PURDUE UNIVERSITY · CS 51500
MATRIX COMPUTATIONS

HOMEWORK 1 SUBMISSION

Jinen Setpal
September 14, 2025

Checklist

1. Cross-checked independent work with Kunal Kapur.
2. No use of AI tools.
3. Code is included!

Problem 1

1. Each element across the diagonal scales the corresponding row vector proportionally. Thus, we have:

$$\begin{bmatrix}2 & 0 & 0\\ 0 & 1/2 & 0\\ 0 & 0 & -1\end{bmatrix}\begin{bmatrix}1 & 2 & 3\\ 6 & 4 & 2\\ -2 & -2 & -2\end{bmatrix} = \begin{bmatrix}2 & 4 & 6\\ 3 & 2 & 1\\ 2 & 2 & 2\end{bmatrix}$$

2. Setting the diagonal matrix as the transformation makes it easier to compute. By transposition, we have: $\boldsymbol{AB} = (\boldsymbol{B}^T\boldsymbol{A}^T)^T$

$$\begin{bmatrix}1 & 2 & 3\\ 6 & 4 & 2\\ -2 & -2 & -2\end{bmatrix}\begin{bmatrix}2 & 0 & 0\\ 0 & 1/2 & 0\\ 0 & 0 & -1\end{bmatrix} = \left(\begin{bmatrix}2 & 0 & 0\\ 0 & 1/2 & 0\\ 0 & 0 & -1\end{bmatrix}\begin{bmatrix}1 & 6 & -2\\ 2 & 4 & -2\\ 3 & 2 & -2\end{bmatrix}\right)^T$$

We can now apply the same diagonal scaling as before:

$$= \left(\begin{bmatrix}2 & 12 & -4\\ 1 & 2 & -1\\ -3 & -2 & 2\end{bmatrix}\right)^T$$

$$\begin{bmatrix}1 & 2 & 3\\ 6 & 4 & 2\\ -2 & -2 & -2\end{bmatrix}\begin{bmatrix}2 & 0 & 0\\ 0 & 1/2 & 0\\ 0 & 0 & -1\end{bmatrix} = \begin{bmatrix}2 & 1 & -3\\ 12 & 2 & -2\\ -4 & -1 & 2\end{bmatrix}$$

3. Using the result from the previous questions, we can merge the two right-most matrices, and then apply diagonal scaling.

$$\begin{bmatrix}1/2 & 0 & 0\\ 0 & 2 & 0\\ 0 & 0 & -1\end{bmatrix}\begin{bmatrix}1 & 2 & 3\\ 6 & 4 & 2\\ -2 & -2 & -2\end{bmatrix}\begin{bmatrix}2 & 0 & 0\\ 0 & 1/2 & 0\\ 0 & 0 & -1\end{bmatrix} = \begin{bmatrix}1/2 & 0 & 0\\ 0 & 2 & 0\\ 0 & 0 & -1\end{bmatrix}\begin{bmatrix}2 & 1 & -3\\ 12 & 2 & -2\\ -4 & -1 & 2\end{bmatrix} = \begin{bmatrix}1 & .5 & -1.5\\ 24 & 4 & -4\\ 4 & 1 & -2\end{bmatrix}$$

4. Applying diagonal scaling on the left transforms the rows of the right matrix, whereas applying diagonal scaling on the right transforms the columns of the left matrix. In either case, the scaling applied corresponds to the diagonal element of the row / column vector respectively.

5. We have $\mathbf{x}, \mathbf{y} \in \mathbb{R}^{1000\times 1}$.

$$\mathbf{x}^T\mathbf{y} = \begin{bmatrix}1 & 1 & \cdots & 1\end{bmatrix}\begin{bmatrix}1\\ 2\\ \dots\\ 1000\end{bmatrix} = \langle \mathbf{x}, \mathbf{y}\rangle = \sum_{i=1}^{1000} i = \frac{n(n+1)}{2} = \frac{1001\cdot 1000}{2} = 500500$$

6. Let $\mathbf{e}$ specifically represent the *column* vector. We are given $\boldsymbol{A} \in \mathbb{R}^{m\times n}$.

   We can select the r^{th} row by left multiplication: $\mathbf{e}_r^T\boldsymbol{A}$, and select the c^{th} column by right multiplication: $\boldsymbol{A}\mathbf{e}_c$.

7. We have $\mathbf{x} = \begin{bmatrix} -5 & 4 & 2 \end{bmatrix}^T$ and $\mathbf{e}_i \in \mathbb{R}^{3\times 1}$. Assuming indexing starts with 1. Then,

$$\mathbf{e}_2\mathbf{x}^T = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \begin{bmatrix} -5 & 4 & 2 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ -5 & 4 & 2 \\ 0 & 0 & 0 \end{bmatrix}$$

and,

$$\mathbf{x}\mathbf{e}_1^T = \begin{bmatrix} -5 \\ 4 \\ 2 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} -5 & 0 & 0 \\ 4 & 0 & 0 \\ 2 & 0 & 0 \end{bmatrix}$$

8. Applying the column-selector idea, each of these operations are simply selecting columns. Thus, we have $\boldsymbol{A}\mathbf{e}_1 = \begin{bmatrix} 2 & 0 & 0 \end{bmatrix}^T$, $\boldsymbol{A}\mathbf{e}_2 = \begin{bmatrix} 0 & 1/2 & 0 \end{bmatrix}^T$, $\boldsymbol{A}\mathbf{e}_3 = \begin{bmatrix} 0 & 0 & -1 \end{bmatrix}^T$

Problem 2

1. The answer to this is highly dependent on the assumptions I make. Consider,
   Scenario 1: OpenAI has competent engineers. In this case:

   a. Their web crawler finds a database containing everyone's personal data, including the social security number. It is then formatted into a paragraph form where the personal data is preceded by the SSN.

   b. GPT-5 is trained on next-token prediction.

   c. Each SSN gets it's own encoding (as opposed to, e.g., each letter getting it's own embedding and then being concatenated to form the final SSN embedding) or each part (first three, next two, and last four digits) get their own embeddings which are then concatenated.

   Consequence: In this case, I'd argue the embeddings do constitute a matrix. A sufficiently well-trained GPT-5 should learn the correlation of SSNs with personal data, especially givenhistorically, the first three digits denote area (e.g., XXX-YY-ZZZZ was born in Austin, Texas). These could be embedded as specific columns within the embedding, or even as an unprivileged basis. Either way, it makes sense to perform linear transformations on this table, hence it's a matrix.

