UNIVERSIDADE ESTADUAL DE CAMPINAS

Faculdade de Engenharia Civil, Arquitetura e Urbanismo

IC639: Métodos Numéricos para Engenharia Civil

List 3

Decomposition and Storage of Matrices

Student:

Carlos Henrique Chama Puga - 195416

Advisors:

Porf. Dr. Philippe Devloo

Dr. Giovane Avancini

Campinas

2024

Contents

1 Introduction

In mathematics, physics, and many engineering fields, the need to find the solution of a linear system of equations appears. Given a $N \times N$ linear system of the form of Eq. (1.1)

$$Ax = b, \tag{1.1}$$

in which the matrix A is the coefficient matrix, and the vector b is the equation's right-hand side, the solution vector x can be easily found by using the inverse of the matrix A as shown in Eq. (1.2)

$$x = A^{-1}b. \tag{1.2}$$

The task of inverting a matrix, however, is computationally expensive, and as the problem grows in size, the computational cost increases exponentially. In this context, the decomposition method arises to ease the computational burden of finding the inverse of a matrix.

This is justified by the fact that inverting the decomposed form $A = LU$ is computationally cheaper than inverting the original matrix A. The final solution is then obtained by solving two systems as shown in Eq. (1.3)

$$\begin{cases} Ax = LUx = b, \\ Ly = b, \\ Ux = y \end{cases} . \tag{1.3}$$

LU, LDU, LLt (or Cholesky), and LDLt decompositions are examples of this methodology, to cite a few.

This work will focus on the LU and LDLt methods, but the main idea behind

all of them is to decompose the original matrix A into two triangular matrices, L (lower triangular), and U (upper triangular). The D stands for a diagonal matrix and might appear in the LDU and LDLt methods. Finally, for the Cholesky and LDLt decompositions, the upper matrix is the transpose of the lower matrix, $U = L^t$, which is achieved due to the symmetry of the problem.

To decompose a matrix, in this work, the rank one update is employed. The rank one update is a method to update a matrix by adding the outer product between the vectors of the line and column of a given equation to the original submatrix formed by all lines and columns below the assessed equation.

This procedure is repeated until the last equation. Once it is over, the resulting matrix is the decomposed form of the original one. The matrix L is the lower triangle of the decomposed matrix (not considering the diagonal) and the matrix U is the upper triangle of the decomposed matrix (including the diagonal). For the cases in which the diagonal matrix is required, the diagonal of the decomposed matrix is the matrix D and U does not include the diagonal.

One problem that might happen is when the pivot is zero. In these cases, the rank one update can not be executed since the division by zero is undetermined. To overcome this issue, the pivoting procedure is employed. It consists of swapping lines and/or columns to ensure that the pivot is not zero. The pivoting procedure is essential to guarantee the convergence of the decomposition method.

Pivoting can be done in two ways: partial pivoting, in which only lines or columns are swapped and full pivoting, in which both lines and columns are swapped. This work focuses on implementing the full pivoting procedure for both LU and LDLt decompositions.

This work also covers structures to storage matrices. Some forms of com-

putational storage matrices include the full matrix, in which all elements are stored, a symmetric matrix, where only the upper or lower triangle is stored, band matrix, where only the diagonal and few elements close to it are stored (the band is more properly the matrix type rather than the storage method itself), and the sparse matrix, in which only the non-zero elements are stored.

Elements in a sparse matrix are stored in three vectors, usually called cols, values, and ptr. The cols vector stores which column each element is, the values vector stores the value of each element, and the ptr vector stores the cumulative number of elements per line in the matrix.

Sparse matrices are, allegedly, the most efficient way to store matrices, especially when the system is large and contains relatively low non-zero elements. For this reason, the sparse matrix is used in this work and compared to the full matrix storage method.

In this list, LU and LDLt decompositions are implemented and a sparse matrix class is developed to compare its storage performance. Finally, a method to multiply a matrix by a vector is implemented to test the sparse matrix class and results are compared. The literature used herein can be found in (1, 2, 3, 4)

2 The Full Matrix Class

The FullMatrix class is developed to store and manipulate dense matrices. The class is implemented in Python and has as methods the basic constructor, the SetDecompositionMethod, RankOneUpdtade, Fill_L, Fill_U, Fill_D, Pivot, Lu_Decompostion, LDLT_Decompostion, FindInverse, and Decompose. Each one of these methods is described in the following sections.

The main goal of this class is to decompose a given matrix and compare not only if the decomposition could be done, but whether the decomposition is

correct. The inverse of the matrix is calculated as a way to check the accuracy of the code.

2.1 Class Implementation

The class FullMatrix has the following attributes:

```
@dataclass
class FullMatrix:
    tolerance: ClassVar[float] = 1e-11 # Tolerance for the decomposition

    A: np.array # Matrix
    pivoting: bool = False # Pivoting flag
    diagonal: bool = False # Diagonal flag
    decomposition_type: str = None # Decomposition type
    size: int = field(init=False) # Matrix size

    A_Decomposed: np.array = field(init=False) # Decomposed matrix
    L: np.array = field(init=False) # Lower matrix
    U: np.array = field(init=False) # Upper matrix
    D: np.array = field(init=False) # Diagonal matrix

    detA: float = field(init=False) # Determinant of A
    inverseA: np.array = field(init=False) # Inverse of A
    decomposed_inverse: np.array = field(init=False) # Inverse of A by
    decomposition

    permuted_rows: list = field(default_factory=list) # permuted rows
    permuted_cols: list = field(default_factory=list) # permuted columns

    A_Memory: int = field(init=False, default=0) # Memory usage of the
    matrix
```

Code Listing 1: FullMatrix constructor.

