Due date: Monday, March 24, at the beginning of the lab.

1 Objectives of this Assignment

Our plans for this assignment are

- To complete the implement of the LU factorization algorithm, by adding code for pivoting;
- To look at elements of the QR factorization algorithm, for linear least squares problems;
- To apply the SLE and LLS solving module to the resolution of a couple of practical problems.

We know that LU factorization will run into problems when the the matrix of the least squares system (or for that matter, of the SLE) is ill-conditioned, so we can’t expect that everything will work perfectly. We will solve most of these problems next week when we implement the QR factorization algorithm.

2 LU Factorization

2.1 Where we stand

Having completed Lab 04, you should have Maple and C++/Java code to decompose a square matrix $A$ into the product of a lower triangular matrix with unit diagonal, $L$, and an upper triangular matrix $U$. The algorithm you implemented is (should be) the following:

```
// for all columns of matrix A
for j from 1 to n do
    // for all elements above or on the diagonal
    for i from 1 to j do
        compute $u_{i,j} = a_{i,j} - \sum_{k=1}^{i-1} l_{i,k} u_{k,j}$
    // for all elements below the diagonal
    for i from j+1 to n do
        compute $l_{i,j} = \frac{1}{u_{j,j}} \left( a_{i,j} - \sum_{k=1}^{j-1} l_{i,k} u_{k,j} \right)$
```
What has been left out of this algorithm is pivoting. What do we do when the term \( u_{jj} \) is too small? We swap rows, as usual. But this brings up three questions (the third, more practical question, was added to last week’s list):

1. Since swapping rows is a very fast operation, why not do it all the time, and always choose the largest pivot (in absolute value)?
2. At what point in the algorithm can we compare the pivots and swap rows?
3. How is row swapping done, in Maple and in C++/Java?

The answer to Question 1 is: Indeed, why not? This is exactly what we will do.

Questions 2 and 3 are somewhat trickier, at first. We will address them in the next subsections.

### 2.2 Choice of a pivot

When we look carefully in the algorithm without pivoting at the expression for \( u_{jj} \) and that of the terms \( l_{ij} \) we see that it is identical, except for the \( u_{jj} \) scaling factor. In other words, we can do the following

1. For \( i = j \ldots n \), compute \( l_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj}, \)
2. Decide which of the above terms (say, the one at row \( i_{\text{max}} \) would constitute the best pivot,
3. If \( i_{\text{max}} \neq j \), swap rows \( i_{\text{max}} \) and \( j \) (this defines the new pivot \( u_{jj} \),
4. For all rows \( i = j + 1 \ldots n \) after the row swap, divide row elements by the term computed in step one to obtain \( l_{ij} \).

We will discuss implementation issues regarding Step 3 (row swapping) in Subsection 2.3, but for the moment, let us consider more carefully Step 2. Obviously, we need to swap rows when the candidate pivot computed at Step 1, \( u_{jj} \) is null or very small. To pick a new pivot, we have two possible choices: “first fit” or “best fit.” The first strategy consists in selecting the first element computed in Step 1 whose absolute value is larger than some predefined threshold. The second strategy consists in estimating a “quality” criterion for each pivot candidate and selecting the one with the highest score. The two most common quality estimates are

- **absolute value** \( |l_{ij}| \): We simply select the candidate pivot with the largest absolute value;
- **relative value** \( |l_{ij}| / \max_{k=1\ldots n} |a_{ik}| \): We select the candidate pivot that is the largest relative to the original row in matrix \( A \).

The second strategy works generally better, so this is the one I use in the following algorithm:
// for all rows of matrix A
for j from 1 to n do
  compute $v_i = \max_{j=1\ldots n} |a_{ij}|$
// for all columns of matrix A
for j from 1 to n do
  // for all elements above the diagonal
  for i from 1 to j-1 do
    compute $u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}$
  // for all elements on or below the diagonal
  for i from j to n do
    compute $l_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj}$
    determine $i_{\text{max}}$ such that $l_{i_{\text{max}},j} = \max_{i=j\ldots n} |l_{ij}|/v_i$
    if $i_{\text{max}} \neq j$ swap rows $j$ and $i_{\text{max}}$ in $A$, $L$, $U$ (set $u_{j,j} = l_{i_{\text{max}},j}$)
    compute $s = 1/u_{jj}$ (if $u_{jj} = 0$, set $s = 0$)
  // for all elements below the (possibly new) diagonal
  for i from j+1 to n do
    compute $l_{ij} = s l_{ij}$

2.3 How to swap rows

Regarding the swapping operation, we must be careful about one thing: we are (or could be) performing the factorization of $A$ before we know the right hand term $b$. Therefore, we cannot simply swap rows. We have to keep track of the swaps we perform so that we can apply them to the right hand term when we try to solve the SLE.

