Brownian Motion Program Documentation

The Brownian Motion Equation

I have written a program to calculate Brownian Motion paths based on this equation:

results[j] = results[j-1] + random * sqrt((double) T / N).

Note the following points about this equation:

- results is the array to store the location of the Brownian Motion particle at each timestep. The index values of the array represent the timesteps.
- results[0] = 0
- random is a pseudo random number generated along the normal distribution with a mean of 0 and a variance of 1
- N is the number of timesteps calculated for each Brownian Motion path
- T is a value that scales the random number. I'm not sure of it's function exactly.
- Nor do I understand the logic of this equation, honestly. I took it from this paper that prof Simen emailed me and Tao: An Algorithmic Introduction to Numerical Simulation of Stochastic Differential Equations by Desmond J. Higham.The program that I wrote is to be imported in the Python interpreter, from which the user can play around the provided functions.

How to Use the Program

The program is called GenerateNumPy3. It's main purpose is to generate a NumPy array that stores the user's desired number of Brownian Paths, all generated in parallel on crispr's 3 GPU's. To access the program, navigate on crispr to /home/thomasnemeh/CudaPrograms/BrownianMotion. Then run the python 3.6 interpreter. You

can do that by typing /home/psimen/anaconda3/bin/python3.6. Then type the following in the python interpreter:

>>>import GenerateNumPy3

In order to lessen the amount you need to type you can name the program something when importing like so:

>>>import GenerateNumPy3 as gnp

Once the program is imported, the following manual will be displayed, listing the functions available to the user. Make sure to read this manual to understand what the program does:

Functions:

getPaths(int T, int N, int numSims, double lowerThreshold, double upperThreshold): returns list of each time step of each Brownian Motion simulation.

getCrossingTimes(): returns list of time that each of the N simulations crossed either threshold. If the number is negative, then the simulation crossed the lower threshold at the absolute value of that timestep. If the number is positive, then the simulation crossed the upper threshold at that timestep. If the number is 0, then the simulation never crossed either threshold. The number at the first index corresponds to the first simulation in the getPaths array, the second index to the second simulation, etc.

getNoCross(): return the number of simulations that never crossed either threshold.

plotHistogram(): plots a histogram of the array returned by getCrossTimes().

The following functions are to be used if you cannot access a graphical interface for the histogram from the command prompt:

witeToFile(fileName): writes the getCrossTimes() array to a file.

readFromFile(fileName): reads the getCrossTimes() array from a file

getNoCross(array): input cross times array after reading and writing to file

plotHistogram(array): input cross times array after reading and writing to file

---

As can be seen from reading the above manual, the program allows the user to generate Brownian paths, generate an array that contains the crossing time for each path of either the upper or lower threshold, calculate the number of simulations that do not cross any threshold, display the crossing times array as a histogram, and read the arrays from and write the arrays to a file if a graphical interface isn't available or to store the arrays for later. After reading the manual, the user just needs to type the functions into the interpreter. For example, if the user wants to generate a list of paths where T = 1, N = 500, the number of paths generated is 50,000, upperThreshold = 1, and lowerThreshold = -1, then the user can type the following in the interpreter:

```
>>>array = gnp.getPaths(1, 500, 50000, -1, 1)
```

*Note about the cross times array:

Alternate Version

I have written another version of the program called GenerateUpperTNP, standing for generate upper threshold NumPy. This program has the same functionality as GenerateNumPy3, but only has as upper threshold. It now occurs to me that you can make it a lower threshold by inputting a value less than zero for the threshold. The idea, however, was to write a version of the program with only an upper threshold to see if that the histogram for this version has an inverse gaussian shape, if I recall correctly. The getPaths function is the only part of this program that is different, since it eliminates the lower threshold parameter lie so:

getPaths(int T, int N, int numSims, double upperThreshold)

Other Versions and Which Files Go Together

I also preserved earlier iterations of the program:

- GenerateNumPy: a simple version of the program that can only calculate the brownian motion path for one simulation. Unlike the other programs, it does not have any additional functionality beyond this.
- GenerateNumPy2: a version of the program that has all the same capabilities as the most advanced version, but it only utilizes one of crispr's 3 GPUs. The most advanced version, GenerateNumPy3, utilizes all 3 of crispr's GPUs.

