CS 111: Homework 4: Due by 11:59 pm Monday, February 10, 2020

Submit your paper as one PDF file, and tell GradeScope which page(s) each problem is on.

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

Experiment with solving $At = b$ for the temperature t, 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 $At = b$ 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 `spla.spsolve()`.
- The `cs111.LUsolve()` dense LU solver from class. (For this solver, you will have to convert A to a dense array with `A.toarray()`. 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.

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 `spla.cholesky()` to compute the lower triangular Cholesky factor of A. Verify that the answer is correct by multiplying the factor by its transpose and comparing with A. Then use `cs111.Usolve()` and/or `cs111.Lsolve()` to compute the solution x to $Ax = b$ from the Cholesky factor, without calling any other factorization routine. (You are allowed to transpose any matrix if you want.) Show the Jupyter input and output for your computations.

3b. Use the `scipy` QR factorization routine `spla.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 `cs111.Usolve()` and/or `cs111.Lsolve()` to compute the solution x to $Ax = b$ from the QR factors, without calling any other factorization routine. (You are allowed to transpose any matrix if you want.) Show the Jupyter input and output for your computations.

4. The moral of this problem is twofold: First, linear least squares can be used to fit data with polynomials, not just with straight lines or planes. (As we've seen, it can be used to fit data with lots of different kinds of models.) Second, fitting data with a very high-degree polynomial can be a bad idea.

We are going to try various ways of predicting the results of this year's 2020 census. The data for our prediction is the population of the United States at each 10-year census from 1900 to 2010, as follows.

```
date = np.array(range(1900,2020,10)) - 1900
population = 1000 * np.array([ 75995,  91972, 105711, 123203, 131669, 150697,
                              179323, 203212, 226505, 249633, 281422, 308746])
for (d,p) in zip(date, population):
    print(d, ':', p)
```

(Notice that we are writing the date as "years since 1900." How come? It turns out to make the fitting computation more stable. We may return to this in the next homework.)

4a. Express the problem of fitting a straight line of the form

$$p = x_0 + x_1 d$$

to the data as a linear least squares problem

$$Ax \approx b$$

where $x = (x_0, x_1)^T$ is the vector of coefficients of the line. (The answer to this question consists just of the 12-by-2 matrix A and the vector b.)

4b. Solve the least squares problem in (4a) for x in two different ways: First, use the QR factorization of A. Second, use the `scipy` routine `npla.lstsq()`. (This uses either QR or SVD under the hood, but it has more bells and whistles; see the python help for the function. You'll need to give it an extra argument `rcond=None`, and the solution it returns is a python tuple whose first element is x.) Verify that the two x vectors are (nearly) the same. Make a plot that shows the original population data as circles, and the least squares fit as a line.

4c. Use x to compute the US population in the year 2020, as predicted by the straight-line fit.

4d. Express the problem of fitting a quadratic polynomial of the form

$$p = x_0 + x_1 d + x_2 d^2$$

to the data as a linear least squares problem

$$Ax \approx b$$

where $x = (x_0, x_1, x_2)^T$ is the vector of coefficients of the polynomial. This time the matrix A will be 12-by-3. Again, solve the least squares problem (using your choice of QR factorization or `npla.lstsq()` but not both), make a plot of the resulting parabola, and use the new x to predict the 2020 population.

4e. Revise the code you used in (4d) to take the polynomial degree as a parameter. Repeat the fitting, plotting, and prediction for polynomials of degrees 3, 4, ..., 8. (Turn in your plots and the values of your predictions for this part, but not your python code.) What do you notice about the last few predictions?