Among the attributes, the most important ones are the matrix A, which storages the full matrix itself, the decomposition_type, responsible for setting which type of decomposition will be used, A_Decomposed, which stores the decom-

posed matrix, L, D, and U, which stores the lower, diagonal and upper decomposed matrices. Notice that regardless of the method, a diagonal matrix is always created but initialized as an Identity matrix.

One cites as well the permutation_rows and permutations_cols attributes, which are used during the pivoting process. The A_Memory attribute stores how much memory is used by the matrix and will be used for further comparisons between full and sparse matrices.

The first method that is implemented is the Decompose method. This function performs the decomposition depending on the decomposition_type attribute. The method is shown in Listing 2.

```
1  def Decompose(self)->None:
2      if self.CheckSingularity():
3          raise Exception("Singular matrix")
4
5      if not self.Check_Symmetry() and (self.decomposition_type == "LDLt"):
6          raise Exception("Matrix not symmetric.")
7
8      if self.pivoting:
9          self.Pivot_Decomposition()
10
11     elif self.decomposition_type == "LU":
12         self.LU_Decomposition()
13
14     elif self.decomposition_type == "LDLt":
15         self.LDLt_Decomposition()
16
17     else:
18         text = f"The '{self.decomposition_type}' decomposition is not valid
     . Please choose one of the following: LU, LDLt "
19         raise Exception(text)
```

Code Listing 2: Decompose method.

In line 2, the method checks if the matrix is singular. If it is, an exception is raised since the decomposition cannot be done. Line 5 checks if the matrix is

symmetric once the LDLt is only valid for symmetric matrices. If the matrix is not symmetric, an exception is raised. Finally, in line 8 the decomposition is performed according to the decomposition_type attribute. If the decompostion type does not match any of the implemented methods, an exception is raised.

Due to their simplicity, LU_Decompostion and LDLt_Decompostion methods are not shown first. Concepts like RankOneUpdate and Fill_L, Fill_U, and Fill_D are explained in the following sections. Then, the Pivot_Decomposition method is commented.

From the implementation standpoint, both LU and LDLt implementations are alike. Basically what is done is to subtract each equation from the submatrix below it, updating the matrix using the rank one update method. Code 3 shows the main idea of the decomposition methods.

```
1  def LU_Decomposition(self)->None:
2      self.RankOneUpdate()
3      self.Fill_L()
4      self.Fill_U()
5
6  def LDLt_Decomposition(self)->None:
7      self.RankOneUpdate()
8      self.Fill_L()
9      self.Fill_D()
10     self.U = self.L.T
```

Code Listing 3: Decomposition method.

Comparing both methods in Code 3, one realizes that the class is implemented in a manner to avoid code repetition. The only difference is that the LDLt method fills the diagonal matrix D with the Fill_D method and the U matrix is the transpose of the L matrix. Code 4 shows the RankOneUpdate method adapted for both LU and LDLt decompositions.

```
1  def RankOneUpdate(self)->None:
2      for i in range(self.size):
```

```
3         matii_correction = 1.0
4
5         if (abs(self.A_Decomposed[i, i]) < self.tolerance):
6             raise Exception("Null Pivot")
7
8         if self.decomposition_type in == "LDLt":
9             matii_correction = self.A_Decomposed[i, i]
10            self.A_Decomposed[i, i+1::] /= self.A_Decomposed[i, i]
11
12        self.A_Decomposed[i+1::, i] /= self.A_Decomposed[i, i]
13        self.A_Decomposed[i+1::, i+1::] -= OUTER(self.A_Decomposed[i+1::, i
    ], self.A_Decomposed[i, i+1::]) * matii_correction
```

Code Listing 4: RankOneUpdate method.

The main difference between the Lu and LDLt rank one update is that, for the former only the column below the pivot is updated, while for the latter both row and column are updated. Due to this fact, for LDLt it is necessary to correct the value of the submatrix formed by the outer product of the row and column vectors (matii_correction = A_Decomposed[i, i], rather than 1).

Line 5 raises the issue when the pivot is null. Since the row and column are divided by it, the rank one update is not possible. To overcome this problem, the Pivot method is implemented. The Pivot_Decompostion method is shown in Code 5.

```
1 def Pivot_Decomposition(self)->None:
2     for index in range(self.size - 1):
3         self.Pivot(index)
4
5         if (abs(self.A_Decomposed[index, index]) < self.tolerance):
6             raise Exception("Null Pivot.")
7
8         matii_correction = 1.0
9
10        if self.decomposition_type == "LDLt":
11            matii_correction = self.A_Decomposed[index, index]
```

```
12              self.A_Decomposed[index, index+1::] /= self.A_Decomposed[index,
    index]
13
14          self.A_Decomposed[index+1::, index] /= self.A_Decomposed[index,
    index]
15          self.A_Decomposed[index+1::, index+1::] -= OUTER(self.A_Decomposed[
    index+1::, index], self.A_Decomposed[index, index+1::]) *
    matii_correction
16
17      self.Fill_L()
18      if self.decomposition_type == "LU":
19          self.Fill_U()
20
21      elif self.decomposition_type == "LDLt":
22          self.Fill_D()
23          self.U = self.L.T
24
25      self.permuted_rows = self.PermutationMatrix(self.permuted_rows)
26      self.permuted_cols = self.PermutationMatrix(self.permuted_cols, rows=
    False)
```

Code Listing 5: Pivot method.

