FCM Report

Ahmer Nadeem Khan

16 September 2024

1 Executive Summary

For this assignment, we will be creating sub-routines and analyzing the algorithms for basic matrix-vector and matrix-matrix multiplications, assuming certain structure on our matrices which will be exploited to improve time and space complexities of the algorithms. We will also consider using different data structures for one of the problem. For the flow of our solutions, we will first derive the algorithms mathematically, then provide concrete code in C++ and Python. The code in C++ will only be used for algorithm analysis, and basic validation with toy examples, and comprehensive testing drivers and reporting will be done in Python using the functionalities of the NumPy and Matplotlib libraries.

2 Statement of the Problems

We will state here the 4 subroutines that we have to compute:

- The first subroutine takes a unit lower triangular matrix L stored in a 2D array and an arbitrary vector v stored in a 1D array and computes the output, $w \leftarrow Lv$.
- The second subroutine takes a unit lower triangular matrix L stored in a 1D array, and and an arbitrary vector v stored in a 1D array and computes the output, $w \leftarrow Lv$.
- The third subroutine takes a banded unit lower triangular matrix L stored in a 1D array, and an arbitrary vector v stored in a 1D array and computes the output, $w \leftarrow Lv$.
- The fourth subroutine takes a unit lower triangular matrix L and an upper triangular matrix U, both stored in a single 2D array, and outputs the matrix product $M = LU$.

2.1 Note on Storage and Languages

Our initial algorithms are run on C++, where the storage patterns and control flow of the algorithm is most obvious. In subroutine 1, when we initialize a static and local 2D array, the values are automatically garbage. We then complete our user initialization of a lower triangular matrices in the specific required indexes, using $\frac{n(n+1)}{2}$ storage locations. We consider the other garbage values "undefined" and consider it free store. We demonstrate this effect in the source file.

Storing 1D arrays does not have this complication as we pre-compute the size of the required arrays, and only use as many storage locations. Since size must be defined and be constant as good practice, we manually change the size variables (RDIM and CDIM, for example) to test and validate.

In Python, we use the NumPy array structures to reproduce the algorithm, and here we again start with uninitialized 2D arrays, and then only specify the values that we want. The NumPy arrays are close to C++ arrays in their implementation. The reason we translate our code to Python is because of the ease of testing using standard libraries and plotting errors, graphs and tables.

3 Mathematical Background and Algorithms

These section should be used with the C++ source file, where additional comments explain algorithmic choices. Each algorithm uses the inner product method to compute products as much as possible. These loops are obvious in the below algorithms.

Algorithm 1 Unit Lower Triangular Matrix-Vector Multiplication

```
1: **Input:** Lower triangular matrix $L$ of size $n \times n$, vector $v$ of size $n$
2: **Output:** Vector $w$ of size $n$, the result of $L \times v$
3: Initialize $w[0] \leftarrow v[0]$
4: for $i \leftarrow 1$ to $n-1$ do
5:     Initialize $w[i] \leftarrow 0$
6:     for $j \leftarrow 0$ to $i-1$ do
7:         $w[i] \leftarrow w[i] + L[i][j] \times v[j]$
8:     end for
9:     $w[i] \leftarrow w[i] + v[i]$
10: end for
```

Algorithm 2 Unit Lower Triangular Matrix-Vector Multiplication (Compressed Row Form)

```
1: Input: Lower triangular matrix $L\_compressed$ stored in 1D array (compressed row form), vector $v$ of size $n$
2: Output: Vector $w$ of size $n$, the result of $L \times v$
3: Initialize $w[0] \leftarrow v[0]$
4: Initialize $matrix\_index \leftarrow 0$
5: for $row \leftarrow 1$ to $n - 1$ do
6:     Initialize $w[row] \leftarrow 0$
7:     Initialize $current\_index \leftarrow matrix\_index$
8:     for $col \leftarrow 0$ to $row - 1$ do
9:         $w[row] \leftarrow w[row] + L\_compressed[current\_index] \times v[col]$
10:        $current\_index \leftarrow current\_index + 1$
11:        $matrix\_index \leftarrow matrix\_index + 1$
12:    end for
13:    $w[row] \leftarrow w[row] + v[row]$
14: end for
```

