CS 111: Homework 3: Due by 11:59 pm Sunday, October 17, 2021

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. Suppose that A is a square, nonsingular, nonsymmetric matrix, b is an n-vector, and that you have called

```
L, U, p = cs111.LUfactor(A)
```

(using the routine from the lecture files). Now suppose you want to solve the system $A^T x = b$ (not $Ax = b$) for x. Show how to do this using calls to `cs111.Lsolve()` and `cs111.Usolve()`, without modifying either of those routines or calling `cs111.LUfactor()` again. You are allowed to transpose matrices L and U, that is, you may work with `L.T` and `U.T`. You may not call any of `numpy`'s built-in solvers (like `npla.solve()`). Test your method in `numpy` on a randomly generated 6-by-6 matrix and show the code and output in Jupyter.

2. The *condition number* of a matrix A is a measure of how close to being singular it is. The definition is

$$\kappa(A) = ||A||\ ||A^{-1}||,$$

with the convention $\kappa(A) = \infty$ if A is singular. Each different matrix norm gives its own definition of condition number; usually we'll consider the 2-norm condition number. The condition number is always at least 1. (I'll prove that in class.) A matrix is *well-conditioned* if its condition number is not too big (think 10^3 or less), and *ill-conditioned* if its condition number is very large (think 10^6 or more). In numpy, the condition number is computed by `npla.cond()`.

In this problem you'll explore the relationship between condition number and the behavior of LU factorization.

2.1. Consider the linear system

$$\begin{pmatrix} \alpha & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} = \begin{pmatrix} 1-\alpha \\ 0 \end{pmatrix},$$

for some $\alpha < 1$. Clearly the solution is $(x_0, x_1)^T = (-1, 1)^T$. For each value of $\alpha = 10^{-4}, 10^{-8}, 10^{-16}, 10^{-20}$, try to solve this system twice, using the routine `cs111.LUsolve()`, first with `pivoting = True` and then with `pivoting = False`. For each solution, print: the condition number of A; the computed x; the norm of the error; and the relative residual norm returned by `cs111.LUsolve()`. Show your `numpy` code and its output.

As α decreases, does A become ill-conditioned or does it remain well-conditioned? Describe and explain the trends in the relative residual norm for both the non-pivoting and pivoting solutions.

2.2. Consider the linear system

$$\begin{pmatrix} 1+\alpha & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} = \begin{pmatrix} -\alpha \\ 0 \end{pmatrix},$$

for some $\alpha < 1$. Again, the solution is $(x_0, x_1)^T = (-1, 1)^T$. For each value of $\alpha = 10^{-4}, 10^{-8}, 10^{-16}, 10^{-20}$, try to solve this system twice, using the routine `cs111.LUsolve()`, first with `pivoting = True` and then with `pivoting = False`. This time, some of the tries will fail to return an answer. For each try, print the condition number of A. If an answer was returned, print: the computed x; the norm of the error; and the relative residual norm returned by `cs111.LUsolve()`. Show your `numpy` code and its output.

As α decreases, does A become ill-conditioned or does it remain well-conditioned? Describe and explain the trends in the relative residual norm for the successful trys. Explain why the other tries were unsuccessful.

3. The temperature problem models our cabin in the woods in two dimensions, but most modern scientific simulations are done in three dimensions. Here you will create the matrix that corresponds to a 3-D version of the temperature problem. The "cabin" is now the unit cube. As before, we will discretize the interior by dividing it into k points in each dimension, but now there are a total of k^3 points rather than k^2.

The partial differential equation still leads to the approximation that the temperature at any given point is the average of the temperatures at the neighboring points, but now there are 6 neighbors, with 2 in each of the three dimensions. The matrix A for the 3D version of the temperature problem expresses the fact that, in a 3D k-by-k-by-k grid, each interior point has a temperature that is the average of its 6 neighbors (left, right, up, down, in, out). The diagonal elements of A are all equal to 6, and the off-diagonal elements are either 0 or -1. Most of the rows of A have 7 nonzeros.

Using the routine `make_A(k)` from `cs111/temperature.py` as a model, write a routine `make_A_3D(k)` that returns the k^3-by-k^3 matrix A. Like `make_A(k)`, your routine should create A as a sparse matrix using `scipy.sparse.csr_matrix()`, after making a table "`triples`" of the nonzero elements of A.

Here below, for your debugging, is the correct matrix for $k = 2$. I converted it to a dense array for printing—you should also print it out as a sparse matrix (that line is commented out below), and indeed for $k > 2$ it's going to be too large to see what's going on in the dense matrix anyway.

```
[In:]
  k = 2
  A = make_A_3D(k)
  print('k:', k)
  print('dimensions:', A.shape)
  print('nonzeros:', A.nnz)
  #print('A as sparse matrix:\n', A)
  print('A as dense matrix:\n', A.toarray())
```

```
[Out:]
  k: 2
  dimensions: (8, 8)
  nonzeros: 32
  A as dense matrix:
  [[ 6. -1. -1.  0. -1.  0.  0.  0.]
   [-1.  6.  0. -1.  0. -1.  0.  0.]
   [-1.  0.  6. -1.  0.  0. -1.  0.]
   [ 0. -1. -1.  6.  0.  0.  0. -1.]
   [-1.  0.  0.  0.  6. -1. -1.  0.]
   [ 0. -1.  0.  0. -1.  6.  0. -1.]
   [ 0.  0. -1.  0. -1.  0.  6. -1.]
   [ 0.  0.  0. -1.  0. -1. -1.  6.]]
```

Print out your matrix for $k = 2$ and $k = 3$ as a check that it's correct. Also use `plt.spy(A)` to make a spy plot of the nonzero structure for $k = 4$ or 5 (you may want to zoom in on the plot to see all the structure).

To complete a realistic simulation you would also write a routine `make_b_3D(k)` to compute the right-hand side b. For this problem, you don't have to do that.