The simplest way to do this is to maintain an array of row indexes. In other words, we replace in our algorithms any row index $i$ by $s_i$. Initially, $p = \{1, 2, \ldots, n\}$. Each time the algorithm calls for a row swap, we simply invert the corresponding elements of $p$. At the end of the factorization, $p$ represents the permutation that has been applied to $A$ and must now be applied to any right hand term $b$ of an SLE we want to solve. So now, instead of manipulating element $LU_{ij}$, we will work with $LU_{s_i,j}$, or if you prefer to write this in “array” notation, $LU[p[i]][j]$.

Now, where you have to be very careful is that the way you apply this row permutation depends on the way you decided to store your matrices. If you overwrite your $A$ matrix with the terms of the $L$ and $U$ matrices, then you should always use $s[i]$ instead of $i$ as the first index whenever you access the matrix. You should of course understand that this permutation vector should be sent back as a result of the factorization since you will need it in your substitution modules.
3 QR Factorization

3.1 What it is about

As we saw in class, the idea behind the QR factorization of the matrix of a linear least squares (LLS) problem is that we know that the solution of the approximate SLE

\[ A x \approx b, \]

where \( A \) is an \( m \times n \) matrix \( (m > n) \), is the solution of the exact SLE

\[ (A^T A) x = A^T b. \]

The problem with this new SLE is that its matrix, \( A^T A \), is typically ill-conditioned, so that applying LU factorization does not give good results. Still, we know that \( A \) somehow contains the information about an exact \( n \times n \) SLE that can be solved to get the best fit solution of the approximate problem.

As we said in class, since \( A \) is rectangular, a straightforward LU factorization would not help us much. The idea, instead is to try to decompose \( A \) into the product of two matrices,

\[ A = QR \]

where \( Q \) is an \( m \times m \) orthogonal matrix and \( R \) is an \( m \times n \) upper-triangular matrix. I remind you that when we say that \( Q \) is orthogonal, we mean that \( Q^T Q = I_n \), the \( n \times n \) identity matrix.

What we would like to do is build \( Q \) as the product of \( n \) elementary orthogonal matrices, \( H_1, H_2, \ldots, H_n \), such that \( H_j \) operates mostly on the \( j \)th column of \( A \) by setting to zero all its terms below the diagonal without disturbing too much the columns to the left of column \( j \).

3.2 Building orthogonal matrices

First, I want you to verify that for any \( m \times 1 \) vector \( v \neq 0 \), the following matrix is orthogonal:

\[ H_v = I - \frac{2vv^T}{v^Tv}. \]

Next, if \( a \) is an arbitrary (non-zero) \( m \times 1 \) vector, we want to find a vector \( v \) such that \( H_v \), when applied to \( a \) produces a vector whose terms in rows 2 to \( m \) are all zero. Your textbook explains how to build such a vector.

What your textbook doesn’t completely explain, and which I would like you to do, is to determine for an arbitrary (non-zero) \( m \times 1 \) vector \( a \) a vector \( v \) such that \( H_v \), when applied to \( a \) produces a vector whose terms in rows 1 to \( k \) are unchanged, the term in row \( k \) is non-zero, and those in rows \( k + 1 \) to \( m \) are equal to zero.
4 Test applications

4.1 Curve fitting

In Lab04, we saw how to compute the coefficients of a polynomial function such that the function’s graph passes through preselected data points. The main weakness of the solution we computed was that the degree of the polynomial function increased with the number of data points (in fact, the degree of the polynomial was equal to the number of points minus one). As we have seen in previous labs, high-degree polynomials are not in general functions we want to work with:

- They will present a large number of local minima and maxima and oscillation that were not in the original data. Try for example to get the curve for a dozen points that are almost aligned.
- They are highly sensitive to perturbations
- In the case of our curve fitting problem, the two curves you get for a set of \( N \) points and for the same \( N \) points and one additional point are typically completely different.

March 17, 2003, we will try again to generate a polynomial function that best fits a set of \( N \) data points, but this time we will impose the degree of the polynomial. Since in practice polynomials of degree higher than 5 are rarely useful, we will restrict ourselves to polynomials of degree 0 to 5.

What this means is that as soon as we have more than 6 points, we won’t have anymore an exact SLE but an approximate one, which we will solve by application of the linear least squares algorithm. What this also means is that the “best fit” curves generated typically will not pass through any of our data points.

The equations are the same as in Lab04, only the way you use them changes.

**What you should do:** First, you should write a Maple procedure that takes as parameters a 2D array of point coordinates, \( (x_i, y_i), i = 1 \ldots n \) and the desired degree of the polynomial function, \( m \), and returns the array of coefficients of the “best fit” polynomial of degree \( m \) for the data points.

