CS 111: Homework 4: Due by 6:00pm Friday, February 1

Homework must be submitted online as a PDF file to GradeScope. When you turn in your homework, tell GradeScope which page(s) contain each problem. Doing this correctly will be worth 2 points.

1. 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 k^3 points in all 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 dimension.

Using the routine `make_A(k)` from `Temperature.ipynb` as a model, write a routine `make_A_3D(k)` that returns the k^3-by-k^3 matrix A for the 3D version of the temperature problem. This matrix 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.

Here below, for debugging, is the correct matrix for $k = 2$. I converted it to dense for printing—you should also print it out as sparse, 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.size)
  #print('A as sparse matrix:'); print(A)
  print('A as dense matrix:'); print(A.todense())

[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; for the experiments in Problem 2 you can just use `np.random.rand()` to generate a random b.

2. Now you will experiment with solving $At = b$ using various solvers from class and from `numpy`. For this problem, you should use the 3-D version of the temperature matrix from Problem 1. (You can get partial credit by using the 2-D temperature matrix from the class instead.) You can use a randomly chosen right-hand side vector b.

Experiment with solving $At = b$ for the temperature t, for various values of k, using five different solvers:

- The `CGsolve()` conjugate gradient solver, from class. (You can vary the arguments `tol` and `max_iters` to make it find a more accurate solution.)
- The `Jsolve()` Jacobi solver, also from class. (Again you can vary `tol` and `max_iters`.)
- The `scipy` sparse conjugate gradient solver `spla.cg()`.
- The `scipy` sparse LU solver `spla.spsolve()`.
- The `LUsolve()` dense LU solver from class. (For this, you will have to use the dense form of A that you get from `A.todense()`. Warning! This will use too much memory if k gets very big at all.)

For each solve, measure the run time and also the relative residual norm. 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; try to see how big a value of k each solver can handle with at most 30 seconds of compute time.

3. Let

$$A = \begin{pmatrix} 4 & -1 & -1 \\ -1 & 4 & -1 \\ -1 & -1 & 4 \end{pmatrix}$$

and let $b = (15, -3, 12)^T$.

3a. Use the `scipy` Cholesky factorization routine `linalg.cholesky()` to compute the triangular Cholesky factor of A. (Either upper or lower triangular is fine, but just compute one of them.) Verify that the answer is correct by multiplying the factor by its transpose and comparing with A. Then use `Usolve()` and/or `Lsolve()` to compute the solution x to $Ax = b$ from the Cholesky factor (without calling any other factorization routine). Show the Jupyter input and output for your computations.

3b. Use the `scipy` QR factorization routine `linalg.qr()` to compute the two matrices (orthogonal and upper triangular) that constitute the QR factorization of A. Verify that the answer is correct by multiplying the factors and comparing with A. Then use `Usolve()` and/or `Lsolve()` to compute the solution x to $Ax = b$ from the QR factors (without calling any other factorization routine). Show the Jupyter input and output for your computations.

4. Which of the following matrices are orthogonal? (Don't show your work, just give the answer.)

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad B = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \quad C = \begin{pmatrix} 2 & 0 \\ 0 & 1/2 \end{pmatrix}, \quad D = \begin{pmatrix} \sqrt{2}/2 & \sqrt{2}/2 \\ -\sqrt{2}/2 & \sqrt{2}/2 \end{pmatrix}$$