FCM Programming Assignment 2

Jonathan Cushman

October 2025

1 Executive Summary

In this report we are asked to compute the matrix-vector product through a variety of techniques and algorithms. We are asked to store the matrix information in data structure that take allow us to avoid unnecessary computations, leaving us with a better complexity. We will then test these algorithms by evaluating there accuracy as the number of dimensions, n, grows along with a timing test to evaluate the speed of the algorithm.

2 Statement of the Problem

For this assignment, we are asked to compute Matrix Vector Products under certain restrictions. The first tasks gives us an in row order coordinate data structure that represents a non symmetric matrix. The routines produced in task 1 store our non-symmetric matrix as a compressed spare row (CSR), modified CSR, and Ellpack-Itpack data structures. We then use these data structures to compute the matrix vector product. Task 2 is similar but this time we are given A matrix in coordinate form such that the rows are not indexed in increasing order and we output a CSR data structure which then computes the matrix vector product. The third task differ from the previous in the fact that we are given a symmetric matrix and need to implement a CSR data structure using only the lower triangular part of our matrix, then compute the matrix vector product using the CSR. The fourth and final task gives us a sparse matrix with nonzero values over 3 diagonal bands. We are then asked to implement the compressed diagonal data structure and use that to compute the matrix vector product.

3 Description of Mathematics

3.1 Converting from in row order COO to CSR

Let

$$A = \begin{bmatrix} \lambda_{11} & 0 & 0 \\ \lambda_{21} & \lambda_{22} & 0 \\ 0 & \lambda_{32} & \lambda_{33} \end{bmatrix}, \quad v = \begin{bmatrix} \nu_1 \\ \nu_2 \\ \nu_3 \end{bmatrix} \tag{1}$$

$$AA = [\lambda_{11}, \lambda_{21}, \lambda_{22}, \lambda_{32}, \lambda_{33}] \tag{2}$$

$$JR = [1, 2, 2, 3, 3] \tag{3}$$

$$JC = [1, 1, 2, 2, 3] \tag{4}$$

For CSR, we need a row pointer IA.
Let $IA(i) = 1$, and

$$IA(i+1) = IA(i) + \text{nonzeros in row } i$$

Giving,

$$IA = [1, 2, 4, 6] \tag{5}$$

Then our CSR data structure is:

$$AA = [\lambda_{11}, \lambda_{21}, \lambda_{22}, \lambda_{32}, \lambda_{33}] \tag{6}$$

$$JA = [1, 1, 2, 2, 3] \tag{7}$$

$$IA = [1, 2, 4, 6] \tag{8}$$

Then,

$$z(i) = \sum_{k=IA(i)}^{IA(i+1)-1} AA(k) \cdot v(JA(k)) \tag{9}$$

to find the product of our matrix-vector multiplication.

3.2 In row order COO to Modified CSR

Let

$$A = \begin{bmatrix} \lambda_{11} & 0 & 0 \\ \lambda_{21} & \lambda_{22} & 0 \\ 0 & \lambda_{32} & \lambda_{33} \end{bmatrix}, \quad v = \begin{bmatrix} \nu_1 \\ \nu_2 \\ \nu_3 \end{bmatrix} \tag{10}$$

In the modified CSR, our matrix A will be stored using 4 Arrays.

Diagonal elements:

$$D(i) = A_{ii} = \lambda_{ij} \tag{11}$$

Off-diagonal elements (stored row by row):

$$\text{off-D}(i) = A_{ij} = \lambda_{ij} \quad \text{such that} \quad i \neq j \tag{12}$$

Off-diagonal column indices, and row pointers:

$$D = [\lambda_{11}, \lambda_{22}, \lambda_{33}], \quad \text{off-D} = [\lambda_{21}, \lambda_{32}]$$

$$\text{off-D-column} = [1, 1, 2, 3], \quad \text{Row pointers} = [1, 1, 2]$$

Then we concatenate the arrays to achieve our desired modified CSR data structure:

$$\text{modified-AA} = [\text{Diagonal elements}, 0, \text{off-Diagonal elements}]$$

$$JA = [\text{Row pointers}, \text{off-diagonal column indices}]$$

Then perform the matrix-vector product using:

$$z(i) = D(i) \cdot V(i) + \sum_{k=\text{rowpointer}(i)}^{\text{rowpointer}(i+1)-1} \text{off-D}(k) \cdot v(\text{off-D-column}(k)) \tag{13}$$