   Scenario 2: OpenAI has less competent engineers. In this case:

   a. They synthetically generate all SSNs, but don't have any associated data about the individual. This is then randomly sampled and spliced together using spaces into paragraph form for each batch.

   b. GPT-5 is trained on next-token prediction.

   c. Each character gets it's own encoding.

   Consequence: In this case, I'd argue the embeddings do not constitute a matrix. During training, the preceding data is entirely disconnected from and offers no insight into the next token. Further, each character is oversubscribed; repeated characters mean repeated encodings, making it harder still to obtain a meaningful result. The model cannot be well-trained and the embeddings are meaningless. The learned embeddings are a table of disconnected numbers. Performing linear transformations on this would lend no insight, hence this is not a matrix.

2. Although I don't identify any structure (e.g., Hankel / Toeplitz), I do think this is a matrix. Audio is processed per second (assuming 44.1kHz) with each element representing the amplitude of the wave. As an example, we can compute the similar chunks of audio by computing the inner product of $\boldsymbol{A}$.

3. Each of the columns represent factors that are relevant to admission. Consider $\mathbf{x} = \begin{bmatrix} 1 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}^T$. If $\boldsymbol{A}$ is the described matrix, $\boldsymbol{A}\mathbf{x}$ is a very rudimentary way to score each applicant. We can then rank applicants by score and evaluate if it correlates strongly with their admission into Purdue's CS program. It makes sense to perform linear transformations on this data; therefore, this is an example of a matrix.

4. For N SSNs, we have $\boldsymbol{A} \in \mathbb{R}^{N \times 3072}$. Thus $\boldsymbol{A}^T\boldsymbol{A} \in \mathbb{R}^{3072 \times 3072}$ renders the correlation between each of the embeddings. If we assume that GPT-5 embeddings are a matrix (in that they are meaningful), this is valuable as high similarity scores would denote correlated features, while low scores denote anti-correlated features. Both are useless and can be safely discarded (or a smaller embedding may be trained). If $\boldsymbol{A}^T\boldsymbol{A} \approx \boldsymbol{I}$, we may conjecture that the embeddings are saturated.

5. Similarity across for N seconds of audio $A \in \mathbb{R}^{44100 \times N}$ results in comparing each second of audio to the other. Overlaps between segments will result in amplified scores, while anti-waves cancel one another out. We studied applications of linear algebra to noise cancelling as well; so I'd say this is safely a matrix too.

6. For N applicants, we have $\boldsymbol{A} \in \mathbb{R}^{N \times 6}$. Thus $\boldsymbol{A}^T\boldsymbol{A} \in \mathbb{R}^{N \times N}$ denotes the correlation between factors; for the same reason as part 4, $\boldsymbol{A}^T\boldsymbol{A}$ is a matrix.

Problem 3

1. Let $\boldsymbol{A} = \begin{bmatrix} a_{11} & & \\ & \ddots & \\ & & a_{dd} \end{bmatrix}$, $\boldsymbol{B} = \begin{bmatrix} b_{11} & & \\ & \ddots & \\ & & b_{dd} \end{bmatrix}$. We have the product rule:

$$\boldsymbol{C} = \boldsymbol{AB} \qquad \text{where} \qquad \boldsymbol{C}_{ij} = \sum_{k=1}^{d} \boldsymbol{A}_{ik}\boldsymbol{B}_{kj}$$

It suffices to show that:

$$\boldsymbol{A}_{ij} = \boldsymbol{B}_{ij} = 0 \implies \boldsymbol{C}_{ij} = 0 \qquad \forall i \neq j$$

Let $[d] = \{1, 2, \ldots, d\}$. We have that, for $i \neq j$,

$$\begin{aligned} \boldsymbol{C}_{ij} &= \sum_{k=1}^{d} \boldsymbol{A}_{ik}\boldsymbol{B}_{kj} \\ &= (\boldsymbol{A}_{ii}\boldsymbol{B}_{ij} + \boldsymbol{A}_{ij}\boldsymbol{B}_{jj}) + \sum_{k \in [d]\setminus\{i,j\}} \boldsymbol{A}_{ik}\boldsymbol{B}_{kj} \\ &= (\boldsymbol{A}_{ii} \cdot 0 + 0 \cdot \boldsymbol{B}_{jj}) + \sum_{k \in [d]\setminus\{i,j\}} 0 = 0 \end{aligned}$$

Thus $\boldsymbol{C}_{ij} = 0 \; \forall i \neq j$, and the resultant matrix $\boldsymbol{C}$ is also diagonal.

2. Let $\boldsymbol{A} = \begin{bmatrix} a_1 & a_2 & \cdots & a_n \\ a_2 & a_3 & \cdots & a_1 \\ \vdots & \vdots & & \vdots \\ a_n & a_1 & \cdots & a_{n-1} \end{bmatrix}$, $\boldsymbol{B} = \begin{bmatrix} b_1 & b_2 & \cdots & b_n \\ b_2 & b_3 & \cdots & b_1 \\ \vdots & \vdots & & \vdots \\ b_n & b_1 & \cdots & b_{n-1} \end{bmatrix}$

Circulant matrices are those s.t:

$$\boldsymbol{C}_{ij} = \boldsymbol{C}_{i+1,j-1} \qquad \text{where} \qquad \boldsymbol{C}_{i,0} = \boldsymbol{C}_{in},\ \boldsymbol{C}_{n+1,j} = \boldsymbol{C}_{1,j}$$

It suffices to show that:

$$\boldsymbol{A}_{ij} = \boldsymbol{A}_{i+1,j-1},\ \boldsymbol{B}_{ij} = \boldsymbol{B}_{i+1,j-1} \implies \boldsymbol{C}_{ij} = \boldsymbol{C}_{i+1,j-1} \qquad \forall i, j \in [n]$$

With invalid indexes re-mapped following the circulant matrix definition. We can show that for arbitrary i, j,

$$\begin{aligned} \boldsymbol{C}_{i+1,j-1} &= \sum_{k=1}^{n} \boldsymbol{A}_{i+1,k}\boldsymbol{B}_{k,j-1} \\ &= \sum_{k=1}^{n} \boldsymbol{A}_{i,k-1}\boldsymbol{B}_{k-1,j} \\ &= \boldsymbol{A}_{i,0}\boldsymbol{B}_{0,j} + \sum_{k=2}^{n} \boldsymbol{A}_{i,k-1}\boldsymbol{B}_{k-1,j} \\ &= \boldsymbol{A}_{in}\boldsymbol{B}_{nj} + \sum_{k=1}^{n-1} \boldsymbol{A}_{ik}\boldsymbol{B}_{kj} \\ &= \sum_{k=1}^{n} \boldsymbol{A}_{ik}\boldsymbol{B}_{kj} \\ \boldsymbol{C}_{i+1,j-1} &= \boldsymbol{C}_{ij} \end{aligned}$$

Thus the product of two circulant matrices is also circulant.

