Homework Assignment 4

Eugene Shvarts
Stats 250 – Baines – Fall 2013 – UC Davis

Problem #1

In this question, we implement a kernel to obtain samples from a truncated normal random variable, and test our code.

---

a. Write a kernel in CUDA C to obtain samples from a truncated normal random variable of the form:

$$X \sim TN\big(\mu, \sigma^2; (a, b)\big) \equiv N(\mu, \sigma^2)\mathbf{1}_{[a,b]}$$

Code attached.

I used a naive rejection sampler with a tolerance of `maxnaive` attempts, and then switched over to the acception-rejection method detailed in Robert (2009) for at most `maxtries` attempts, after which I return `NA` in frustration. The method allows either or both truncation end point to be finite.

b. Compile your CUDA kernel using `nvcc` and check it can be launched properly.

I used the RCUDA instructions and (eventually) had no issues. My rng was initialized to $[1, 2, 3]$. Output was correctly returned, captured to file on `lipschitz`, and sent to my machine. I used

```
.cuda(mykernel, out = x, n, mu, sigma, lo, hi, rnga, rngb, rngc, maxn, maxt,
gridDim = A$grid_dims, blockDim = A$block_dims, outputs = 'out')
```

where A is the output of the gridsize utility, and everything else is vectorized where necessary.

c. Sample 10,000 random variables from $TN(2, 1; (0, 1.5))$, and verify the expected value (roughly) matches the theoretical value.

The theoretical expected value, if ϕ is the standard normal and $\phi_{\mu,\sigma}$ is $\mathcal{N}(\mu, \sigma^2)$, is

$$\frac{\int_A x\phi_{\mu,\sigma}}{\int_A \phi_{\mu,\sigma}} = \mu + \sigma\frac{\int_{A'} x\phi}{\int_{A'} \phi} \quad ,$$

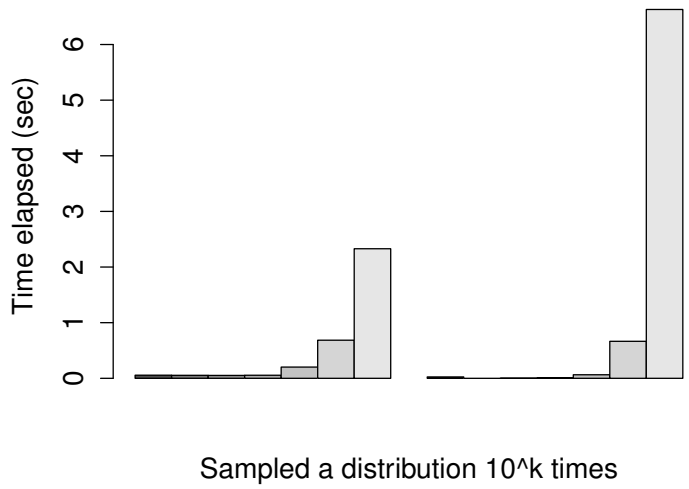
where A is the original interval, and A' is the interval with endpoints mapped through $x \mapsto (x-\mu)/\sigma$. For our parameters, Matlab computes an expected value of 0.9570066. The simulated values from `rtruncnorm.cu` have mean 0.9513158; one part in one hundred error looks like good agreement (obeying the square-root law, since we had ten thousand samples).

d. Write an R function for sampling truncated normal random variables, and sample 10,000 random variables from this function. Verify the mean (roughly) matches the theoretical values.

Code attached.

I tried several test runs; the samples produced consistently have a mean between 9.44 and 9.60. This is a good enough agreement for 10,000 samples.

e. Time the GPU and CPU code for a range of samples / iterations.



I ran my CUDA (through RCUDA) and R code both on an AWS GPU instance, via the included `compute_times.R`. I plotted the third component of the system time, i.e., the elapsed time. 10^8 samples was too many for CUDA to handle via my code, but it's clear that the GPU will vastly outstrip the CPU at that order; on the order of 10^7 samples (keeping in mind each is running potentially thousands of sub-computations in the accept-reject routine) they both take about 2 seconds, and then the relative difference in computing time grows exponentially.

f. , g. Verify that both codes work when a and/or b are infinite. Check an extreme region, e.g., $(\mu, \sigma, a, b) = (0, 1, -\infty, -10)$.

I took 10,000 samples from 3 different distributions using both GPU and CPU, and computed their means along with the actual expected value according to `quadgk` in Matlab.

	GPU mean	CPU mean	Expectation
TN(4,4,-Inf,Inf)	3.9648	4.0051	4.0000
TN(-1,2,0,Inf)	1.2800	1.2717	1.2822
TN(0,1,-Inf,-10)	-10.0980	-10.0984	-10.0981

Problem #2

In this question you will implement Probit MCMC i.e., fitting a Bayesian Probit regression model using MCMC. This model turns out to be computationally nice and simple, lending itself to a Gibbs sampling algorithm with each distribution available in sample-able form. The model is as follows:

$$\begin{aligned} Y_i \,\big|\, Z_i &\sim \mathbf{1}_{Z_i>0} \\ Z_i \,\big|\, \beta &\sim \mathcal{N}(x_i^T\beta, 1) \\ \beta &\sim \mathcal{N}(\beta_0, \Sigma_0) \quad , \end{aligned}$$

where β_0 is a $p \times 1$ vector corresponding to the prior mean, and Σ_0^{-1} is the prior precision matrix.

a. , b. Write an R function `probit_mcmc` to sample from the posterior distribution of β using the CPU only. Then, modify the code so that all samples of Z_i (i.e., all samples from a truncated normal) are performed by the GPU.

Code attached.
Running `source('probit_mcmc')` loads RCUDA, all of the other relevant code I've made for the module, and your grid-size utility. β is returned as an `mcmc` object, and both thinning and burn-in are available.

c. Test both of your codes on the test file `mini_data` generated via `sim_probit` .

While I can get my CPU regression code to work fine, and the stand-alone GPU-based sampler works like a charm, for some reason no matter how much I debug, it is absolutely impossible for me to combine the probit on CPU with sampling via GPU – I repeatedly get a cornucopia of CUDA errors (CUDA_ERROR_LAUNCH_FAILED , error launching CUDA kernel 400 , ...). I must be missing something simple; in any case my code is attached and on Github.

d. Analyze as many of the large datasets generated by `sim_probit` as you can, with both CPU and GPU code, and compare the run times. Use `niter = 2000, burnin = 500`.

For the reasons mentioned above, I could only get CPU results (running on AWS, in this case). I ran

```
system.time({
a <- as.matrix(read.table('./data/data_0i.txt',header=TRUE,sep=" "));
probit_mcmc(a[,-1],a[,1]) } )
```

with the asked-for parameters set as defaults, with $i = 1, 2, 3, 4, \not{5}$. The results:

data_01	data_02	data_03	data_04	data_05
00:16	02:36	23:22	3:50:00	Loooong

I would've let the code run overnight, but I felt it was more prudent to turn it in. It would have been nice to do the GPU comparison; because the number of truncated-normal samples taken is $n + n\#_{mcmc} = O(n)$, so we should get roughly the same exponential (at least?) rate of improvement in the relative computation speed, where the CPU is outstripped within an order of magnitude or so of $n = 10^6$.