3.3 In row order COO to Ellpack-Itpack

$$A = \begin{bmatrix} \lambda_{11} & 0 & 0 \\ \lambda_{21} & \lambda_{22} & 0 \\ 0 & \lambda_{32} & \lambda_{33} \end{bmatrix}, \quad v = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} \quad \text{in COO form:}$$

$$AA = [\lambda_{11}, \lambda_{21}, \lambda_{22}, \lambda_{32}, \lambda_{33}]$$

$$JR = [1, 1, 2, 3, 3]$$

$$JC = [1, 1, 2, 2, 3]$$

We then count the number of nonzero elements in each row and take the max to determine the number of entries per row in COEF and JCOEF:

$$\text{COEF} = \begin{bmatrix} \lambda_{11} & 0 \\ \lambda_{21} & \lambda_{22} \\ \lambda_{32} & \lambda_{33} \end{bmatrix}, \quad \text{JCOEF} = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 2 & 3 \end{bmatrix}$$

where JCOEF is the associated column index for λ_{ij}.

Then we compute $z = A \cdot v$ by

$$z(i) = \sum_{j=1}^{N_d} \text{COEF}(i,j) \cdot v(\text{JCOEF}(i,j)) \tag{14}$$

Then

$$z = \begin{bmatrix} \lambda_{11} v_1 \\ \lambda_{21} v_1 + \lambda_{22} v_2 \\ \lambda_{32} v_2 + \lambda_{33} v_3 \end{bmatrix}$$

3.4 Unordered Row COO to CSR

Let

$$A = \begin{bmatrix} \lambda_{11} & 0 & 0 \\ 0 & \lambda_{22} & \lambda_{23} \\ \lambda_{31} & 0 & \lambda_{33} \end{bmatrix}, \quad V = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

and

$$AA = [\lambda_{22}, \lambda_{31}, \lambda_{11}, \lambda_{23}, \lambda_{33}]$$
$$JR = [2, 3, 1, 2, 3]$$
$$JC = [1, 1, 1, 3, 3]$$

Reordering $(JR(k), JC(k), AA(k))$ so row order index are increasing order:

$$(JR', JC', AA') = [(1, 1, \lambda_{11}), (2, 2, \lambda_{22}), \ldots, (3, 3, \lambda_{33})]$$

Then we build IA such that $IA(1) = 1$ and

$$IA(i+1) = IA(i) + \#\{k : JR'(k) = i\}$$

then

$$IA = [1, 2, 4, 6]$$

and

$$AA_{\text{CSR}} = [\lambda_{11}, \lambda_{22}, \lambda_{23}, \lambda_{31}, \lambda_{33}]$$
$$JA_{\text{CSR}} = [1, 2, 3, 1, 3]$$

Then,

$$Z(i) = \sum_{k=IA(i)}^{IA(i+1)-1} AA_{\text{CSR}}(k) \cdot V(JA_{\text{CSR}}(k))$$

$$Z = \begin{bmatrix} \lambda_{11} v_1 \\ \lambda_{22} v_2 + \lambda_{23} v_3 \\ \lambda_{31} v_1 + \lambda_{33} v_3 \end{bmatrix}$$

3.5 Symmetric CSR storage algorithm

$B \in R^{n \times n}, \quad B = B^T$ s.t. $\lambda_{ij} = \lambda_{ji}$

where we will only store elements below and including the diagonal

$$\lambda_{ij} = e_i^T B e_j, \quad i \geq j \tag{15}$$

$$B = \begin{bmatrix} \lambda_{11} & \lambda_{12} & \lambda_{13} \\ \lambda_{21} & \lambda_{22} & \lambda_{23} \\ \lambda_{31} & \lambda_{32} & \lambda_{33} \end{bmatrix} \tag{16}$$

to build the CSR arrays.