3. Let $\boldsymbol{A} \in \mathbb{R}^{n \times n} : \boldsymbol{A}_{ij} = 0 \; \forall j \in [n] \setminus \{i-1, i, i+1\}$. T.P.T. for $\boldsymbol{C} = \boldsymbol{AA}$,

$$\boldsymbol{A}_{ij} = 0 \quad \forall j \notin \{i-1, i, i+1\} \implies \boldsymbol{C}_{il} = 0 \quad \forall l \in [n] \setminus \{i-2, i-1, i, i+1, i+2\}$$

Selecting arbitrary $l : l \in [n] \setminus \{i-2, i-1, i, i+1, i+2\}$:

$$\boldsymbol{C}_{il} = \sum_{k=1}^{n} \boldsymbol{A}_{ik}\boldsymbol{A}_{kl} = \sum_{k=1}^{n} \boldsymbol{A}_{ik} \cdot 0 = 0$$

Thus $\boldsymbol{C}$ is a matrix with 5 diagonals.

Problem 4

1. We have $\boldsymbol{A} = \begin{bmatrix} \boldsymbol{C} & \boldsymbol{I} \\ -\boldsymbol{I} & \boldsymbol{C}^T \end{bmatrix}$. T.P.T:

$$\exists \boldsymbol{A}^{-1} \quad \text{s.t.} \quad \boldsymbol{A}^{-1}\boldsymbol{A} = \boldsymbol{I} \quad \forall \boldsymbol{C} \in \mathbb{R}^{n\times k}$$

$$\text{Let } \boldsymbol{A}^{-1} = \begin{bmatrix} \boldsymbol{P} & \boldsymbol{Q} \\ \boldsymbol{R} & \boldsymbol{S} \end{bmatrix} \quad \text{where} \quad \boldsymbol{P} \in \mathbb{R}^{k\times n}, \boldsymbol{Q} \in \mathbb{R}^{k\times k}, \boldsymbol{R} \in \mathbb{R}^{n\times n}, \boldsymbol{S} \in \mathbb{R}^{n\times k}$$

$$\begin{aligned}
\text{Then } \boldsymbol{A}\boldsymbol{A}^{-1} = \boldsymbol{I} &\implies \begin{bmatrix} \boldsymbol{C} & \boldsymbol{I} \\ -\boldsymbol{I} & \boldsymbol{C}^T \end{bmatrix}\begin{bmatrix} \boldsymbol{P} & \boldsymbol{Q} \\ \boldsymbol{R} & \boldsymbol{S} \end{bmatrix} = \boldsymbol{I} \\
&\implies \begin{bmatrix} \boldsymbol{C}\boldsymbol{P} + \boldsymbol{I}\boldsymbol{R} & \boldsymbol{C}\boldsymbol{Q} + \boldsymbol{I}\boldsymbol{S} \\ -\boldsymbol{I}\boldsymbol{P} + \boldsymbol{C}^T\boldsymbol{R} & -\boldsymbol{I}\boldsymbol{Q} + \boldsymbol{C}^T\boldsymbol{S} \end{bmatrix} = \boldsymbol{I} \\
&\implies \begin{bmatrix} \boldsymbol{C}\boldsymbol{P} + \boldsymbol{R} & \boldsymbol{C}\boldsymbol{Q} + \boldsymbol{S} \\ -\boldsymbol{P} + \boldsymbol{C}^T\boldsymbol{R} & -\boldsymbol{Q} + \boldsymbol{C}^T\boldsymbol{S} \end{bmatrix} = \begin{bmatrix} \boldsymbol{I}_{n\times n} & \boldsymbol{0}_{n\times k} \\ \boldsymbol{0}_{k\times n} & \boldsymbol{I}_{k\times k} \end{bmatrix}
\end{aligned}$$

Now we can solve for each partition separately!

$$\begin{aligned}
-\boldsymbol{P} + \boldsymbol{C}^T\boldsymbol{R} = \boldsymbol{0}_{k\times n} &\implies \boldsymbol{P} = \boldsymbol{C}^T\boldsymbol{R} \\
\boldsymbol{C}\boldsymbol{P} + \boldsymbol{R} = \boldsymbol{I}_{n\times n} &\implies \boldsymbol{C}\boldsymbol{C}^T\boldsymbol{R} + \boldsymbol{R} = \boldsymbol{I}_{n\times n} \\
&\implies (\boldsymbol{C}\boldsymbol{C}^T + \boldsymbol{I}_{n\times n})\boldsymbol{R} = \boldsymbol{I}_{n\times n} \\
&\implies \boldsymbol{R} = (\boldsymbol{C}\boldsymbol{C}^T + \boldsymbol{I}_{n\times n})^{-1}\boldsymbol{I}_{n\times n} \\
&\implies \boldsymbol{R} = (\boldsymbol{C}\boldsymbol{C}^T + \boldsymbol{I}_{n\times n})^{-1} \\
\therefore \boldsymbol{P} = \boldsymbol{C}^T\boldsymbol{R} &= \boldsymbol{C}^T(\boldsymbol{C}\boldsymbol{C}^T + \boldsymbol{I}_{n\times n})^{-1}
\end{aligned}$$

$$\begin{aligned}
\boldsymbol{C}\boldsymbol{Q} + \boldsymbol{S} = \boldsymbol{0}_{n\times k} &\implies \boldsymbol{S} = -\boldsymbol{C}\boldsymbol{Q} \\
-\boldsymbol{Q} + \boldsymbol{C}^T\boldsymbol{S} = \boldsymbol{I}_{k\times k} &\implies -\boldsymbol{Q} - \boldsymbol{C}^T\boldsymbol{C}\boldsymbol{Q} = \boldsymbol{I}_{k\times k} \\
&\implies \boldsymbol{Q} + \boldsymbol{C}^T\boldsymbol{C}\boldsymbol{Q} = -\boldsymbol{I}_{k\times k} \\
&\implies (\boldsymbol{I}_{k\times k} + \boldsymbol{C}^T\boldsymbol{C})\boldsymbol{Q} = -\boldsymbol{I}_{k\times k} \\
&\implies \boldsymbol{Q} = -(\boldsymbol{I}_{k\times k} + \boldsymbol{C}^T\boldsymbol{C})^{-1}\boldsymbol{I}_{k\times k} \\
&\implies \boldsymbol{Q} = -(\boldsymbol{I}_{k\times k} + \boldsymbol{C}^T\boldsymbol{C})^{-1} \\
\therefore \boldsymbol{S} = -\boldsymbol{C}\boldsymbol{Q} &= \boldsymbol{C}(\boldsymbol{I}_{k\times k} + \boldsymbol{C}^T\boldsymbol{C})^{-1}
\end{aligned}$$

