CS 111: Homework 4: Due by 11:59 pm Sunday, February 12, 2023

Submit your paper as one PDF file, and tell GradeScope which page(s) each problem is on. If you worked with a partner, you must each turn in your own homework paper, and report the name and perm number of your partner. No groups of more than two allowed.

1. A symmetric matrix A is *positive definite* (SPD for short) if and only if $x^T Ax > 0$ for every nonzero vector x.

1.1 Find a 2-by-2 matrix A that (1) is symmetric, (2) is not singular, and (3) has all its elements greater than zero, but (4) is *not* SPD. Show a nonzero vector x such that $x^T Ax < 0$.

1.2 Let B be an m-by-n matrix (m and n may or may not be equal) whose rank is n. Prove that the matrix $A = B^T B$ is SPD (mathematically from the condition above, not experimentally).

2. If A is symmetric, we don't need to store all n^2 of its elements; we can just store the $n(n + 1)/2$ elements of the upper triangle of A, for example. If A is symmetric and also positive definite then there is a symmetric version of Gaussian elimination called *Cholesky factorization.* You can read about Cholesky and his factorization in NCM problem 2.5 (pages 35–36), but don't do that problem.

The Cholesky factorization of an SPD matrix is

$$A = R^T R,$$

where R is an upper triangular matrix with all its diagonal elements positive. Notice that there's only one triangular matrix R involved, so computing the factorization should only need to compute $n(n + 1)/2$ numbers, not n^2 numbers like LU factorization. There's also no pivoting permutation; it's a theorem that the Cholesky factorization can be computed stably without pivoting for any SPD matrix.

One way to get R from A is to factor $A = LU$ with no pivoting; then write $U = DV$ where D is diagonal and V is upper triangular with ones on the diagonal; then show that $L = V^T$ so that $A = V^T DV$; then finally take $R = \sqrt{D}V$, where $\sqrt{D}$ is the diagonal matrix of square roots of diagonal elements of D; then we have $A = V^T DV = R^T R$ as desired. However, this method does twice as much work as it needs to, because it computes all n^2 elements of L and U.

Your assignment is to write a routine `R = Cfactor(A)` that returns the factor R without ever touching the lower triangle of A or the lower triangle of R (or of any other n-by-n matrix). For full credit, your routine should also only do about half as many arithmetic operations as `L, U = cs111.LUfactorNoPiv(A)`. For debugging, you can generate a random n-by-n SPD matrix A by saying

```
B = np.random.randn(n, n)
A = B.T @ B
```

Explain in English (in LaTeX) how your `Cfactor()` works. Demonstrate that it works by generating a 10-by-10 SPD matrix A as above, generating a random 10-vector b, and comparing the solution to $Ax = b$ from `x = cs111.LUsolve(A,b)` to the solution you get by saying

```
R = Cfactor(np.triu(A))
y = cs111.Lsolve(R.T, b, unit_diag=False)
x = cs111.Usolve(R, y, unit_diag=False)
```

Note that this would still work without calling `np.triu()` on A, but that call makes it impossible for `Cfactor()` to use the lower triangle of A.

Finally, do an experiment to compare the running times of `Cfactor(A)` and `LUfactorNoPiv(A)`, for a range of values of n up to large enough that the routines take several seconds to run. Report your running times, and make a plot of the ratio of `Cfactor(A)` time to `LUfactorNoPiv(A)` time against n. (You can time one line of code in Jupyter by saying `%time line-of-code`, or you can time a whole window by starting it with `%%time`.)

3. Here you will experiment with solving $Ax = b$ using various solvers from class and from `numpy`. For this problem, you should use the 3-D version of the temperature matrix from `make_A_3D()`. You can use the version of `make_A_3D()` you wrote for Homework 3, or if you prefer you can use my version (which is in the latest update of `cs111/temperature.py` on GauchoSpace). For a right-hand side b, use the vector of row sums, `b = A @ np.ones(n)`, so that you know the exact solution to $Ax = b$ is the vector of all ones.

Experiment with solving $Ax = b$ for the temperature x, for various values of k, using five different solvers as follows. For each solver, you should report (showing code and output) the largest value of k for which that solver could solve $Ax = b$ within 30 seconds. For all but the last solver, use the sparse version of A from `make_A_3D()`.

- The `cs111.CGsolve()` conjugate gradient solver, from class. (You can vary the arguments `tol` and `max_iters` to make it find a more accurate solution.)
- The `cs111.Jsolve()` Jacobi solver, also from class. (Again you can vary `tol` and `max_iters`.)
- The `scipy` sparse conjugate gradient solver `scipy.sparse.linalg.cg()`.
- The `scipy` sparse LU solver `scipy.sparse.linalg.spsolve()`.
- The dense LU solver `cs111.LUsolve()` from class. For this solver, you will have to convert A to a dense array with `A.toarray()`. Warning! This will run out of memory if k gets very big.

For each solve, measure and report the run time, the relative residual norm, and the relative error norm $||x_{\text{exact}} - x||/||x_{\text{exact}}||$. Which solvers are more accurate? Which are faster? How do the answers to these questions change as you change k?

Warning: Start with very small values of k, and be cautious as you increase k! The matrices get big in a hurry. Different solvers will fall over for different values of k.