Notice that the rank one update is now evaluated in each iteration to find the pivot, instead of from start to beginning. This ensures that the pivoting will guarantee the convergence of the decomposition method. Line 6 shows the Pivot method being called (see Code 6). Lines 11 - 19 evaluate the rank one updated. Lines 20 - 26 fill the matrices L, D, and U. Finally, lines 28 and 29 create the permutation matrices for rows and columns.

```
1 def Pivot(self, index:int)->None:
2     rows = [i for i in range(self.size)]
3     cols = [i for i in range(self.size)]
4
5     submatrix = self.A_Decomposed[index::, index::].copy()
6
7     row_max, col_max = self.FindMaxRowAndCol(submatrix)
8     row_max += index
```

```
    col_max += index

    rows, cols = self.UpdatedPermutationVector(rows, cols, row_max, col_max
    , index)

    perm_row = self.PermutationMatrix(rows)
    perm_col = self.PermutationMatrix(cols, rows=False)

    self.A_Decomposed = perm_row @ self.A_Decomposed @ perm_col
```

Code Listing 6: Pivot method.

The pivot method is simple and consists of finding the maximum value in the submatrix below the pivot to permute the rows and columns. Special attention is paid to the LDLt decomposition in which the maximum value is found in the diagonal to maintain the symmetry of the matrix. A flag, diagonal, can also be set to force the LU decomposition to look for the maximum value in the diagonal as well.

The Fill_L, Fill_U, and Fill_D methods are simple and are shown in Code 7.

```
def Fill_L(self)->None:
    for i in range(diagonal_aux1, self.size):
        for j in range(i + diagonal_aux2):
            self.L[i, j] = self.A_Decomposed[i, j]

def Fill_U(self)->None:
    diagonal_aux1 = 0 if self.decomposition_type == "LU" else 1 # row used
    to start filling the matrix
    diagonal_aux2 = 1 if self.decomposition_type == "LU" else 0 # column
    used to start filling the matrix

    for j in range(diagonal_aux1, self.size):
        for i in range(j + diagonal_aux2):
            self.U[i, j] = self.A_Decomposed[i, j]

def Fill_D(self)->None:
    for i in range(self.size):
```

```
16          self.D[i, i] = self.A_Decomposed[i, i]
```

Code Listing 7: Fill_L, Fill_U, and Fill_D methods.

Lastly, the FindInverse method, used to compare the inverse of the original matrix and the one obtained after the decomposition, is shown in Code 8.

```
def FindInverse(self)->None:
    if self.pivoting:
        self.decomposed_inverse = self.permuted_rows.T @ INVERSE(self.U) @ INVERSE(self.D) @ INVERSE(self.L) @ self.permuted_cols.T

    elif self.decomposition_type in ["LU", "LDLt"]:
        self.decomposed_inverse = INVERSE(self.U) @ INVERSE(self.D) @ INVERSE(self.L)

    else:
        raise Exception(f"The '{self.decomposition_type}' decomposition is not valid.")
```

Code Listing 8: FindInverse method.

Equations (2.1) and (2.2) display the inverse of the matrix with and without the need for pivoting

$$pR^T.U^{-1}.D^{-1}.L^{-1}.pC^T, \tag{2.1}$$

$$U^{-1}.D^{-1}.L^{-1} \tag{2.2}$$

where pR and pC are the permutation matrices for rows and columns, respectively, and the . operator indicates the matrix multiplication.

The following sections show the results for decompositions LU and LDLt and the comparison between their inverse and the original matrix. The evaluation of the determinant shows that the matrix is not singular, i.e., the determinant is different from zero. Being so, the decomposition can be done. However, due to a zero on the diagonal, pivoting is necessary to avoid division by zero. Matrix

A and its inverse are given in figues 2.1 and 2.2.

816.	419.	0	420.	-1.	282.	232.	398.	65.1	0	421.	2.35	0.0571	-0.14	420.
419.	521.	0	520.	-1.	488.	335.	527.	-9.12	0	521.	2.61	-1.55	-0.182	520.
0	0	0	0	0	0	-1.	-1.	1.	1.	0	0	0	0	0
420.	520.	0	3.33×10^{10}	-1.	486.	337.	528.	-11.6	0	520.	2.58	-0.264	-0.877	1.67×10^{10}
-1.	-1.	0	-1.	0	-1.	-1.	-1.	0	0	-1.	0	0	0	-1.
282.	488.	0	486.	-1.	949.	349.	566.	-86.6	0	485.	2.56	-1.13	0.0777	487.
232.	335.	-1.	337.	-1.	349.	-62000.	-30700.	107000.	0	337.	2.02	0.489	0.555	336.
398.	527.	-1.	528.	-1.	566.	-30700.	-8230.	53400.	0	528.	2.96	0.437	1.5	527.
65.1	-9.12	1.	-11.6	0	-86.6	107000.	53400.	-160000.	0	-12.4	-1.09	-2.	-1.09	-9.9
0	0	1.	0	0	0	0	0	0	0	0	0	0	0	0
421.	521.	0	520.	-1.	485.	337.	528.	-12.4	0	520.	2.61	0.469	-0.183	520.
2.35	2.61	0	2.58	0	2.56	2.02	2.96	-1.09	0	2.61	-0.318	-0.29	0.0279	3.3
0.0571	-1.55	0	-0.264	0	-1.13	0.489	0.437	-2.	0	0.469	-0.29	-0.581	-0.29	-0.814
-0.14	-0.182	0	-0.877	0	0.0777	0.555	1.5	-1.09	0	-0.183	0.0279	-0.29	-0.318	-0.153
420.	520.	0	1.67×10^{10}	-1.	487.	336.	527.	-9.9	0	520.	3.3	-0.814	-0.153	3.33×10^{10}

Figure 2.1: Matrix A.

