CS 138 PROBLEM SET

Yenzy Urson S. Hebron | 202003090
Sir Jaymar Soriano | CS138 THY 1st Sem AY 2022-2023
Problem Set

1. The null space of a matrix and the general solution of $Ax = b$. (item1.m)

a. Understanding of the problem.

My understanding of the problem is that this is a general SLE problem with the assumption that the given system $Ax = b$ is always consistent. Here, $A \in \mathbb{R}^{m,n}, x \in \mathbb{R}^n$, and $b \in \mathbb{R}^m$ wherein we receive A and b as inputs and solve for x. This is in contrast to the special case of the SLE problem that we encountered in our discussions where $m = n$ always, i.e. A is always a square matrix. Hence, we expect here that we will have to adjust the discussed methods to accommodate $m \neq n$ or when A is a rectangular matrix, particularly the indexing of our loops. We also expect that given the hints in the Problem Set guide, we are going to use Gaussian solvers that give us the $REF(A)$ and the $RREF(A)$ as these matrix forms will help us solve for a particular and the general solution of $Ax = b$.

b. Implementation of the solution (analytical derivations/equation setup, paste code into the PDF and discuss, etc.).

Heads up! I left a lot of comments in my code which I hope would bring further clarity to its whats, whys, and hows. I hope the reader finds them helpful.

Now showing my code for this problem which we will discuss afterwards.

```
% item1.m [Hebron, Yenzy]
% Test cases also submitted
A = [1 -2 -1 3; 2 -4 1 0; 1 -2 2 -3]
b = [1 5 4]'
% ans = [2 0 1 0; 2 1 0 0; -1 0 2 1]'

x = gaussjordan(A,b)    % get solution x

% To check, see if norm of residual is within our tolerance 1E-10.
% Check base case: free variables are set to 0
if norm(b - A*x(:,1), 'inf') < 1E-10
    fprintf('Base case: xhat is correct.\n')
else
    fprintf('Base case: xhat is incorrect!\n')
end

% Check particular: free variables can be nonzero
% Test 10000 random test cases
% This might be very slow for huge matrices, so please comment this out
% if ever the performance impact is very bad.
free = size(x,2) - 1;
check = 1;
for i = 1:10000
    % plug serves as the modifier of x.
    % For Pset example, think of plug as [1 s t]'.
    plug = [ 1 randi([-10000 10000], 1, free) ]';
    if norm(b - A*(x*plug), 'inf') >= 1E-10
        check = 0;
        break
    end
end

if check == 1
    fprintf('All tests passed!\n')
    fprintf('Particular soln xhat*plug is correct.\n')
else
    fprintf('Failed test case detected!\n')
    fprintf('!! Particular soln xhat*plug is incorrect!\n')
end

% Solve for the solution x of Ax=b.
function ret = gaussjordan(A,b)
    % Get the RREF of A

    [m,n] = size(A);
    p = 0;          % Current number of leading 1s or pivots. p = pivots
                    % Becomes rank(A) upon getting REF(A).
                    % rank(A) := # of nonzero rows or leading 1s of REF(A).

    % Augment A with b so all transformations done to A also impacts b.
    A = [A b];  clear b;
    % Get physical limit of unit pivots
    % i.e. max # of nonzero rows or leading 1s (pivot cols) of REF(A).
    if m >= n
        limit = n; % if tall/sqr matrix, limit dictated by # of cols n
    else
        limit = m; % if fat matrix, limit dictated by # of rows m
    end

    %% Part 1: A to REF(A) (Gaussian Elimination with Pivoting)
    for j = 1:n
        % If number of leading 1s already at matrix limit, stop.
        if p == limit
            break
        end

        % Pivoting, increases stability of the method, and also collects
        % all eventually non-zero rows of REF(A) on top rows of the matrix.
        % ERO1: Switch row with largest magnitude with active row.
        [~,k] = max(abs(A(p+1:end,j)));
        % Adjust k to account for subarray A(p+1:end,j).
        % Bug Fix: We use subarray A(p+1:end,j) to avoid switching with
        % "finished rows" (rows already with pivot column) on the
        % next jth (column) iterations.
        k = p+k;
        % k now contains the index of row with largest magnitude.
        % Active row index is given by p+1 (since p starts at 0).
        if p+1 ~= k     % If switching by same row ignore switch
            A([p+1 k], j:end) = A([k p+1], j:end);
        end
        % Also nice to know that the max() function behaves in such a
        % way that we don't switch when row entries in the subarray have
        % the same magnitude.
        % ERO2: Normalize active row
        % Set to zero numerically zero entry, then continue to next
        % column (i.e. there cannot be a pivot column in this column)
        if abs(A(p+1,j))<1E-10
            A(p+1,j)=0;
            continue
        end
        % If a non-numerically zero leading entry in active row is found,
        % then we will have a unit pivot. Increase number of found pivots,
        % then proceed to normalize.
        p = p + 1;  % Active row is now given by just p.
        A(p,j:end) = A(p,j:end)/A(p,j);

        % ERO3: Annihilate entries below leading entry of active row.
        for i = p+1:m
            if abs(A(i,j)) < 1E-10
                A(i,j) = 0;
                continue
            end
            A(i,j:end) = A(i,j:end)-A(i,j)*A(p,j:end);
        end
    end

    clear limit;
    REF = A   % Show REF(A)

    %% Part 2: REF(A) to RREF(A) (Gauss-Jordan Reduction)
    % Look for pivot points then annihilate entries above pivot points.
    % Note that due to the pivoting implemented here,
    % the nonzero rows are going to be contiguous on the top part
    % of the matrix.
    pcols = [];     % Columns with unit pivots
    for i = p:-1:1  % Start from highest index nonzero row, step backwards.
        for j = 1:1:n   % Scan left to right to guarantee leading entry.
            if A(i,j) == 0
                continue
            end
            % Leading entry (1) found at A(i,j)
            % Annihilate entries above A(i,j:end), just j:end since
            % zero entries don't have effect on other entries.
            for k = i-1:-1:1
                A(k,j:end) = A(k,j:end) - A(k,j)*A(i,j:end);
            end
            % Record pivot column at pcols
            pcols(size(pcols,2)+1) = j;

            break
        end
    end

    RREF = A   % Show RREF(A)

    % Get fcols AFTER getting RREF(A) for better performance.
    % Note: This algo does not see zero columns as free variables.
```

```
% This is as expected, since zero columns mean variables set to 0.
pcols = sort(pcols) % Sort first to also get fcols from left to right
fcols = [];     % Columns with free variables
% Start checking at col to the right of a pcol
for i = 1:p
    % Check if all free variable columns found.
    % n-p := expected number of free variables
    if size(fcols,2) == n-p
        break
    end
    for j = pcols(i)+1:n
        % Also check: if j is already recorded in fcols, move on
        if A(i,j) == 0 || ismember(j,fcols) == 1
            continue
        end
        % Free variable column found at column j, record
        fcols(size(fcols,2)+1) = j;
    end
end
% Possible optimization: Just record all j's and
% filter at the end using unique(). Tried this, got Index Error.

%% Part 3: Extracting ret from RREF(A)
% Note that p also counts the number of nonzero rows.
% Sort pcols and fcols in preparation for return extraction.
fcols = sort(fcols)

% Create placeholder matrix for ret, n-p columns for free variables
% i.e. basis of nullspace(A), and 1 column for particular solution
ret = zeros(n,n-p);
particular = zeros(n,1);

% Particular Solution
% Assign constants b to rows corresponding to pcols
% This assumes a consistent system
% i.e. no zero row corresponding to a nonzero constant
particular(pcols,1) = A(1:p, end);

% If no free variable or trivial input, skip code for free variables.
% Trivial input := no pivots found, i.e. zero matrix
% Do NOT use below n-p as # of free vars to avoid bug on zero columns
% (It happens that n-p > 0 when there really are no free variables)
% Just check if fcols contains anything or if there are pivots.
if size(fcols,2) == 0 || p == 0
    ret = particular;
    return;
end

% Basis of Nullspace(A)
for j = 1:n-p
    for i = 1:p
        % Use pcols and fcols to "translate" indices bet ret and A
        % Get negative values since we are "transposing" them.
        ret(pcols(i),j) = -A(i,fcols(j));
        % Breakdown: A(i,fcols(j)) takes free variable coefficients
```

```
            % from A, with j representing which free variable we are
            % dealing with (e.g. in the Pset example, j=1=s, j=2=t).
            % This value is translated to ret(pcols(i),j) which represents
            % how the transposed free variable j values are captured by the
            % solution variables x, with pcols(i) telling which
            % pivot x_k gets the free variable value.
        end
        % Insert 1 to row representing free variable
        ret(fcols(j),j) = 1;
    end

    % Augment particular solution with basis of nullspace of A
    ret = [particular ret];

    return;
end
```

Discussion

Assuming that our input A, b is as provided in the PSet guide, let us step into the gaussjordan(A,b) function. Note that "gaussjordan" here is actually a misnomer due to the many modifications we've done to what was discussed, but we use the name because the algorithm we made still relies heavily on getting the $RREF(A)$ which is the end product of a Gauss-Jordan Reduction method.

Inside the function, we first do the standard stuff, such as taking the size of A and augmenting A with b. What's new is that we have a variable p which corresponds to the number of nonzero rows or leading 1s of $REF(A)$. Essentially, p is the number of pivots of $REF(A)$ and by extension $RREF(A)$. This information will be very useful, and we take advantage of this a lot. We initialize $p = 0$, and after getting REF(A), p will then contain the $rank(A)$. We also check if A is a tall, square, or fat matrix and obtain the appropriate physical limit of unit pivots for that matrix. E.g. if A is 3×4, then we can only have 3 nonzero rows in $REF(A)$, hence we set $limit = 3$.

Now, we proceed to the first of three (3) parts of this function.

Part 1. A to $REF(A)$ via Gaussian Elimination with Pivoting. This is very similar to the function we discussed in class. The modifications are that we also now check if $p == limit$ as guard condition, possibly improving performance, and we also conveniently use p for our row indexing.

The reason for our use of p is that because we now have a rectangular matrix, we have no guarantee of which column we will find the leading 1 that denotes a nonzero row, so the index j just iterates over all the columns while p stays the same until a leading 1 or pivot is found, at which point we increment p to protect "finished rows" collected at the top of the matrix from being computed again. Hence, p here is also our indirect index of what we call the "active row", i.e. the current row for which a leading 1 is being attempted to be found.

Speaking of "collected at the top of the matrix", this is a consequence of pivoting, which aside from increasing the stability of the method, also makes sure that at $REF(A)$, the nonzero rows of A are contiguous at the top of the matrix, and using p helps us ensure this helpful property.

After Part 1, we now have $REF(A)$ and $p = rank(A)$, we now proceed to Part 2.

Part 2. $\boldsymbol{REF(A)}$ to $\boldsymbol{RREF(A)}$ via Gauss-Jordan Reduction. This is again very similar to what we did in the class, much more similar than our Gaussian Elimination with Pivoting in Part 1. However, this time, every time we found a leading entry 1 for which we would annihilate the entries above it (below it are already 0s due to $REF(A)$), we also record the column position of that entry in $pcols$, which helps us later interpret the $RREF(A)$ we got and extract our results. Note here that because this is a rectangular matrix, we start scanning the columns for a leading entry (expected to be a 1) from left to right to guarantee that the entry we found is indeed "leading". Some optimizations on starting indices are also applied here.

We now have $RREF(A)$ and $pcols$ containing the pivot columns of A. Before we end Part 2, we also get the $fcols$ or free variable columns of A to complete the information needed to extract the results from $RREF(A)$. We first sort $pcols$ so that the $fcols$ we get are also in ascending order (this is consistent with the output formatting observed from the PSet guide). To get $fcols$, we simply check the columns to the right of each pivot column (because to the left of the first of these pivot columns are guaranteed to be all 0s), and if a nonzero entry is found, we treat that as a free variable column and record j to $fcols$. Because we may end up going over the same columns more than once (because the entries in a column can also be 0, leading to false negatives, which we avoid), we also have to include a check if j is already recorded in $fcols$ and if so, we skip that column.

Now, we have $RREF(A)$, $p = rank(A)$, $pcols =$pivot columns of A, $fcols =$free variable columns of A. Note that we also sort $fcols$ to fit the required output format. We are now ready for Part 3.

Part 3. Extracting $\boldsymbol{ret}$ from $\boldsymbol{RREF(A)}$. First, we create placeholder matrices for ret and $particular$ which we initialize to 0s so that undealt with entries are appropriately 0. Getting the particular solution is easy, we simply map the entries of b to their corresponding $pcols$ (i.e. solution variables x_i) and store them in $particular$ while ignoring $fcols$. This effectively just gets the solution when the free variables $s = t = \cdots = 0$.

Then if we actually have no free variable or if we just have a trivial input (no pivots found, i.e. zero coefficient matrix), then we simply return the particular solution.

Else, we also return the basis of the nullspace of A, $\eta(A)$ alongside our particular solution. To get this, we now use both pcols and fcols to translate indices between ret and $RREF(A)$ so the entries get to the variables that they correspond to. We use negative values since we are "transposing" them. A more detailed breakdown can be found in the code.

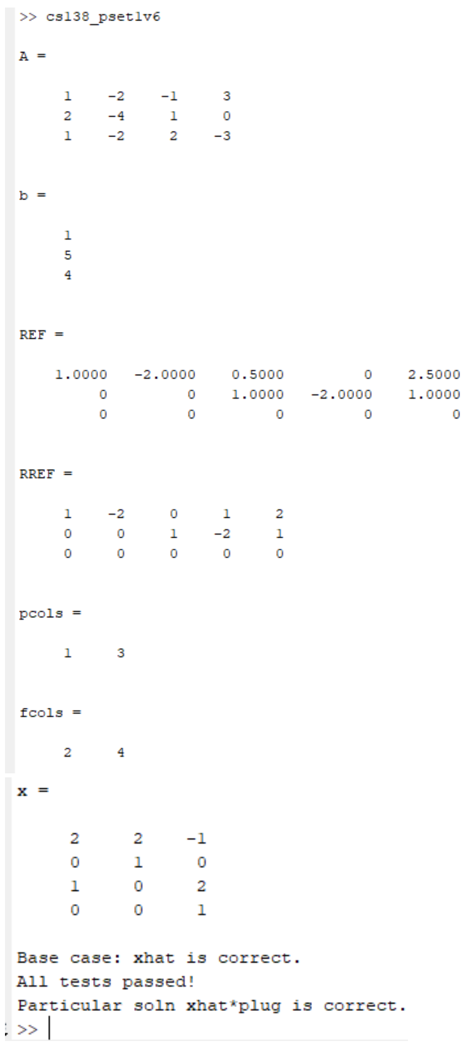
And with that, we have computed for the general solution of $Ax = b$.

To test the correctness of this algorithm (see code after x = gaussjordan(A,b)), we check if the norm of the residual $r = b - A\hat{x}$ is within our small enough tolerance 1E-10. Note here that x in the code is essentially $\hat{x}$. For the base case, i.e. the free variables are 0, we simply test using the first column of $\hat{x}$. For the non-trivial case, i.e. the free variables can be nonzero, we test using all the columns of x, modified into properly sized $\hat{x}$ by multiplying x with a $plug$ vector which contains random values for the free variables. We do this enough times and we can be sufficiently confident with the correctness of our solution (though this is still not a universal proof).

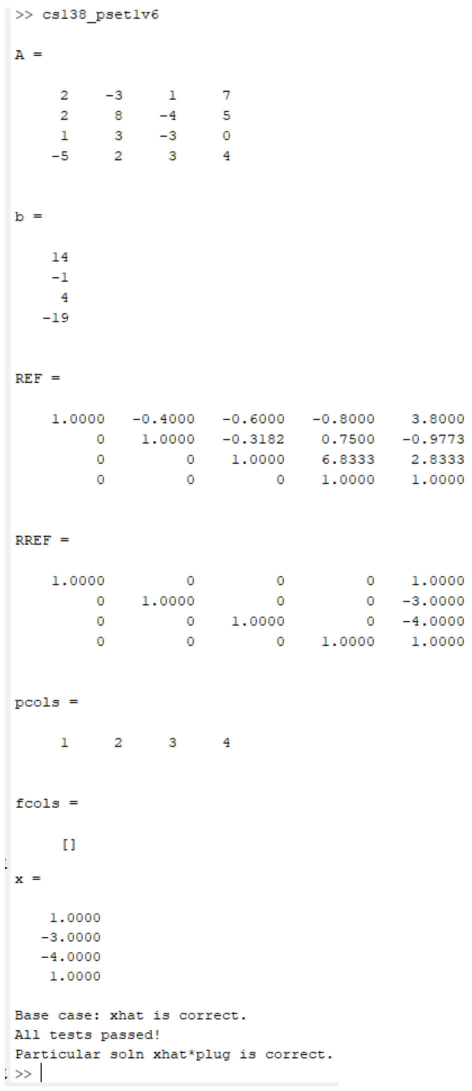
c. Results. Show some snapshots of solution process and plots appropriate in the discussion of the results. Problems requiring numerical solution must be implemented using non-built-in methods or libraries.

Our algorithm also outputs on the command window the $REF(A)$, $RREF(A)$, $pcols$, $fcols$, the solution x (essentially $\hat{x}$), and also if $\hat{x}$ passed our correctness tests.

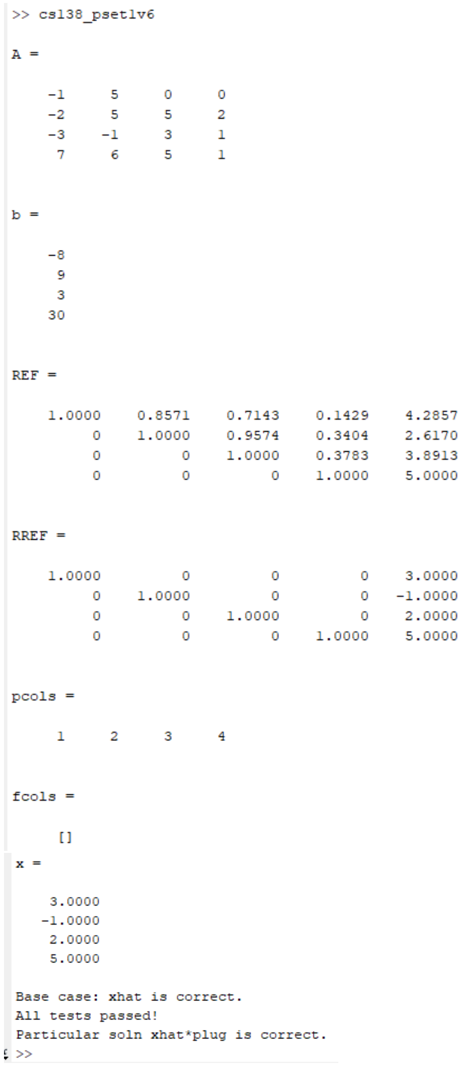
I'll just show these outputs for three test cases. Sorry for the small photos, please just zoom in.



```
>> cs138_pset1v6

A =

     1    -2    -1     3
     2    -4     1     0
     1    -2     2    -3


b =

     1
     5
     4


REF =

    1.0000   -2.0000    0.5000         0    2.5000
         0         0    1.0000   -2.0000    1.0000
         0         0         0         0         0


RREF =

     1    -2     0     1     2
     0     0     1    -2     1
     0     0     0     0     0


pcols =

     1     3


fcols =

     2     4

x =

     2     2    -1
     0     1     0
     1     0     2
     0     0     1

Base case: xhat is correct.
All tests passed!
Particular soln xhat*plug is correct.
>> |
```



```
>> cs138_pset1v6

A =

     2    -3     1     7
     2     8    -4     5
     1     3    -3     0
    -5     2     3     4


b =

    14
    -1
     4
   -19


REF =

    1.0000   -0.4000   -0.6000   -0.8000    3.8000
         0    1.0000   -0.3182    0.7500   -0.9773
         0         0    1.0000    6.8333    2.8333
         0         0         0    1.0000    1.0000


RREF =

    1.0000         0         0         0    1.0000
         0    1.0000         0         0   -3.0000
         0         0    1.0000         0   -4.0000
         0         0         0    1.0000    1.0000


pcols =

     1     2     3     4


fcols =

     []

x =

    1.0000
   -3.0000
   -4.0000
    1.0000

Base case: xhat is correct.
All tests passed!
Particular soln xhat*plug is correct.
>> |
```



```
>> cs138_pset1v6

A =

    -1     5     0     0
    -2     5     5     2
    -3    -1     3     1
     7     6     5     1


b =

    -8
     9
     3
    30


REF =

    1.0000    0.8571    0.7143    0.1429    4.2857
         0    1.0000    0.9574    0.3404    2.6170
         0         0    1.0000    0.3783    3.8913
         0         0         0    1.0000    5.0000


RREF =

    1.0000         0         0         0    3.0000
         0    1.0000         0         0   -1.0000
         0         0    1.0000         0    2.0000
         0         0         0    1.0000    5.0000


pcols =

     1     2     3     4


fcols =

     []

x =

    3.0000
   -1.0000
    2.0000
    5.0000

Base case: xhat is correct.
All tests passed!
Particular soln xhat*plug is correct.
>>
```

Note that for the full set of test cases (including edge cases) that I tested the algorithm on, please see **item1_tc.txt** which I also submitted. All of those test cases were passed. Thank you so much!

2. Image compression using Singular Value Decomposition. (item2.m)

a. Understanding of the problem.

