Problem Set 7

Due Fri. November 13, 10 am

Comments

- This covers material in Unit 10.
- It's due at 10 am (Pacific) on Friday Nov. 13, both submitted as a PDF to Pensive as well as committed to your GitHub repository.
- Note that is is fine to hand-write solutions to the the non-coding questions, but make sure your writing is neat and insert any hand-written parts **in order** into your final submission.

Problems

1. We've seen how to use Gaussian elimination (i.e., the LU decomposition) to solve $Ax = b$ and that we can do the solution in $n^3/3$ operations (plus lower-order terms). Suppose I want to know how inefficient it is to explicitly invert the matrix A and then multiply, thereby finding $x = A^{-1}b$ via matrix-vector multiplication. If we look into the C++ code of the `numpy.linalg.inv` function, we see it solves the system $AZ = I$ to find $Z = A^{-1}$, using a Lapack routine DGESV, which uses the LU decomposition. Count the number of computations for

   a. transforming $AZ = I$ to $UZ = I^*$ (where I^* is no longer a diagonal matrix),
   b. for solving for $A^{-1} = Z$ given $UZ = I^*$, and
   c. for calculating $x = Zb$.

   Then compare the total cost to the $n^3/3$ cost of what we saw in class.

   Notes:

   - In counting the computations you should be able to make use of various results we derived in class concerning the Gaussian elimination computations and computations involved in a backsolve, so your answer should be able to simply combine together results we've already discussed without any detailed new derivation.
   - Given that numpy's call to `dgesv` doesn't take account of the special (diagonal) structure of I on the right-hand side, you do not need to take account of the fact that because I has zeroes and ones, one can actually save some computation. You can simply count the calculations as if I were filled with arbitrary values. (Note: if we did actually try to be careful about making use of the structure of I, it turns out we could save $n^3/3$ calculations.)

2. Compare the speed of $x = A^{-1}b$ using: (i) `np.linalg.inv(A) @ b`, (ii) `np.linalg.solve(A, b)`, and (iii) Cholesky decomposition followed by solving triangular systems. To ensure that

A is positive definite (needed of course for the Cholesky and helpful, as a stricter condition than invertibility, for the other two), you can construct a test matrix A as $A = W^\top W$, with the elements of the $n \times n$ matrix W generated randomly.

a. Using a single thread and $n = 5000$, how do the timing and relative ordering amongst methods compare to computational cost from problem 1?

b. Are the results for the solution `x` the same numerically for methods (ii) and (iii) (up to double precision)? Comment on how many digits in the elements of `x` agree, and relate this to the condition number of the calculation. You can do this for a value of n rather smaller than 5000 to reduce the time to estimate the condition number.

c. How does the timing scale with the number of cores (i.e., threads, using a parallelized BLAS) for each of the three approaches? You may need to use the SCF for this to ensure you are using a parallel BLAS.

d. Now switch from numpy/scipy to JAX. Compare timing to (a) and (c), including using multiple cores with JAX.

e. Now run the JAX-based code on a GPU and compare timing. In doing your comparison, also compare using single vs. double precision for the JAX arrays. You can use the old GPU in the `gpu` partition on the SCF or you can use one of the other partitions that have GPUs; it doesn't matter what GPU you use – just make sure to note which GPU type you used in your solution. These other partitions have GPUs purchased for use by faculty research groups and are available on a preemptible basis - if your job is running and a job from a group member needs the GPU your job is using, your job may be cancelled and you may need to resubmit it. When running your job (either by `srun` or `sbatch`), please set a time limit of 10 minutes or less on your job to make sure that stale/unused jobs don't prevent your classmates from accessing the GPU.

Note: Section 6.1 of Unit 10 has some information on BLAS and on using JAX, including links to sections of the SCF Parallelization Tutorial.

3. Two-stage least squares (2SLS) is a way of implementing a causal inference method called instrumental variables that is commonly used in economics. Consider the following set of regression equations:

$$\begin{aligned} \hat{X} &= Z(Z^\top Z)^{-1} Z^\top X \\ \hat{\beta} &= (\hat{X}^\top \hat{X})^{-1} \hat{X}^\top Y \end{aligned}$$

which can be interpreted as regressing Y on X after filtering such that we only retain variation in X that is correlated with the instrumental variable Z. An economics graduate student asked me how he could compute $\hat{\beta}$ if Z is 60 million by 630, X is 60 million by 600, and Y is 60 million by 1, but both Z and X are sparse matrices.

a. Describe briefly why I can't do this calculation in two steps as given in the equations, even if I use the techniques for OLS discussed in class for each stage.

b. Figure out how to rewrite the equations such that you can actually calculate $\hat{\beta}$ on a computer without using a huge amount of memory. You can assume that any matrix multiplications involving sparse matrices can be done on the computer (e.g., using `scipy.sparse`). Describe the specific steps of how you would do this and/or write out in pseudo-code.

Notes:

- The product of two sparse matrices is not (in general) sparse and would not be sparse in this case.
- As discussed in Section 6 of Unit 10, there are approaches in Python (and software packages more generally) for efficiently storing (to save memory) and efficiently doing matrix manipulations (to save computation time) with sparse matrices.