We will store the elements $\lambda_{ij},\ i \geq j$ into AA and put their corresponding row and column index into JR and JC.

Then we will compute IA similar as before and perform the matrix-vector multiplication.

Let

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} \tag{17}$$

then $B\mathbf{v} \to \mathbf{z}$ s.t.

$$z(i) = z(i) + \lambda_{ij} \cdot v(j) \tag{18}$$

Then

$$\mathbf{z} = \begin{bmatrix} z_1 = \lambda_{11}v_1 + \lambda_{12}v_2 + \lambda_{13}v_3 \\ z_2 = \lambda_{21}v_1 + \lambda_{22}v_2 + \lambda_{23}v_3 \\ z_3 = \lambda_{31}v_1 + \lambda_{32}v_2 + \lambda_{33}v_3 \end{bmatrix} \tag{19}$$

3.6 Compressed Diagonal Data Structure

$C \in R^{n \times n}$ s.t. C has nonzero elements on 3 diagonals - the main diagonal and then a sub and super diagonal, offset by k.

Main diagonal:

$$\lambda_{ii} \tag{20}$$

Subdiagonal:

$$\lambda_{ij}, \quad j = i - k \tag{21}$$

Superdiagonal:

$$\lambda_{ij}, \quad j = i + k \tag{22}$$

$$C = \begin{bmatrix} \lambda_{11} & \lambda_{12} & 0 & 0 & 0 \\ \lambda_{21} & \lambda_{22} & \lambda_{23} & 0 & 0 \\ 0 & \lambda_{32} & \lambda_{33} & \lambda_{34} & 0 \\ 0 & 0 & \lambda_{43} & \lambda_{44} & \lambda_{45} \\ 0 & 0 & 0 & \lambda_{54} & \lambda_{55} \end{bmatrix} \tag{23}$$

We first find K by finding the distance between the main diagonal and the next nonzero diagonal:

$$K = \min\{|i - j| : \lambda_{ij} \neq 0 \text{ and } i \neq j\} \tag{24}$$

We then store C into 2 arrays, Diag with dimensions $n \times 3$ and IOFF with dimensions 1×3 s.t.:

$$\text{Diag} = \begin{bmatrix} 0 & \lambda_{11} & \lambda_{12} \\ \lambda_{21} & \lambda_{22} & \lambda_{23} \\ \lambda_{32} & \lambda_{33} & \lambda_{34} \\ \lambda_{43} & \lambda_{44} & \lambda_{45} \\ \lambda_{54} & \lambda_{55} & 0 \end{bmatrix} \tag{25}$$

$$\text{IOFF} = [-1, 0, 1] \tag{26}$$

Then, to compute $z \leftarrow Cv$ we compute

$$z(i) = \sum_{j=1}^{3} \text{Diag}(i, j) \cdot v\left(i + \text{IOFF}(j)\right) \tag{27}$$

to produce

$$z = \begin{bmatrix} z_1 = \lambda_{11}v_1 + \lambda_{12}v_2 \\ z_2 = \lambda_{21}v_1 + \lambda_{22}v_2 + \lambda_{23}v_3 \\ z_3 = \lambda_{32}v_2 + \lambda_{33}v_3 + \lambda_{34}v_4 \\ z_4 = \lambda_{43}v_3 + \lambda_{44}v_4 + \lambda_{45}v_5 \\ z_5 = \lambda_{54}v_4 + \lambda_{55}v_5 \end{bmatrix} \tag{28}$$

4 Description of the Algorithms and Implementation

4.1 Task 1 (a): In Row Order COO to CSR, Matrix-Vector Product

The first algorithm converts a sparse matrix from in row order coordinate form to compressed sparse row formatting, which it then uses to prefrom the matrix vector product. The row pointers array, IA, is computed by counting the nonzero elements in each row using JR as it index. Then the dot product is applied between elements of AA and the corresponding vector,v, being indexed by JC.