Here, what we want to do is to obtain the Singular Value Decomposition of a given rectangular matrix $A \in R^{m,n}$, factorizing A such that $A = U\Sigma V^T$ wherein

- U is an orthogonal $m \times m$ matrix. Contains the left eigenvectors of A.
- Σ is a diagonal $m \times n$ matrix. Entries are principal square root of the eigenvalues of D_U and D_V from $AA^T = UD_UU^T$ and $A^TA = VD_UV^T$ wherein D_U and D_V contains the same set of positive eigenvalues (possibly in a different order). Contains the singular values of A arranged in increasing order:

$$\sigma_{11} > \sigma_{22} > \cdots > \sigma_k > \sigma_{k+1,k+1} = \sigma_{k+2,k+2} = \cdots = 0$$

- V is an orthogonal $n \times n$ matrix. Contains the right eigenvectors of A.

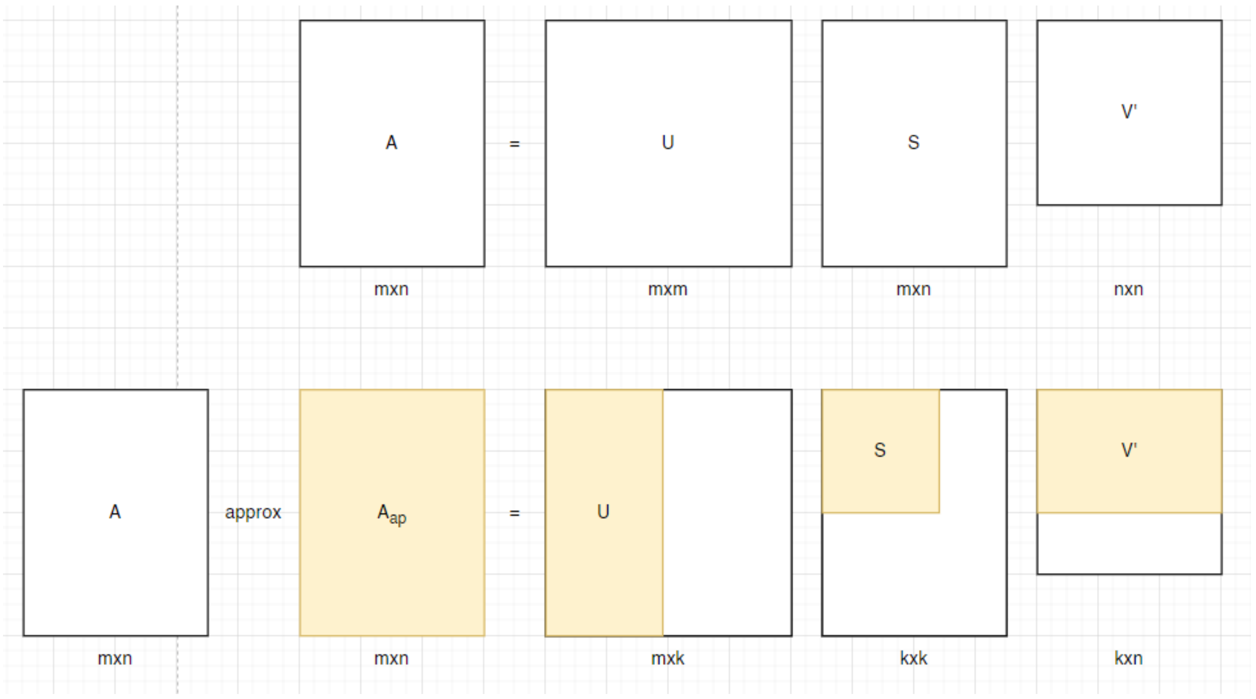
From $A = U\Sigma V^T$, we can observe that A can actually be compressed (saving space) by reconstructing A from $U\Sigma V^T$ as A_{ap} (presumably meaning "approximate A") using just a certain number k of singular values σ stored in Σ without great loss of data quality. In other words, we can simply use a $k \times k$ subarray of Σ and subarrays of U and V also selected according to k in order to approximate A. In equation form, what we're saying is that

$$A = U\Sigma V^T$$

wherein,

$$A \approx A_{ap} = U(:,1{:}k) * \Sigma(1{:}k,1{:}k) * V(1{:}k,:)^T$$

provided that k is chosen appropriately. Graphically, this is



Considering the arrangement of the contents of Σ, k is chosen as the index of the singular value just before all the next diagonal entries are 0 (or numerically zero). This means that we only require k singular values, k left eigenvectors, and k right eigenvectors to approximately reconstruct A (these can be seen in the illustration above as the yellow subarrays of the matrices). This means we can truncate the insignificant portions of the matrices and keep only the significant subarrays, saving space.

More so, the optimal k is essentially the smaller of m and n, and this hinges on the following theorem:

Theorem 1: Let $A \in \mathbb{R}^{m,n}$. Then AA^T is an $n \times n$ matrix with a total of n eigenvalues, and A^TA is an $m \times m$ matrix with a total of m eigenvalues. It can be shown that AA^T has an eigenvalue $\lambda \neq 0$ iff A^TA has an eigenvalue $\lambda \neq 0$. In other words, AA^T and A^TA have the same nonzero, positive eigenvalues. And if one has more eigenvalues than the other, then these eigenvalues are all equal to 0.[1]

This compression is more apparent visually when A is equivalent to a matrix I denoting black and white image data, and we expect that as we use increasing values k closer to the most significant k — let's call it $sigk$ — we'll get a better looking image that requires less space. If k goes over $sigk$, we'll find that there's no more increase in image quality because the next diagonal entries are already 0, so the eigenvectors "corresponding" to those entries don't have any impact to A_{ap}.

Hence, from here we can already say that compression is achieved for values of $k \leq sigk$ with k close enough to $sigk$ (i.e. the resulting data quality is still tolerable), with optimal compression (best data quality on smallest space consumption) at $k = sigk$. We'll demonstrate this later.

As Sir JM discussed, this can also be extended to RGB image data by applying SVD on the Red, Green, and Blue matrices separately, but that's a complicated topic for another day.

Now, the meat of the problem is how we can compute the Singular Value Decomposition of A, and for that, we rely on Methods for the Simultaneous Solution for the Eigenvalues of Symmetric Matrices, particularly QR Iteration with its subroutine QR Factorization based on Gram-Schmidt Orthonormalization. We can do this because we know that AA^T and A^TA are square and non-defective (fulfilling a Spectral Decomposition Theorem precondition), and symmetric (fulfilling a QR iteration precondition).

In particular, if A is a square matrix that is symmetric and non-defective, its eigenvectors corresponding to distinct eigenvalues of A are linearly independent (via non-defectivity) and orthogonal (via symmetry). We also note that QR iteration is rooted on Singular Value Decomposition, hence it works fine on either AA^T or A^TA which we said are always symmetric.

Many statements I made here are based on theorems that I've researched online. These notions and theorems are in pages 17-19 and 26-28 of my submitted notes. If ever I'll also attach these pages in the appendix of this document, but I don't think I have the time.

We discuss our SVD approach in the next section.

[1] https://math.stackexchange.com/questions/1087064/non-zero-eigenvalues-of-aat-and-ata

b. Implementation of the solution (analytical derivations/equation setup, paste code into the PDF and discuss, etc.).

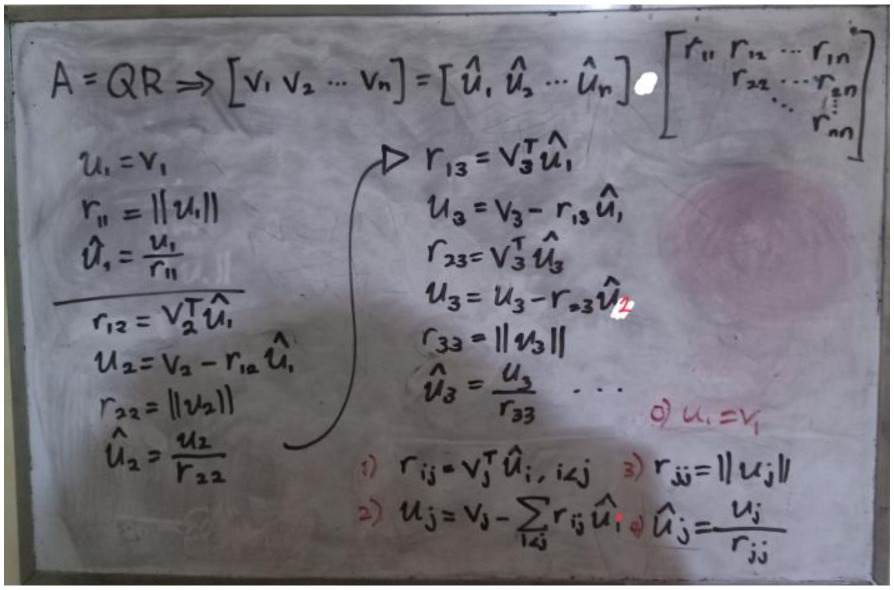
I think it is best to discuss the code at item2.m starting from the most elementary subroutine, in this case is QR Factorization.

My code for QR factorization is as follows:

```
%% QR Factorization equiv. to GS Orthonormalization
% Produces the QR factorization of A
% A : symmetric and non-defective
% Q : contains orthonormalized columns of A
% R : upper triangular matrix relative to the Gram-Schmidt projections.
function [Q,R] = QRFact(A)
    n = size(A,1);  % A is square, so size(A,2) would also do.
    Q = A;          % Initialize Q as A = [v1, v2, ..., vn].
                    % Eventually, Q = [uhat1, uhat2, ..., uhatn].
    R = zeros(n);   % Initialize R as 0 nxn matrix.
                    % This is what we "factor out" of A
    % Base Case
    % Implied u_1 = v_1
    R(1,1) = norm(Q(:,1));          % r_11 = norm(u_1)
    if R(1,1) >= 1E-6
        Q(:,1) = Q(:,1)/R(1,1);     % uhat_1 = u_1/r_11
    end

    % Iterative Case
    % Note that the column vectors of Q undergoes the following transform:
    % Q: v_j → u_j → uhat_j
    for j = 2:n
        % Solve r_ij entries and u_j = v_j - summ(rij * uhat_i), i<j
        % r_ij are the entries of R above the diagonal
        % u_j, by reusing Q(:,j), accumulates the effects of subtracting
        % R(i,j)*Q(:,i), i<j, repeatedly
        % This emulates summ(rij * uhat_i), i<j.
        for i = 1:j-1
            R(i,j) = Q(:,j)'*Q(:,i);            % r_ij = v_j' * uhat_i, i<j
            Q(:,j) = Q(:,j)-(R(i,j)*Q(:,i));    % u_j = v_j - r_ij*uhat_i
                                                % At i = 1, u_j = v_j
        end
        % Finally, get r_jj and uhat_j
        % Get diagonals of R, giving r_jj
        R(j,j) = norm(Q(:,j));      % r_jj = norm(u_j)
        % Normalize jth vector of Q, giving uhat_j
        Q(:,j) = Q(:,j)/R(j,j);     % uhat_j = u_j / r_jj
    end
end
% Note: This subroutine, which always goes through the entire
% matrix Q and is often called by QRiter, is what slows down the code.
```

Again, I have made annotations to the code which I hope would help make it more digestible for you and my future self. Seriously, comments are good practice!

Essentially, what happens in QR Factorization is based on the illustration below which I made and which I hope was the way this topic was thought to us.



$A = QR \Rightarrow [v_1\ v_2 \cdots v_n] = [\hat{u}_1\ \hat{u}_2 \cdots \hat{u}_n] \cdot \begin{bmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ & r_{22} & \cdots & r_{2n} \\ & & \ddots & \vdots \\ & & & r_{nn} \end{bmatrix}$

$u_1 = v_1$

$r_{11} = \|u_1\|$

$\hat{u}_1 = \frac{u_1}{r_{11}}$

$r_{12} = v_2^T \hat{u}_1$

$u_2 = v_2 - r_{12}\hat{u}_1$

$r_{22} = \|u_2\|$

$\hat{u}_2 = \frac{u_2}{r_{22}}$

$r_{13} = v_3^T \hat{u}_1$

$u_3 = v_3 - r_{13}\hat{u}_1$

$r_{23} = v_3^T \hat{u}_3$

$u_3 = u_3 - r_{23}\hat{u}_2$

$r_{33} = \|u_3\|$

$\hat{u}_3 = \frac{u_3}{r_{33}}$ . . .

0) $u_1 = v_1$

1) $r_{ij} = v_j^T \hat{u}_i,\ i<j$ 3) $r_{jj} = \|u_j\|$

2) $u_j = v_j - \sum_{i<j} r_{ij}\hat{u}_i$ 4) $\hat{u}_j = \frac{u_j}{r_{jj}}$

A concrete application is shown below:

That is, given $A = QR \Longrightarrow [v_1\ v_2 \ldots v_n] = [\hat{u}_1\ \hat{u}_2 \ldots \hat{u}_n] \begin{bmatrix} r_{11} & r_{12} & \ldots & r_{1n} \\ 0 & r_{22} & \ldots & r_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & r_{nn} \end{bmatrix}$,

the key ideas are

- The base case, in which we assume that $u_1 = v_1$ and $r_{11} = \|u_1\|$ and $\hat{u}_1 = \frac{u_1}{r_{11}}$.
- The iterative case, in which we get in sequence the following

$$r_{ij} = v_j^T \hat{u}_i, \qquad i < j$$

$$u_j = v_j - \sum_{i<j} r_{ij}\hat{u}_i$$

$$r_{jj} = \|u_j\|$$

$$\hat{u}_j = \frac{u_j}{r_{jj}}$$

- The transformation of the matrix Q in the implementation, wherein it goes from being initialized as $Q = A$, to containing intermediate values, to actually containing the orthonormalized columns of A as follows

$$[v_1\ v_2 \ldots v_n] \to [u_1\ u_2 \ldots u_n] \to [\hat{u}_1\ \hat{u}_2 \ldots \hat{u}_n]$$

The annotated code explains these equations in more detail.

```
%% QR Iteration
```

```
% Computes the eigenvalues of A stored in the diagonal matrix D
% corresponding to the eigenvectors of A eventually stored in Q.
% The motivation here is the Spectral Decomposition Theorem in which
% A = X*D*inv(X) = X*D*X' if A is symmetric (meaning inv(X)=X' because
% the eigenvectors of a symmteric matrix are orthogonal, notes p.18).
function [D,Q] = QRIter(A0)
    n = size(A0,1);
    Ak = A0;        % Note that original A serves as our initial matrix
    Q = eye(n);     % Prepare for accumulating products of Qi
    while (1)
        [Qk,Rk] = QRFact(Ak);
        A = Rk*Qk;  % Equiv. to A = Qk'*Ak*Qk (similarity transform.)
        if norm(diag(A)-diag(Ak)) < 0.1     % use large tol to speed-up
            break;
        end
        Ak = A;
        Q = Q*Qk;   % Q will store the eigenvectors as its columns
                    % Cumulative Products(Qi) from i = 1 to k
    end
    D = A;          % If A is symmteric, Ak converges to diagonals D of A
end
```

Essentially, we have the base case that $A_0 = Q_1R_1$ (applying the subroutine QRFact(A_0)). Then, $A_1 = R_1Q_1$. We check if the norm of $A_1 - A_0$ is already within our tolerance, which we made to be a large tolerance to increase the speed of convergence, and if not, we continue. $A_1 = Q_2R_2$, then $A_2 = R_2Q_2$. Eventually, $A_{k-1} = Q_kR_k$, then $A_k = R_kQ_k$, and we converge as $k \to \infty$. And the reason this is still based on spectral decomposition is because from the previous sentence, $R_k = Q_k^{-1}A_{k-1}$, and since Q_k contains orthogonal eigenvectors in the making, $Q_k^{-1} = Q_k^T$. Therefore, $A_k = Q_k^TA_{k-1}Q_k$. This is the so-called **similarity transformation** used to solve simultaneously for the eigenvalues D of A (or eigenvalues D_V of A^TA) in our case.

Moreover, we also notice that Q multiplicatively accumulates factors of Q_k, because knowing $A_k = Q_k^TA_{k-1}Q_k$, applying the substitution iteratively, we get $D = A_k = Q_k^TQ_{k-1}^T \dots Q_1^TA_0Q_1Q_2 \dots Q_k = \bar{Q}^TA\bar{Q}$, noting that $A_0 = A$ (i.e. the initial A_k is the given by A). And we can transform this into:

$$A = \bar{Q}D\bar{Q}^T$$

And comparing this with the Spectral Decomposition Theorem form of $A = XDX^T$, we see that

$$X = \prod_{i=1}^{k} Q_i$$

gives the eigenvectors corresponding to the eigenvalues of A. That's why Q behaves as such.

More details are given in the code annotations.

Finally, we discuss how we obtain the SVD with the help of QR iteration and its subroutine QR Factorization. The code for this is given as:

```
%% SVD using QR Iteration (and QR Factorization)
% Every mxn matrix A factors into A = USV'
% U : orthogonal mxm matrix
% S : diagonal mxn matrix
% V : orthogonal nxn matrix
% Wherein orthogonal implies lin. ind. columns (or eigenvectors).
% U and V can be derived from the following equations:
% AA' = U*D_U*U' and A'A = V*D_V*V', where D_U = D_V = SS' = S'S
% Theorem: AA' and A'A are square, symmetric, and non-defective,
% allowing spectral decomposition via QRIter to be applied to either.
function [U,S,V,sigk] = svdQR(A)
    [m,n] = size(A);

    % First, we obtain sigk from the smaller of m and n
    if m <= n
        sigk = m;
    else
        sigk = n;
    end

    % We then apply QR Iteration to either AA' or A'A
    % which would make them converge to D_U or D_V respectively
    % (essentially the same set of eigenvalues).
    % Note that QR Factorization is a subroutine of QR Iteration, and it
    % computes for the transformation matrices Q and R to be used.
    % Moreover, note that as either AA' or A'A converges to the
    % eigenvalues D, Q also converges to the eigenvectors corr. to
    % those eigenvalues, so we also return it from QRIter.

    % Here, we choose to solve for D_V and V first,
    % and then we compute for U using the equation AV = US
    % (We already know A, right eigenvectors V, and eigenvalues S
    % by that point).
    [D_V, V] = QRIter(A'*A);
    % D_V and V are nxn

    % Compute S which will be mxn.
    % Simultaneously compute inv(S) for later use.
    % S := principal square roots of D_V
    % S must be mxn
    S = zeros(m,n);
    invS = zeros(m,n);
    temp = sqrt(diag(D_V));
    for i = 1:sigk
        S(i,i) = temp(i);       % sqrt of diagonals of D_V
        invS(i,i) = 1/temp(i);  % reciprocal of those diagonals
    end
    % NOTE: Rectangular matrices, like S, don't have inverses in general.
    % What they do have are right and left inverses. Here we will be
    % using that notion instead, wherein invS' is both the left and
    % right inverse of S, hence we can treat it as a "proper" inverse here.

    % Use now known A, V, and inv(S) to compute U in U=A*V*inv(S)'
    U = A*V*invS';
end
```

First, as explained in a theorem in section (a) of this item, $sigk$ is simply the smaller of m and n. We first take that for later use.

Then, because AA^T and A^TA yields the same set of positive eigenvalues D_U and D_V when QR iteration is applied (with possibly different ordering), we can solve for either. Note that solving for D_U also gives us U and solving for D_V also gives us V which is a fortunate byproduct of QR iteration. Hence, if we solve for either of the two (in our case D_V and V), we can use the obtained information to solve for the remaining unknowns: Σ, and in our case, U, via the notion that Σ simply contains the principal square roots of the eigenvalues of D_U and D_V, and the equation $A = U\Sigma V^T \Longrightarrow AV = U\Sigma$, respectively.

Note here that we choose to solve for V (and D_V) because it is much easier to solve for U in $AV = U\Sigma$ than for V. That is, $U = AV\Sigma^{-1}$ wherein the inverse of the diagonal matrix Σ are obtained by replacing its diagonal entries with their reciprocals, compared to $V = A^{-1}U\Sigma$ wherein the inverse of non-diagonal A are much harder to ascertain, requiring Gaussian solvers to solve for U.

Caveat on Σ^{-1}! Rectangular matrices, like Σ, don't have "inverses" in general. What they do have are right and left inverses[2]. We will be using that notion instead, wherein $\Sigma^{-1} = (1./\Sigma)^T$ is both the left and right inverse of Σ, hence we can treat it as a "proper" bidirectional inverse of Σ here.

We verified this by taking the following norms which we found to result in 1:

$$norm(\Sigma\Sigma^{-1}) = norm(\Sigma^{-1}\Sigma) = 1$$

As we solve for Σ in the implementation using information from D_V, we also solve for its "inverse" for later use.

Finally, we solve for U in $U = AV\Sigma^{-1}$.