It is important to note here that since $\boldsymbol{C}\boldsymbol{C}^T$ and $\boldsymbol{C}^T\boldsymbol{C}$ are symmetric matrices $\forall \boldsymbol{C} \in \mathbb{R}^{n\times k}$, to which adding $\boldsymbol{I}$ does not change this property ($\because$ it only updates the diagonal). Thus inverses for these are guaranteed to exist. We can extend this guarantee to $\boldsymbol{A}$, as each partition within $\boldsymbol{A}^{-1}$ is also guaranteed to exist:

$$\boldsymbol{A}^{-1} = \begin{bmatrix} \boldsymbol{P} & \boldsymbol{Q} \\ \boldsymbol{R} & \boldsymbol{S} \end{bmatrix} = \begin{bmatrix} \boldsymbol{C}^T(\boldsymbol{C}\boldsymbol{C}^T + \boldsymbol{I}_{n\times n})^{-1} & -(\boldsymbol{I}_{k\times k} + \boldsymbol{C}^T\boldsymbol{C})^{-1} \\ (\boldsymbol{C}\boldsymbol{C}^T + \boldsymbol{I}_{n\times n})^{-1} & \boldsymbol{C}(\boldsymbol{I}_{k\times k} + \boldsymbol{C}^T\boldsymbol{C})^{-1} \end{bmatrix}$$

2. We can solve $\begin{bmatrix} \mathbf{x} & \mathbf{y} \end{bmatrix}^T$ by computing the inverse:

$$\begin{bmatrix} \mathbf{x} \\ \mathbf{y} \end{bmatrix} = \boldsymbol{A}^{-1}\begin{bmatrix} \mathbf{f} \\ \mathbf{g} \end{bmatrix} = \begin{bmatrix} \boldsymbol{C}^T(\boldsymbol{C}\boldsymbol{C}^T + \boldsymbol{I}_{n\times n})^{-1} & -(\boldsymbol{I}_{k\times k} + \boldsymbol{C}^T\boldsymbol{C})^{-1} \\ (\boldsymbol{C}\boldsymbol{C}^T + \boldsymbol{I}_{n\times n})^{-1} & \boldsymbol{C}(\boldsymbol{I}_{k\times k} + \boldsymbol{C}^T\boldsymbol{C})^{-1} \end{bmatrix}\begin{bmatrix} \mathbf{f} \\ \mathbf{g} \end{bmatrix}$$

3. Copying the result from part 1, we have:

$$\boldsymbol{A}^{-1} = \begin{bmatrix} \boldsymbol{C}^T(\boldsymbol{C}\boldsymbol{C}^T + \boldsymbol{I}_{n\times n})^{-1} & -(\boldsymbol{I}_{k\times k} + \boldsymbol{C}^T\boldsymbol{C})^{-1} \\ (\boldsymbol{C}\boldsymbol{C}^T + \boldsymbol{I}_{n\times n})^{-1} & \boldsymbol{C}(\boldsymbol{I}_{k\times k} + \boldsymbol{C}^T\boldsymbol{C})^{-1} \end{bmatrix}$$

4. Let $\boldsymbol{I} - \sigma\mathbf{u}\mathbf{v}^T \in \mathbb{R}^{n\times n}$ be an elementary matrix. $\boldsymbol{I}$ is a rank-n matrix and the right hand term is a rank-1 matrix. $\boldsymbol{I}^{-1} = \boldsymbol{I}$ and we can absorb σ into $\mathbf{u}$. This allows us to directly apply the Sherman–Morrison formula, which states that:

$$\exists(\boldsymbol{A} + \mathbf{u}\mathbf{v}^T)^{-1} \iff 1 + \mathbf{v}^T\boldsymbol{A}^{-1}\mathbf{u} \neq 0$$

Further, if the inverse exists, it is given as:

$$(\boldsymbol{A} + \mathbf{u}\mathbf{v}^T)^{-1} = \boldsymbol{A}^{-1} - \frac{\boldsymbol{A}^{-1}\mathbf{u}\mathbf{v}^T\boldsymbol{A}^{-1}}{1 + \mathbf{v}^T\boldsymbol{A}^{-1}\mathbf{u}}$$

For our case, $\boldsymbol{A} = \boldsymbol{I}$, $\mathbf{u} = -\sigma\mathbf{u}$. Therefore our condition for invertibility is:

$$\exists(\boldsymbol{I} - \sigma\mathbf{u}\mathbf{v}^T)^{-1} \iff \sigma\mathbf{v}^T\mathbf{u} \neq 1$$