0.00211	-0.000601	0	0	-0.222	0.000434	-0.0000133	4.9×10^{-6}	-6.4×10^{-6}	-1.97×10^{-6}	-0.00194	0.000798	-0.00249	0.003	0
-0.000601	0.00206	0	0	1.09	-0.00194	0.0000698	-0.000135	1.5×10^{-6}	-0.000067	0.000547	0.498	-0.497	0.496	0
0	0	0	0	0	0	0	0	0	1.	0	0	0	0	0
0	0	0	0	-1.34×10^{-10}	0	0	0	0	0	0	0	0	0	0
-0.222	1.09	0	-1.34×10^{-10}	-508.	-0.126	-0.0289	0.0154	-0.0141	0.000493	-1.73	-7.9	0.00084	-0.185	2.17×10^{-10}
0.000434	-0.00194	0	0	-0.126	0.00211	4.9×10^{-6}	-0.0000133	-1.97×10^{-6}	-6.4×10^{-6}	-0.000601	-0.003	0.00249	-0.000798	0
-0.0000133	0.0000698	0	0	-0.0289	4.9×10^{-6}	0.000148	-0.0000738	0.0000738	5.12×10^{-8}	-0.000135	-0.00063	2.21×10^{-8}	-0.000355	0
4.9×10^{-6}	-0.000135	0	0	0.0154	-0.0000133	-0.0000738	0.000148	4.67×10^{-8}	0.0000738	0.0000698	0.000355	-4.27×10^{-8}	0.00063	0
-6.4×10^{-6}	1.5×10^{-6}	0	0	-0.0141	-1.97×10^{-6}	0.0000738	4.67×10^{-8}	0.000043	0.0000309	-0.000067	-0.000267	-0.0000125	9.25×10^{-6}	0
-1.97×10^{-6}	-0.000067	1.	0	0.000493	-6.4×10^{-6}	5.12×10^{-8}	0.0000738	0.0000309	0.000043	1.5×10^{-6}	-9.25×10^{-6}	0.0000125	0.000267	0
-0.00194	0.000547	0	0	-1.73	-0.000601	-0.000135	0.0000698	-0.000067	1.5×10^{-6}	0.00206	-0.496	0.497	-0.498	0
0.000798	0.498	0	0	-7.9	-0.003	-0.00063	0.000355	-0.000267	-9.25×10^{-6}	-0.496	-3.17	0.000265	-0.277	0
-0.00249	-0.497	0	0	0.00084	0.00249	2.21×10^{-8}	-4.27×10^{-8}	-0.0000125	0.0000125	0.497	0.000265	-0.000259	0.00026	0
0.003	0.496	0	0	-0.185	-0.000798	-0.000355	0.00063	9.25×10^{-6}	0.000267	-0.498	-0.277	0.00026	-3.17	0
0	0	0	0	2.17×10^{-10}	0	0	0	0	0	0	0	0	0	0

Figure 2.2: Matrix A inverse.

Appendix B shows the code for the main function, used throughout this report.

2.2 Matrix LU Decomposition

LU decomposition follows the methods aforementioned. Results for the decomposition can be read in the .txt written by the Python algorithm. The main results for L, U and the error between the original matrix inverse and the decomposed matrix inverse are shown in figures 2.3 - 2.5.

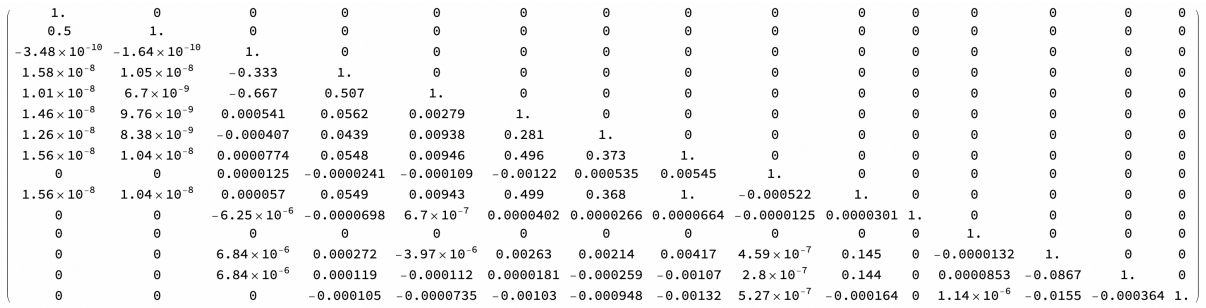

Figure 2.3: Matrix L (LU decomposition).