Note that we didn't have to solve for D_U because the information contained in it is already contained in D_V.

c. **Results. Show some snapshots of solution process and plots appropriate in the discussion of the results. Problems requiring numerical solution must be implemented using non-built-in methods or libraries.**

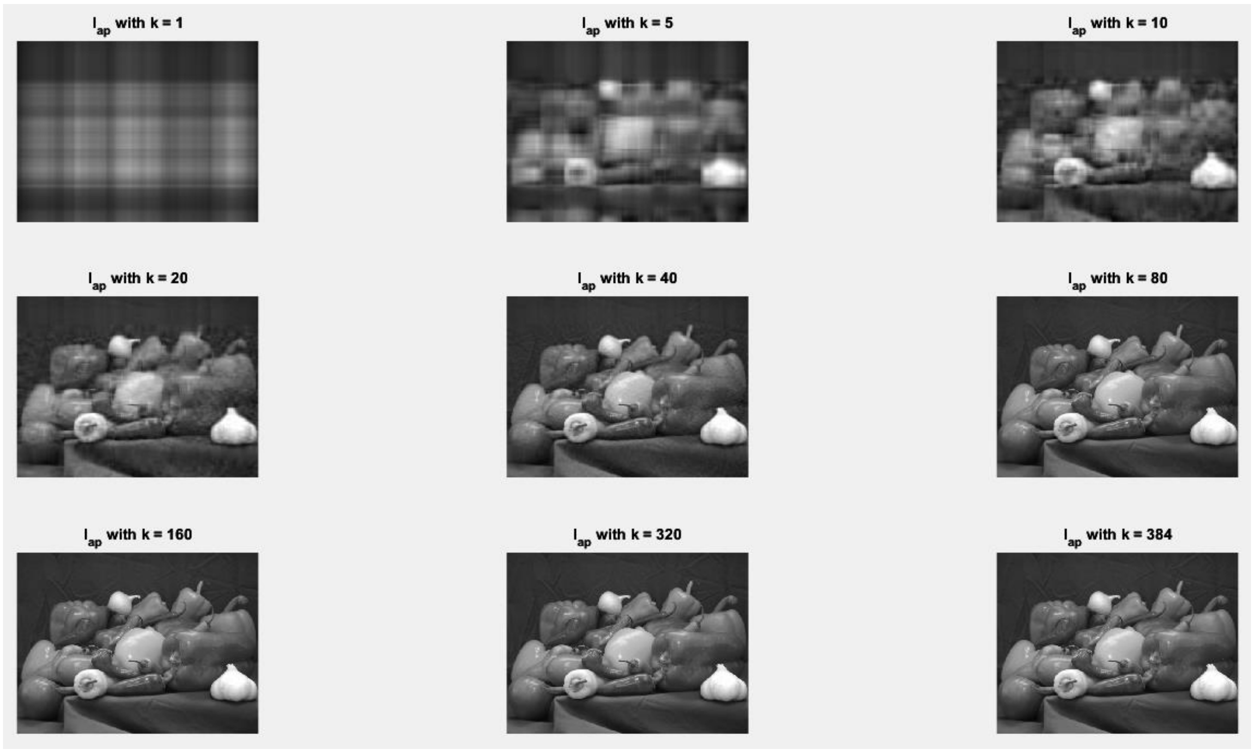
For testing my implementation we use the following driver code:

```
% item2.m [Hebron, Yenzy]

% imread produces mxnx3 array (3 for RGB)
% im2gray removes third dimension
I = im2gray(imread('peppers.png')); % 'peppers.png' is a built-in
                                    % sample file, no need to submit
I = im2double(I);
% Compute singular value decomposition of I = U*S*V'
% sigk is the number of significant eigenvalues
[U,S,V,sigk] = svdQR(I);
I_ap = U*S*V';

% Verify correctness of computed SVD
if norm(I - I_ap, 'inf') < 1E-6
    fprintf("Success!\n");
else
```

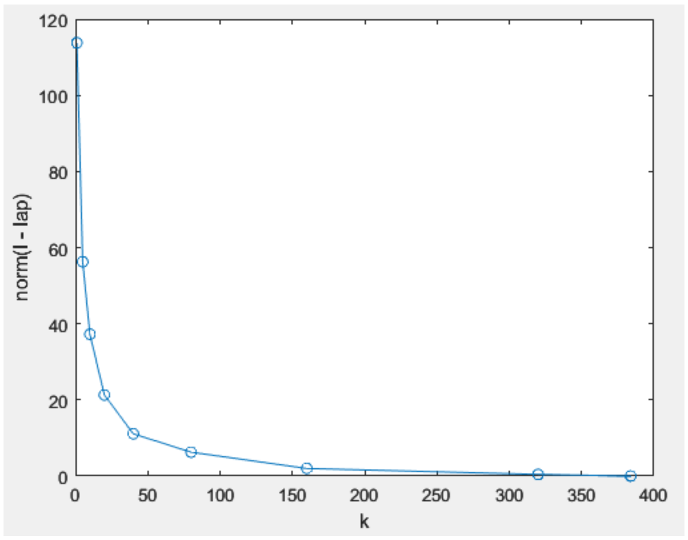
[2]https://people.math.carleton.ca/~kcheung/math/notes/MATH1107/wk06/06_left_and_right_inverses.html#:~:text=Inverse%20matrix&text=If%20MA%3DIn,a%20right%20inverse%20of%20A

```
    fprintf("Failed!!!\n")
end

% Generate comparisons of original image I and compressed image I_ap
% using increasing values of k until sigk. We expect the I_ap using k
% values closer to sigk to be of higher quality.
% For peppers.png, sigk is expected to be 384, we use this as our
% test case and we create the subplots around this expectation.
tcs = [1 5 10 20 40 80 160 320 384];
% tcs = 0:sigk/9:sigk;  % Uncomment for more automated testing.
tcs = round(tcs);
norms = zeros(9,1);
for i = 1:9
    k = tcs(i);
    I_ap = U(:,1:k)*S(1:k,1:k)*V(:,1:k)';
    figure(1);
    subplot(3,3,i); imshow(I_ap);
    title(sprintf('I_{ap} with k = %d', k));
    norms(i) = norm(I-I_ap, 'inf');
end
figure(2);
plot(tcs, norms, 'o-');
xlabel('k'); ylabel('norm(I - Iap)');
```

The driver code works on the built-in Matlab demo file named 'peppers.png'. The SVD of I corresponding to the grayscale data of 'peppers.png' is first computed, then the correctness is verified by checking $norm(I - I_{ap}, \infty)$, then a demo of the image compression is shown as follows (on the hardcoded test cases tcs for increasing values of k):



$\|I - I_{ap}\|$ is afterwards plotted for the same increasing values of k:



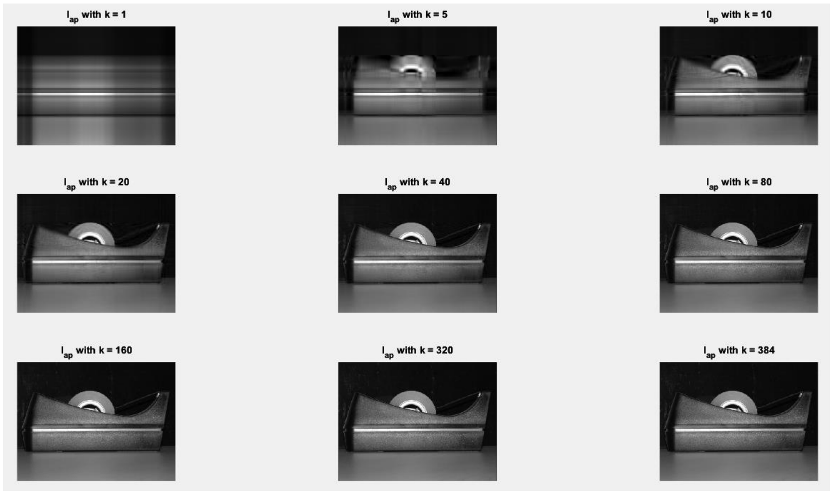
Showing the error decrease as k approaches $sigk = 384$ in the case of 'peppers.png'.

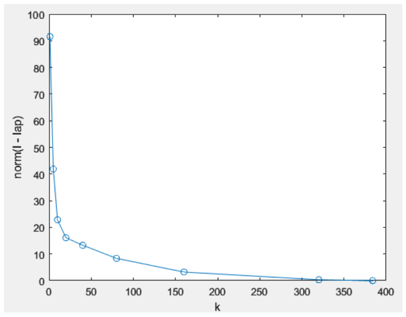
And to answer the last question in Item 2, paraphrasing from section (a):

Good compression is achieved for values of $k \leq sigk$ with k close enough to $sigk$ (i.e. the resulting data quality is still tolerable), with optimal compression (best data quality on smallest space consumption) at $k = sigk$. Above $sigk$, there might still be compression but there is no more improvement in quality as the highest quality is already attained at $k = sigk$. And as discussed earlier, $sigk$ is simply the smaller of m and n considering $A \in \mathbb{R}^{m,n}$ and the properties of AA^T and A^TA. [3] Recall Theorem 1 for that matter.

Other results:

- tape.png, $I \in \mathbb{R}^{384,512}$, same test cases as peppers.png since same size.

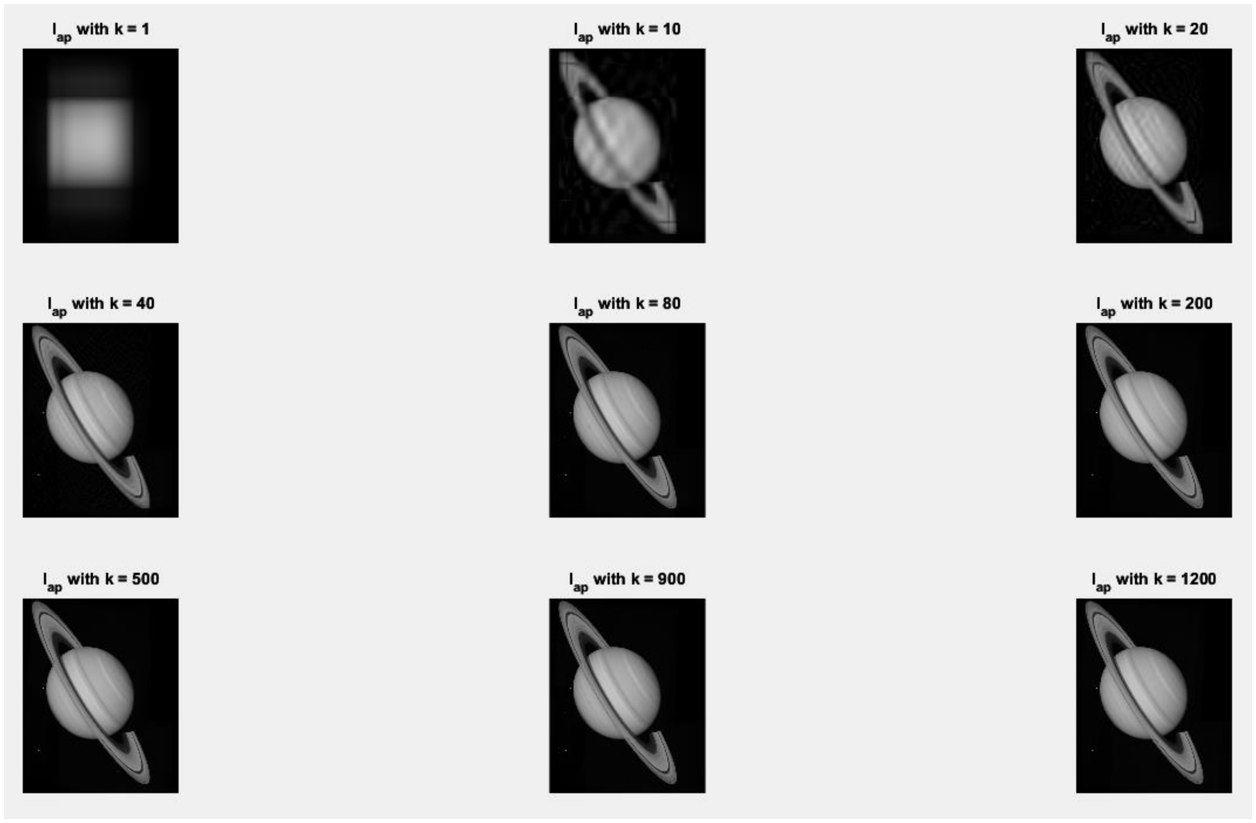


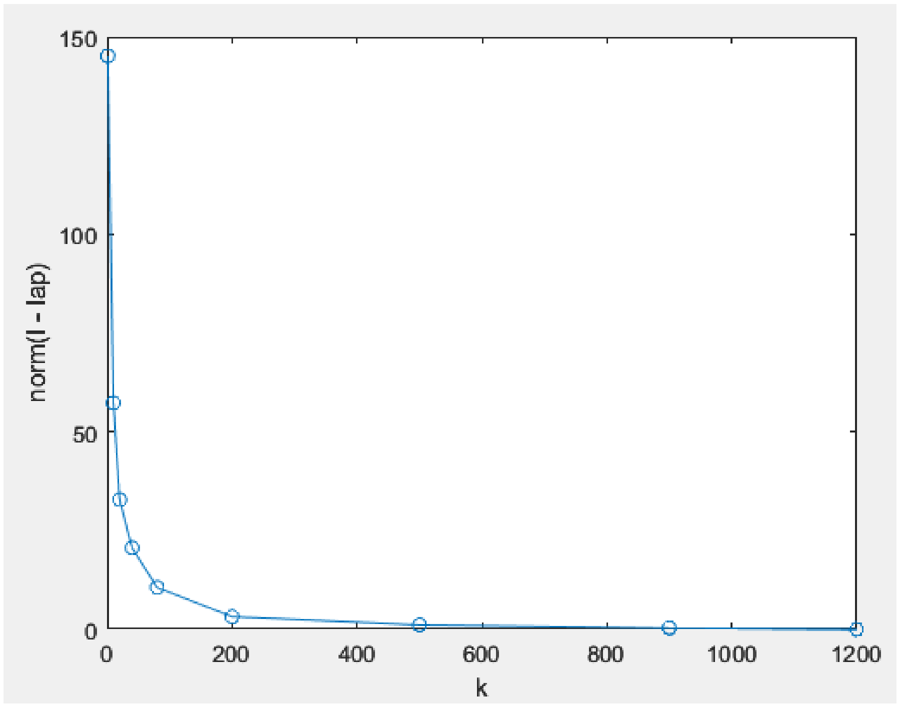


[3] https://math.stackexchange.com/questions/158219/is-a-matrix-multiplied-with-its-transpose-something-special

- saturn.png, $I \in \mathbb{R}^{1500,1200}$, test cases = $[1, 10, 20, 40, 80, 200, 500, 900, 1200]$

Very slow, takes about 5 minutes. Also, this is very important, please **change the correctness tolerance** to match the tolerance check in QRIter to avoid 'Failed!!!' printout when the result is actually successful.





3. The Airy Equation (item3.m)

a. Understanding of the problem.

My understanding of the problem is that we have to derive the power series solution of the Airy equation $y'' = xy$ in the form $y = c_1 y_1 + c_2 y_2$.

Then, taking $c_1, c_2 = 1$, we plot the derived Airy functions of the first kind $Ai(x)$ and the second kind $Bi(x)$ considering both the real domain with $x \in [-5,5]$ and the complex domain with $x = a + bi$ and $a, b \in [-5,5]$ (or $x \in [-5-5i, 5+5i]$) at intervals of 0.1.

- In the given real domain, we plot using 5, 10, and 20 terms of the series solution y.
- In the given complex domain, we plot the real and the imaginary parts of the functions' ranges separately using sufficient number of terms of the series solution y.
    - We note that in the complex domain, the resulting plot will be in three dimensions such that $Ai(a,b)$ and $Bi(a,b)$.
- NOTE: Using these domains and obtained ranges as they are would make it hard for us to see the interesting parts (which are mainly in the area close to the origin), hence we adjust the axis limits of the Matlab plots to get these interesting results more conveniently (i.e. without having to zoom in a LOT).

We then try to relate this in some way with radio waves and molecular vibrations, hence we can expect the Airy function plots to have an oscillatory or trigonometric behavior consistent with waves.

b. Implementation of the solution (analytical derivations/equation setup, paste code into the PDF and discuss, etc.).

Taking the series solution of the Airy equation:

$$y'' - xy = 0$$

$$y = \sum_{n=0}^{\infty} a_n x^n$$

$$y'' = \sum_{n=0}^{\infty} (n-1)n a_n x^{n-2} = \sum_{n=0}^{\infty} (n+1)(n+2) a_{n+2} x^n$$

$$\sum_{n=0}^{\infty} (n+1)(n+2) a_{n+2} x^n - \sum_{n=0}^{\infty} a_n x^{n+1} = 0$$

$$2a_2 + \sum_{n=1}^{\infty} (n+1)(n+2) a_{n+2} x^n - \sum_{n=0}^{\infty} a_n x^{n+1} = 0$$

$$2a_2 + \sum_{n=1}^{\infty} (n+1)(n+2) a_{n+2} x^n - \sum_{n=1}^{\infty} a_{n-1} x^n = 0$$

$$2a_2 + \sum_{n=1}^{\infty} [(n+1)(n+2) a_{n+2} - a_{n-1}] x^n = 0$$

The LHS of the above equation only equals 0 iff $a_2 = 0$ and $(n+1)(n+2)a_{n+2} - a_{n-1} = 0$ assuming that the power series solution is non-trivial. This gives us the following recurrence relation:

$$a_{n+2} = \frac{a_{n-1}}{(n+1)(n+2)}, \qquad n \in \mathbb{N}, \text{and } a_0, a_1 \in \mathbb{R}$$

$n = 1$: $a_3 = \frac{a_0}{2\cdot 3}$

$n = 2$: $a_4 = \frac{a_1}{3\cdot 4}$

$n = 3$: $a_5 = \frac{a_2}{4\cdot 5} = 0$

$n = 4$: $a_6 = \frac{a_3}{5\cdot 6} = \frac{a_0}{2\cdot 3\cdot 5\cdot 6}$

$n = 5$: $a_7 = \frac{a_4}{6\cdot 7} = \frac{a_1}{3\cdot 4\cdot 6\cdot 7}$

$n = 6$: $a_8 = \frac{a_5}{7\cdot 9} = 0$

$n = 7$: $a_9 = \frac{a_6}{8\cdot 9} = \frac{a_0}{2\cdot 3\cdot 5\cdot 6\cdot 8\cdot 9}$

$n = 8$: $a_{10} = \frac{a_7}{9\cdot 10} = \frac{a_1}{3\cdot 4\cdot 6\cdot 7\cdot 9\cdot 10}$

$n = 9$: $a_{11} = \frac{a_8}{10\cdot 11} = 0$

Letting $a_0 = c_1$ and $a_1 = c_2$, the series solution $y = c_1y_1 + c_2y_2$ is given by

$$y = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7 + a_8x^8 + a_9x^9 + a_{10}x^{10} + a_{11}x^{11} + \cdots$$

$$y = a_0 + a_1x + 0 + \frac{a_0}{2\cdot 3}x^3 + \frac{a_1}{3\cdot 4}x^4 + 0 + \frac{a_0}{2\cdot 3\cdot 5\cdot 6}x^6 + \frac{a_1}{3\cdot 4\cdot 6\cdot 7}x^7 + 0 + \frac{a_0}{2\cdot 3\cdot 5\cdot 6\cdot 8\cdot 9}x^9 + \frac{a_1}{3\cdot 4\cdot 6\cdot 7\cdot 9\cdot 10}x^{10} + 0 + \cdots$$

$$y = a_0\left(1 + \frac{1}{2\cdot 3}x^3 + \frac{1}{2\cdot 3\cdot 5\cdot 6}x^6 + \frac{1}{2\cdot 3\cdot 5\cdot 6\cdot 8\cdot 9}x^9 + \cdots\right) + a_1\left(x + \frac{1}{3\cdot 4}x^4 + \frac{1}{3\cdot 4\cdot 6\cdot 7}x^7 + \frac{1}{3\cdot 4\cdot 6\cdot 7\cdot 9\cdot 10}x^{10} + \cdots\right)$$

Recognizing the pattern, we write y in the following more explicit form:

$$y = a_0\left(1 + \sum_{n=1}^{\infty}\frac{x^{3n}}{\prod_{k=0}^{n-1}(3k+2)(3k+3)}\right) + a_1\left(x + \sum_{n=1}^{\infty}\frac{x^{3n+1}}{\prod_{k=0}^{n-1}(3k+3)(3k+4)}\right)$$

And taking $a_0, a_1 = 1$, then

$$y_1 = 1 + \sum_{n=1}^{\infty}\frac{x^{3n}}{\prod_{k=0}^{n-1}(3k+2)(3k+3)}$$

$$y_2 = x + \sum_{n=1}^{\infty}\frac{x^{3n+1}}{\prod_{k=0}^{n-1}(3k+3)(3k+4)}$$

Which we use with $\alpha \approx 1.35412$ and $\beta \approx 2.67894$ on

$$Ai(x) = \frac{y_1}{3^{2/3}\alpha} - \frac{y_2}{3^{1/3}\beta}$$

$$Bi(x) = \frac{y_1}{3^{1/6}\alpha} + \frac{y_2}{3^{-1/6}\beta}$$

to get the Airy functions of the first kind and the second kind.

Important to note that the zero terms of the series solution are still terms. Do not forget to count them in when considering, say, N terms of the series.

c. **Results. Show some snapshots of solution process and plots appropriate in the discussion of the results. Problems requiring numerical solution must be implemented using non-built-in methods or libraries.**

Using the analytical expressions obtained from section (b) for the power series solution of y, we write the following pieces of code deriving the required results for this item.

Note that we require the Symbolic Math Toolbox add-on installed for the Matlab implementation. This allows us to use symbolic computation with the functions symsum() and symprod() in order to solve our power series solution y whose more explicit form contains the Σ and Π operators. Although slow, they're more convenient than having to explicitly hardcode all the terms we are considering from y, we just have to pay the time-cost with patience (lots of it).

Nonetheless, we use a more numerical approach in plotting the functions on the complex domain (for loops) and they're tremendously faster than what I did for the plotting the functions on the real domain.