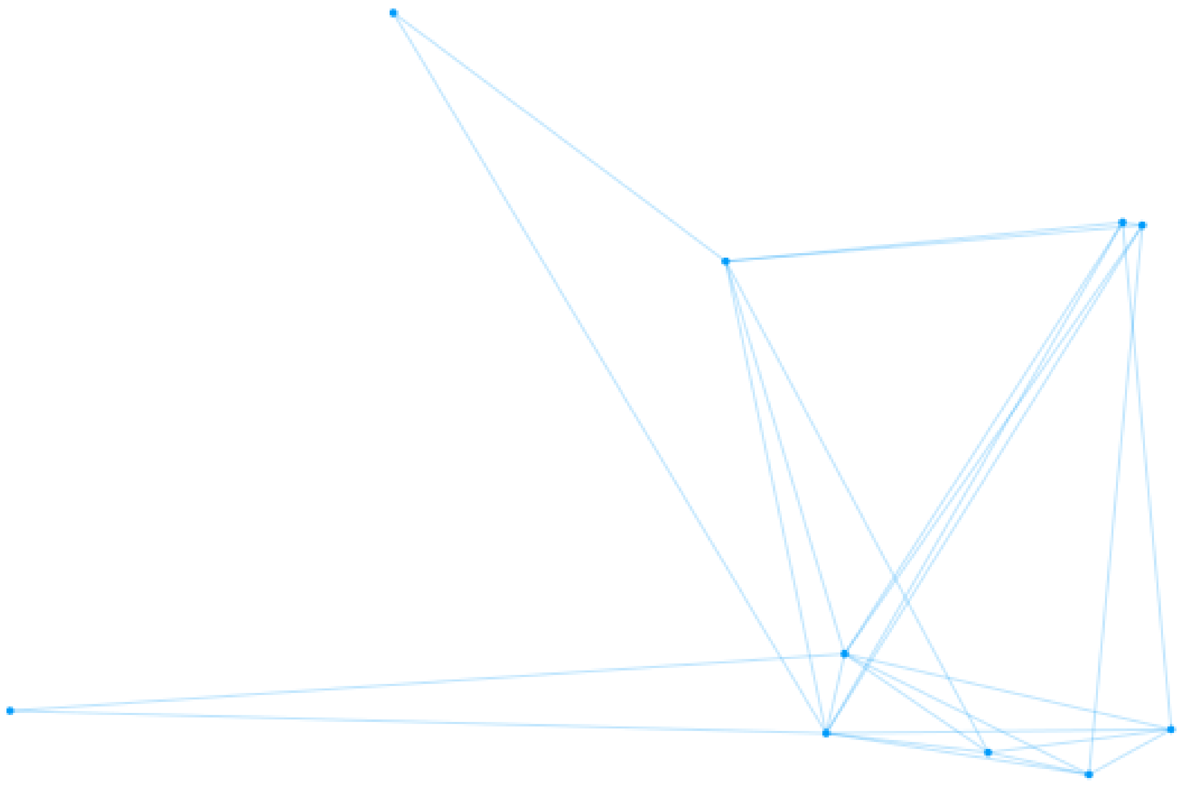
If the above condition is satisifed, then we can take the inverse as:

$$(\boldsymbol{I} - \sigma\mathbf{u}\mathbf{v}^T)^{-1} = \boldsymbol{I} + \frac{\sigma\mathbf{u}\mathbf{v}^T}{1 - \sigma\mathbf{v}^T\mathbf{u}}$$

Problem 5

1.
```
julia> A
10×10 SparseMatrixCSC{Int64, Int64} with 52 stored entries:
 0  1  0  1  1  0  0  1  1  0
 1  0  1  1  1  1  0  1  1  1
 0  1  0  1  0  0  0  0  0  0
 1  1  1  0  1  1  1  1  1  1
 1  1  0  1  0  0  0  0  1  1
 0  1  0  1  0  0  0  1  1  1
 0  0  0  1  0  0  0  0  1  0
 1  1  0  1  0  1  0  0  0  1
 1  1  0  1  1  1  1  0  0  0
 0  1  0  1  1  1  0  1  0  0

julia> xy
2×10 Matrix{Float64}:
 0.876903  0.677109  0.11713   0.664807  0.863706  0.773448  0.374355  0.8411
 ↪  0.597275  0.89628
 0.717874  0.272765  0.213706  0.190383  0.720768  0.17047   0.938415  0.147618
 ↪  0.680474  0.19432
```

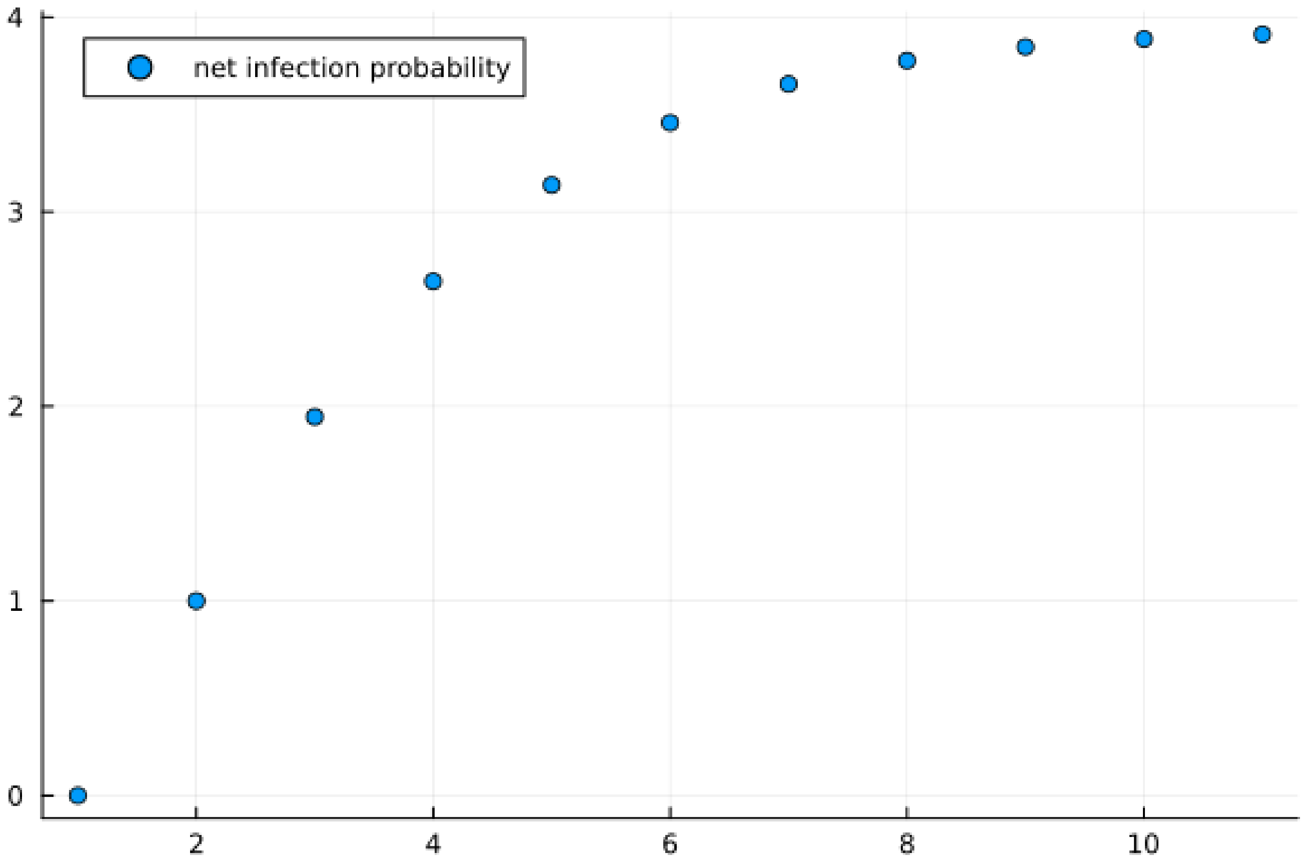
2.