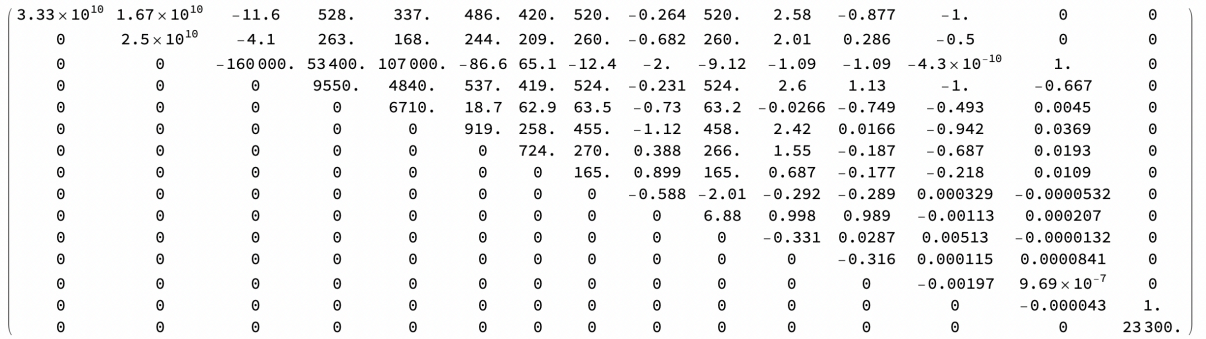

Figure 2.4: Matrix U (LU decomposition).

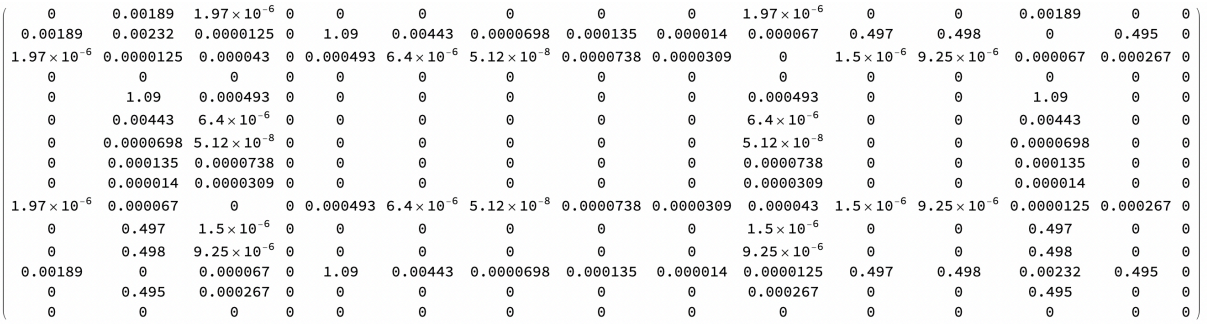

Figure 2.5: Error between the matrix inverses (LU decomposition).

Firstly, the pivot could be found in any position of the matrix. However, the error between the inverses yields relatively high values, as depicted in Fig. 2.5. Seeking to reduce the error, LU decomposition is performed again, but now only pivoting diagonal elements is allowed. Figures 2.6 - 2.8 show the results for the decomposition.

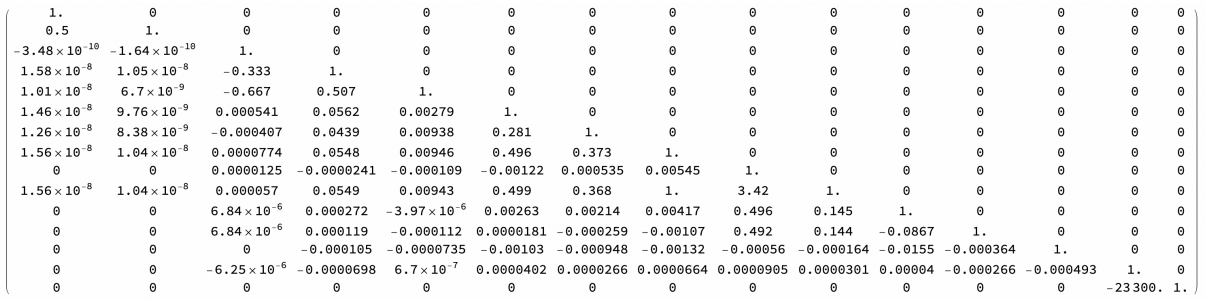

Figure 2.6: Matrix L (LU diagonal pivoting decomposition).

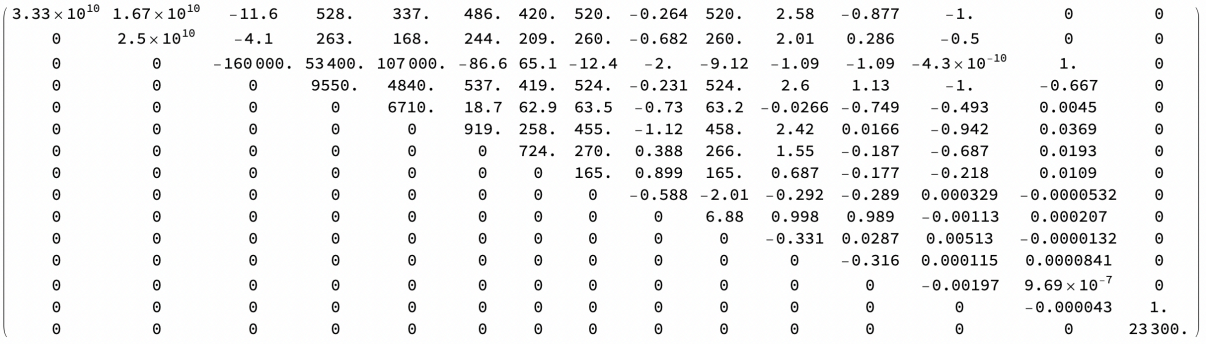

Figure 2.7: Matrix U (LU diagonal pivoting decomposition).