Now the code is given as follows:

```
% item3.m [Hebron, Yenzy]
syms n k
alpha = gamma(2/3); beta = gamma(1/3);

%% Part 1: Plotting on the real domain.
% We later adjust axis limits to make visible the interesting parts
x = -15:0.1:5;      % Note: Use linspace for other intervals.

% 5 term y (also considering 0 terms of y)
y1 = @(x) 1+symsum( x.^(3*n)/symprod( (3*k+2)*(3*k+3),k,0,n-1 ),n,1,1 );
y2 = @(x) x+symsum( x.^(3*n+1)/symprod( (3*k+3)*(3*k+4),k,0,n-1 ),n,1,1 );
Ai = @(x) y1(x)./(3^(2/3)*alpha) - y2(x)./(3^(1/3)*beta);
Bi = @(x) y1(x)./(3^(1/6)*alpha) + y2(x)./(3^(-1/6)*beta);
figure(1);
subplot(1,3,1);
plot(x,double(Ai(x)),'-r');
axis([-10 5 -10 10]);
title("Ai(x) with 5 terms of y")
xlabel('x'); ylabel('Ai(x)');
figure(2);
subplot(1,3,1);
```

```
plot(x,double(Bi(x)),'-b');
axis([-10 5 -10 10]);
title("Bi(x) with 5 terms of y")
xlabel('x'); ylabel('Bi(x)');

% 10 term y (also considering 0 terms of y)
y1 = @(x) 1+symsum( x.^(3*n)/symprod( (3*k+2)*(3*k+3),k,0,n-1 ),n,1,3 );
y2 = @(x) x+symsum( x.^(3*n+1)/symprod( (3*k+3)*(3*k+4),k,0,n-1 ),n,1,2 );
Ai = @(x) y1(x)./(3^(2/3)*alpha) - y2(x)./(3^(1/3)*beta);
Bi = @(x) y1(x)./(3^(1/6)*alpha) + y2(x)./(3^(-1/6)*beta);
figure(1);
subplot(1,3,2);
plot(x,double(Ai(x)),'-r');
axis([-10 5 -10 10]);
title("Ai(x) with 10 terms of y")
xlabel('x'); ylabel('Ai(x)');
figure(2);
subplot(1,3,2);
plot(x,double(Bi(x)),'-b');
axis([-10 5 -10 10]);
title("Bi(x) with 10 terms of y")
xlabel('x'); ylabel('Bi(x)');

% 20 term y (also considering 0 terms of y)
y1 = @(x) 1+symsum( x.^(3*n)/symprod( (3*k+2)*(3*k+3),k,0,n-1 ),n,1,6 );
y2 = @(x) x+symsum( x.^(3*n+1)/symprod( (3*k+3)*(3*k+4),k,0,n-1 ),n,1,6 );
Ai = @(x) y1(x)./(3^(2/3)*alpha) - y2(x)./(3^(1/3)*beta);
Bi = @(x) y1(x)./(3^(1/6)*alpha) + y2(x)./(3^(-1/6)*beta);
figure(1);
subplot(1,3,3);
plot(x,double(Ai(x)),'-r');
axis([-10 5 -10 10]);
title("Ai(x) with 20 terms of y")
xlabel('x'); ylabel('Ai(x)');
figure(2);
subplot(1,3,3);
plot(x,double(Bi(x)),'-b');
axis([-10 5 -10 10]);
title("Bi(x) with 20 terms of y")
xlabel('x'); ylabel('Bi(x)');

% Supplement: Superimposing the Ai(x) and Bi(x) graphs
% using 50 terms of y (0 terms inclusive)
y1 = @(x) 1+symsum( x.^(3*n)/symprod( (3*k+2)*(3*k+3),k,0,n-1 ),n,1,16 );
y2 = @(x) x+symsum( x.^(3*n+1)/symprod( (3*k+3)*(3*k+4),k,0,n-1 ),n,1,15 );
Ai = @(x) y1(x)./(3^(2/3)*alpha) - y2(x)./(3^(1/3)*beta);
Bi = @(x) y1(x)./(3^(1/6)*alpha) + y2(x)./(3^(-1/6)*beta);
figure(3);
plot(x,double(Ai(x)),'-r'); hold on;
plot(x,double(Bi(x)),'-b'); hold off;
axis([-10 5 -10 10]);
title("Ai(x) and Bi(x) with 50 terms of y")

%% Part 2: Plotting on the complex domain.
% We later adjust axis limits to make visible the interesting parts
```

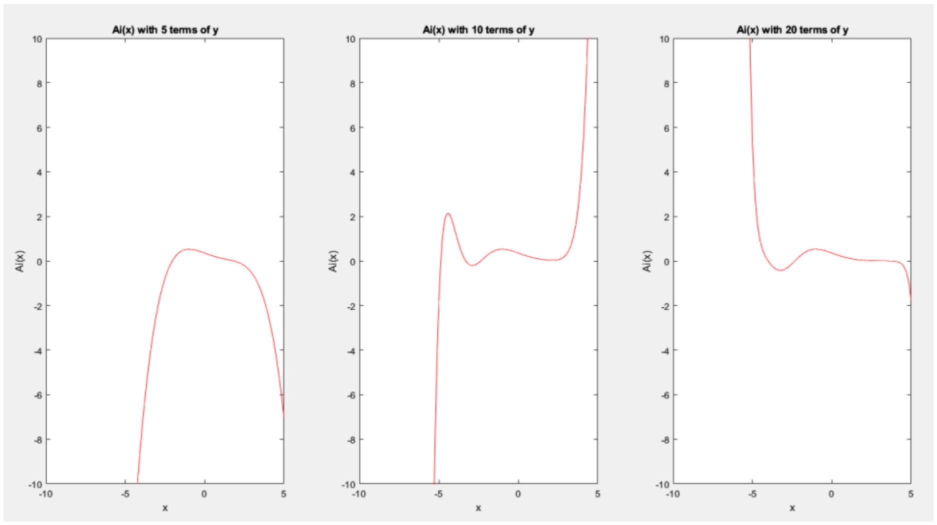
```
step = 0.1;
a = -5:step:5;
b = -5:step:5;
p = numel(a);
q = numel(b);
X = zeros(p,q);
for j = 1:p
    for k = 1:q
        X(j,k) = a(j) + b(k)*i;
    end
end

% Evaluate eigenfunctions on X
y1 = zeros(p,q);
y2 = zeros(p,q);
for k = 1:numel(X)
    y1(k) = 1;
    y2(k) = X(k);
    for n = 1:16
        temp1 = X(k)^(3*n);
        temp2 = X(k)^(3*n+1);
        for l = 0:n-1
            temp1 = temp1 / ((3*l+2)*(3*l+3));
            temp2 = temp2 / ((3*l+3)*(3*l+4));
        end
        y1(k) = y1(k) + temp1;
        y2(k) = y2(k) + temp2;
    end
end

% 20 term y
Ai = y1/(3^(2/3)*alpha) - y2/(3^(1/3)*beta);
Bi = y1/(3^(1/6)*alpha) + y2/(3^(-1/6)*beta);

figure(4);
subplot(2,2,1);
contour(a,b,real(Ai),-5:0.1:5);
title("Re(Ai(x)) with 51 terms of y")
xlabel('a'); ylabel('b');

subplot(2,2,2);
contour(a,b,imag(Ai),-5:0.1:5);
title("Im(Ai(x)) with 51 terms of y")
xlabel('a'); ylabel('b');

subplot(2,2,3);
contour(a,b,real(Bi),-5:0.1:5);
title("Re(Bi(x)) with 51 terms of y")
xlabel('a'); ylabel('b');

subplot(2,2,4);
contour(a,b,imag(Bi),-5:0.1:5);
title("Im(Bi(x)) with 51 terms of y")
xlabel('a'); ylabel('b');
```

Discussion

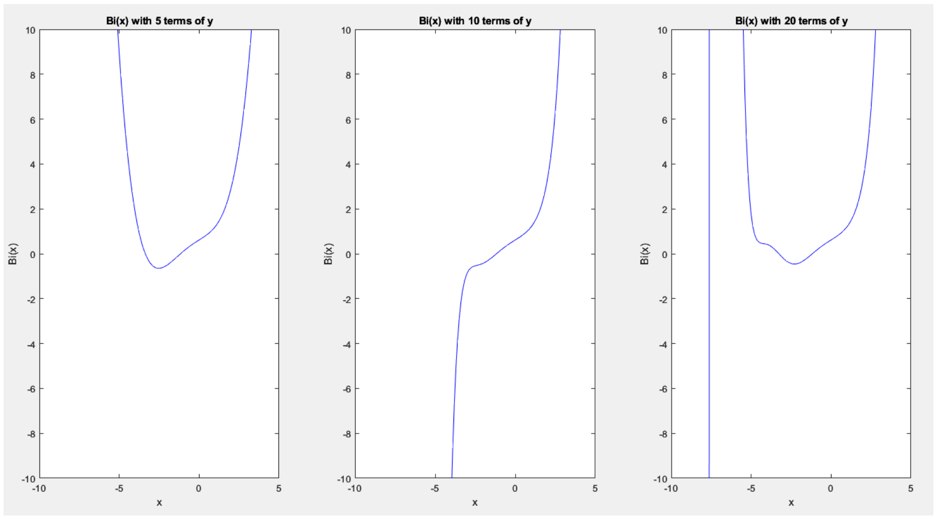
First, we have the assumptions on the constants $\alpha \approx 1.35412$ and $\beta \approx 2.67894$. We also declare n and k as symbols for use in our symbolic summation Σ and big product Π computations.

Part 1: Plotting on the real domain. In this part of the code, we plot the graphs of $Ai(x)$ and $Bi(x)$ on $x \in [-15,5]$ using 5, 10, and 20 terms of the power series solution of y. Note again that we consider here the terms of y that evaluate to 0 when counting the number of terms we use from y (i.e. the terms with a $a_2 = 0$ factor or those given by $a_{3n+2} = 0, n \in 0,1,2, \ldots$).

Because I wasn't able to find a pattern that gives the upper bound of the summations in y given the desired number of terms to be used from y, I resorted to just manually specifying said upper bounds for each of the 5, 10, and 20-term form of the power series solution.



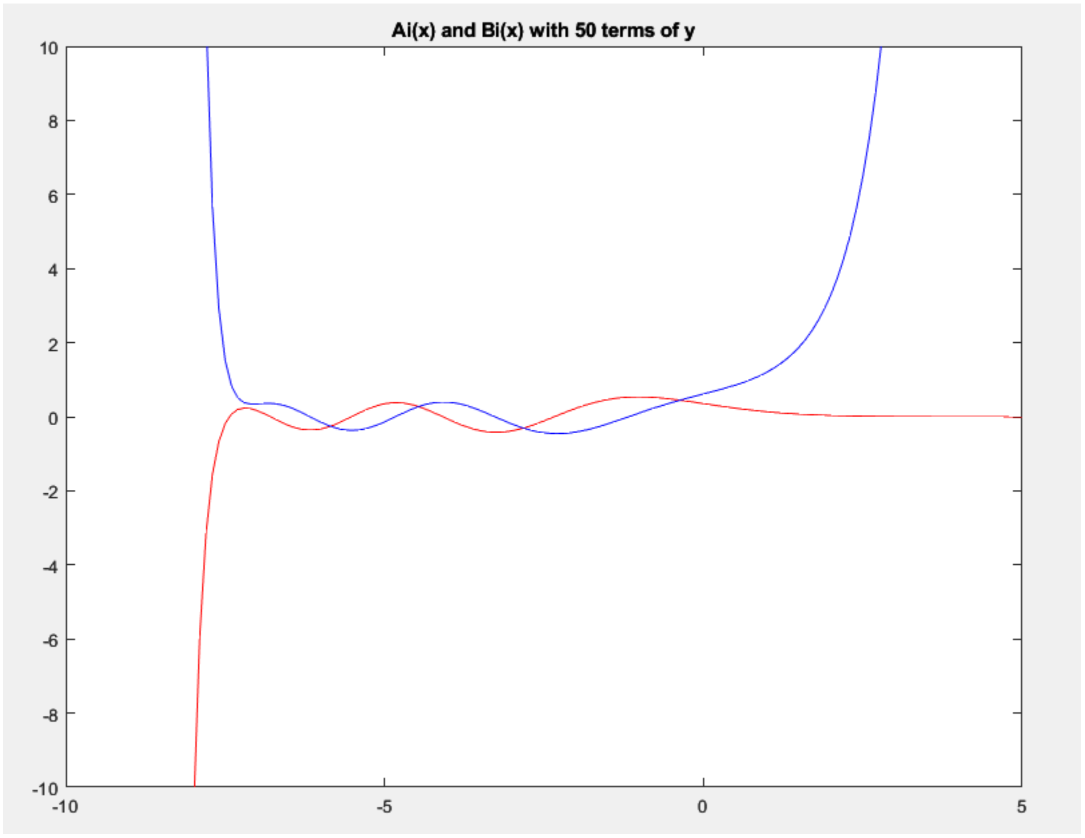
The subplots in Figure 1 (the ones with the red curves) correspond to the graphs of $Ai(x)$ on $x \in [-15,5]$ using 5, 10, and 20 terms respectively. From these subplots, we can deduce that $Ai(x)$ behaves in a way similar to trigonometric (when $x < 0$) and exponential growth functions (when $x \geq 0$).



The subplots in Figure 2 (the ones with the blue curves) correspond to the graphs of $Bi(x)$ on $x \in [-15,5]$ using 5, 10, and 20 terms respectively. From these subplots, we can deduce that $Bi(x)$ behaves in a way similar to trigonometric (when $x < 0$) and exponential decay functions (when $x \geq 0$).

In all subplots, we have adjusted the axis limits to make visible the more interesting parts of the graphs, which happen to be the parts close to the $\mathbb{R}^2$ origin. We also took care to convert symbol values to double values.

As supplement, to emulate the $Ai(x)$ and $Bi(x)$ plots in Edwards and Penney, I also plotted the functions using 50 terms from y (0 terms inclusive), as shown below ($Ai(x)$ is red, $Bi(x)$ is blue).

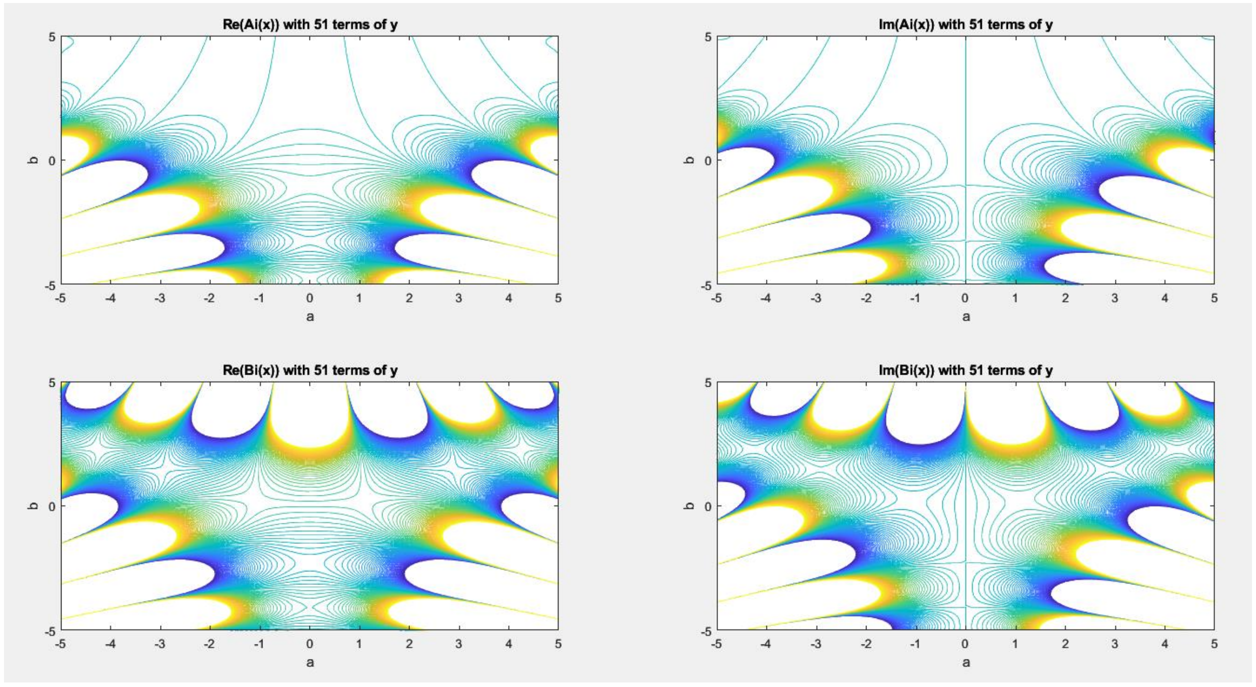


Part 2. Plotting on the complex domain. To plot the required graphs in the complex domain, we first create our desired complex interval defined by $x = a + bi$ such that $a, b \in [-5,5]$ with the required stepsize of 0.1. These x values will be stored in the matrix X.

We also deem 51 terms of y to be a "sufficient number of terms in the power series" (0 terms inclusive) as the real domain plots of $Ai(x)$ and $Bi(x)$ with 50 terms already show a discernible form corresponding to their discussed trigonometric and exponential behaviors. Hence, we set the upper bounds of the y_1 and y_2 summations to 16 respectively.

Afterwards, we evaluate $y_1(x)$ and $y_2(x)$ over $x \in X$ using for loops as stand-ins for the summations and product notations (temporary variables are very helpful here). We then plug the evaluated y_1 and y_2 into $Ai(x)$ and $Bi(x)$, which we then finally use to plot the contour plots of $\Re(Ai(x))$, $\Im(Ai(x))$, $\Re(Bi(x))$, and $\Im(Bi(x))$, taking care to filter their contents using the real() and imag() functions.

The contour plots now follow:



They're mesmerizing.

Attention: Please clear your Matlab workspace first before running item3.m especially after you have just run the saturn.png test case of item2.m to avoid issues caused by memory overlap.

4. Projectile motion with air resistance. (item4.m)

a. Understanding of the problem.

My understanding of the problem is that we should use the collocation method with various parameters (i.e. how many collocation points; uniformly-spaced points or Chebyshev points; etc.) to solve the given coupled SODE that models projectile motion with significant air resistance c and considering the other given constant parameters. We then compare the plots of these solutions. We are also tasked to plot this without drag, which I did with the understanding it's simply referring to a simulation where $c = 0$.

Finally, we compute the initial speed of the projectile and the angle at which it was launched using the approximated values we got from the collocation method (specifically the average of the points near the projectile's origin).

b. Implementation of the solution (analytical derivations/equation setup, paste code into the PDF and discuss, etc.).

We have the coupled SODEs

$$x(t)'' = -\frac{c}{m}\sqrt{x'^2 + y'^2}x'$$

$$y(t)'' = -g - \frac{c}{m}\sqrt{x'^2 + y'^2}y'$$

Over the interval $t_1 = 0$ to $t_n = 10$ with Dirichlet boundary conditions

$$x_1 = \alpha_x = 0, \qquad x_n = \beta_x = 8000, \qquad y_1 = \alpha_y = 0, \qquad y_n = \beta_y = 0.$$

We also have the following parameters: $m = 20, c = 0.00032, g = 9.80665$, and $\sigma = 1,2$.

We assume that x_i and $y_i$have solutions of the form

$$x_i(t_i) = \sum_{j=1}^{n} a_j \phi_j(t_i)$$

$$y_i(t_i) = \sum_{j=1}^{n} b_j \phi_j(t_i)$$

Where $\phi_j(t_i)$ is any basis function and $2(n-2)$ is the number of collocation points we desire to obtain (4 of the points are given by the 4 boundary conditions, while there are $n-2$ collocation points corresponding to x_i, and another $n-2$ collocation points corresponding to y_i). The unknowns here are the coefficients a_j and b_j, and we represent said coefficients using $\theta = [a_j \quad b_j]$ such that

$$\theta = \begin{bmatrix} a_1 & \cdots & a_j & \cdots & a_n \\ b_1 & \cdots & b_j & \cdots & b_n \end{bmatrix}^T,$$

$$x_i(t_i) = \sum_{j=1}^{n} \theta_{a_j}\phi_j(t_i), \qquad y_i(t_i) = \sum_{j=1}^{n} \theta_{b_j}\phi_j(t_i)$$

Note that $\theta \in \mathbb{R}_{n\times 2}$. Moreover, $\theta_{a_j} = \theta(:,1)$ accesses a_j entries, while $\theta_{b_j} = \theta(:,2)$ accesses b_j entries.

Our initial guess θ^0 is defined in the Matlab implementation as random values between -100 and 100 (using randi()) and this is valid because what we're trying to approximate are just real coefficients.

We take the Multiquadratic Function as our basis function $\phi_j(t_i)$, and we take note of the following:

$$\phi_j(t) = \sqrt{\left(t - t_j\right)^2 + \sigma^2}$$

$$\phi_j'(t) = \frac{t - t_j + \epsilon}{\phi_j(t)}, \qquad \epsilon = 10^{-6}$$

$$\phi_j''(t) = \frac{\sigma^2}{\phi_j^3(x)}$$

Note that we add a small number ϵ to the numerator of $\phi_j'(t)$ in order to avoid NaNs from arising during our computation (because at some point $t_i = t_j$ and we use $\phi_j'(t)$ as a factor in the denominator of many expressions in the Jacobian).