3. The starting condition ρ determines the potency of spreading, as it scales the probability of being infected given the probability that a given neighbor is infected. To begin the simulation, we must first infect somebody (quite evil but anything for science). So we seal the fate of the first node by infecting it with probability 1, this is patient zero.

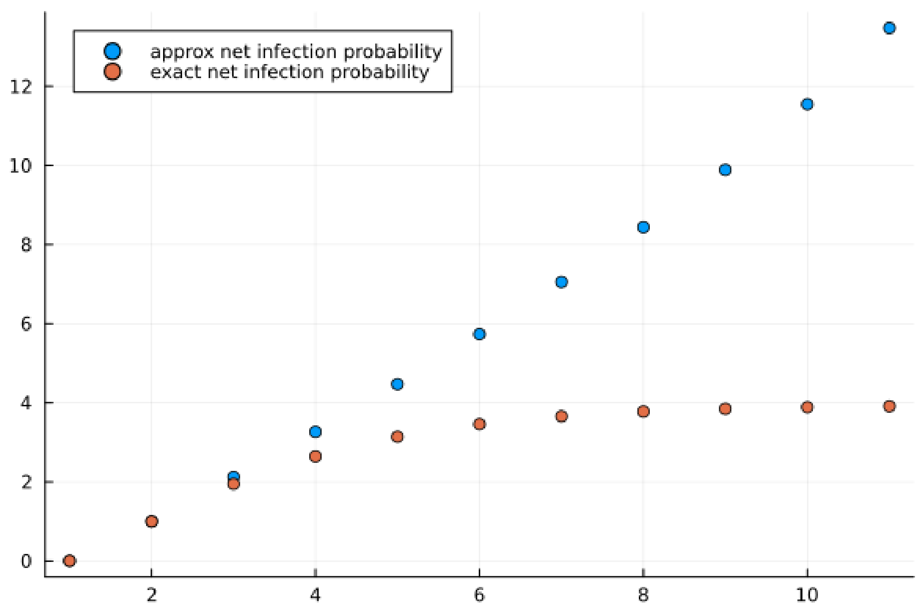
```
function evolve_steps(x0::Vector, p::Real, A::AbstractMatrix, k::Int)
        X = zeros(length(x0),k+1)
        X[:, 1] = x0
        for i=1:k
                X[:,i+1] = max.(evolve(X[:,i], p, A), X[:,1]) # fix the
                    initial probability x0
        end
        return X
end
```

4.

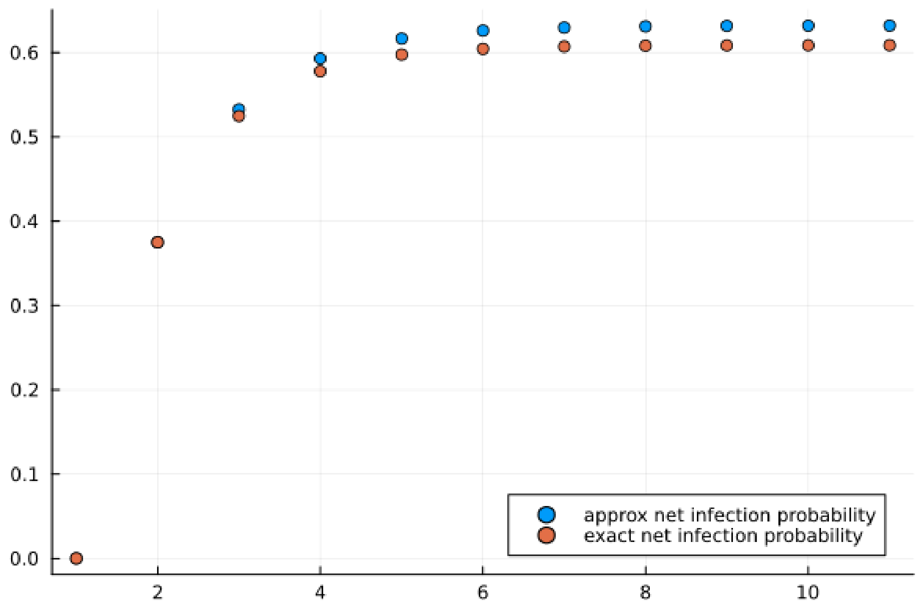


I think this plots looks looks fairly intuitive given it is monotonic. However, it seems to be approaching a steady state that doesn't leave everyone infected, so to me this is somewhat surprising. The change I would make is that instead of setting x0 as the minimum, I'd set `X[:, i]` as the minimum when computing the updated probability states, given that probability of infection for each node must remain monotonic.

5.
```
function approx_evolve_steps(x0::Vector, p::Real, A::AbstractMatrix, k::
    Int)
        X = zeros(length(x0),k+1)
        X[:, 1] = x0
        for i=1:k
                X[:,i+1] = max.(p.*(A*X[:, i]), X[:, i])
        end
        return X
end
```



Overlaying both, they're not very close. $\rho = .2$ is pretty aggressive so that does check out. I tried a couple other values, and it seems to be pretty similar as far as $\rho = .075$.



6. Computing exactly, we have:

```
0.05: 2.2147609258487027
0.1: 13.284931849129581
0.15: 91.25316599696535
0.2: 187.01853110400157
```

Computing the approximations, we have:

```
0.05: 2.245887055423145
0.1: 17.525090005300004
0.15: 414.43051462288486
0.2: 5935.081178726405
```

7. $f = 0.1$:

```
Exact -
0.05: 2.1855337586969155
0.1: 10.986070393749465
0.15: 79.19958635793614
0.2: 175.4514119107722

Approximate -
0.05: 2.2158157881100586
0.1: 14.316362738700006
0.15: 298.2168370196164
0.2: 4190.310200012802
```

$f = 0.2$

```
Exact -
0.05: 2.069686553575516
0.1: 7.212625479439187
0.15: 48.22007277926233
0.2: 137.27831253501301

Approximate -
0.05: 2.090935256316993
0.1: 8.622585344600003
0.15: 120.20576733644496
0.2: 1594.8595357696008
```

