Stat243: Problem Set 4, Due Nov. 13

November 4, 2013

This covers material in Units 10 and 11 on computer numbers and linear algebra.

It's due at the start of class on Wed., Nov. 13.

Please follow the guidelines we have discussed previously on formatting your answers. Solutions that are not formatted will have points taken off. **Also note that handwritten solutions to the non-computer problems are fine, so long as your handwritten solutions are easy to follow. If you do have some handwritten solutions, in your paper submission, please have your answers to problems be in order, by stapling together printed and handwritten pages as needed.**

Please note my comments in the syllabus about when to ask for help and about working together.

Problems

1. In class we discussed that integers as large as 2^{53} can be stored exactly in the double precision floating point representation. Demonstrate how the integers 1, 2, 3, ..., $2^{53}-1$ can be stored exactly in the $(-1)^S \times 1.d \times 2^{e-1023}$ format where d is represented as 52 bits (I'm not expecting anything particularly formal - just write out for a few numbers and show the pattern). Then show that 2^{53} and $2^{53}+2$ can be represented exactly but $2^{53}+1$ cannot, so the spacing of numbers of this magnitude is 2. Finally show that for numbers starting with 2^{54} that the spacing between integers that can be represented exactly is 4.

2. Here we'll consider the effects of adding together numbers of very different sizes. Let's consider adding the number 1 to 10000 copies of the number 1×10^{-16}. Mathematically the answer is obviously $1 + 1 \times 10^{-12} = 1.000000000001$ by multiplication, but we want to use this as an example of the accuracy of summation with numbers of very different magnitudes, so consider the sum $1 + 1 \times 10^{-16} + \cdots + 1 \times 10^{-16}$.

   (a) How many decimal places of accuracy are the most we can expect of our result (i.e., assuming we don't carry out the calculations in a dumb way). In other words, if we store 1.000000000001 on a computer, how many decimal places of accuracy do we have?

   (b) In R, create the vector $x = c(1, 1 \times 10^{-16}, \ldots, 1 \times 10^{-16})$. Does the use of *sum()* give the right answer up to the accuracy expected from part (a)?

   (c) Do the same as in (b) for Python.

   (d) Using a *for* loop to do the summation, $((x_1 + x_2) + x_3) + \ldots$. Does this give the right answer? Now use a *for* loop to do the summation with the 1 as the last value in the vector instead of the first value. Right answer? If either of these don't give the right answer, how many decimal places of accuracy does the answer have? Do the same in Python.

   (e) What do the results suggest about how R's *sum()* function works. Is it simply summing the numbers from left to right?

This should suggest that the *sum()* function works in a smart way, but that if you calculate the sum manually, you need to be careful in some situations.

3. Details of the Cholesky decomposition.

   (a) Work out the operation count (multiplies and divides) for the Cholesky decomposition, including the constant c, not just the order, for terms involving n^3 or n^2, e.g., $5n^3/2 + 75n^2$, NOT the order of the calculation (i.e., not $O(n^3)$). You can ignore the square root and any additions/subtractions. You can ignore pivoting for the purpose of this problem. Remember not to count any steps that involve multiplying by 0 or 1. Compare your result to that given in the notes.

   (b) Suppose I've written out the Cholesky calculation based on for loops. If I wanted to save storage space, can I store the Cholesky upper triangular matrix, U, in the storage space that is used for the original matrix as I go along, assuming I'm willing to lose the original matrix, or do I overwrite anything I need later in the calculation of the Cholesky?

   (c) Now, using a test matrix X, compute the Cholesky and monitor memory use using *top* in UNIX or based on the maximum memory reported by *gc()* in R. Does memory use match that from your answer in part (b)? If not, how much more or less memory is used than you might expect? For a variety of values of n (make sure you have matrices with n in the thousands), find the maximum memory use and the processing time and plot these as a function of n. Empirically how do memory use and processing time scale with n? [Note: make sure the only objects of any substantial size in your workspace are X and the resulting Cholesky matrix so that you only consider memory use from this operation.] For this problem make sure your calculations use only a single thread/core.
   Note: You can compute a positive definite test matrix either using the correlation function examples for stochastic (Gaussian) processes we've seen in class, or simply with $X^\top X$ for an X with random entries.