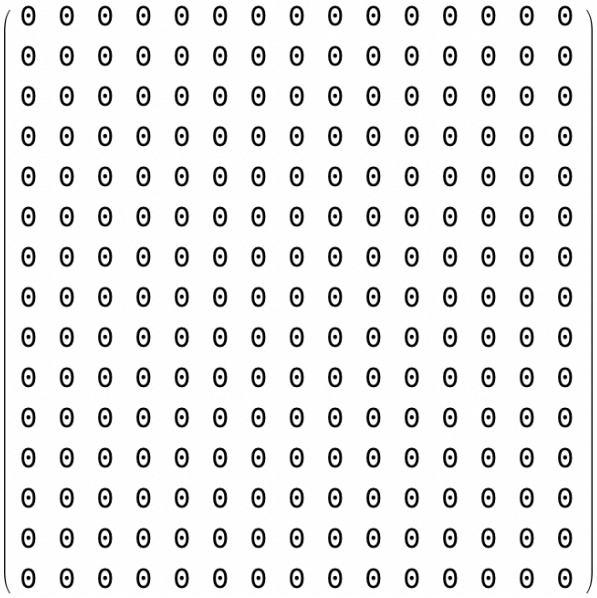

Figure 2.8: Error between the matrix inverses (LU diagonal pivoting decomposition).

A quick comparison between both decompositions shows that matrices L and U pivoting any element have residuals that do not appear when only diagonal elements are pivoted. Results for pivoting only the diagonal elements seem to

be more accurate for this matrice, as the error between the inverses is smaller. Notice that the error matrice is processed by Mathematica using the Chop filter, which sets to zero any value below a certain threshold.

Appendix C shows the permutation matrices for both LU decompositions.

2.3 Matrix LDLt Decomposition

Results for matrices L, D and U are depicted in figures 2.9 - 2.11. T

$$\begin{pmatrix}
1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0.5 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-3.48\times10^{-10} & -1.64\times10^{-10} & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1.58\times10^{-8} & 1.05\times10^{-8} & -0.333 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1.01\times10^{-8} & 6.7\times10^{-9} & -0.667 & 0.507 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1.46\times10^{-8} & 9.76\times10^{-9} & 0.000541 & 0.0562 & 0.00279 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1.26\times10^{-8} & 8.38\times10^{-9} & -0.000407 & 0.0439 & 0.00938 & 0.281 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1.56\times10^{-8} & 1.04\times10^{-8} & 0.0000774 & 0.0548 & 0.00946 & 0.496 & 0.373 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0.0000125 & -0.0000241 & -0.000109 & -0.00122 & 0.000535 & 0.00545 & 1. & 0 & 0 & 0 & 0 & 0 & 0 \\
1.56\times10^{-8} & 1.04\times10^{-8} & 0.000057 & 0.0549 & 0.00943 & 0.499 & 0.368 & 1. & 3.42 & 1. & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 6.84\times10^{-6} & 0.000272 & -3.97\times10^{-6} & 0.00263 & 0.00214 & 0.00417 & 0.496 & 0.145 & 1. & 0 & 0 & 0 & 0 \\
0 & 0 & 6.84\times10^{-6} & 0.000119 & -0.000112 & 0.0000181 & -0.000259 & -0.00107 & 0.492 & 0.144 & -0.0867 & 1. & 0 & 0 & 0 \\
0 & 0 & 0 & -0.000105 & -0.0000735 & -0.00103 & -0.000948 & -0.00132 & -0.00056 & -0.000164 & -0.0155 & -0.000364 & 1. & 0 & 0 \\
0 & 0 & -6.25\times10^{-6} & -0.0000698 & 6.7\times10^{-7} & 0.0000402 & 0.0000266 & 0.0000664 & 0.0000905 & 0.0000301 & 0.00004 & -0.000266 & -0.000493 & 1. & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -23300. & 1.
\end{pmatrix}$$

Figure 2.9: Matrix L (LDLt decomposition).

$$\begin{pmatrix}
3.33\times10^{10} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 2.5\times10^{10} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -160000. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 9550. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 6710. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 919. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 724. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 165. & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -0.588 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 6.88 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -0.331 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -0.316 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -0.00197 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -0.000043 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 23300.
\end{pmatrix}$$

Figure 2.10: Matrix D (LDLt decomposition).

$$\begin{pmatrix}
1. & 0.5 & -3.48\times10^{-10} & 1.58\times10^{-8} & 1.01\times10^{-8} & 1.46\times10^{-8} & 1.26\times10^{-8} & 1.56\times10^{-8} & 0 & 1.56\times10^{-8} & 0 & 0 & 0 & 0 & 0 \\
0 & 1. & -1.64\times10^{-10} & 1.05\times10^{-8} & 6.7\times10^{-9} & 9.76\times10^{-9} & 8.38\times10^{-9} & 1.04\times10^{-8} & 0 & 1.04\times10^{-8} & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1. & -0.333 & -0.667 & 0.000541 & -0.000407 & 0.0000774 & 0.0000125 & 0.000057 & 6.84\times10^{-6} & 6.84\times10^{-6} & 0 & -6.25\times10^{-6} & 0 \\
0 & 0 & 0 & 1. & 0.507 & 0.0562 & 0.0439 & 0.0548 & -0.0000241 & 0.0549 & 0.000272 & 0.000119 & -0.000105 & -0.0000698 & 0 \\
0 & 0 & 0 & 0 & 1. & 0.00279 & 0.00938 & 0.00946 & -0.000109 & 0.00943 & -3.97\times10^{-6} & -0.000112 & -0.0000735 & 6.7\times10^{-7} & 0 \\
0 & 0 & 0 & 0 & 0 & 1. & 0.281 & 0.496 & -0.00122 & 0.499 & 0.00263 & 0.0000181 & -0.00103 & 0.0000402 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1. & 0.373 & 0.000535 & 0.368 & 0.00214 & -0.000259 & -0.000948 & 0.0000266 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0.00545 & 1. & 0.00417 & -0.00107 & -0.00132 & 0.0000664 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 3.42 & 0.496 & 0.492 & -0.00056 & 0.0000905 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0.145 & 0.144 & -0.000164 & 0.0000301 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & -0.0867 & -0.0155 & 0.00004 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & -0.000364 & -0.000266 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & -0.000493 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & -23300. \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.
\end{pmatrix}$$