The 4 boundary points satisfy $\sum \theta_{a_j}\phi_j(t_1) - \alpha_x = 0$, $\sum \theta_{a_j}\phi_j(t_n) - \beta_x = 0$, $\sum \theta_{b_j}\phi_j(t_1) - \alpha_y = 0$, and $\sum \theta_{b_j}\phi_j(t_n) - \beta_y = 0$. Meanwhile, the $2(n-2)$ collocation points (i.e. "internal nodes", with $(n-2)$ for each of x_i and y_i respectively) satisfy the equations

$$x_i'' + \frac{c}{m}\sqrt{x_i'^2 + y_i'^2}\,x_i' = 0 \text{ and } y_i'' + g + \frac{c}{m}\sqrt{x_i'^2 + y_i'^2}\,y_i' = 0.$$

These considered, for the Collocation Method derived using Newton's Method, we have the following Newton Step

$$\theta^{(k+1)} = \theta^{(k)} - J_F^{-1}\left(\theta^{(k)}\right)F\left(\theta^{(k)}\right) = \theta^{(k)} - \delta$$

Wherein using the coupled representation consistent with $\theta = [a_j \quad b_j]$, we write

$$F = \begin{bmatrix} x_1 - \alpha_x & y_1 - \alpha_y \\ \vdots & \vdots \\ x_i'' + \dfrac{c}{m}\sqrt{x_i'^2 + y_i'^2}\,x_i' & y_i'' + g + \dfrac{c}{m}\sqrt{x_i'^2 + y_i'^2}\,y_i' \\ \vdots & \vdots \\ x_n - \beta_x & y_n - \beta_n \end{bmatrix}, \qquad \theta = \begin{bmatrix} a_1 & b_1 \\ \vdots & \vdots \\ a_j & b_j \\ \vdots & \vdots \\ a_n & b_n \end{bmatrix}$$

Noting that $F \in \mathbb{R}_{n\times 2}$. Because of the ubiquity of x_i, x_i', and x_i'', and y_i, y_i', and y_i'', we also write them below for our convenience (while turning them into functions in the Matlab implementation).

$$x_i(t_i) = \sum_{j=1}^{n} \theta_{a_j}\phi_j(t_i), \qquad x_i'(t_i) = \sum_{j=1}^{n} \theta_{a_j}\phi_j'(t_i), \qquad x_i''(t_i) = \sum_{j=1}^{n} \theta_{a_j}\phi_j''(t_i)$$

$$y_i(t_i) = \sum_{j=1}^{n} \theta_{b_j}\phi_j(t_i), \qquad y_i'(t_i) = \sum_{j=1}^{n} \theta_{b_j}\phi_j'(t_i), \qquad y_i''(t_i) = \sum_{j=1}^{n} \theta_{b_j}\phi_j''(t_i)$$

For the purposes of the Jacobian, we immediately solve

$$\begin{aligned}
\frac{\partial}{\partial a_j}\left(x_i'' + \frac{c}{m}\sqrt{x_i'^2 + y_i'^2}x_i'\right) &= \phi_j''(t_i) + \frac{c}{m}\frac{\partial}{\partial a_j}\left(\sqrt{x_i'^2 + y_i'^2}x_i'\right) \\
&= \phi_j''(t_i) + \frac{c}{m}\left(\frac{\partial}{\partial a_j}\left(\sqrt{x_i'^2 + y_i'^2}\right)x_i' + \sqrt{x_i'^2 + y_i'^2}\frac{\partial}{\partial a_j}x_i'\right) \\
&= \phi_j''(t_i) + \frac{c}{m}\left(\frac{\frac{d}{da_j}(x_i'^2)}{2\sqrt{x_i'^2 + y_i'^2}}x_i' + \sqrt{x_i'^2 + y_i'^2}\phi_j'(t_i)\right) \\
&= \phi_j''(t_i) + \frac{c}{m}\left(\frac{2x_i'\phi_j'(t_i)}{2\sqrt{x_i'^2 + y_i'^2}}x_i' + \sqrt{x_i'^2 + y_i'^2}\phi_j'(t_i)\right) \\
&= \phi_j''(t_i) + \frac{c}{m}\left(\sqrt{x_i'^2 + y_i'^2} + \frac{x_i'^2}{\sqrt{x_i'^2 + y_i'^2}}\right)\phi_j'(t_i)
\end{aligned}$$

Similarly, $\frac{\partial}{\partial b_j}\left(y_i'' + g + \frac{c}{m}\sqrt{x_i'^2 + y_i'^2}y_i'\right) = \phi_j''(t_i) + \frac{c}{m}\left(\sqrt{x_i'^2 + y_i'^2} + \frac{y_i'^2}{\sqrt{x_i'^2+y_i'^2}}\right)\phi_j'(t_i)$.

We write the Jacobian (two rows to better fit into the paper) as

$$J_F = \begin{bmatrix} \frac{\partial F_i}{\partial a_j} \\ \frac{\partial F_i}{\partial b_j} \end{bmatrix} = \begin{bmatrix} \phi_j(t_1) \\ \vdots \\ \frac{\partial}{\partial a_j}\left(x_i'' + \frac{c}{m}\sqrt{x_i'^2 + y_i'^2}x_i'\right) \\ \vdots \\ \phi_j(t_n) \\ \phi_j(t_1) \\ \vdots \\ \frac{\partial}{\partial b_j}\left(y_i'' + g + \frac{c}{m}\sqrt{x_i'^2 + y_i'^2}y_i'\right) \\ \vdots \\ \phi_j(t_n) \end{bmatrix} = \begin{bmatrix} \phi_j(t_1) \\ \vdots \\ \phi_j''(t_i) + \frac{c}{m}\left(\sqrt{x_i'^2 + y_i'^2} + \frac{x_i'^2}{\sqrt{x_i'^2 + y_i'^2}}\right)\phi_j'(t_i) \\ \vdots \\ \phi_j(t_n) \\ \phi_j(t_1) \\ \vdots \\ \phi_j''(t_i) + \frac{c}{m}\left(\sqrt{x_i'^2 + y_i'^2} + \frac{y_i'^2}{\sqrt{x_i'^2 + y_i'^2}}\right)\phi_j'(t_i) \\ \vdots \\ \phi_j(t_n) \end{bmatrix}$$

While noting that $J \in \mathbb{R}_{2n\times n}$. Our goal in Matlab is then to construct these $F \in \mathbb{R}_{n\times 2}$ and $J_F \in \mathbb{R}_{2n\times n}$ matrices for every kth iteration of $\theta \in \mathbb{R}_{n\times 2}$. And by solving

$$\delta = J_F^{-1}(\theta^{(k)})F(\theta^{(k)}) \Longrightarrow J_F(\theta^{(k)})\delta = F(\theta^{(k)})$$

for $\delta \in \mathbb{R}_{n\times 2}$ using a Gaussian solver (which we do separately for a_j and b_j such that $\delta = [\delta_x \quad \delta_y]$ to maintain dimensional consistency), we then apply the Newton step $\theta^{(k+1)} = \theta^{(k)} - \delta$ and then check if the difference in $\theta^{(k+1)}$ and $\theta^{(k)}$ is now within our tolerance. If so, we break the loop as we have found the coefficients of the collocation points now stored in θ. If not, we continue.

Using the obtained coefficients of the collocation points, we generate the values of x_i and y_i (on a finer interval z relative to t), effectively reconstructing the trajectory of the projectile.

With these established, we are now ready to implement our solution in Matlab. The code (including the driver code and plotter code) is as follows:

```
% item4.m [Hebron, Yenzy]
function cs138_pset4v9()
    global sigma;   % Try sigma = 1,2
    figure(1);
    sigma = 1;
    subplot(221);
    nonlinear_colloc(1,12);
    sigma = 2;
    subplot(222);
    nonlinear_colloc(1,12);
    sigma = 1;
    subplot(223);
    nonlinear_colloc(2,12);
    sigma = 2;
    subplot(224);
    nonlinear_colloc(2,12);
end

function nonlinear_colloc(pts, N)
    % Projectile Motion, coupled SODE
    % Boundary Conditions and Assumptions
    t1 = 0; tn = 10;
    alpha_x = 0; beta_x = 8000;
    alpha_y = 0; beta_y = 0;
    global sigma;
    m = 20; c = 0.00032;
    g = 9.80665;

    %N = 12; % N-2 Number of collocation points
    if pts == 1
        % Use Uniformly spaced collocation points
        t = linspace(t1,tn,N)'; % time steps
    elseif pts == 2
        % Use Chebyshev collocation points
        t = -cos((0:N-1)*pi/(N-1))';
        t = t1 + ((tn-t1)/2)*(t+1); % time steps
    end

    % Basis Function: Multiquadratic Basis
    phi = @(t,tj) sqrt((t-tj).^2 + sigma^2);
    phi_p = @(t,tj) ((t-tj)+1E-6)./phi(t,tj);  % modified with sigma
    phi_pp = @(t,tj) sigma^2./phi(t,tj).^3;

    % Auxiliary Functions for xi and yi and their derivatives
    xi = @(aj,t,tj) sum(aj.*phi(t,tj));
```

```
xi_p = @(aj,t,tj) sum(aj.*phi_p(t,tj));
xi_pp = @(aj,t,tj) sum(aj.*phi_pp(t,tj));
yi = @(bj,t,tj) sum(bj.*phi(t,tj));
yi_p = @(bj,t,tj) sum(bj.*phi_p(t,tj));
yi_pp = @(bj,t,tj) sum(bj.*phi_pp(t,tj));

% Initial Guess (random values would do bcz these are just coeffs)
% Note, theta(:,1) corr to aj, theta(:,2) corr to bj
theta_0 = randi([-100 100],N,2);

F = zeros(N,2);
J = zeros(N*2,N);

theta = theta_0;       % set theta to initial guess
while (1)
    % Construct current Collocation Points F
    % Boundary Nodes
    F(1,1) = xi(theta(:,1),t(1),t) - alpha_x;
    F(N,1) = xi(theta(:,1),t(N),t) - beta_x;
    F(1,2) = yi(theta(:,2),t(1),t) - alpha_y;
    F(N,2) = yi(theta(:,2),t(N),t) - beta_y;
    % Collocation Points
    for i = 2:N-1
        F(i,1) = xi_pp(theta(:,1),t(i),t) + ...
                (c/m)*sqrt(xi_p(theta(:,1),t(i),t)^2 + ...
                yi_p(theta(:,2),t(i),t)^2)* ...
                xi_p(theta(:,1),t(i),t);
        F(i,2) = yi_pp(theta(:,2),t(i),t) + g + ...
                (c/m)*sqrt(xi_p(theta(:,1),t(i),t)^2 + ...
                yi_p(theta(:,2),t(i),t)^2)* ...
                yi_p(theta(:,2),t(i),t);
    end

    % Construct current Jacobian J
    % Boundary Nodes
    J(1,:) = phi(t(1),t);
    J(N,:) = phi(t(N),t);
    J(N+1,:) = phi(t(1),t);
    J(N*2,:) = phi(t(N),t);
    % Internal Nodes
    for i = 2:N-1
        J(i,:) = phi_pp(t(i),t) + (c/m)* ...
                 (sqrt(xi_p(theta(:,1),t(i),t)^2 + ...
                  yi_p(theta(:,2),t(i),t)^2) + ...
                  xi_p(theta(:,1),t(i),t)^2 / ...
                  sqrt(xi_p(theta(:,1),t(i),t)^2 + ...
                  yi_p(theta(:,2),t(i),t)^2)).* ...
                  phi_p(t(i),t);
        J(i+N,:) = phi_pp(t(i),t) + (c/m)* ...
                 (sqrt(xi_p(theta(:,1),t(i),t)^2 + ...
                  yi_p(theta(:,2),t(i),t)^2) + ...
                  yi_p(theta(:,2),t(i),t)^2 / ...
                  sqrt(xi_p(theta(:,1),t(i),t)^2 + ...
                  yi_p(theta(:,2),t(i),t)^2)).* ...
                  phi_p(t(i),t);
        end

        % Solve Delta for x and y separately
%         Delx = J(1:N,:)\F(:,1);
%         Dely = J(N+1:end,:)\F(:,2);
        % We use GE from prev discussions, valid since Jacobian is square
        Delx = ge(J(1:N,:),F(:,1));
        Dely = ge(J(N+1:end,:),F(:,2));
        if norm(Delx, 'inf') < 1E-6 && norm(Dely, 'inf') < 1E-6
            break
        else
            theta(:,1) = theta(:,1) - Delx;
            theta(:,2) = theta(:,2) - Dely;
        end
    end

    % Coefficients of xi and yi now stored in theta_j (aj and bj resp)
    a = theta(:,1);
    b = theta(:,2);

    % Reconstructing the trajectory (numerical on top of analytical)
    z = (t1:(tn-t1)/50:tn)';    % Finer interval than t
    % Analytical solution
    x_ana = 800.*z;
    y_ana = (-1/2)*g.*z.^2 + 5*g.*z;
    plot(x_ana,y_ana,'g'); hold on; % green, superimposed
    % Numerical Solution
    x = a(1)*phi(z,t(1));
    y = b(1)*phi(z,t(1));
    for j = 2:N
        x = x+a(j)*phi(z,t(j));
        y = y+b(j)*phi(z,t(j));
    end
    if pts == 1
        plot(x,y,'r+');
        title(sprintf("%d Uniform Colloc Pts. with sigma = %d", N-2, sigma));
    elseif pts == 2
        plot(x,y,'b+');
        title(sprintf("%d Chebyshev Colloc Pts. with sigma = %d", N-2, sigma));
    end
    xlabel('x'); ylabel('y');

    % Limit x and y axis for easy comparison
    axis([0 8000 0 max(y)+10]); % use y = 0 to 8000 to make angle visible

    % Compute approx initial velocity and direction
    vx_0 = xi_p(theta(:,1),t(1),t)
    vy_0 = yi_p(theta(:,2),t(1),t)
    v_0 = sqrt(vx_0^2 + vy_0^2)         % combine velocity components
    angle = rad2deg(atan(y(2)/x(2)))    % compute angle
end

function x = ge(A,b)
    % Assumption: Matrix is square and has a unique solution.
    n = size(A, 1);
    A = [A b]; % Augmented Matrix
    % Elimination
    for j = 1:n
        % Find index of largest magnitude
        [~, k] = max(abs(A(j:end,j)));

        % Adjust k so it reflects row index for entire matrix
        k = (j-1)+k;    % ex. if j = 2 and k = 1, k is actually 2

        % ERO1 switch to get row with pivot into working row
        A([j k], j:end) = A([k j], j:end);

        % ERO2 scale to get a unit pivot
        A(j,j:end) = A(j,j:end)/A(j,j);
        % zero the entries under the unit pivot in column j
        for i = j+1:n
            if abs(A(i,j)) < 1E-10
                A(i,j) = 0;
                continue;
            end
            % ERO3 annihilate
            A(i,j:end) = A(i,j:end)-A(i,j)*A(j,j:end);
        end
    end
    % A %REF (A|b)

    %Backward Sub
    x = A(:, end); % x(n) is correct
    for i = n-1:-1:1
        x(i) = x(i)-A(i,i+1:end-1)*x(i+1:end);
        % ex. x3 = b3
        %     x2 = b2 - a3x3 = b2 - [a3]*[x3]'
        %     x1 = b1 - (a2x2 + a3x3) = b1 - [a2 a3]*[x2 x3]'
    end
end
```

Now, we derive the analytical solution for the projectile motion:

Without drag, there is no change in the horizontal velocity x' of the projectile. Hence,

$$x'' = 0$$

$$x' = A$$

$$x = At + B$$

Meanwhile, because there is still the gravitational acceleration,

$$y'' = -g$$

$$y' = -gt + C$$

$$y = -\frac{1}{2}gt^2 + Ct + D$$

Applying the initial conditions,

$$\begin{aligned} x_1 &= At_1 + B \\ B &= 0 \end{aligned}$$

$$\begin{aligned} x_n &= At_n + B \\ 8000 &= A10 + 0 \\ A &= 800 \end{aligned}$$

Hence,

$$x = 800t$$

Moreover,

$$\begin{aligned} &y_1 = -\frac{1}{2}gt_1^2 + Ct_1 + D \\ &D = 0 \end{aligned}$$

$$\begin{aligned} &y_n = -\frac{1}{2}gt_n^2 + Ct_n + D \\ &0 = -50g + C10 + 0 \\ &C = 5g \end{aligned}$$

Hence,

$$y = -\frac{1}{2}gt^2 + 5gt$$

In short, the analytical solution for the trajectory of the projectile is given by the coupled equations

$$x = 800t$$

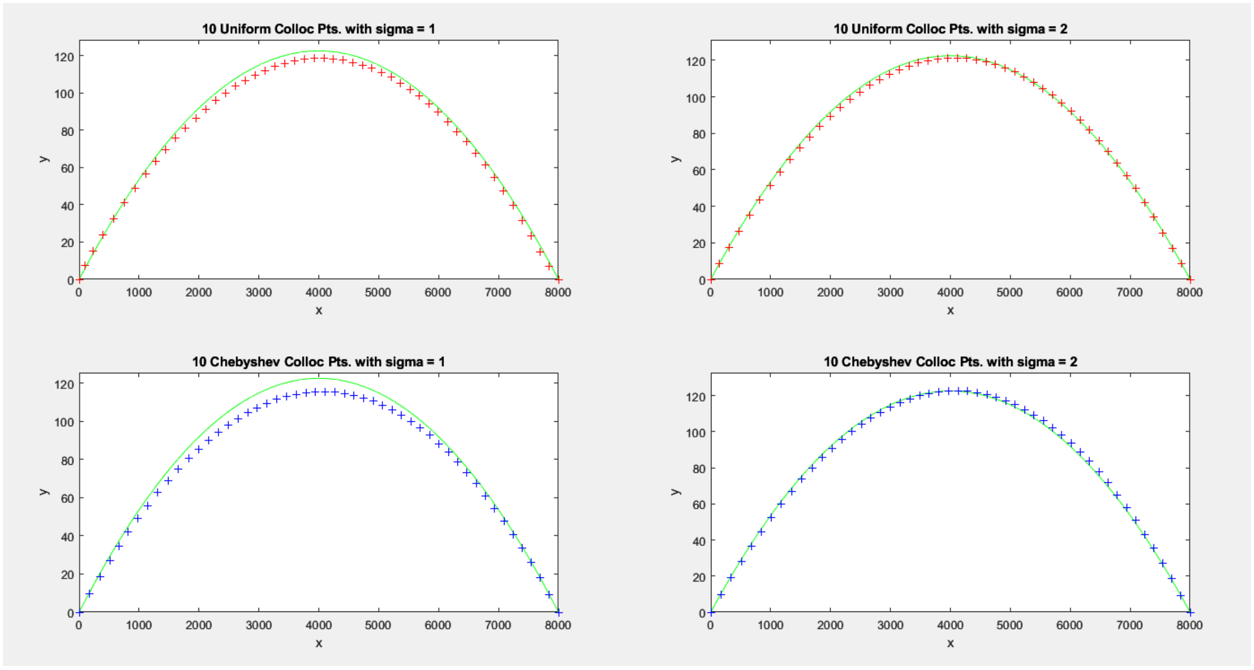
$$y = -\frac{1}{2}gt^2 + 5gt$$

Moreover, we can already see that the initial velocity for the no air resistance case is going to be $x'(0) = 800$, and we can expect the approximated initial velocity for the nontrivial air resistance case to be somewhat close to that (provided that the drag coefficient is small). We leave the discussion of the initial angle on the next part.

We then compare the plots of these analytical solutions with those we got using the Collocation method.

c. **Results. Show some snapshots of solution process and plots appropriate in the discussion of the results. Problems requiring numerical solution must be implemented using non-built-in methods or libraries.**

The required plots can be seen below. The numerical solutions (red points for uniform collocation points and blue points for Chebyshev collocation points) are superimposed over the analytical solution (green curve).

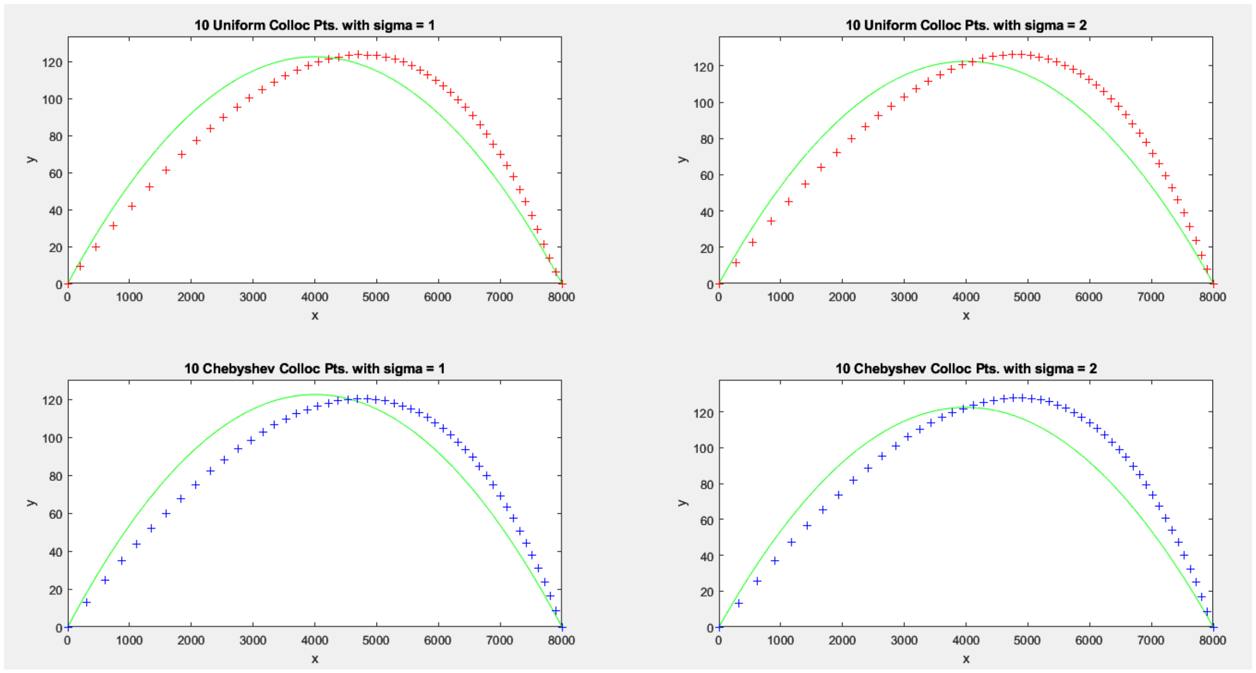


The most striking detail that can be observed from the plots is that higher σ values (which are used to modulate the radial basis functions ϕ) result in a closer approximation of the coupled linear SODEs case (without drag).

