components of X and Y. Also the h in the X component is a constant. It does not appear in the leading coefficient because it is absorbed into the proportionality.

The de-normalized log of the distribution is given

$$\begin{aligned} l(D|\theta) = \quad & -(N_{steps}) \sum_{j=1}^{NX} \log(\sigma_j) \quad -(N_{obs}) \sum_{j=1}^{NY} \log(\zeta_j) \\ & - \frac{1}{2} \left(\sum_{t=1}^{N_{steps}} \sum_{j=1}^{NX} \frac{(X_{j,pred}(t) - X_j(t))^2}{h\sigma_j^2} + \sum_{t_{obs}=1}^{N_{obs}} \sum_{j=1}^{NY} \frac{(Y_{j,pred}(t_{obs}) - Y_{j,obs}(t_{obs}))^2}{\zeta_j^2} \right) \end{aligned}$$

This is the form shown in the examples.

The examples uses the standard setup for inference problems

$$P(\theta|D) = \frac{P(\theta)\ P(D|\theta)}{P(D)}$$

The value $P(D)$ is a constant, unknown and ignored.

The example runs simulated annealing, burn-in and samples. It estimates the posterior means. It computes histograms and outputs them to files.

6 Questions and issues

Here are some issues that you may run across in using the code.

- The code does not compile.

  The code needs to link to various OpenCL headers. The makefile assumes they are in a standard place. If they are not, you must modify the makefile accordingly.

  Sometimes with AMD compilers, there is difficultly finding the random number generator file. If so, set the AMD flag in "constants.h" to one to change the include path.

  Contact me if this doesn't work.

- What are walkers and how should I set them up?

  The algorithms run with an ensemble of random walks, which itself is a Markov chain. On every iteration after convergence, each walker is an independent sample of the invariant distribution. For the algorithm to be valid, you must have more walkers than the dimension of the problem. The walkers must span the space, in a linear algebra sense, so they cannot lie in a hyperplane that is a proper subset of the space. Generally, for performance you want to use a lot of walkers, think thousands or tens of thousands.

  Default initialization is to put a random uniform(0,1) value in each component of each walker. If your walkers are all very far from a region with lots of probability mass, they may move completely randomly or not move at all. If there is a region with zero probability, and the walkers are initialized there, the algorithm will almost certainly have trouble converging. If you think you know where the interesting regions of probability mass are, you can set the walkers to start near there. This is a risk, because they may not leave and explore the rest of the space.

- My sampler is not converging and my histograms are noise.

  Some suggestions:

  - Test and debug your PDF. Subtle errors in the PDF may produce very subtle wrong answers, or may overwhelm the sampler and just give noise.
  - Try running simulated annealing as shown in the inference example. Run for a lot of steps.
  - Run the burn-in for very long.
  - What is the dimension of your problem? Experiments have shown that in high dimensions or with very many walkers a very long burn in may be required. If the dimension is extremely large, the size of the state space may be so large that convergence may not occur at all.

- What is this data packing thing?

  As mentioned above, OpenCL does not handle structures involving variable numbers of pointers or pointers to pointers. This is the best compromise that allows flexibility for users. Note that there are better solutions, but they require modifying much more of the code. See the following question and email for help.

- I have a complicated data structure that I need to evaluate my likelihood function. I hate packing my data. What should I do?

  This must be handled manually. Modify the current code and add arguments. Email for help.

- How do I check what GPUs are available on my system and pick one?

  If you run the initialize kernel with the "CHOOSE_INTERACTIVELY" option the code will output lists of the available platform. When you choose interactively, you will see a menu that looks something like this:

  ```
  Choose platform:
  [0] Advanced Micro Devices, Inc.
  [1] NVIDIA Corporation
  [2] Intel(R) Corporation
  Enter choice:
  ```

  Suppose we want to use NVIDIA. Type 1 at the terminal, and the code will output another menu.

  ```
  Choose device:
  [0] GeForce GTX 590
  [1] GeForce GTX 590
  Enter choice:
  ```

  Type 0 or 1 to pick your desired GPU.

- How do I set the GPU so I don't have to pick interactively?

  Run in interactive mode to display the available devices. Suppose we wanted to follow the above example, we would change the platform and device strings to:

  ```
  const char *plat_name = "NVIDIA";
  const char *dev_name  = "GeForce GTX 590";
  ```

- My performance is terrible.

  Some suggestions:

  – How many walkers are you using? Try using more. Think 2^{10}, 2^{14} or more as memory allows. Make sure you run lots of burn-in when using so many walkers.

  – Are you running a GPU? The code may give decent performance running threaded on CPUs, but it may not.

  – Now that you're on the GPU, what is your work group size? It should decently big, try 64 or 128 or even larger. Powers of two are good.

  – Is the GPU also running your monitor at the same time? This will make everything slow. Laptop GPUs are frequently slow.

  – Is the time to evaluate your PDF highly variable? If you have, for example, an iterative solve like Newton's method, the SIMD nature of the GPU means all your work-items (GPU threads) run as slow as your slowest work-item. If you can improve the worst case performance you may improve the overall very much.

  - Is the time to evaluate you PDF very fast? You may be spending all your time waiting for memory to be sent to the host. This is not very common.

- Why are these cl_whatever datatypes all over the place?

  These datatypes guarantee a particular size and alignment to the compiler. Use them or risk difficult to diagnose, very confusing bugs. They cover the full set of basic C types and more, so you might as well.

- I get an error saying "invalid arg size." What is this?

  In OpenCL, data types have to be the specified width and have correct alignment. This means that you must match the data exactly — if the kernel calls for an int, you should pass a "cl_int" and not a "cl_long." Passing an int value to a function which expects a long int frequently doesn't matter in C. This is not true in OpenCL and will cause bugs. This is a good reason to use OpenCL datatypes throughout your C code.

- I get an error regarding local memory.

  As mentioned above, the default code places the data and proposal in local memory. If either data is too large remove

  ```
  #define USE_LOCAL_DATA
  ```

  and if the proposals are too large remove

  ```
  #define USE_LOCAL_PROPOSAL
  ```

  from "pdf.h."

References

[1] Goodman, J., and Weare, J. Ensemble samplers with affine invariance. *Commun. Appl. Math. Comput. Sci. 5* (2010), 65–80.

[2] Nikolaisen, I. U. ranluxcl v1.3.1, 2011. https://bitbucket.org/ivarun/ranluxcl/.