PURDUE UNIVERSITY · CS 51500
MATRIX COMPUTATIONS

HOMEWORK
David F. Gleich
September 3, 2018

Homework 2

Please answer the following questions in complete sentences in a clearly prepared manuscript and submit the solution by the due date on Blackboard (around Sunday, September 9nd, 2018.)

Remember that this is a graduate class. There may be elements of the problem statements that require you to fill in appropriate assumptions. You are also responsible for determining what evidence to include. An answer alone is rarely sufficient, but neither is an overly verbose description required. Use your judgement to focus your discussion on the most interesting pieces. The answer to "should I include 'something' in my solution?" will almost always be: Yes, if you think it helps support your answer.

Problem 0: Homework checklist

- Please identify anyone, whether or not they are in the class, with whom you discussed your homework. This problem is worth 1 point, but on a multiplicative scale.
- Make sure you have included your source-code and prepared your solution according to the most recent Piazza note on homework submissions.

Problem 1: The expected length of a Candyland game

In class, we showed that we could form a linear system of equations in order to determine the expected time that a random walk on the integers $[-4, 6]$ spends before it reaches the endpoints -4 and 6. We can do the same thing to determine the expected length of a game of Candyland!

Recall that the data for our Candyland game is available from the website urls:

- https://www.cs.purdue.edu/homes/dgleich/cs515-2018/julia/candyland-matrix.csv
- https://www.cs.purdue.edu/homes/dgleich/cs515-2018/julia/candyland-coords.csv
- https://www.cs.purdue.edu/homes/dgleich/cs515-2018/julia/candyland-cells.csv

The starting cell is given by cell 140 and the ending cell is given by cell 134. Cells 135 and 136 are are bridge cells. Cells 137, 138, and 139 are used to model the sticky cells.

Let $T_{i,j}$ be the probability of moving from Candyland cell j to Candyland cell i given a draw from the desk of cards.

1. (Warm-up) Show how to identify the game-cell associated with sticky cells 137, 138, and 139 based on the transition matrix T. Your answer must include both the answer in the given data and and procedure that would apply more generally.

2. Use the same type of implicit formulation where x_i is the expected length of a Candyland game starting from cell i to derive a linear system of equations. Build and solve this linear system of equations in Julia. For this problem, assume that you use one turn for *any* cell other than the ending cell 134. (That is, we are doing no modeling associated with the bridges or sticky states.) What is the expected length of the game starting from the start? (Hint, I get 33.66804940823773). You should provide the linear system in terms of the matrix $\boldsymbol{T}$, but you should not provide explicit entries. Also, presumably, the creators would have made the start state the place of maximum game length. Is this the case or is there another cell or set of cells that result in a longer game?

   For this problem, I found it helpful to visualize a solution to make sure it made sense. I used the following line of code to visualize a solution $\mathbf{x}$:

   ```
   using Plots
   pyplot(size=(600,600))
   scatter(xc,yc,markersize=x, zcolor=x)
   gui()
   ```

   where `xc` and `yc` are the coordinates from the coords file.

3. Recall that in class we showed that $\mathbf{p}_k = \boldsymbol{T}^{k-1}\mathbf{t}_{140}$ gave the probability of being in each state after k steps. By definition, the expected length of the game is:

   $$\sum_{k=1}^{\infty} k[\mathbf{p}_k]_{134}.$$

   Develop a computer program to approximately compute this sum. Is there a good way to determine when to stop adding terms? What value do you get?

4. Now, ideally, the answer to parts 3 and 4 are the same. (Hint: they are.) Use the Neumann series of a matrix to prove that this is the case.

5. Going back to part 2, we didn't get our model of Candyland quite right because we assumed that each of the bridge and sticky states should be associated with one turn, including a transition to the virtual state $135-139$. Show how to modify your linear system of equations so that these virtual transitions are no longer included. (Note, there is more than one way to solve this.) Just to be precise, we want to assume the following game rules:

   - you remain at a sticky state (and keep using turns) until you exit that state.
   - a bridge state does not consume a turn, that is, if you land on a bridge (cell 5, 35) you move directly from the start of the bridge to the end cell of the bridge consuming no turns.

   How does this change the expected length of the game? Also, are there still a cell or set of cells that results in a longer game than the actual start?

Problem 2: Poisson's equation

In this problem, we'll meet one of the most common matrices studied in numerical linear algebra: the $2d$-Laplacian. We arrive at this matrix by discretizing a partial differential equation. Poisson's equation is:

$$\Delta u = f$$

where $u(x, y)$ is a continuous function defined over the unit-plane (i.e. $0 \le x \le 1, 0 \le y \le 1$), $f(x, y)$ is a continuous function defined over the same region, and Δ is the Laplacian operator:

$$\Delta u = \partial^2 u/\partial x^2 + \partial^2 u/\partial y^2.$$

Given a function f, we want to find a function u that satifies this equation. There are many approaches to solve this problem theoeretically and numerically. We'll take a numerical approach here.