As an aside, I actually tried plotting with higher σ and the higher it gets, the less apparent the improvement in the approximation seems to be, that is, the solution stabilizes with higher σ. So if we are to reconstruct a dataset using the Collocation method, I think it is best to use higher σ. The caveat is that it appears the method stops converging after a certain σ threshold is breached for a particular number of collocation points. In my case, it appears that the highest σ for which my algorithm can obtain a solution for 15 Chebyshev collocation points is about $\sigma = 6$.

At a glance, it really is surprising that our Collocation method is winding up with a trajectory that approximates the case without air resistance (almost perfectly parabolic) when we're trying to approximate a case with non-trivial air resistance. However, this actually isn't really mysterious because the drag coefficient we used here, $c = 0.00032$, is actually very small, effectively translating our coupled nonlinear SODE case to a coupled linear SODE case.

So as supplement, and because I'm very curious, if we want to see the kind of trajectory where the projectile is being visibly affected by the air resistance (as discussed in the class) and where the outcome differs significantly from that of the negligible air resistance case, we have to set the drag coefficient c closer to 1. We show a sample plot where c is closer to 1 below (specifically $c = 0.0032$):



The most interesting thing in these new plots, I think, is that instead of what one would naively expect to be like a force uniformly pushing back against the projectile everywhere from every direction, because the drag expression $-\frac{c}{m}\sqrt{x'^2 + y'^2}x'$ is also dependent on time t, the air resistance (and its x and y components) also actually varies in time as demonstrated by the plots where it significantly slowed the projectile's ascent but offered less resistance on the horizontal component when it was just launched, and significantly sped up the projectile's descent and pushed back harder on the horizontal component as it was about to reach its destination. And if we are to decouple the drag expression from t, we could also expect a trajectory that is highly similar to the case without air resistance.

Another thing that one can notice is that the uniformly-spaced points and Chebyshev points approach vary most at the points surrounding the maximum height of the projectile and at the points near the end of the trajectory. I believe this is because the Chebyshev points tend to concentrate at these "extreme ends" with respect to time, and we can see this reflected in how they get interpolated. And again, if we think of the unit circle as a clock and get our t from there, we will notice that the Chebyshev points concentrates at the beginning $t = 0$, at the halfway point $t = \pi$ (near the t when projectile is at max. height) and when it lands $t = 2\pi$.

What are the initial speed of the projectile and the angle at which it was launched?

We first declare that we use the given m, c, and g parameters here.

Then for the initial speed, using 15 Chebyshev collocation points and $\sigma = 3$ for better accuracy (inferring from the previous discussion that higher σ may lead to a more accurate solution), we solve the horizontal and vertical components of the velocity:

$$v_{x0} = x_1'(0) = \sum_{j=1}^{n} \theta_{a_j}\phi_j'(0) = 853.4440, \qquad v_{y0} = y_1'(0) = \sum_{j=1}^{n} \theta_{b_j}\phi_j'(0) = 50.1474$$

Which we then take the magnitude of to get the approximate initial speed, as follows

$$v_0 \approx \sqrt{{x_1'}^2 + {y_1'}^2} = 854.9160$$

Giving us an approximate initial speed of $854.9160\ \mathrm{m/s}$ for the coupled SODE model.

We use the Chebyshev collocation points because as discussed earlier, they tend to better interpolate the extreme ends of a data set, and the initial point (the origin) can be seen as an extreme end.

Finally, recalling that

$$v_x = |v| \cos\theta\,, \qquad v_x = v_{x0} = 853.4440$$

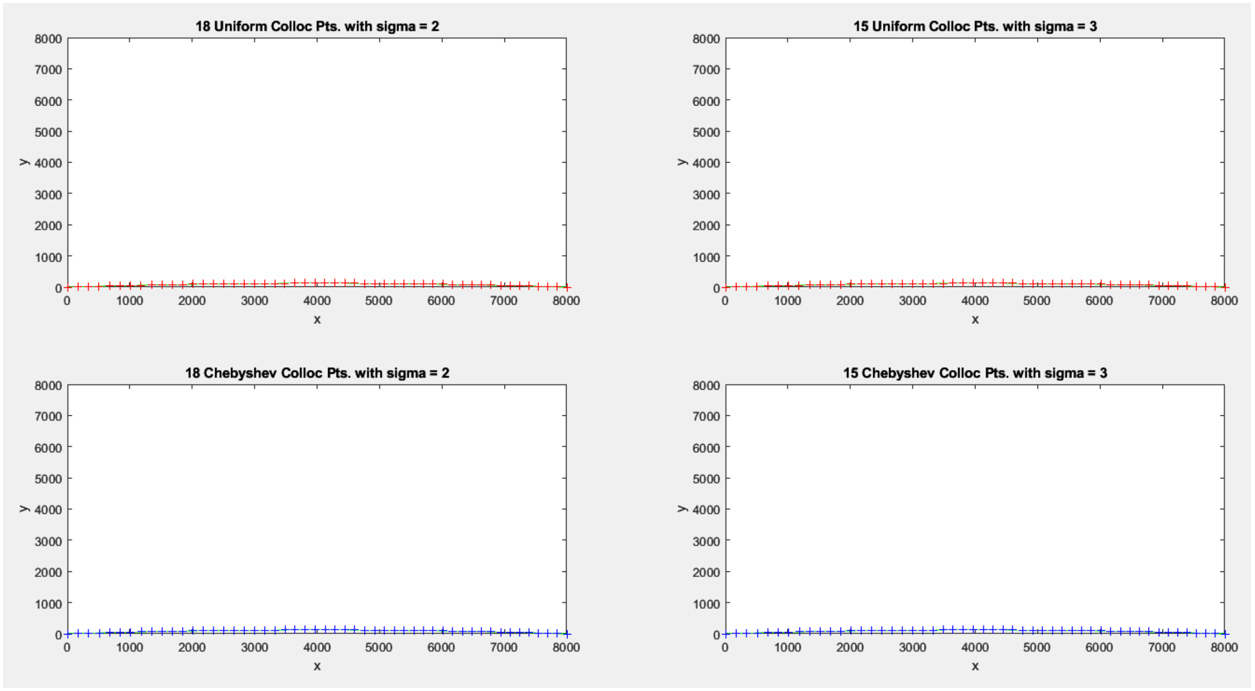
We solve for the initial angle as follows:

$$\theta_0 = \cos^{-1}\left(\frac{v_{x0}}{v_0}\right) = \cos^{-1}\left(\frac{853.4440}{854.9160}\right) = 3.3627°, \text{North of East}$$

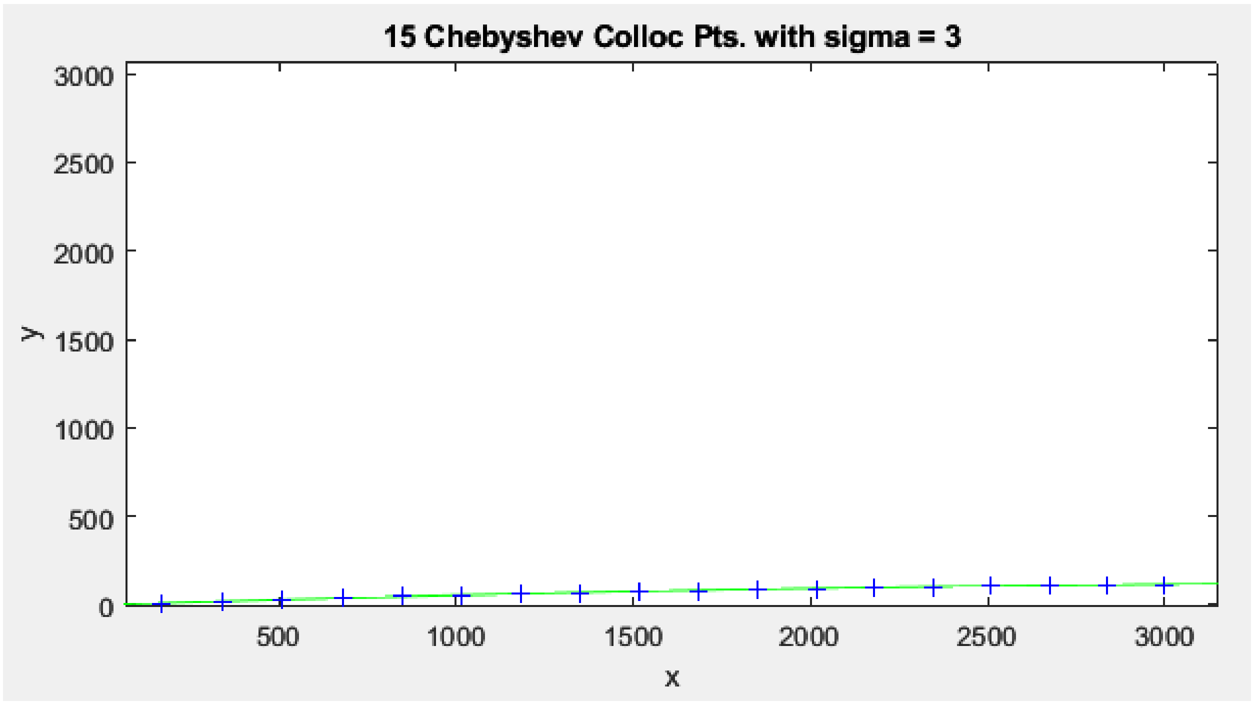
Alternatively, with the help of the very fine interval z, the initial angle is then

$$\theta_0 \approx \tan^{-1}\left(\frac{y_2 - y_1}{x_2 - x_1}\right) \approx 3.2971°, \text{North of East}$$

This "surprising" angle is more visible when we plot the trajectories on a proper square scale:



Zooming in on one of the plots, we can better see that the trajectory is indeed one that begins with a small initial angle (makes sense because launching at a higher angle to reach $8000\ m$ in 10 s with just $854.9160\ m/s$ of initial velocity is quite hard to believe):



Note: The initial speed and the initial angle of the projectile are also computed at the tail-end of nonlinear_colloc in item4.m.

5. Heat Conduction with Variable Conduction Coefficient

Solve the one-dimensional heat conduction equation using Generalized Fourier series method

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(D(x)\frac{\partial u}{\partial x}\right) - u$$

on $x \in [1,3], t > 0$ with $D(x) = x^2$, Dirichlet boundary conditions $u(1,t) = u(3,t) = 0$ and initial condition $u(x,0) = 1 - |x-2|$.

Provide the surface plot of the solution from $t = 0$ until steady state is achieved using sufficient number of terms of the series. You may use numerical integration such as Composite Simpson's 1/3 in lieu of solving the integrals.

Afterwards, solve the same problem using finite difference approximation of partial derivatives using Crank-Nicolson Scheme. Provide the surface plot of the solution similar to the one above then compare the two solutions.

a. Understanding of the problem.

We are to compare the difficulty of deriving the almost analytical solution we got from the difficulty of setting-up the numerical solution. We are also to assess the accuracy of the Crank-Nicolson scheme in solving the special PDE, possibly also touching on Crank-Nicolson's stability, by comparing its results to the ones we're getting from our almost analytical solution.

We say "almost" because we have involved numerical integration in our analytical solution as a matter of convenience.

b. Implementation of the solution (analytical derivations/equation setup, paste code into the PDF and discuss, etc.).

For the analytical solution:

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(x^2\frac{\partial u}{\partial x}\right) - u$$

$$\frac{\partial u}{\partial t} - \frac{\partial}{\partial x}\left(x^2\frac{\partial u}{\partial x}\right) + u = 0$$

$$\frac{\partial u}{\partial t} - 2x\frac{\partial u}{\partial x} - x^2\frac{\partial^2 u}{\partial x^2} + u = 0$$

Separation of variables: Let $u = ST \Leftrightarrow u(x,t) = S(x)T(t)$.

$$\left(S\frac{dT}{dt} - 2xT\frac{dS}{dx} - x^2T\frac{d^2S}{dx^2} + ST = 0\right)\frac{1}{ST}$$

$$\frac{1}{T}\frac{dT}{dt} - \frac{1}{S}2x\frac{dS}{dx} - \frac{1}{S}x^2\frac{d^2S}{dx^2} + 1 = 0$$

Isolate S and T respectively:

$$\frac{1}{T}\frac{dT}{dt}+1=\frac{1}{S}x^2\frac{d^2S}{dx^2}+\frac{1}{S}2x\frac{dS}{dx}=-k, \qquad k\geq 0$$

This converts the special PDE to a system of eigenvalue problems:

$$\begin{cases} \frac{1}{T}\frac{dT}{dt}+1=-k \\ \frac{1}{S}x^2\frac{d^2S}{dx^2}+\frac{1}{S}2x\frac{dS}{dx}=-k \end{cases}$$

Solving for $T(t)$:

$$\frac{1}{T}\frac{dT}{dt}+1=-k$$

$$\frac{1}{T}dT=(-k-1)dt$$

$$\ln T=-kt-t+C$$

$$T=e^{-kt-t+C}$$

$$T(t)=Ae^{-kt-t}, \qquad A=e^C$$

Solving for $S(x)$:

$$\frac{1}{S}x^2\frac{d^2S}{dx^2}+\frac{1}{S}2x\frac{dS}{dx}=-k$$

$$x^2\frac{d^2S}{dx^2}+2x\frac{dS}{dx}=-kS$$

$$x^2S''+2xS'+kS=0$$

This takes the form of a Cauchy-Euler equation, with $a=1$, $b=2$, and $c=k$. (Keep this in mind as we will be using this to solve for the Sturm-Liouville Form later). The indicial equation is then

$$r^2+r+k=0$$

$$r=\frac{-1\pm\sqrt{1-4k}}{2}$$

Case 1 ($1-4k>0$): This gives us distinct roots of r:

$$S=c_1x^{r_1}+c_2x^{r_2}$$

$$S(1)=c_1+c_2=0\Longrightarrow c_1=-c_2$$

$$S(3)=c_13^{r_1}-c_13^{r_2}=0$$

$$c_1(3^{r_1}-3^{r_2})=0$$

$$\therefore c_1=c_2=0$$

Hence, Case 1 is a trivial case.

Case 2 $(1-4k=0)$: This gives us same roots of r (specifically $r=\frac{1}{2}$):

$$S=c_1x^r+c_2x^r\ln x$$

$$S(1)=c_1=0$$

$$S(3)=c_2 3^r\ln 3=0$$

$$\therefore c_1=c_2=0$$

Hence, Case 2 is also a trivial case. We can now expect Case 3 to be the nontrivial case.

Case 3 $\left(1-4k<0\Rightarrow k>\frac{1}{4}\right)$: This gives us complex roots of r:

$$r=\frac{-1\pm\sqrt{1-4k}}{2}=-\frac{1}{2}\pm\frac{\sqrt{4k-1}}{2}i$$

Letting $w=\frac{\sqrt{4k-1}}{2}$, we have

$$S=c_1x^{-\frac{1}{2}}\cos w\ln x+c_2x^{-\frac{1}{2}}\sin w\ln x$$

Applying boundary conditions:

$$S(1)=c_1=0$$

$$S(3)=c_2 3^{-\frac{1}{2}}\sin w\ln 3=0$$

$$c_2\sin w\ln 3=0$$

Forcing $c_2\neq 0$ and $\sin w\ln 3=0$:

$$w\ln 3=n\pi,\qquad n\in\mathbb{N}$$

$$w_n=\frac{n\pi}{\ln 3}$$

Moreover,

$$\frac{\sqrt{4k-1}}{2}=\frac{n\pi}{\ln 3}$$

$$4k-1=\frac{4n^2\pi^2}{\ln^2 3}$$

$$k_n=\frac{n^2\pi^2}{\ln^2 3}+\frac{1}{4}$$

And finally, we have

$$S(x)=\sum_{n=1}^{\infty}c_nx^{-\frac{1}{2}}\sin w_n\ln x$$

Combining S and T back into u, we have

$$u(x,t) = T(t)S(x)$$

$$u(x,t) = Ae^{-kt-t}\sum_{n=1}^{\infty} c_n x^{-\frac{1}{2}} \sin w_n \ln x$$

$$u(x,t) = \sum_{n=1}^{\infty} c_n e^{-k_n t - t} x^{-\frac{1}{2}} \sin w_n \ln x\,, \qquad w_n = \frac{n\pi}{\ln 3}, \qquad k_n = \frac{n^2\pi^2}{\ln^2 3} + \frac{1}{4}$$

The eigenfunctions of which are $\{\sin w_n \ln x\}$.

Solving for the coefficients c_n by treating $u(x,t)$ as a Generalized Fourier Series problem:

Using the initial condition $u(x,0) = 1 - |x-2| = f(x)$:

$$u(x,0) = \sum_{n=1}^{\infty} c_n e^0 x^{-\frac{1}{2}} \sin w_n \ln x = f(x)$$

$$f(x) = \sum_{n=1}^{\infty} c_n x^{-\frac{1}{2}} \sin w_n \ln x = \sum_{n=1}^{\infty} c_n \phi_n$$

$$\therefore c_n = \frac{\left\langle 1 - |x-2|, x^{-\frac{1}{2}} \sin w_n \ln x \right\rangle_r}{\left\langle x^{-\frac{1}{2}} \sin w_n \ln x\,, x^{-\frac{1}{2}} \sin w_n \ln x \right\rangle_r}$$

The final piece: Solving for the $r(x)$ to define $\langle \cdot,\cdot \rangle_r$ where the eigenfunctions of $f(x)$ are pairwise orthogonal. We use the Sturm-Liouville form for this:

$$x^2 S'' + 2xS' + kS = 0$$

$$S'' + 2\frac{1}{x}S' = -k\frac{1}{x^2}S$$

$$\frac{dp}{p} = \frac{2}{x}dx$$

$$\ln p = \ln x^2$$

$$p = x^2$$

Multiplying back to $S'' + 2\frac{1}{x}S' = -k\frac{1}{x^2}S$:

$$x^2 S'' + 2xS' = -k(1)S$$

$$(x^2 S')' = -k(1)S$$

Therefore, $r(x) = 1$, and hence

$$\langle v_1, v_2 \rangle_r \coloneqq \int_1^3 v_1 v_2\, dx\,, \qquad v_1 \neq v_2$$

Consequently,

$$c_n = \frac{\int_1^3 (1-|x-2|)\left(x^{-\frac{1}{2}} \sin w_n \ln x\right) dx}{\int_1^3 \left(x^{-\frac{1}{2}} \sin w_n \ln x\right)^2 dx}$$

With this inner product, the non-matching components will be zeroed out, leaving us with c_n.

And for completeness, we also write here $u(x,t)$:

$$u(x,t) = \sum_{n=1}^{\infty} c_n e^{-k_n t - t} x^{-\frac{1}{2}} \sin w_n \ln x\,, \qquad w_n = \frac{n\pi}{\ln 3}, \qquad k_n = \frac{n^2\pi^2}{\ln^2 3} + \frac{1}{4}$$

Now, to solve the integrals making up c_n, we will have to use Composite Simpson's 1/3 Rule (CSR 1/3) to ease our computations. We refer to this as the function "Newton_Cotes" in our program, because CSR 1/3 belong to the same family of Newton-Cotes numerical integration formulas.

In particular, CSR 1/3 provides the following formula:

$$\int_a^b f(x)dx \approx \frac{h}{3}\left[f(x_1) + 4\sum_{\substack{i=1 \\ i=\text{odd}}}^{n-1} f(x_i) + 2\sum_{\substack{i=2 \\ i=\text{even}}}^{n-2} f(x_i) + f(x_n)\right]$$

In my case, I use 100 test points for the Newton-Cotes method.

This formula is translated into Matlab as follows:

```
function I = Newton_Cotes(n,a,b,int)
    % int pertains to the integrand, pass it onto here
    % as a function using @func syntax
    N = 100;
    if (b-a == 0)
        I=0;
        return;
    end
    h = (b-a)/(N-1);
    x = (a:h:b)'; % test points at h stepsize
    % Composite Simpson's 1/3 Rule
    % Preemptively apply the constant factors of the terms
    w = h/3*ones(N,1);
    w(2:2:end-1) = 4*w(2:2:end-1);
    w(3:2:end-1) = 2*w(3:2:end-1);
    f = int(x);     % solve for y=f(x) values
    % Combine applied constant factors with the f(x) values.
    I = dot(w,f);   % Return value is a scalar
end
```

The parameters a and b denote the given bounds of $x \in [1,3\,]$ which are also used as the upper and lower lomit of integration respectively. Meanwhile, the parameter int receives a function object corresponding to the integrand of the integral being numerically approximated.

