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 the following methodology, 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. Although it may vary from method to method, a simple example of rank one update in LU decomposition follows

$$A_{step1} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix},$$

In the matrix A_{step1}, $1x_1 + 2x_2 + 3x_3 = b_1$ is the first equation. To update the matrix, the line and column are divided by the pivot (the term in the diagonal of the equation's line, in this case, 1), and the outer product between the vectors is calculated

$$\{4, 7\}^T \otimes \{2, 3\} = \begin{bmatrix} 8 & 12 \\ 14 & 21 \end{bmatrix},$$

and the result is subtracted from the original matrix

$$A_{step2} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & -3 & -6 \\ 7 & -6 & -12 \end{bmatrix}.$$

This procedure is repeated until the last equation. Once it is over, the re-

sulting 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 computational 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, espe-

cially 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 decomposed 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.

```
def LU_Decomposition(self)->None:
    self.RankOneUpdate()
    self.Fill_L()
    self.Fill_U()

def LDLt_Decomposition(self)->None:
    self.RankOneUpdate()
    self.Fill_L()
    self.Fill_D()
    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.

```
def RankOneUpdate(self)->None:
    for i in range(self.size):
        matii_correction = 1.0

        if (abs(self.A_Decomposed[i, i]) < self.tolerance):
            raise Exception("Null Pivot")

        if self.decomposition_type in == "LDLt":
            matii_correction = self.A_Decomposed[i, i]
            self.A_Decomposed[i, i+1::] /= self.A_Decomposed[i, i]

        self.A_Decomposed[i+1::, i] /= self.A_Decomposed[i, i]
        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
9      col_max += index
10
11     rows, cols = self.UpdatedPermutationVector(rows, cols, row_max, col_max
       , index)
12
13     perm_row = self.PermutationMatrix(rows)
14     perm_col = self.PermutationMatrix(cols, rows=False)
15
16     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):
        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:
```

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

Code Listing 8: FindInverse method.

In the case of pivoting, the inverse is obtained by $pR^T.U^{-1}.D^{-1}.L^{-1}.pC^T$. However, if no pivoting has been done, the inverse is simply $U^{-1}.D^{-1}.L^{-1}$. If the decomposition type is not valid, an exception is raised.

2.2 Matrix LU Decomposition

2.3 Matrix LDLt Decomposition

3 The Sparse Matrix Class

3.1 Class Implementation

3.2 Storage Structure and Memory Usage

3.3 Product Matrix Vector

4 Conclusions

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.