Listing 1: Algorithm 1

```
%Task 1 part 1

function[z] = COO_to_CSR_Mv_mult(AA,JR,JC,v,n)

  z=zeros(n,1);

  %Gather elements of IA
  IA(1)=1;
  for i=1:n
       rowsum = 0;
       for j=1:length(JR)
           if JR(j) == i
               rowsum = rowsum+1;
           end
       end
       IA(i+1) = IA(i)+rowsum;
  end

  %preforms multiplication of Mv using dot product method from notes
  for i = 1:n
      k_1 = IA(i);
      k_2 = IA(i+1)-1;
      z(i) = dot(AA(k_1:k_2),v(JC(k_1:k_2)));
  end

  disp(AA)
  disp(JC)
  disp(IA)

return

%Works during testing
```

4.2 Task 1 (b): In Row Order COO to Modified CSR, Matrix-Vector Product

The second algorithm converts a sparse matrix from coordinate form into a modified CSR data structure. We begin by separating the entries of our matrix into diagonal and non diagonal elements. We then construct a row pointer array indicating where each non diagonal element begins in our structure. We then concatenate these arrays together to from our modified AA and JA containing our elements, row pointers, and column index. We then use this to compute the matrix-vector product, optimizing storage for sparse matrix multiplication.

Listing 2: Algorithm 2

```
%Task 1 part 2

function [z] = COO_to_modified_CSR_Mv_mult(AA, JR, JC, v, n)

diagonal_elements = zeros(1, n);
nondiagonal_elements = [];
```

```
nondiagonal_elements_columns = [];
rowpointers = zeros(1, n + 1);
z = zeros(n, 1);

% input diagonal and non-diagonal elements (and associated columns)
for i = 1:length(AA)
    if JR(i) == JC(i)
        diagonal_elements(JR(i)) = AA(i);
    else
        nondiagonal_elements = [nondiagonal_elements, AA(i)];
        nondiagonal_elements_columns = [nondiagonal_elements_columns, JC(i)];
    end
end

%find row pointers and fill array
p = 1;
for i = 1:n
    rowpointers(i) = p;
    p = p + sum(JR ~= JC & JR == i); % Count non-diagonal elements for row i
end
rowpointers(n + 1) = p;

% Append arrays to form modified CSR storage
modified_AA = [diagonal_elements, 0, nondiagonal_elements];
JA = [rowpointers, nondiagonal_elements_columns];

disp(modified_AA)
disp(JA)

% Matrix Vector Mult
for i = 1:n
    z(i) = modified_AA(i)*v(i);
    for j = JA(i):JA(i+1)-1
        col_index = JA(n+1+j);
        z(i) = z(i) + modified_AA(j+n+1)*v(col_index);
    end
end

end
```

4.3 Task 1 (c): In Row Order COO to Ellpack-Itpack, Matrix-Vector Product

The third part of the first task is to convert our sparse matrix from Coordinate from to Ellpack-itpack form and the compute the matrix vector multiplication. The algorithm starts by counting the number of non zero elements in each row and finding the maximum of the between each row. This determine the Column size of COEF and JCOEF. COEF holds the nonzero values of our matrix and JCOEF is the corresponding column index. The algorithm then computes the product by iterating through the rows and summing the product of the coefficient and the vector element.

Listing 3: Algorithm 3

```
%Task 1 part 3

function [z] = COO_to_ELL_Mv_mult(AA,JR,JC,v,n)

z=zeros(n,1);
rowcounter = zeros(n,1);
rowpointer = zeros(n,1);

% Determine # nonzero elements in A
```

```
for i =1:length(JR)
    rowcounter(JR(i)) = rowcounter(JR(i)) + 1;
end

num_nonzeros = max(rowcounter);
COEF = zeros(n,num_nonzeros);
JCOEF = zeros(n, num_nonzeros);

%Determine elements of COEF & JCOEF
for k=1:length(AA)
    i = JR(k);
    j = rowpointer(i) +1;
    COEF(i,j) = AA(k);
    JCOEF(i,j) = JC(k);
    rowpointer(i) = j;
end

%Matrix Vector mult
for i = 1:n
    for j = 1:num_nonzeros
        if JCOEF(i,j) > 0
            z(i) = z(i) + COEF(i,j)*v(JCOEF(i,j));
        end
    end
end

disp(COEF)
disp(JCOEF)
end
```