Also note here that the local variable N is distinct from the parameter n. N denotes the number of test points used for the Newton-Cotes CSR 1/3 approximation while n denotes which term of the summation in $u(x,t)$ the Newton-Cotes method is currently working with. n is important because it is used for evaluating w_n and k_n which we encoded as functions:

```
%% Auxiliary Functions for the Analytical Solution
function ret = w_n(n)
    ret = n*pi/log(3);
end

function ret = k_n(n)
    ret = n^2*pi^2/log(3)^2 + 1/4;
end
```

Using the function Newton_Cotes, we can now solve for the coefficient c_n on the nth term of $u(x,t)$, which returns its numerator cn_{num} divided by its denominator cn_{den}:

```
function ret = c_n(n,a,b)
    % The xs inside cn_num and cn_den will be vectors wrt CSR 1/3
    cn_num = @(x) (1-abs(x-2)).*x.^(-1/2).*sin(w_n(n).*log(x));
    cn_den = @(x) (x.^(-1/2).*sin(w_n(n).*log(x))).^2;
    ret = Newton_Cotes(n,a,b,cn_num)/Newton_Cotes(n,a,b,cn_den);
    % return value is a scalar

end
```

Observe how we are passing cn_num and cn_den into Newton_Cotes.

We can then apply these results to $u(x,t)$, wherein we only take $N = 20$ terms of the summation:

$$u(x,t) \approx \sum_{n=1}^{20} c_n e^{-k_n t - t} x^{-\frac{1}{2}} \sin w_n \ln x = \text{sum}\begin{bmatrix} c_1 e^{-k_1 t - t} x^{-\frac{1}{2}} \sin w_1 \ln x \\ \vdots \\ c_j e^{-k_j t - t} x^{-\frac{1}{2}} \sin w_j \ln x \\ \vdots \\ c_{20} e^{-k_{20} t - t} x^{-\frac{1}{2}} \sin w_{20} \ln x \end{bmatrix}$$

And this is coded as:

```
function ret = u(N,x,t)
    % Pass onto here values from fine intervals of x and t.
    % i.e. x and t are not vectors here, neither is c_n
    global e;
    ret = 0;
    for n = 1:N
        ret = ret + c_n(n,1,3)*e^(-k_n(n)*t-t)*x^(-1/2)*sin(w_n(n)*log(x));
    end
    % return value is a scalar
end
```

We decided to solve these summations of $u(x,t)$ per x and t pair using nested for loops, constructing the solution $u(x,t)$ (which we named usoln) in the process:

```
%% Part 1: Analytical Solution
function ana()
    global e;
    e = exp(1);
    N = 20;                 % Adjust N for no. of summation terms considered
    x = linspace(1,3,50);      % x in [1,3]
    t = linspace(0,1,50);     % t > 0, same size as x
    usoln = zeros(numel(x),numel(t));
    for j = 1:numel(t)
        for i = 1:numel(x)
            usoln(i,j) = u(N,x(i),t(j));
        end
    end
    figure(1);
    plotter(x,t,usoln');
    xlabel('space x'); ylabel('time t'); zlabel('u(x,t)');
    title("Analytical solution of u(x,t)")
    view(2);
end
```

Some more remarks:

- We are using a 50x50 meshgrid for usoln corresponding to the dimensions of x and t (both 50×1 vectors).
- Before plotting, usoln has to be first **transposed** in order to be consistent with the diffusion model and especially the boundary conditions. This is because in Matlab and other math softwares in general, the axes of a plane are transposed when stored in an array as per the $A(\text{row}, \text{col}) \equiv A(y, x)$ notation. Please keep this in mind for the other plots later.
- The **plotter** modified from https://www.mathworks.com/matlabcentral/answers/387362-how-do-i-create-a-3-dimensional-surface-from-x-y-z-points and has the following code:

```
function plotter(x,y,z)
    % Prep for plotting (Borrowed online)
    % https://www.mathworks.com/matlabcentral/answers/387362-how-do-i-
    % create-a-3-dimensional-surface-from-x-y-z-points
    % figure(fig*2+1)
    % stem3(x, y, z)
    grid on

    xv = linspace(min(x), max(x), 100);
    yv = linspace(min(y), max(y), 100);
    [X,Y] = meshgrid(xv, yv);
    Z = griddata(x,y,z,X,Y);

    % figure(fig*2+2)
    surf(X, Y, Z);
    grid on
    xlabel('space x'); ylabel('time t'); zlabel('heat u(x,t)')
    set(gca, 'ZLim',[0 100])
    shading interp
    view(2)
end
```

The above code also serves as the driver code of program's first part: Analytical Solution.

For the Forward in Time, Central in Space (FTCS) Scheme:

Given

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(D(x)\frac{\partial u}{\partial x}\right) - u, \qquad D(x) = x^2$$

$$\frac{\partial u}{\partial t} = D'(x)\frac{\partial u}{\partial x} + D(x)\frac{\partial^2 u}{\partial x^2} - u$$

$$u(1,t) = u(3,t) = 0 = \alpha = \beta, \qquad u(x,0) = 1 - |x-2|$$

We first list down the important Finite Difference Approximations that we will use to construct the explicit method:

- First order Forward Approximation, to be used for partials with respect to time t

$$f'(x) = \frac{f(x+h) - f(x)}{h}$$

- Second Order Central Approximations, to be used for partials with respect to space x

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h}$$

$$f''(x) = \frac{f(x-h) - 2f(x) + f(x+h)}{h^2}$$

We now substitute these to our PDE. Note that here, we will be using the notation $U\big(x(i), t(j)\big) = u_i^j = u\big(x_i, t_j\big)$ to create expressions that are immediately translatable to Matlab. Note that $D(x)$ doesn't need to be approximated because we already have the formula for it.

Solving for either time or space and freezing the other:

$$\frac{\partial u}{\partial t} = D'(x)\frac{\partial u}{\partial x} + D(x)\frac{\partial^2 u}{\partial x^2} - u$$

$$\frac{u_i^{j+1} - u_i^j}{\Delta t} = D'(x)\left(\frac{u_{i+1}^j - u_{i-1}^j}{2\Delta x}\right) + D(x)\left(\frac{u_{i-1}^j - 2u_i^j + u_{i+1}^j}{(\Delta x)^2}\right) - u_i^j$$

Solving for u_i^{j+1}: Multiply both sides by Δt (while also simplifying D),

$$u_i^{j+1} - u_i^j = 2x_i\Delta t\left(\frac{u_{i+1}^j - u_{i-1}^j}{2\Delta x}\right) + x_i^2\Delta t\left(\frac{u_{i-1}^j - 2u_i^j + u_{i+1}^j}{(\Delta x)^2}\right) - u_i^j\Delta t$$

$$u_i^{j+1} = x_i\Delta t\left(\frac{u_{i+1}^j - u_{i-1}^j}{\Delta x}\right) + x_i^2\Delta t\left(\frac{u_{i-1}^j - 2u_i^j + u_{i+1}^j}{(\Delta x)^2}\right) - u_i^j\Delta t + u_i^j$$

Refactor for readability:

$$u_i^{j+1} = \left[-\frac{x_i\Delta t}{\Delta x} + \frac{x_i^2\Delta t}{(\Delta x)^2}\right]u_{i-1}^j + \left[-\frac{2x_i^2\Delta t}{(\Delta x)^2} - \Delta t + 1\right]u_i^j + \left[\frac{x_i\Delta t}{\Delta x} + \frac{x_i^2\Delta t}{(\Delta x)^2}\right]u_{i+1}^j$$

$$u_i^{j+1} = \left[\frac{x_i^2\Delta t}{(\Delta x)^2} - \frac{x_i\Delta t}{\Delta x}\right]u_{i-1}^j + \left[1 - \frac{2x_i^2\Delta t}{(\Delta x)^2} - \Delta t\right]u_i^j + \left[\frac{x_i^2\Delta t}{(\Delta x)^2} + \frac{x_i\Delta t}{\Delta x}\right]u_{i+1}^j$$

And for the initial guess, we use the initial condition $u(x,0) = 1 - |x-2|$.

~~We ignore the restriction on Δt given by the CFL stability criterion (as we don't know how to derive this for non-constant $D(x)$) and simply trust our guts that the values we provide will work.~~

Aside from this, we will have to compute the CFL stability criterion of the FTCS scheme because when I tried just ignoring it, the program lashed out at me by not converging at all. Actually no, I cannot figure out how to solve for the Courant number. According to this[4], we have to reduce both Δx and Δt logarithmically to get a more accurate FTCS. So what I'm going to try is to adjust Δx and Δt independently of each other, and the mesh grid that we'll use to plot may not be necessarily square anymore.

YES! I tried brute forcing my way into getting a convergent simulation and just a few minutes in, it worked!

The working values are $\Delta x = 0.2500$ and $\Delta t = 0.0045$. The relatively large Δx means that the x-axis will have rougher stepsize but we have to work with that if we want to finish this. This situation also means that we will have to adjust BTCS and CNS to be compatible with the dimensions generated by these uneven intervals. I call the adjusted BTCS and CNS schemes BTCS_REDUCED and CNS_REDUCED respectively, as we'll see later. However, we'll still present BTCS and how its output is already great when it's working alone.

The FTCS scheme derived here now corresponds to this portion of the program:

```
%% Part 2: Numerical Solution
function U = ftcs()
    %% Heat Equation (with variable coefficients) (FTCS)
    % x in [x1,xn], t in [t1, tn], D = x^2 %% WARNING: UNSTABLE
    x1 = 1; xn = 3; t1 = 0; tn = 1;
    % Dirichlet BCs: u(x1,t)=alpha, u(xn,t)=beta;
    alpha = 0; beta = 0;
    % IC: u(x,0)=f(x)
    f = @(x) 1 - abs(x - 2);

    % Discretization
    % We'll have to abandon plotter for this to work, adjust dx and dt
    % independently of each other, U may not be square
    dx = 0.25000;
    x = (1:dx:3)';
    dt = 0.0045;
    t = (0:dt:1)';

    U = zeros(numel(x),numel(t));      % initialize U, the zeros are placeholders
only
    U(1,:) = alpha;
    U(end,:) = beta;
    U(2:end-1,1) = f(x(2:end-1)); % set init guess here, use IC f(x)
    for j = 1:numel(t)-1
        for i = 2:numel(x)-1
```

[4] https://web.cecs.pdx.edu/~gerry/class/ME448/notes/1Dmodels/pdf/FTCS_slides.pdf

```
                U(i,j+1) = (x(i)^2*dt/dx^2 - x(i)*dt/dx)*U(i-1,j) + ...
                    (1 - 2*x(i)^2*dt/dx^2 - dt)*U(i,j) + ...
                    (x(i)^2*dt/dx^2 + x(i)*dt/dx)*U(i+1,j);
            end
        end

        U   % approximate temp gradient
        figure(2);
        surf(x,t,U');
        title("Numerical solution of u(x,t) using FTCS (UNSTABLE)")
        xlabel('space x'); ylabel('time t'); zlabel('heat u(x,t)')
        view(2)
        % NOTE: TO combine this with btcs, merge btcs backwards to ftcs
        % i.e. filter data from btcs to fit ftcs.
    end
```

For the Backward in Time, Central in Space (BTCS) Scheme:

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(D(x)\frac{\partial u}{\partial x}\right) - u, \qquad D(x) = x^2$$

$$u(1,t) = u(3,t) = 0 = \alpha = \beta, \qquad u(x,0) = 1 - |x-2|$$

Goal: Set up a system of nonlinear equations by isolating the known u_i^{j-1} on the RHS and the unknowns u_{i-1}^j, u_i^j, and u_{i+1}^j on the LHS.

We begin by applying the finite difference approximations here as listed in the FTCS scheme.

$$\frac{\partial u}{\partial t} = D'(x)\frac{\partial u}{\partial x} + D(x)\frac{\partial^2 u}{\partial x^2} - u$$

$$\left(\frac{u_i^j - u_i^{j-1}}{\Delta t} = D'(x_i)\frac{u_{i+1}^j - u_{i-1}^j}{2\Delta x} + D(x_i)\frac{u_{i+1}^j - 2u_i^j + u_{i-1}^j}{(\Delta x)^2} - u_i^j\right)\Delta t(\Delta x)^2$$

$$u_i^j(\Delta x)^2 - u_i^{j-1}(\Delta x)^2 = D'(x_i)\frac{u_{i+1}^j - u_{i-1}^j}{2}\Delta t\Delta x + D(x_i)\left(u_{i+1}^j - 2u_i^j + u_{i-1}^j\right)\Delta t - u_i^j\Delta t(\Delta x)^2$$

$$u_i^j(\Delta x)^2 - D'(x_i)\frac{u_{i+1}^j - u_{i-1}^j}{2}\Delta t\Delta x - D(x_i)\left(u_{i+1}^j - 2u_i^j + u_{i-1}^j\right)\Delta t + u_i^j\Delta t(\Delta x)^2 = u_i^{j-1}(\Delta x)^2$$

Expanding:

$$(\Delta x)^2 u_i^j - D'(x_i)\Delta t\Delta x\frac{u_{i+1}^j - u_{i-1}^j}{2} - D(x_i)\Delta t\left(u_{i+1}^j - 2u_i^j + u_{i-1}^j\right) + \Delta t(\Delta x)^2 u_i^j = (\Delta x)^2 u_i^{j-1}$$

$$\left[\frac{D'(x_i)\Delta t\Delta x}{2} - D(x_i)\Delta t\right]u_{i-1}^j + \left[(\Delta x)^2 + 2D(x_i)\Delta t + \Delta t(\Delta x)^2\right]u_i^j + \left[-\frac{D'(x_i)\Delta t\Delta x}{2} - D(x_i)\Delta t\right]u_{i+1}^j = (\Delta x)^2 u_i^{j-1}$$

$$\left[x_i\Delta t\Delta x - x_i^2\Delta t\right]u_{i-1}^j + \left[(\Delta x)^2 + 2x_i^2\Delta t + \Delta t(\Delta x)^2\right]u_i^j + \left[-x_i\Delta t\Delta x - x_i^2\Delta t\right]u_{i+1}^j = (\Delta x)^2 u_i^{j-1}$$

Converting this to an SNLE problem:

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ x_i\Delta t\Delta x - x_i^2\Delta t & (\Delta x)^2 + 2x_i^2\Delta t + \Delta t(\Delta x)^2 & -x_i\Delta t\Delta x - x_i^2\Delta t & \cdots & 0 \\ 0 & x_i\Delta t\Delta x - x_i^2\Delta t & (\Delta x)^2 + 2x_i^2\Delta t + \Delta t(\Delta x)^2 & \cdots & 0 \\ 0 & 0 & x_i\Delta t\Delta x - x_i^2\Delta t & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_1^j \\ u_2^j \\ u_3^j \\ u_4^j \\ u_5^j \\ \vdots \\ u_n^j \end{bmatrix} = \begin{bmatrix} \alpha \\ (\Delta x)^2 u_2^{j-1} \\ (\Delta x)^2 u_3^{j-1} \\ (\Delta x)^2 u_4^{j-1} \\ (\Delta x)^2 u_5^{j-1} \\ \vdots \\ \beta \end{bmatrix}$$

We solve this in the algorithm using LU Factorization. Note that since $D(x)$ is given, we can first solve for the values of $D(x_i)$ and use that as a constant substitute for $D(x)$, allowing us to solve this as an SLE Problem.

Note that compared to when D is constant, we need more lines to complete the array set-up here, notably a for loop to set up the internal nodes of the coefficient matrix and another for loop working with $u(x,0) = 1 - |x-2|$ to set up the initial guess for the vector of unknowns u^j.

The BTCS scheme derivation here corresponds to this portion of the program (which also comes with an LU factorization implementation in order to accomplish the implicit “update” step):

```
function U = btcs()
    %% Heat Equation (with variable coefficients) (BTCS)
    % x in [x1,xn], t in [t1, tn], D = x^2
    x1 = 1; xn = 3; t1 = 0; tn = 1;
    % Dirichlet BCs: u(x1,t)=alpha, u(xn,t)=beta;
    alpha = 0; beta = 0;
    % IC: u(x,0)=f(x)

    % Discretization
    n = 30; % number of points in space and time, total of n*n points
    % Note: Prog only works when nx = nt, issue with plotter when nx != nt
    dx = (xn-x1)/(n-1);            % stepsize in space
    x = x1 + (0:dx:(n-1)*dx);      % steps in space
    dt = (tn-t1)/(n-1);            % stepsize in time, no restrictions
    t = t1 + (0:dt:(n-1)*dt);      % steps in time

    % Construct Au = b  % Where b = [alpha, ..., dx^2*u_prev(i), ..., beta]
    A = zeros(n,n);     % Becomes almost a constant tridiagonal matrix
                        % Can use LU fact because A is constant
                        % Zeros are (useful) placeholders
    % Boundary Nodes
    A(1,1) = 1;
    A(n,n) = 1;
```

```
% Internal Nodes (Manual Construction)
for i = 2:n-1
    A(i,i-1) = x(i)*dt*dx-x(i)^2*dt;       % coeff of u_(i-1)^j (left)
    A(i,i)   = dx^2+2*x(i)^2*dt+dt*dx^2;   % coeff of u_i^j (center)
    A(i,i+1) = -x(i)*dt*dx-x(i)^2*dt;      % coeff of u_(i+1)^j (right)
end

% Construct solution matrix V
% Will dynamically contain:
% Knowns u_i^(j-1) or u_prev and
% Unknowns u_i^j or u (should both be 2D)

% Initial guess on u_i^1 (use this instead of given IC)
% Note: Use V as container of both u and u_prev
%  (use V instead of U to avoid confusion with LU fact)
V = ones(n,n);
% V = dx^2*V;        %% mult dx^2 once column is next to be used (!!)
                     % bcos if we do this here, u_prev will contain
                     % updated values already
V(1,1:end) = alpha;
V(n,1:end) = beta;
for i = 2:n-1
    V(i,1) = 1-abs(x(i)-2); % set init guess here, USE IC
end

LU = lufact(A,n);    % Workable since A is square

% No updating, merely iterating through what we already know.
% Don't confuse this too much with linear FDA.

% Solve u^j using u^(j-1) (fun to watch contents of V)
for j = 2:n
    %u = A\b;            % for verifying result of LU fact: okay

    % Solve LUu = u_prev, let Uu = y.
    % Forward sub Ly = u_prev. Solve for y.
    % Take u_prev from V, then modify to reflect b
    %   i.e. mult internal nodes by dx^2
    u_prev = V(:,j-1);
    u_prev(2:end-1,1) = dx^2*u_prev(2:end-1,1);     %% (!!)
    b = u_prev;          % rename for consistency
    y = b;               % y(1) is correct
    for i=2:n
        y(i)=y(i)-LU(i,1:i-1)*y(1:i-1);
    end

    % Backward sub Uu = y. Solve for u.
    u = y;
    u(n) = u(n);         % u(1) is correct (boundary)
    for i=n-1:-1:1
        u(i) = (u(i) - LU(i,i+1:end)*u(i+1:end))/LU(i,i);
    end

    V(:,j) = u;  % insert solution to u^j
end
```

```
    U = V   % approximate temp gradient
    % Use U' for plotter to rectify x and t axes
    figure(3);
    plotter(x,t,U');
    title("Numerical solution of u(x,t) using BTCS")
end

function [LU] = lufact(A,n)
    LU = A;
    % Iterate through columns, pivoting at diag entries
    for j=1:n
        % Check for failure point.
        if LU(j,j)==0
            disp('Zero pivot encountered');
            return;
        end
        for i=j+1:n
            % Check for numerically zero entries below LU(j,j)
            if abs(LU(i,j)) < 1E-10
                LU(i,j) = 0;
                continue;
            end
            % Get scalar used to annihilate LU(i,j),
            % store scalar in corresponding LU entry
            LU(i,j) = LU(i,j)/LU(j,j);
            % ERO3 to annihilate L(i,j), row subtraction assumed
            LU(i,j+1:end) = LU(i,j+1:end) - LU(i,j)*LU(j,j+1:end);
        end
    end
end
```

However, as mentioned earlier, we have to adjust the BTCS function to fit the edgy FTCS scheme that we're currently tolerating. Said adjustment took liberty to use the left-divide or the matrix solver operator of Matlab because with the working dimension now rectangular, my implementation of LU factorization now fails and I have no time to fix that. We now present BTCS_REDUCED:

```
function U = btcs_reduced()
    %% Heat Equation (with variable coefficients) (BTCS)
    % x in [x1,xn], t in [t1, tn], D = x^2
    x1 = 1; xn = 3; t1 = 0; tn = 1;
    % Dirichlet BCs: u(x1,t)=alpha, u(xn,t)=beta;
    alpha = 0; beta = 0;
    % IC: u(x,0)=f(x)

    % Diff of this with regular BTCS is that we make this match
    % the dimensions of the FTCS
    dx = 0.25000;
    x = (1:dx:3)';
    dt = 0.0045;
    t = (0:dt:1)';

    % Construct Au = b  % Where b = [alpha, ..., dx^2*u_prev(i), ..., beta]
    p = numel(x); q = numel(t);
    A = zeros(p,q);
    % Boundary Nodes
    A(1,1) = 1;
    A(p,q) = 1;
    % Internal Nodes (Manual Construction)
    for i = 2:p-1
        A(i,i-1) = x(i)*dt*dx-x(i)^2*dt;       % coeff of u_(i-1)^j (left)
        A(i,i)   = dx^2+2*x(i)^2*dt+dt*dx^2;   % coeff of u_i^j (center)
        A(i,i+1) = -x(i)*dt*dx-x(i)^2*dt;      % coeff of u_(i+1)^j (right)
    end

    % Construct solution matrix V
    % Will dynamically contain:
    % Knowns u_i^(j-1) or u_prev and
    % Unknowns u_i^j or u (should both be 2D)

    % Initial guess on u_i^1 (use this instead of given IC)
    % Note: Use V as container of both u and u_prev
    %  (use V instead of U to avoid confusion with LU fact)
    V = ones(p,q);
    % V = dx^2*V;       %% mult dx^2 once column is next to be used (!!)
                        % bcos if we do this here, u_prev will contain
                        % updated values already
    V(1,1:end) = alpha;
    V(p,1:end) = beta;
    for i = 2:p-1
        V(i,1) = 1-abs(x(i)-2); % set init guess here, USE IC
    end

%     LU = lufact(A,n);   % Workable since A is square

    % No updating, merely iterating through what we already know.
    % Don't confuse this too much with linear FDA.

    % Solve u^j using u^(j-1) (fun to watch contents of V)
    for j = 2:q
        % Because U is now not square, rendering my LU Fact
        % implementation useless, we have taken the liberty
        % of using the built in matrix solver
        u_prev = V(:,j-1);
        u_prev(2:end-1,1) = dx^2*u_prev(2:end-1,1);     %% (!!)
        b = u_prev;          % rename for consistency
        u = A\b;
        V(:,j) = u(1:9);  % insert solution to u^j
    end

    U = V   % approximate temp gradient
    % Use U' for plotter to rectify x and t axes
    figure(4);
    surf(x,t,U');
    title("Numerical solution of u(x,t) using BTCS (REDUCED)")
    xlabel('space x'); ylabel('time t'); zlabel('heat u(x,t)')
    view(2);
end
```

Note again that BTCS_REDUCED has to be done because we have to take the average of the FTCS and BTCS scheme later and for that their approximated heat gradient matrices U_1 and U_2 (see CNS function below) needs to be compatible with each other.