Figure 2.11: Matrix U (LDLt decomposition).

The error between the original matrix inverse and the decomposed matrix inverse is shown in Fig. 2.12. Appendix C shows the permutation matrices for the LDLt decomposition.

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 2.12: Error between the matrix inverses (LDLt decomposition).

As a way to compare the methods, Table 2.1 shows the norm of the error between the inverse matrices in each decomposition. The norm of the error is calculated as the sum of the absolute values of the difference between the inverse matrices.

Table 2.1: Error between the inverse matrices.

Decomposition	Error
LU	2.77
LU (Diagonal Pivoting)	5.72e-13
LDLt	6.30e-13

One understands that the error is smaller for LU decomposition with diagonal pivoting, followed by LDLt decomposition. However, since the values are near to machine precision, numerical errors might influence the results.

3 The Sparse Matrix Class

A sparse matrix is an option to store matrices without compromising computational memory usage. Since the number of non-zero elements in a matrix is relatively lower than the total number of elements, the sparse matrix is a good choice. In this section, the SparseMatrix class is explained and, for the matrix A shown in Section 2.1, the memory usage is compared to the full matrix storage method.

3.1 Class Implementation

SparseMatrix class' attributes are as follows by Code 9:

```
@dataclass
class SparseMatrix:
    data: list = field(init=False, default_factory=list) # List of non-zero elements
    cols: list = field(init=False, default_factory=list) # List of columns
    ptr: list = field(init=False, default_factory=list) # List of cumulative sums of non-zero elements
    sparsity: float = field(init=False, default=0.0) # Sparsity of the matrix
    density: float = field(init=False, default=0.0) # Density of the matrix
    size: int = field(init=False, default=0) # Size of the full matrix

    data_memory: int = field(init=False, default=0) # Memory usage of the data list
    cols_memory: int = field(init=False, default=0)
    ptr_memory: int = field(init=False, default=0)
    total_memory: int = field(init=False, default=0)
```

Code Listing 9: SparseMatrix Class Attributes

Among them, highlights the data, in which the non-zero elements are stored, cols, in which the column indexes of each non-zero element are stored, and ptr, in which the cumulative sum of non-zero elements are stored. Sparsity is a

way to measure the rate of non-zero numbers in the matrix. Depending on the sparsity value, it is possible to decide whether it is more efficient to use a sparse matrix or a full one.

SparseMatrix has the ParseFullMatrix, which parses a full matrix to a sparse matrix structure, Multiply, which performs the product of a sparse matrix and a vector, and the EvaluateSparsity method, which evaluates the matrix sparsity and density. The ParseFullMatrix method is shown in Code 10

```
def ParseFullMatrix(self, matrix: np.array)->None:
    self.size = len(matrix)

    self.ptr.append(0)
    for row in matrix:
        for j, item in enumerate(row):
            if item != 0:
                self.data.append(item)
                self.cols.append(j)

        self.ptr.append(len(self.data))
```

Code Listing 10: ParseFullMatrix Method

ParseFullMatrix reads the full matrix and stores the non-zeros elements in the data, cols, and ptr vectors. Code 11 shows the Multiply method.

```
def Multiply(self, vector:np.array)->np.array:
    prod = np.zeros(len(vector))
    for i in range(len(self.ptr) - 1):
        sum = 0.0
        for k in range(self.ptr[i], self.ptr[i+1]):
            sum += self.data[k] * vector[self.cols[k]]

        prod[i] = sum

    return prod
```

Code Listing 11: Multiply Method

The Multiply function takes advantage of the sparse structure to evaluate the product of a matrix by a vector. The sparsity is calculated by the EvaluateSparsity method, shown in Code 12.

```
def EvaluateSparsity(self)->None:
    self.sparsity = 1 - len(self.data) / (self.size * self.size)
    self.density = 1 - self.sparsity
```

Code Listing 12: EvaluateSparsity Method

3.2 Storage Structure and Memory Usage

Once the FullMatrix and SparseMatrix classes are working, the comparison between the memory usage of both structures can be made. To do so, 4 bits are assigned to each element that is an integer, and 8 bits to those that are floating-point numbers.

The FullMatrix is composed of a 15×15 matrix, in which each element is a floating number. On the other hand, the SparseMatrix object has three vectors, two of which are int vectors and one is a float vector. Table 3.1 compares the size of each structure and its memory usage.