4.4 Task 2: Unordered Row CSR, Matrix-Vector Product

For the second task, our algorithm converts a sparse matrix stored in COO form to A compressed sparse row structure with elbow room, then computes the matrix-vector product. First we organize the row indices in increasing order and builds the row pointer array, IA, indicating the starting row for the values. then the CSR array is build for AA and JC. after each row extra space is added for new elements that get inserted which is the relaxed version of our AA, JR, and JC arrays. then using the dot product between the rows corresponding value and elements of v.

Listing 4: Algorithm 4

```
function [z] = COO_to_relaxed_CSR_Mv_mult(AA,JR,JC, v, n, elbow)

    sorted_JR = sort(JR);
    IA = zeros(n+1,1);
    IA(1) = 1;

    for i = 1:n
        rowsum = sum(sorted_JR == i);
        IA(i+1) = IA(i) + rowsum;
    end

    AA_CSR = zeros(length(AA),1);
    JC_CSR = zeros(length(JC),1);
    p = 1;
    for i = 1:n
        for j = 1:length(JR)
            if JR(j) == i
                AA_CSR(p) = AA(j);
                JC_CSR(p) = JC(j);
```

```
                p = p + 1;
            end
        end
    end

    elbowroom = n * elbow;
    AA_relaxed = zeros(length(AA_CSR) + elbowroom, 1);
    JC_relaxed = zeros(length(JC_CSR) + elbowroom, 1);
    IA_relaxed = zeros(n+1,1);
    IA_relaxed(1) = 1;

    h = 1;
    for i = 1:n
        k1 = IA(i);
        k2 = IA(i+1)-1;
        rowlength = k2 - k1 + 1;

        AA_relaxed(h:h+rowlength-1) = AA_CSR(k1:k2);
        JC_relaxed(h:h+rowlength-1) = JC_CSR(k1:k2);

        h = h + rowlength + elbow;
        IA_relaxed(i+1) = h;
    end
    IA = IA_relaxed;

    for i = 1:n
        k1 = IA(i);
        k2 = IA(i+1)-elbow-1;
        if k1 <= k2
            z(i) = dot(AA_relaxed(k1:k2), v(JC_relaxed(k1:k2)));
        end
    end
end
```

4.5 Task 3: Symmetric CSR, Matrix-Vector Product

This algorithm takes a symmetric matrix A and converts it into CSR formatting by first scanning over the lower triangular portion of the matrix and storing those nonzero values in AA along with its associated index in JR and JC. Doing this is important as we now do not have to store as many values as in a non symmetric matrix with the same number of elements, helping improve our computation efficiency. We next create the row pointer array where IA(i) is the starting position of starting element in row i. Then the matrix vector product is computed by iterating through each row's stored value and for each value, adds $a_{ij}v_j$ to z_i. the mirrored contribution are also added in as well ensuring our calculation run properly without having to store them individually.

Listing 5: Algorithm 5

```
%Task 3 works - Do not mess with it

function [z] = Task3(A, v, n)

  AA = [];
  JR = [];
  JC = [];
  for i = 1:n
    for j = 1:i
      if A(i,j) ~= 0
        AA(end+1) = A(i,j);
        JR(end+1) = i;
        JC(end+1) = j;
```

```
      end
    end
  end

  %Converts to CSR Form
   IA(1)=1;
   for i=1:n
        rowsum = 0;
        for j=1:length(JR)
            if JR(j) == i
                rowsum = rowsum+1;
            end
        end
        IA(i+1) = IA(i)+rowsum;
   end


  z = zeros(n,1);
  for i = 1:n
    k1 = IA(i);
    k2 = IA(i+1) - 1;

    for k = k1:k2
      j = JC(k);
      aij = AA(k);
      z(i) = z(i) + aij * v(j);

      if i ~= j
        z(j) = z(j) + aij * v(i);

      end
    end
  end

  disp(IA)
return
```