Each version of the program comprises several files. To demonstrate this, I will list all the files that form GenerateNumPy3 and explain their function:

- *3gpubm.cu*: a cuda/c++ file that essentially does all the heavy lifting of the program by creating the array storing the brownian motion paths and the array storing the cross time for each simulation, and the storing these arrays on the heap.
- 3gpubm.h: the file where the functions contained in 3gpubm.cu are defined. 3gpubm.cu contains the actually code, 3gpubm.h is simply where the functions are originally defined. This file is necessary because cython cannot interface with .cu files, only c++ files.
- libgpubm.so: 3gpubm.cu is compiled into this shared library. This step is also necessary so that cython can interface with the Cuda code.
- GenerateNumPy3.pyx: this cython code that wraps the arrays created by the cuda code into NumPy arrays that can be used by python. Cython is a programming language that

allows for both Python and C code. Happily, this code does not copy the underlying data allocated in the heap, just converts it to a NumPy array. This is where all the functions available to the user from the python interpreter are defined.

- SetupNumPy3.py: cython files require a python file to compile them. This file is to compile GenerateNumPy.pyx
- SetupNumPy3.py is compiled into a shared library called GenerateNumPy3.cpython-36m-x86_64-linux-gnu.so. When the user imports GenerateNumPy3 in the python interpreter, this is the file that they are importing.

Here is a list of the files corresponding to the other versions of the program:

- GenerateNumPy2: CythonBM.cu | CythonBM.h | libCythonBM.so | GenerateNumPy2.pyx | SetupNumPy2.py | GenerateNumPy2.cpython-36m-x86_64-linux-gnu.so
- GenerateNumPy: SimpleBrownianMotion.cu | CythonSBM.h | libCythonSBM.so | GenerateNumPy.pyx | SetupNumPy.py | GenerateNumPy.cpython-36m-x86_64-linux-gnu.so
- GenerateUpperTNP: UpperBM.cu | UpperBM.h | libUpperBM.so | GenerateUpperTNP.pyx | SetupUpperTNP.py | GenerateUpperTNP.cpython-36m-x86_64-linux-gnu.so

I also have .cu files for testing purposes that simply print out the arrays and report how long they took to generate, without saving them in the heap to be imported into python. The values used are hardcoded into the program rather than passed as input arguments.

- 3gpubmTest.cu corresponds to GenerateNumPy3

- ❏ UpdatedBm.cu corresponds to GenerateNumPy2

I have others there but I only ever use these two.

Compilation Instructions

In my CudaPrograms/BrownianMotion folder, I have a file called CompilationInstructions.txt that describes how to compile the program. I will copy the text and paste it here.

1. Put the following files in the same directory: 3gpubm.cu, 3gpubm.h, SetupNumPy3.py, GenerateNumPy3.pyx

2. Add the nvcc compiler to your path by typing the following:

PATH="/usr/local/cuda-9.1/bin:$PATH"

3. Compile CythonBM.cu into a shared library by typing the following:

nvcc --compiler-options '-fPIC' -shared -o libgpubm.so 3gpubm.cu
-I/home/psimen/anaconda3/lib/python3.6/site-packages/numpy/core/include/numpy
-I/home/psimen/anaconda3/include/python3.6m

You should now have file in your directory called libgpubm.so.

4. Now you must modify the SetupNumPy3.py file. Open it and you should see the following:

Extension('GenerateNumPy3',
sources=['GenerateNumPy3.pyx'],
library_dirs=['/usr/local/cuda-9.1/lib64',
'/home/thomasnemeh/CudaPrograms/BrownianMotion'],

libraries=['cudart', gpubm],

language='c++',

runtime_library_dirs=['/usr/local/cuda-9.1/lib64',
'/home/thomasnemeh/CudaPrograms/BrownianMotion'],

include_dirs =
['/home/psimen/anaconda3/lib/python3.6/site-packages/Cython/Includes',
'/usr/local/cuda-9.1/include', '/home/psimen/anaconda3/lib/python3.6/site-packages']

There are places where "/home/thomasnemeh/CudaPrograms/BrownianMotion" is listed. One is in library_dirs and the other is in runtime_library_dirs. Delete "/home/thomasnemeh/CudaPrograms/BrownianMotion" and replace it with the name of the directory you put the files in.

4. Now compile SetupNumPy3.py by typing the following:

/home/psimen/anaconda3/bin/python3.6 SetupNumPy3.py build_ext --inplace

---

Final Note:

To get a 200 timestep average crossing time use these values:

T = 2.8

N = 600

Upper Threshold = 1

Lower Threshold = -1