Table 3.1: Memory Usage Comparison between Full and Sparse Matrix

Full Matrix			**Sparse Matrix**		
Structure	**Size (elements)**	**Memory Usage (bits)**	**Structure**	**Size (elements)**	**Memory Usage (bits)**
Matrix	225	1800	data	168	1344
			cols	168	672
			ptr	16	64
Total	1800 bits			2080 bits	

In terms of memory usage, the FullMatrix class has a better performance than the SparseMatrix. With approximately 15% more memory used, the SparseMatrix is not advantageous in this case. Another important factor that corroborates

this result is the sparsity of the matrix A, which is 25.33%, a low value that does not justify the use of a sparse matrix. The code in which these evaluations are made can be found in Appendix B.

3.3 Product Matrix Vector

The Multiply function is tested using the unitary vector. This way, each element of the resulting vector is the sum of the elements of the corresponding row of the matrix. The result is shown in Fig. 3.1.

$$\begin{pmatrix} 3470. \\ 3840. \\ 0 \\ 5. \times 10^{10} \\ -8. \\ 4000. \\ 15\,900. \\ 17\,500. \\ -90.1 \\ 1. \\ 3840. \\ 19.3 \\ -5.47 \\ -1.08 \\ 5. \times 10^{10} \end{pmatrix}$$

Figure 3.1: Product of the Matrix A by the Unitary Vector

Comparing this result with the one obtained by numpy dot function, it is clear that the function is working properly.

4 Conclusions

This work implemented two classes: FullMatrix and SparseMatrix. FullMatrix has LU and LDLt decomposition methods for solving linear systems,

which after analyses it was observed that LU decomposition with diagonal pivoting has resulted in the smallest errors in terms of inverse recovery, followed by LDLt decomposition.

When it comes to sparse matrices, it is observed that the decision whether to use it or not is based on a series of factors, such as the sparsity of the matrix, the number of non-zero elements, and the size of the matrix. Not always sparse matrices result in less memory usage.

Finally, the process of operating a sparse matrix is not trivial, and the algorithm must be adapted to this structure. Otherwise, several errors may arise during the code's execution and more necessary than the necessary might be allocated to finish the process.

References

1 CUNHA, M. C. de C. *Métodos numéricos*. [S.l.]: Editora da UNICAMP, 2000.

2 GOLUB, G. H.; LOAN, C. F. V. *Matrix computations*. [S.l.]: JHU press, 2013.

3 STRANG, G. *Introduction to linear algebra*. [S.l.]: SIAM, 2022.

4 JENNINGS, A. Matrix computation for engineers and scientists. *(No Title)*, 1977.

A GitHub Repository

The source code for this report and every code inhere mentioned can be found in the following GitHub repository: CarlosPuga14/MetodosNumericos_-2024S1.

B Main Function Code

The main function code is shown in Listing 13.

```
def Main()->None:
    decomposition = "LDLt"
    pivoting = True
    diagonal = False

    output_file = f"{decomposition}"
    output_file += f"{'_Pivoting' if pivoting else ''}"
    output_file += f"{'_Diagonal' if diagonal else ''}"
    output_file += ".txt"

    matrix = # given Matrix

    full_matrix = FullMatrix(matrix, pivoting, diagonal, decomposition)

    full_matrix.PrintMathematica(True)

    full_matrix.Decompose()
    full_matrix.FindInverse()
    full_matrix.Print(output_file)

    full_matrix.CalcMemoryUsage()

    vec = np.random.rand(full_matrix.A.shape[0])
    sparse_matrix = SparseMatrix()
    sparse_matrix.ParseFullMatrix(full_matrix.A)

    sparse_matrix.CalcMemoryUsage()

    a = sparse_matrix.Multiply(vec)

    b = np.dot(full_matrix.A, vec)

    print(np.allclose(a, b))

```

```
35    sparse_matrix.EvaluateSparsity()
```

Code Listing 13: Main function.

Lines 2 to 13 set the input data for the FullMatrix object. Line 15 turns on the Mathematica way of printing output. Lines 17 to 19 decompose, invert, and print the output for the given matrix decomposition. Line 21 calculates the memory usage, used later to compare with the sparse matrix.

Line 23 creates a vector of random numbers to test the multiplication of the sparse matrix. Lines 24 to 27 create the SparseMatrix object, parse the full matrix, and calculate the memory usage. Line 29 calculates the multiplication of the sparse matrix by the vector, while line 31 does the same for the full matrix. Line 33 compares the results.

C Permutation Matrices

Figures C.1 and C.2 show the row and column permutation matrices for both cases, regular pivoting and diagonal pivoting, for LU decomposition, respectively.

Figures C.3 and C.4 show the row and column permutation matrices for LDLt decomposition, respectively. Notice that due to the symmetry of the matrix, the column permutation matrix is the transpose of the row permutation matrix.

$$\begin{pmatrix}
0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 \\
0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 \\
0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}$$

(a) Regular pivoting.

$$\begin{pmatrix}
0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 \\
0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 \\
0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0
\end{pmatrix}$$

(b) Diagonal pivoting.

Figure C.1: Row permutation matrix for LU decomposition.

$$\begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 \\
1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. \\
0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 \\
0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}$$

(a) Regular pivoting.

$$\begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 \\
1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 \\
0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}$$

(b) Diagonal pivoting.

Figure C.2: Columns permutation matrix for LU decomposition.

$$\begin{pmatrix}
0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 \\
0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 \\
0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0
\end{pmatrix}$$

Figure C.3: Row permutation matrix for LDLt decomposition.

$$\begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 \\
1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1. & 0 & 0 & 0 \\
0 & 1. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}$$

Figure C.4: Columns permutation matrix for LDLt decomposition.