For the Crank-Nicolson Scheme: We simply take the average of the FTCS scheme and BTCS_REDUCED scheme.

```
function U = cns_reduced()
    %% Example: Heat Equation (with variable coefficients) (CNS)
    x1 = 0; xn = 10; t1 = 0; tn = 1; D = 2;
    % Dirichlet BCs: u(x1,t)=alpha, u(xn,t)=beta;

    % Discretization
    n = 30; % number of points in space and time, total of n*n points
    % Note: Prog only works when nx = nt, issue with plotter when nx != nt
    dx = 0.25000;
    x = (1:dx:3)';
    dt = 0.0045;
    t = (0:dt:1)';

    % U1: from FTCS, U2: from BTCS_REDUCED
    U1 = ftcs();
    U2 = btcs_reduced();
    U = (U1 + U2) ./ 2;     % take the average

    U
    figure(5);
    surf(x,t,U');
    title("Numerical solution of u(x,t) using CNS (REDUCED)")
    xlabel('space x'); ylabel('time t'); zlabel('heat u(x,t)')
    view(2);
end
```

c. **Results. Show some snapshots of solution process and plots appropriate in the discussion of the results. Problems requiring numerical solution must be implemented using non-built-in methods or libraries.**

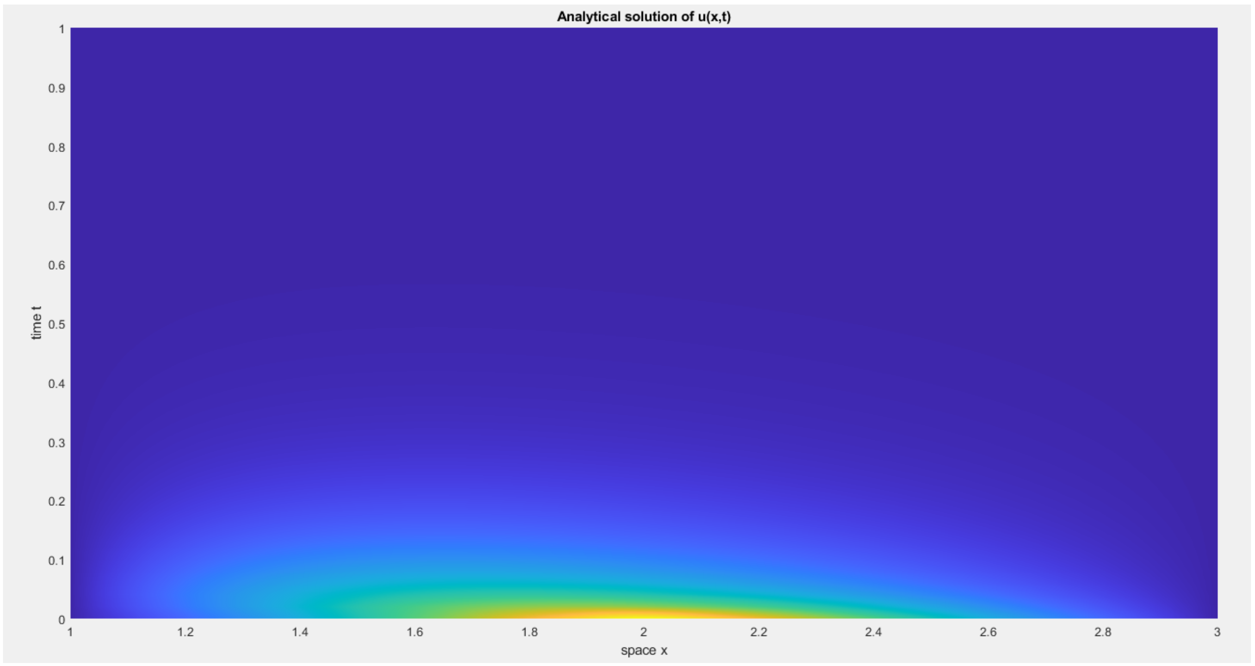
The driver code for the entire program is simply:

```
% item5.m [Hebron, Yenzy]
ana();
% ftcs();
btcs();
cns_reduced();
```

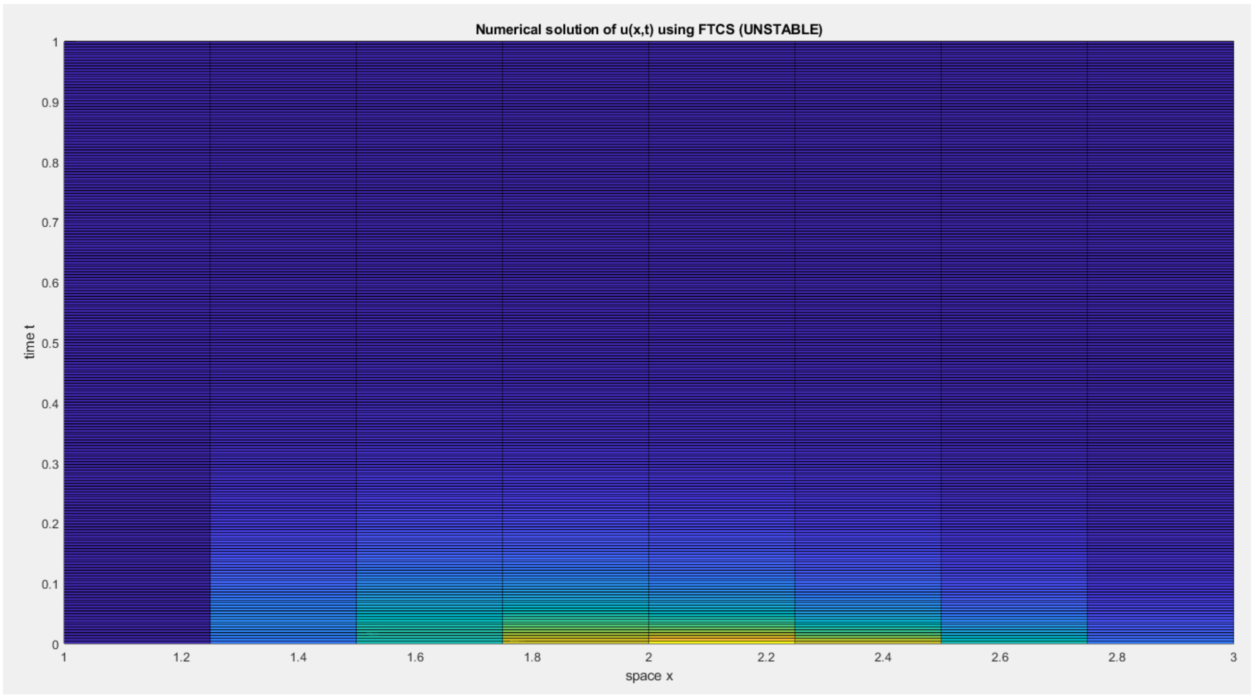
I am still running the regular BTCS scheme (btcs()) aside from the BTCS_REDUCED scheme inside CNS_REDUCED, because I feel bad about abandoning it when its graph looks pretty.

Alright, without further ado, I present to you the plots of $u(x,t)$ obtained using various methods.

Below is the plot of $u(x,t)$ using the analytical solution (the visually pleasing plot is again courtesy of the borrowed plotter function):

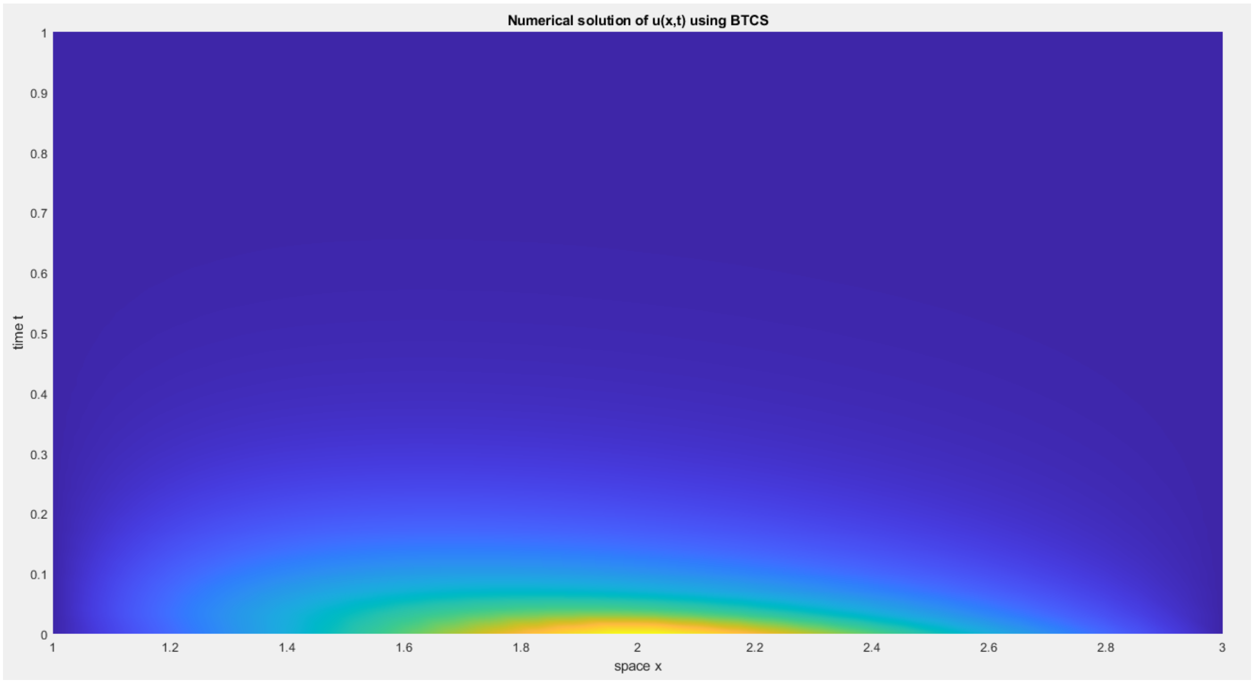


Below is the plot of $u(x,t)$ using the unstable FTCS scheme:



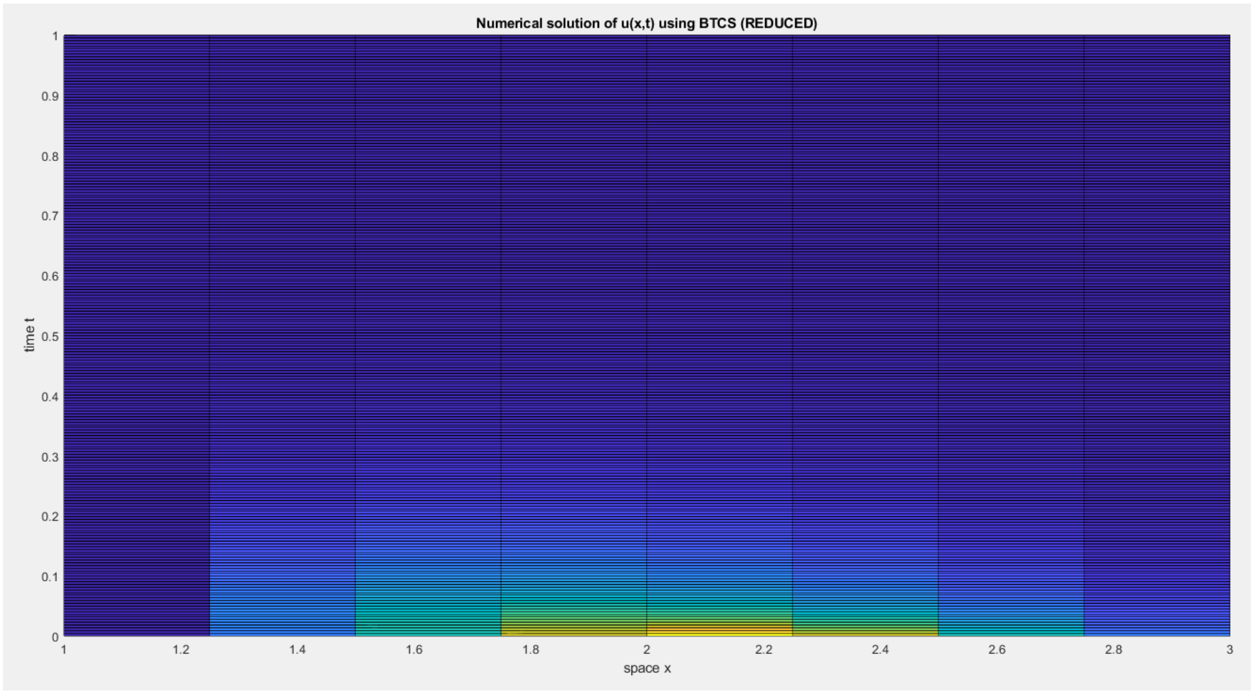
It’s regrettable that we can’t use the borrowed plotter function for the schemes involved with this rough FTCS scheme due to issues with dimensional compatibility.

Below is the plot of $u(x,t)$ using the regular BTCS scheme:

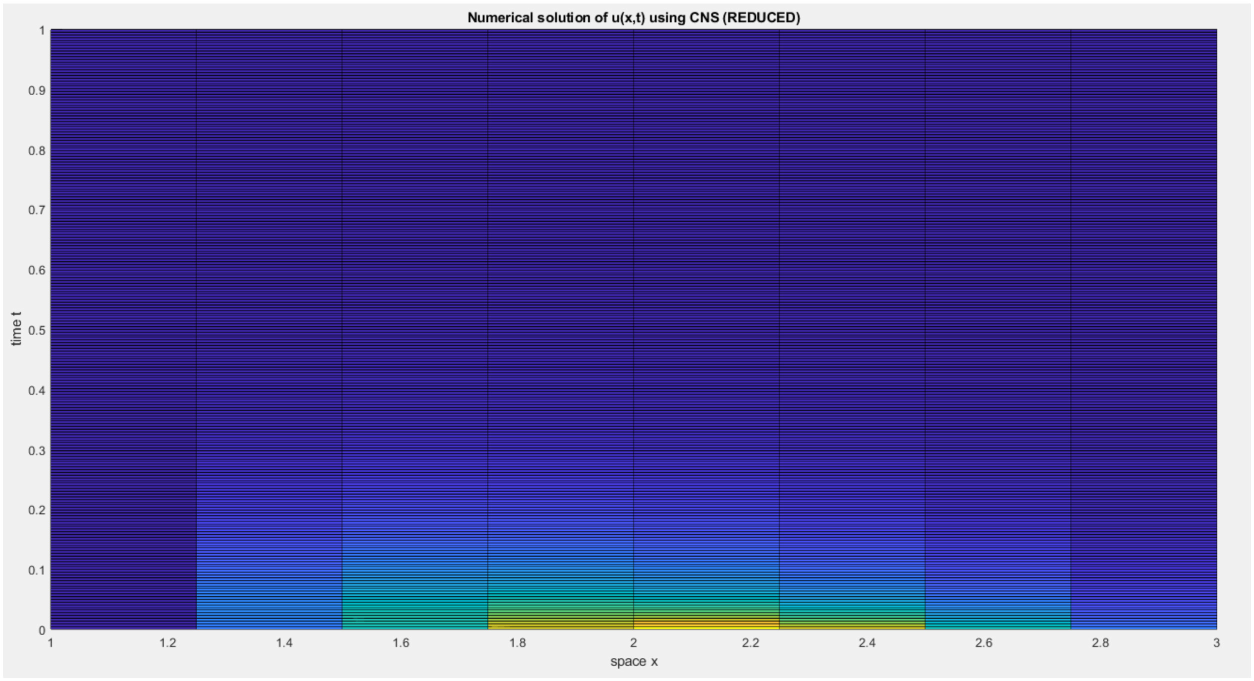


Observe how the above plot is a close approximation of the analytical plot. More so, the BTCS scheme is stable enough that we were able to create a square mesh grid over which to solve and plot $u(x,t)$. This exercise is really warming me up to BTCS: despite the difficulty of setting it up, it really pays off in the end.

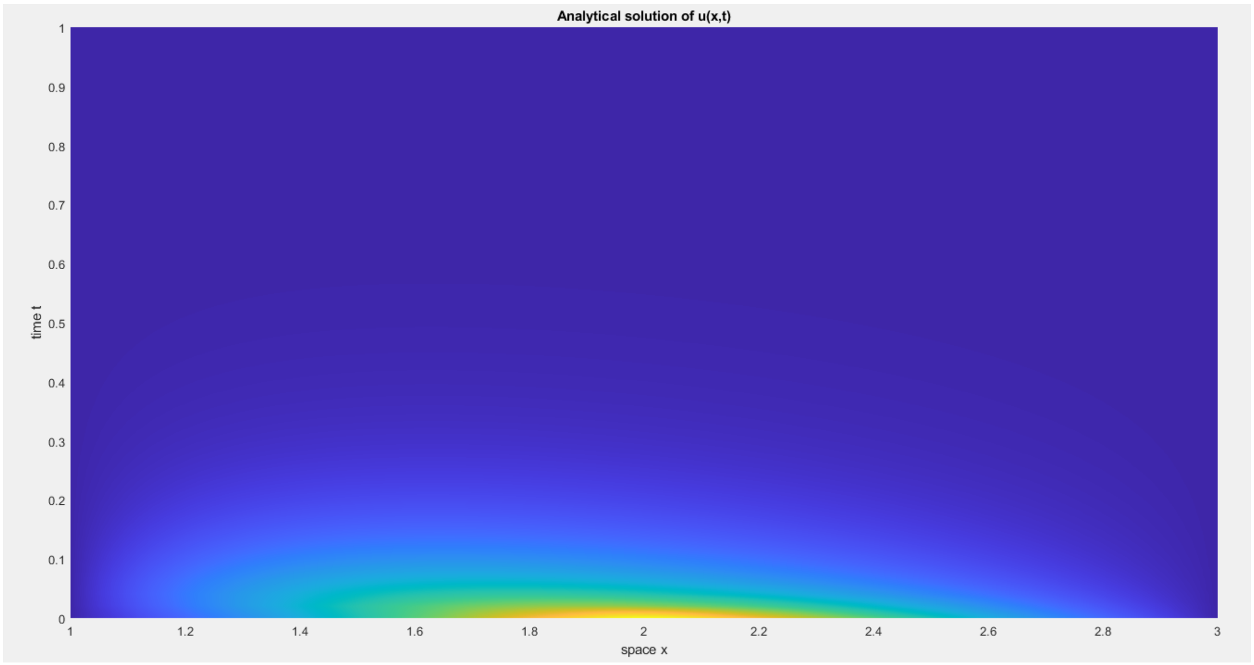
Up next is the plot of $u(x,t)$ using the BTCS_REDUCED scheme, which looks like so because we want it to be compatible with the FTCS data once the time for CNS comes.



Lastly, the plot for CNS_REDUCED is shown below, combining and taking the average of the FTCS and the BTCS_REDUCED $u(x,t)$ solutions:



The analytical solution is again shown below for easy comparison:



As we can see, despite the roughness of the above CNS plot, it's already well on its way to approximate the analytical solution. We can also conclude that if the stability issues or the noise of its contributing FTCS scheme is resolved, we can definitely expect that CNS will converge to the analytical solution, as even with stability issues it's already "tolerable".

Also, we have to remind ourselves that the analytical solution is not purely analytical as we have involved numerical integration in it. So when we speak about the difficulty between the analytical solution and the numerical solution (i.e. avg(FTCS + BTCS) = CNS), we can still say that the numerical solution is a better choice in this case because solving such integrals, i.e.

$$c_n = \frac{\int_1^3 (1 - |x-2|)\left(x^{-\frac{1}{2}} \sin w_n \ln x\right) dx}{\int_1^3 \left(x^{-\frac{1}{2}} \sin w_n \ln x\right)^2 dx},$$

would have been a nightmare. However, the stability problems of FTCS and the possible necessity of computing its CFL stability criterion (which I was unable to do) is also something worth considering when choosing our approach. And that is why I insisted on showing the result of the regular BTCS solution, because for me, it's already a satisfying approximation of what the analytical solution would have given us.

As an aside, throughout this problem set, we also got to combine different numerical methods, with one usually being a subroutine to another, and that way, their intrinsic properties like stability and speed of convergence and such also end up overlapping with each other. So that's something we have to keep in mind when analyzing our numerical methods.

I think that's it for this problem set.

Thank you so much for everything you've shared!