4.6 Task 4: Compressed Diagonal, Matrix-Vector Product

The final algorithm assumes A has a banded structure with non zero elements on the main diagonal and sub/super diagonals with some offset value, k. We first create a matrix to hold our nonzero diagonal elements and determine the values of k, storing that into its own array. Afterwards we use this now extracted diagonal data structure and its associated offset value to find the product. For each row i and diagonal offset in IOFF, multiply the element of Diag(i,j) by v indexed by i + IOFF(j).

Listing 6: Algorithm 6

```
%Task 4 works, dont mess with it

function [z] = task4(A,v,n)

    Diag = zeros(n, 3);

    k = 0;
    for i = 1:n-1
        if A(i, i+1) ~= 0
            k = i;
            break;
        elseif A(i+1, i) ~= 0
            k = i;
```

```
            break;
        end
    end

    IOFF = [-k, 0, k];

    for i = 1:n
        Diag(i, 2) = A(i, i);
    end

    for i = (k+1):n
        Diag(i, 1) = A(i, i-k);
    end

    for i = 1:(n-k)
        Diag(i, 3) = A(i, i+k);
    end

    disp(Diag);
    disp(IOFF);

    z = zeros(n,1);
    for i =1:n
        for j = 1:length(IOFF)
            if i + IOFF(j) >= 1 && i + IOFF(j) <= n
                z(i) = z(i) + Diag(i,j) * v(i + IOFF(j));
            end
        end
    end

    disp("A*v done by Matlab");
    disp(A*v);
return
```

5 Testing and Results

To ensure the algorithms work properly we need to test them on a large number of cases for sufficiently large n. In this project we were given the testing file for task 1. We were then instructed to test Tasks 2,3, and 4 in a similar way. During testing we are looking at computational efficiency (timing) of our algorithms and seeing how they fair when then number of dimensions in large. Additionally we will use the relative error to test our algorithms against the prebuilt Matlab functions. Relative error := $\frac{\|x_{\text{comp}} - x_{\text{true}}\|}{\|x_{\text{true}}\|}$

5.1 Task1

Task 1 successfully output the correct vector for each predefined data structure in COO Form to the required data structure. NOTE: The name "COO to modified CSR conversion" is used twice. The second iteration is actually converting to Ellpack-Itpack.

Figure 1: Task 1: Matrix 1 and Vector 1

```
>> Task1_tester
Perform the multiplication of Matrix 1 and Vector 1 using the COO to CSR conversion
    10    -1     3    11     5    12     7    13     9     2     3

     1     5     1     2     2     3     3     4     4     1     5

     1     3     5     7     9    12

     8
     3
   -12
    -7
     8

Perform the multiplication of Matrix 1 and Vector 1 using the COO to modified CSR conversion
    10    11    12    13     3     0    -1     3     5     7     9     2

     1     2     3     4     5     7     5     1     2     3     4     1

     8
     3
   -12
    -7
     8

Perform the multiplication of Matrix 1 and Vector 1 using the COO to modified CSR conversion
    10    -1     0
     3    11     0
     5    12     0
     7    13     0
     9     2     3

     1     5     0
     1     2     0
     2     3     0
     3     4     0
     4     1     5

     8
     3
   -12
    -7
     8
```

Figure 2: Task 1: Matrix 2 and Vector 2

```
>> Task1_tester
Perform the multiplication of Matrix 2 and Vector 2 using the COO to CSR conversion
     1     5    -2     4     1     2     1     2     3

     1     4     1     2     1     4     2     3     4

     1     3     5     7    10

    -2
    14
     1
     0

Perform the multiplication of Matrix 2 and Vector 2 using the COO to modified CSR conversion
     1     4     0     3     0     5    -2     1     2     1     2

     1     2     3     5     7     4     1     1     4     2     3

    -2
    14
     1
     0

Perform the multiplication of Matrix 2 and Vector 2 using the COO to modified CSR conversion
     1     5     0
    -2     4     0
     1     2     0
     1     2     3

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

    -2
    14
     1
     0
```