Suppose we discretize the function u at regular points $x_0, \ldots, x_n$, and $y_0, \ldots, y_n$ where $x_i = y_i = i/n$ so that we have:

$$u(x,y) \approx \text{ grid of } \begin{matrix} u(x_0,y_0) & \cdots & u(x_n,y_0) \\ \vdots & \ddots & \vdots \\ u(x_0,y_n) & \cdots & u(x_n,y_n). \end{matrix}$$

For this discretization, note that

$$\begin{aligned} \Delta u(x_i,y_j) &\approx n^2 \left(u(x_{i-1},y_j) - 2u(x_i,y_j) + u(x_{i+1},y_j)\right) \\ &\quad + n^2 \left(u(x_i,y_{j-1}) - 2u(x_i,y_j) + u(x_i,y_{j+1})\right) \\ &= n^2 \left(u(x_{i-1},y_j) + u(x_i,y_{j-1}) - 4u(x_i,y_j) + u(x_{i+1},y_j) + u(x_i,y_{j+1})\right) \\ &= f(x_i,y_j). \end{aligned}$$

What we've done here is use the approximation:

$$\partial^2 u/\partial x^2 \approx \frac{1}{h^2}\left(u(x-h) - 2u(x) + u(x+h)\right)$$

for both partial terms.
We need this equation to hold at each point x_i, y_j. But note that there are some issues with this equation at the boundary values (where x=0 or 1, or where y=0 or 1).
For this problem, we'll make it very simple and set:

$$u(0,y_j) = u(1,y_j) = u(x_i,0) = u(x_i,1) = 0.$$

Now, we'll do what we always do! Turn this into some type of matrix equation!

Let $\boldsymbol{U}$ be an $n+1 \times n+1$ matrix that we'll index from zero instead of one:

$$\boldsymbol{U} = \begin{bmatrix} U_{0,0} & \cdots & U_{0,n} \\ \vdots & \ddots & \vdots \\ U_{n,0} & \cdots & U_{n,n} \end{bmatrix}.$$

where $U_{i,j} = u(x_i,y_j)$. At this point, we are nearly done. What we are going to do is turn Poisson's equation into a linear system. This will be somewhat like how we turned image resampling into a matrix vector equation in the first homework.

In order to write $\boldsymbol{U}$ as a vector, we'll keep the convention from last time:

$$\boldsymbol{U} = \begin{bmatrix} u_1 & \cdots & u_{n+1} \\ u_{n+2} & \cdots & u_{2(n+1)} \\ \vdots & \ddots & \vdots \\ u_{n(n+1)+1} & \cdots & u_{(n+1)(n+1)} \end{bmatrix}.$$

Let $\mathbf{u}$ be the vector of elements here. Note that our approximation to Δu, just involved a linear combination of the elements of $\mathbf{u}$. This means we have a linear system:

$$\boldsymbol{A}\mathbf{u} = \mathbf{f}$$

where the rows of $\boldsymbol{A}$ and $\mathbf{f}$ correspond to equations of the form:

$$\frac{1}{h^2}\left(u(x_{i-1},y_j) + u(x_i,y_{j-1}) - 4u(x_i,y_j) + u(x_{i+1},y_j) + u(x_i,y_{j+1})\right) = f(x_i,y_j).$$

1. Let $n = 3$. Write down the 16×16 linear equation for $\mathbf{u}$ including all the boundary conditions. Note that you can encode the boundary conditions by adding a row of $\boldsymbol{A}$ where: $u_i = 0$.
2. Write a Julia or Matlab/Python code to construct a sparse matrix $\boldsymbol{A}$ and vector $\mathbf{f}$ when $n = 10$ and $f(x, y) = 1$. Here's some pseudo-code to help out:

```
function laplacian(n::Integer, f::Function)
    N = (n+1)^2
    nz = <fill-in>
    I = zeros(Int,nz)
    J = zeros(Int,nz)
    V = zeros(nz)
    fvec = zeros(N)
    # the transpose mirrors the row indexing we had before.
    G = reshape(1:N, n+1, n+1)' # index map, like we saw before;
    h = 1.0/(n)
    index = 1
    for i=0:n
        for j=0:n
            row = G[i+1,j+1]
            if i==0 || j == 0 || i == n || j == n
                # we are on a boudnary
                fvec[row] = 0.0
                # fill in entries in I,J,V and update index
            else
                fvec[row] = f(i*h, j*h)*h^2
                # fill in entries in I,J,V and update index
            end
        end
    end
    A = sparse(I,J,V,N,N)
    return A, fvec
end
```

3. Solve for $\mathbf{u}$ using Julia's or Matlab's backslash solver, and show the result using the `mesh` function (Matlab) or `surface` function (Plots.jl in Julia).
4. We can use the Neumann series to turn an inverse into an infinite summation. What happens if we try and use that approach to solve this linear system of equations? Does it work or not?

Problem 3: Sparse matrix operations

Write working code for the following operations for a matrix given by Compressed Sparse Column arrays: `colptr`, `rowval`, `nzval` along with dimensions `m` and `n` as in the Julia sparse matrix storage.

1. Sparse matrix-transpose multiplication by a vector

```
""" Returns y = A'*x where A is given by the CSC arrays
colptr, rowval, nzval, m, n and x is the vector. """
function csc_transpose_matvec(colptr, rowval, nzval, m, n, x)
end
```

2. Row-inner-product

```
""" Returns  = A[i,:]*x where A is given by the CSC arrays
colptr, rowval, nzval, m, n and x is the vector. """
```

```
function csc_row_projection(colptr, rowval, nzval, m, n, i, x)
end
```

3. Column inner-product

```
""" Returns  = A[:,i]'*x where A is given by the CSC arrays
colptr, rowval, nzval, m, n and x is the vector. """
function csc_column_projection(colptr, rowval, nzval, m, n, i, x)
end
```