Algorithm 3 Unit Lower Banded Matrix-Vector Multiplication (Compressed Row Form)

```
1: Input: Lower banded matrix $L\_banded$ stored in 1D array (compressed row form), vector $v$ of size $n$
2: Output: Vector $w$ of size $n$, the result of $L \times v$
3: Initialize $w[0] \leftarrow v[0]$                ▷ Row 1 is just identity: $[1, 0, 0, \ldots]$
4: Initialize $w[1] \leftarrow L\_banded[0] \times v[0]$
5: Initialize $index \leftarrow 1$
6: for $i \leftarrow 2$ to $n - 1$ do
7:     $w[i] \leftarrow 0$
8:     $w[i] \leftarrow w[i] + L\_banded[index] \times v[i - 2]$      ▷ First sub-diagonal element
9:     $w[i] \leftarrow w[i] + L\_banded[index + 1] \times v[i - 1]$      ▷ Second sub-diagonal element
10:    $index \leftarrow index + 2$
11: end for
12: for $i \leftarrow 1$ to $n - 1$ do
13:    $w[i] \leftarrow w[i] + v[i]$        ▷ Add the vector back because of the 1s diagonal
14: end for
```

Algorithm 4 Multiplication of Unit Lower Triangular Matrix and Upper Triangular Matrix (Stored in One 2D Array)

```
1: Input: Combined matrix $LU$ of size $n \times n$, where $L$ is unit lower triangular
   and $U$ is upper triangular, stored in a single 2D array.
2: Output: Matrix $M$ of size $n \times n$, the result of $L \times U$
3: Initialize $M$ as a zero matrix of size $n \times n$          ▷ Output matrix
4: Step 1: Compute the first row and first column
5: for $i \leftarrow 0$ to $n-1$ do
6:     $M[0][i] \leftarrow LU[0][i]$          ▷ First row is directly from $U$
7: end for
8: for $i \leftarrow 1$ to $n-1$ do
9:     $M[i][0] \leftarrow LU[i][0] \times LU[0][0]$          ▷ First column involves only $L$ and $U$
10: end for
11: Step 2: Compute the remaining entries of $M$
12: for $i \leftarrow 1$ to $n-1$ do
13:     for $j \leftarrow 1$ to $n-1$ do
14:         $min\_limit \leftarrow \min(i, j)$
15:         $index \leftarrow 0$
16:         for $counter \leftarrow 0$ to $min\_limit$ do
17:             $M[i][j] \leftarrow M[i][j] + LU[i][counter] \times LU[counter][j]$
18:             $index \leftarrow index + 1$
19:         end for
20:         if $i \leq j$ then
21:             $M[i][j] \leftarrow M[i][j] + LU[index][j]$          ▷ Handle diagonal element
   multiplication with implicit 1 for $L$
22:         end if
23:     end for
24: end for
```

The first algorithm computes the product $w = L \cdot v$, where L is a unit lower triangular matrix stored in a 2D array, and v is a vector. The result vector w is computed as:

$$w[i] = \sum_{j=0}^{i-1} L[i][j] \cdot v[j] + v[i] \quad \text{for } 1 \leq i < n$$

The diagonal elements of L are assumed to be 1, so the corresponding element of v is added to account for this.

The second algorithm computes the product $w = L \cdot v$, where L is a unit lower triangular matrix stored in compressed row format (1D array), and v is a vector. The result vector w is computed as:

$$w[i] = \sum_{j=0}^{i-1} L[k] \cdot v[j] + v[i] \quad \text{for } 1 \leq i < n$$

The diagonal elements are assumed to be 1, and k is the index in the compressed row storage, updated as required.

The third algorithm computes the product $w = L \cdot v$, where L is a unit lower banded matrix with two sub-diagonals below the main diagonal (stored in a 1D compressed row format), and v is a vector. The result vector w is computed as:

$$w[i] = L_{\text{banded}}[i-2] \cdot v[i-2] + L_{\text{banded}}[i-1] \cdot v[i-1] + v[i] \quad \text{for } 2 \leq i < n$$

For $i = 0$ and $i = 1$, the matrix is handled manually due to the banded structure, and these are pre-determined.

The fourth algorithm computes the product $M = L \cdot U$, where both L (unit lower triangular) and U (upper triangular) are stored in a single 2D array. The matrix multiplication is computed as:

$$M[i][j] = \sum_{k=0}^{\min(i,j+1)} A[i][k] \cdot A[k][j] + A[i][j] \quad \text{for } i \leq j$$

Here, LU stores both L and U in a single 2D array. The diagonal elements of L are treated implicitly as 1, simplifying the computation for each entry in M.

4 Testing

Testing has been done extensively on the Jupyter notebook, where the graphs are also available!