5.2 Task 2

During testing for the second task we were requested to insert padding into our algorithm and then recompute our results. Through validation the time needed to compute the dense matrix using Matlab's built in function has a faster runtime then our relaxed algorithm. After insertion of an element we additionally notice that our relative error goes up as well on average. This can be associated to more operations being computed so this checks out.

Figure 3: Task 2: Speed Comparison and Initial/Post Insertion Error

```
Testing Relaxed CSR Matrix-Vector Multiplication
Matrix Size:
   100

  Initial product relative error: 2.23e-17
  After insertion relative error: 1.04e-16
Warning: The measured time for F may be inaccurate because it is running too fast. Try measuring something that takes longer.
> In timeit (line 158)
In Task2_testerfile (line 58)
Dense Time:
   4.5693e-07

Relaxed Time:
   1.9655e-04

Matrix Size:
   500

  Initial product relative error: 1.25e-16
  After insertion relative error: 1.45e-16
Dense Time:
   8.6220e-06

Relaxed Time:
    0.0041

Matrix Size:
        1000

  Initial product relative error: 1.40e-16
  After insertion relative error: 1.51e-16
Dense Time:
   9.7181e-06

Relaxed Time:
    0.0215
```

5.3 Task 3

Once again we notice that the algorithm works slower than the Matlab function for dense matrices. This is especially obvious when n = 1000 and our difference in computational efficiency is extreme. We produce a good error for large n verifying our accuracy measurements in product the correct output vector.

Figure 4: Task 3: Speed Comparison and Relative Error

```
>> TaskThreeTesting
Testing Symmetric CSR Matrix-Vector Multiplication
Matrix Size:
   100

  Relative Error (initial): 8.19e-17
Warning: The measured time for F may be inaccurate because it is running too fast. Try measuring something that takes longer.
> In timeit (line 158)
In TaskThreeTesting (line 40)
Dense Time:
   4.4776e-07

Symmetric Time:
   4.2250e-05

Matrix Size:
   500

  Relative Error (initial): 7.48e-17
Dense Time:
   1.0674e-05

Symmetric Time:
   7.0105e-04

Matrix Size:
        1000

  Relative Error (initial): 1.20e-16
Dense Time:
   3.5436e-05

Symmetric Time:
    0.0036
```

5.4 Task 4

During Validation for the fourth routine we notice some errors appear. Our algorithm works well for large cases of n when the diagonal offset is 1. We have a low relative error and as usual, our algorithm is slower than the Matlab function for each cases of k. When k is greater than 1, the algorithm breaks and does not produce the correct vector product. This is due to some issue in the algorithm.

Figure 5: Task 4: Speed Comparison and Relative Error with k Bandwidth

```
Matrix Size:
        1000

Band offset:
     1

    Relative Error: 8.44e-17
Dense Time:
   9.8348e-06

Task 4 Time:
   1.1623e-05

Band offset:
     5

    Relative Error: 1.43e+00
Dense Time:
   9.7358e-06

Task 4 Time:
   1.6023e-05

Band offset:
    10

    Relative Error: 1.44e+00
Dense Time:
   3.0498e-05

Task 4 Time:
   1.5901e-05
```

6 Conclusions

Through validation we can conclude that we have most of the algorithms working, except for last algorithm that does not produce the correct vector product when k is greater than 1 for the diagonal offset. We also noticed that even though we had very low relative error for our working algorithms, we were not able to beat the speed at which the prebuilt Matlab function can operate at. This is most likely due to the fact that we are not correctly allocating our variable for speed in our program. Something I would try to correct with more time.

7 Programming Files

Task 1 includes 4 files: 3 files for the functions and 1 provided for testing. The rest of the tasks are also separated into a file for the function and an associated testing file