Next, I want you to write a procedure that takes as parameter a 2D array of point coordinates, \( (x_i, y_i), i = 1 \ldots n \), and returns the coefficients of the polynomial function of degree \( 0 \leq m \leq 5 \) that best fits the data.

4.2 Surface fitting

This time, we have triplets of point coordinates \( (x_i, y_i, z_i), i = 1 \ldots n \). And we want to find the coefficients of the polynomial function of degree three that best fits the data. Modelling the problem and working out a solution should not pose you any major difficulty.

5 C++/Java Implementation

From now on I will give much clearer specifications for these implementations. For one thing, this should give you a better idea of what is really required. For another, this will make the TA’s
job much easier, when grading the code: I will give her a test application that uses your classes according to the specs. Failure to meet the specifications will result in failure of the tests.

5.1 LU FActorization and SLE solver

I want you to reorganize your LU factorization, forward and backward substitution, and SLE solving code into a class LinearSolver that offers the following public methods (feel free to add more if you want):

- public LinearSolver(), the default constructor
- public int[] LUDecompose(float ALU[][])
  This method receives as parameter the matrix A to decompose and it overwrites it with the values of the elements of the L and U matrices. The method returns the array of permutations resulting from row swaps. Matrix A is expected to be properly allocated and to be a square matrix. You do not need to check for that.
- public float[] forwardSubstitute(float L[][], float b[], int p[]).
  L is supposed to be a properly allocated square matrix that is lower triangular with a unit diagonal (hence the 1 in the method’s name). b is the SLE’s right hand term. It is expected to have the same dimensions as L. p is the array of permutations that have been applied to the rows of the matrix.
- public float[] backwardSubstitute(float U[][], float b[], int p[]) throws ArithmeticException.
  U is supposed to be a properly allocated square matrix that is upper triangular. b is the SLE’s right hand term. It is expected to have the same dimensions as L. p is the array of permutations that have been applied to the rows of the matrix. The exception is thrown when a zero diagonal term is encountered.
- public float[] solveSLE(float A[], float b[]) throws ArithmeticException.

You are not expected to verify that the matrices have been properly allocated, are square, that the right hand terms have the same size as the SLE’s matrix, etc. On the other hand, if you feel like doing it, make sure that this is mentioned in your report and comments so that you get the appropriate extra credit for your efforts.

5.2 QR factorization

No C++/Java implementation in this assignment.
6 Evaluation

6.1 What to hand in

A: You should upload to the EnVision server a folder containing the following:

1. A complete, cleaned-up CodeWarrior project folder.
   - Complete means that everything needed to compile and execute the project is there: source files, project file, header files (.h files) and precompiled header source files (.pch files) if you program in C++, test data files.
   - Cleaned-up means that files resulting from the compilation should be removed: .exe files, .sym files, precompiled header binary files (if in C++), and the entire <name of project>data folder (close the project before you throw away this folder).

2. The file of a well-commented Maple or muPad worksheet.


B: You should hand in hard copies of your report and of your Maple/muPad worksheet.

6.2 Point distribution

The maximum number of points is 100, but extra points could be awarded for excellent aspects of the project or report. The point distribution for this assignment is as follows:

<table>
<thead>
<tr>
<th>Maple/muPad Modelling</th>
<th>20 pts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accomplishes what was demanded</td>
<td></td>
</tr>
<tr>
<td>Comments and analysis</td>
<td>10 pts</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C++/Java Code</th>
<th>20 pts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accomplishes what was demanded</td>
<td></td>
</tr>
<tr>
<td>Good class design</td>
<td>10 pts</td>
</tr>
<tr>
<td>General quality &amp; readability</td>
<td>10 pts</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Report</th>
<th>20 pts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discussion and analysis of the results</td>
<td></td>
</tr>
<tr>
<td>General quality of the writing and presentation</td>
<td>10 pts</td>
</tr>
</tbody>
</table>

6.3 Various point penalties

Hopefully we won’t have to apply many of these:

- Project folder incomplete or not properly cleaned up -5 pts
- Report file missing from the project folder -5 pts
- Maple file missing 0 for that part

Late penalties

- Printed copy of the report, 1 day late -5 pts
- Project folder (uploaded to EnVision server), per day late -5%
If you submit a project late, then it is your responsibility to notify the TA (with CC. to me) that the project is finally available for download on the EnVision server. If you fail to do so, then the “late penalty clock” will keep ticking until the TA gets around to checking your folder on the EnVision server and notices your project. Unless asked explicitly to do so, do not mail your project folder as an attachment.