$f = 0.3$

```
Exact -
0.05: 1.85344784101363
0.1: 4.73449001259144
0.15: 19.810011769879956
0.2: 76.3573538688463

Approximate -
0.05: 1.8692448553495118
0.1: 5.5370666049000015
0.15: 45.870099183829666
0.2: 539.1967444992002
```

$f = 0.4$

```
Exact -
0.05: 1.6508958815006438
0.1: 3.314971265463761
0.15: 9.216509725827118
0.2: 28.680290981075718

Approximate -
0.05: 1.6612361319829108
0.1: 3.6671809465000003
0.15: 16.970630170221185
0.2: 154.77591828480007
```

$f = 0.5$

```
Exact -
0.05: 1.6148047520808846
0.1: 2.897113115881908
0.15: 6.0135671139733216
0.2: 12.654701109935086

Approximate -
0.05: 1.6234464478638668
0.1: 3.124832097600001
0.15: 9.75216472124746
0.2: 67.11750686720002
```

$f = 0.6$

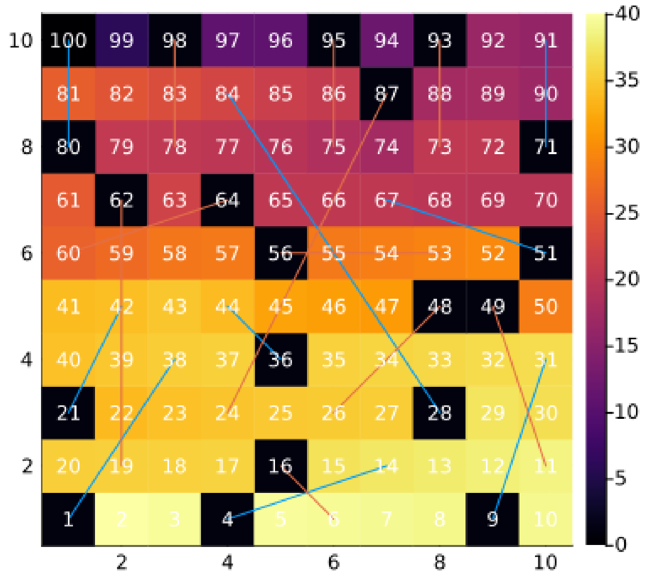
```
Exact -
0.05: 1.3884126229957134
0.1: 2.119473632223249
0.15: 3.8920328951181467
0.2: 7.9593681770090745

Approximate -
0.05: 1.3916229429317384
0.1: 2.1842057561000003
0.15: 4.794743637666894
0.2: 18.154333388800005
---
```

8. Wearing masks (correctly!) would reduce the potency of the spread, given the chance of infecting another person is reduced by the barrier induced by the mask. Thus $\rho_{\text{masks}} \ll \rho_{\text{orig}}$, if ρ is small enough, it will take significantly longer to converge. This however also de-incentivizes social distancing. Therefore treating these two variables as independent is perhaps not a solid assumption, but assuming our approximate analytical model. After k steps, $\rho_{\text{masks}}^k \ll \rho_{\text{orig}}^k$ even if the impact over an individual step is small, the network effect is large.

Problem 6

1.
```
using DelimitedFiles, LinearAlgebra, SparseArrays, Plots


coords = readdlm("chutes-and-ladders-coords.csv",',')
data = readdlm("chutes-and-ladders-matrix.csv",',')

xc = coords[:, 1]
yc = coords[:, 2]

TI = Int.(data[:,1])
TJ = Int.(data[:,2])
TV = data[:,3]
T = sparse(TI, TJ, TV, 101, 101)
A = -T' + I  #  linear system in terms of T

y = ones(101)
y[100] = 0

x = A\y
```

Here's the resultant plot:



$x_{\{101\}} \approx 39.598$, whereas $x_{\{2\}} \approx 40.071$, which means that the start is actually *not* the state with the maximum expected length. This makes intuitive sense, because we observe that the ladder on state 1 gives a significant boost to a player, sending them straight to state 38. This ladder cannot be accessed by a player on state 2.

2. Once we are past the maximum expected number of moves for a given state, the series reduces monotonically, thus we have a lower bound of the infinite series by computing the partial sum. Once summands reduce below a given threshold (arbitrarily set as 10^{-6}), we can stop the count.

```
expected_length = 0.0
delta = 1.
flag = true
k = 3
```

```
while delta > 1e-6 || flag
        global delta = k * (T^(k-1)*T[:, 101])[100]
        global expected_length += delta
        global k += 1

        if delta > 1e-6
                global flag = false # trip flag
        end
end
```

This program computes upto $k = 429$ and gets an expected length of ≈ 39.598, matching our analytical solution.

Problem 7

1. We have:

$$\boldsymbol{U} = \begin{bmatrix} u(x_0, y_0) & \cdots & u(x_0, y_n) \\ \vdots & \ddots & \vdots \\ u(x_n, y_0) & \cdots & u(x_n, y_n) \end{bmatrix}$$

   Flattened in row-major order to $\mathbf{u}$; thus $\mathbf{u}_{(n+1)i+j} = \boldsymbol{U}_{ij}$. To simplify notation, we'll permute the order and use matrix indexing (also similarly for $\mathbf{f}$ as $\boldsymbol{F}$) with the understanding that it maps following the rule above. We linearly approximate the 2D Laplacian with $\boldsymbol{A}\mathbf{u} = \mathbf{f} = \Delta \mathbf{u}$. We can generalize the form up to $\boldsymbol{B}$:

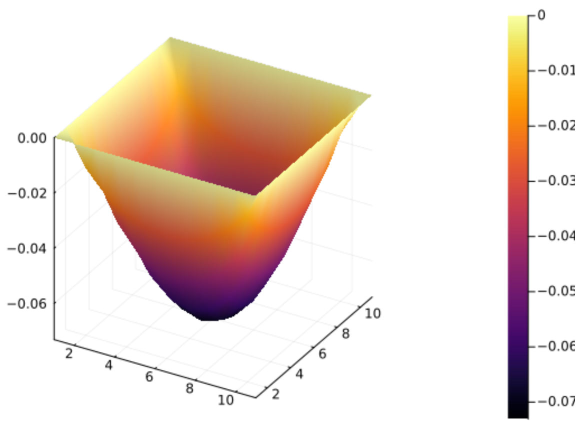
$$\begin{bmatrix} \boldsymbol{I}_{4n\times(n+1)^2} \\ \boldsymbol{B} \end{bmatrix} \begin{bmatrix} \boldsymbol{U}_{00} \\ \boldsymbol{U}_{01} \\ \vdots \\ \boldsymbol{U}_{0,n} \\ \boldsymbol{U}_{n,0} \\ \boldsymbol{U}_{n,1} \\ \vdots \\ \boldsymbol{U}_{nn} \\ \boldsymbol{U}_{10} \\ \vdots \\ \boldsymbol{U}_{n-1,0} \\ \boldsymbol{U}_{1,n} \\ \vdots \\ \boldsymbol{U}_{n-1,n} \\ \boldsymbol{U}_{1,1} \\ \boldsymbol{U}_{1,2} \\ \vdots \\ \boldsymbol{U}_{1,n-1} \\ \boldsymbol{U}_{2,1} \\ \vdots \\ \boldsymbol{U}_{n-1,n-1} \end{bmatrix} = \frac{1}{n^2} \begin{bmatrix} \mathbf{0}_{1\times 4n} \\ \boldsymbol{F}_{1,1} \\ \boldsymbol{F}_{1,2} \\ \vdots \\ \boldsymbol{F}_{1,n-1} \\ \boldsymbol{F}_{2,1} \\ \vdots \\ \boldsymbol{F}_{n-1,n-1} \end{bmatrix}$$

   For $n = 3$, $\boldsymbol{B} = \mathbf{0}_{(n+1)^2-4n\times(n+1)^2}$ except for $\boldsymbol{B}_{1,2} = \boldsymbol{B}_{1,2n+3} = \boldsymbol{B}_{1,4n+2} = \boldsymbol{B}_{1,4n+4} = \boldsymbol{B}_{2,3} = \boldsymbol{B}_{2,3n+3} = \boldsymbol{B}_{2,4n+1} = \boldsymbol{B}_{2,4n+4} = \boldsymbol{B}_{3,3n+2} = \boldsymbol{B}_{3,n+3} = \boldsymbol{B}_{3,4n+1} = \boldsymbol{B}_{3,4n+4} = \boldsymbol{B}_{4,n+2} = \boldsymbol{B}_{4,4n} = \boldsymbol{B}_{4,4n+2} = \boldsymbol{B}_{4,4n+3} = 1$, and $\boldsymbol{B}_{1,4n+1} = \boldsymbol{B}_{2,4n+2} = \boldsymbol{B}_{3,4n+3} = \boldsymbol{B}_{4,4n+4} = -4$.

2.
```julia
using LinearAlgebra, SparseArrays, Plots

function map_index(i::Integer, j::Integer, n::Integer)
        if 1 < i < n+1 && 1 < j < n+1
                return 4n + (i - 2)*(n-1) + j-1
        elseif i == 1
                return j
        elseif i == n+1
                return n + 1 + j
        elseif j == 1
                return 2(n+1) + i - 1
        elseif j == n+1
                return 2(n+1) + n - 2 + i
        end
end

function map_index_inv(k::Integer, n::Integer)
        if k <= n+1
                return 1, k
        elseif k <= 2(n+1)
                return n + 1, k - (n+1)
        elseif k < 2(n+1) + n
                return k - 2(n + 1) + 1, 1
        elseif k <= 4n
                return k - 2(n + 1) - n + 2, n + 1
        else
                return div(k-1 - 4n, n-1) + 2, (k-1 - 4n)%(n-1) + 2
        end
end

function laplacian(n::Integer, f::Function)
        A = sparse(1I, (n+1)^2, (n+1)^2)
        A[diagind(A)[4n+1:end]] .= -4

        fvec = zeros((n+1)^2)

        global row_index = 4n + 1
        for i in 2:n
                for j in 2:n
                        A[row_index, map_index(i-1, j, n)] = 1
                        A[row_index, map_index(i+1, j, n)] = 1
                        A[row_index, map_index(i, j-1, n)] = 1
                        A[row_index, map_index(i, j+1, n)] = 1
                        fvec[row_index] = f(i, j)

                        global row_index += 1
                end
        end

        return A, fvec/n^2
end

n = 10
A, fv = laplacian(n, (x, y) -> 1)
```

3.

```
u = A\fv

U = spzeros(n+1, n+1)
for k in 4n+1:(n+1)^2
        i, j = map_index_inv(k, n)
        U[i, j] = u[k]
end

surface(U)
```



4. On trimming out the boundary conditions, we get a new solution $\boldsymbol{U}'$. $\|\boldsymbol{U} - \boldsymbol{U}'\|_F \approx 1.96 \cdot 10^{-16}$. Thus we can exclude the boundary condition and get the same solution.

```
A_smol = A[4n+1:end,4n+1:end]
fv_smol = fv[4n+1:end]
u_smol = A_smol\fv_smol

norm(u_smol - u[4n+1:end])
```

Problem 8

1.
```
function csc_transpose_matvec(colptr, rowval, nzval, m, n, x)
        res = zeros(n)

        for col_idx in 1:n
                for col_ptr in colptr[col_idx]:(colptr[col_idx+1]-1)
                        res[col_idx] += x[rowval[col_ptr]] * nzval[
                            col_ptr]
                end
        end

        return res
end
```

2.
```
function csc_column_projection(colptr, rowval, nzval, m, n, i, x)
        res = 0
        for col_ptr in colptr[i]:(colptr[i+1]-1)
                res += x[rowval[col_ptr]] * nzval[col_ptr]
        end

        return res
end
```

3.
```
function csc_col_col_prod(colptr, rowval, nzval, m, n, i, j)
        res = 0
        for col_ptr_i in colptr[i]:(colptr[i+1]-1)
                for col_ptr_j in colptr[j]:(colptr[j+1]-1)
                        if rowval[col_ptr_i] == rowval[col_ptr_j]
                                res += nzval[col_ptr_i] * nzval[col_ptr_j
                                    ]
                        end
                end
        end

        return res
end
```

4.
```
function csc_lookup(colptr, rowval, nzval, m, n, i, j)
        for col_ptr in colptr[j]:(colptr[j+1]-1)
                if rowval[col_ptr] == i
                        return nzval[col_ptr]
                end
        end

        return 0
end
```

5.
```
function csc_lookup_row(colptr, rowval, nzval, m, n, i)
        res = zeros(n)
        for col_idx in 1:n
                for col_ptr in colptr[col_idx]:(colptr[col_idx+1]-1)
```

```
                        if rowval[col_ptr] == i
                                res[col_idx] = nzval[col_ptr]
                        end
                end
        end

        return res
end
```