4. Compare the speed of $b = X^{-1}y$ using: (1) *solve()* followed by '%*%'; (2) `solve(X,y)`; (3) Cholesky decomposition followed by solving triangular systems. Do this for a matrix of size 5000×5000 using a single thread.

   (a) How do the timing and relative ordering amongst methods compare to the order of computations we discussed in class and the notes?

   (b) Are the results for b the same numerically for the different methods (up to machine precision)? Comment on how many decimal places in b agree, and relate this to the condition number of the calculation.

5. Suppose I need to compute the generalized least squares estimator, $\hat{\beta} = (X^\top\Sigma^{-1}X)^{-1}X^\top\Sigma^{-1}Y$, for X $n \times p$, Σ $n \times n$ and assume that $n > p$. Assume n could be of order several thousand and p of order in the hundreds. First write out in pseudo-code how you would do this in an efficient way - i.e., the particular linear algebra steps and the order of operations. Then write efficient R code in the form of a function, *gls()*, to do this - you can rely on the various high-level functions for matrix decompositions and solving systems of equations, but you should not use any code that already exists for doing generalized least squares.

6. For section for 10/21, I asked you to think about different ways of parallelizing a matrix multiplication problem. Here you'll expand on your answers and formally assess memory use and communication

costs. Assume again for simplicity that you are multiplying XY where each matrix is $n \times n$ and that you have p cores available for calculation, with at most p tasks, one per core, operating at any given moment in time. Recall the setup from the problem for section.

- Approach A: divide Y into p blocks of equal numbers of columns, where the first block has columns $1, \ldots, m$ where $m = \frac{n}{p}$ and so forth. Pass X and the jth submatrix of Y to the jth task.
- Approach B: divide X into p blocks of rows, where the first block has rows $1, \ldots, m$ and Y into p blocks of columns as above. Pass pairs of a submatrix of X and submatrix of Y to the different tasks.

(a) For each approach, count the amount of memory used at any single moment in time, when all p workers are doing their calculations, including memory use in storing the result (unless you can re-use any of the memory used for the inputs). Count the total number of numbers that need to be passed to the workers as well as the numbers passed from the workers back to the master when returning the result.

(b) Compare and contrast the two approaches in terms of memory use, communication cost, and any other relevant factors, such as overhead in starting up tasks and load-balancing. When would you choose Approach A and when Approach B?

(c) You can answer part (c) in one of two ways. The first way is to consider Approach C, in which you divide X into $\sqrt{p}$ blocks of rows and Y into $\sqrt{p}$ blocks of columns. Assume $\sqrt{p}$ is an integer. Analyze Approach C as in part (a) and compare with Approaches A and B, as in part (b). Alternatively, come up with an approach different than A, B, or C and analyze it. Particularly clever approaches may receive extra credit.

In your calculations, assume that the matrix chunks need to be passed to the various workers (i.e., those workers do not have direct access to the memory of the master). Note that a better way to do this is with threaded code in which the elements of the matrix never need to be copies because the different threads access the same memory locations of the matrices (as is done with threaded BLAS implementations). But this is good practice for the case where computations use distributed memory or if you use something like *parSapply()* or *foreach()*, which create new processes and make copies.

7. Extra credit: Figure out how R stores character strings in memory. As you do so, consider the following questions. Are character vectors stored contiguously in memory? How much space does R allocate for strings. If I do the following, how does R handle the fact that I am replacing one element with a very large element? (Consider the memory locations of the elements of the vector.)

```r
vec <- c("a", "b", "c")

vec[2] <- paste(sample(letters, 1e+05, replace = TRUE), collapse = "")
```