Due date: Maple: Monday, March 07, at the beginning of the lab
C++/Java: Monday, March 14, at the beginning of the lab.

1 What this Lab is About

Our plans for this assignment are to implement a first version of

- the LU factorization algorithm
- the forward substitution and backsubstitution algorithms.

These are only “first versions” because today’s algorithms do not perform pivoting yet.

2 Warmup: Forward and Backward Substitution

2.1 Solving an upper triangular SLE by backsubstitution

We consider an upper triangular matrix

\[
U = \begin{pmatrix}
    u_{1,1} & u_{1,2} & \cdots & u_{1,n} \\
    0 & u_{2,2} & \cdots & u_{2,n} \\
    \vdots & \ddots & \ddots & \vdots \\
    0 & \cdots & 0 & u_{n,n}
\end{pmatrix}
\]

and we want to solve the SLE \( U \mathbf{x} = \mathbf{b} \).

Maple 1. Implement the forward substitution algorithm as presented in the notes and in class. Since this procedure will be receive as one of its parameters the output of an LU factorization, you don’t need to verify that the matrix is indeed upper triangular: you just know it is.

Needless to say, you should test your procedure with a few upper triangular SLEs.

2.2 Solving a lower triangular SLE by forward substitution

This time, the SLE’s matrix is lower triangular with a unit diagonal,

\[
L = \begin{pmatrix}
    1 & 0 & \cdots & 0 \\
    l_{2,1} & 1 & \cdots & 0 \\
    \vdots & \ddots & \ddots & \vdots \\
    l_{n,1} & \cdots & l_{n,n-1} & 1
\end{pmatrix}
\]
and we want to solve the SLE $L \mathbf{x} = \mathbf{b}$.

**Maple 2.** Implement a Maple procedure to solve such an SLE backsubstitution. Of course, you will want to exploit the fact that you know that your matrix has a unit diagonal. Since this speed gain would come at the cost of generality the name of your procedure (and, later, that of its C++/Java implementation) and the description and comments in your code should reflect that.

### 3 LU Factorization

#### 3.1 Step 1: Crout’s algorithm without pivoting

If we ignore the issue of pivoting (that is, if we are lucky enough not to encounter a diagonal element equal to 0), we saw last Wednesday in class that the following algorithm will perform the factorization while overwriting the elements of $\mathbf{A}$ with that of $L$ and $U$. In what follows, $\mathbf{A}$, $L$, $U$, and $LU$ all refer to the same “array.” Simply $a_{i,j}$ refers to an element of the original matrix, $lu_{i,j}$ refers to the result of a calculation that has not been “committed” yet to being an $L$ or a $U$ element. After the choice has been made (that will be the role of pivoting), we will write $l_{i,j}$ or $u_{i,j}$.

```plaintext
// for all rows of matrix A
for j from 1 to n do
  // 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 $u_{i,j} = a_{i,j} - \sum_{k=1}^{i-1} l_{i,k} u_{k,j}$
    // for all elements on or below the diagonal
    for i from j to n do
      compute $lu_{i,j} = a_{i,j} - \sum_{k=1}^{j-1} l_{i,k} u_{k,j}$
    // Pivoting will be performed here (later)
    $u_{j,j} = lu_{j,j}$
    compute $s = 1/lu_{j,j}$ (if $lu_{j,j} = 0$, set $s = 0$
    // for all elements below the diagonal
    for i from j + 1 to n do
      $l_{i,j} = s lu_{i,j}$
```

Although when we begin the algorithm none of the terms $u_{k,j}$ or $l_{i,k}$ is known, they are all computed just in time to be known when they appear on the right side of an equation in the algorithm.
Furthermore, once a term \( a_{i,j} \) has been used, it will not be used anymore. This is why it is possible to store the \( u \) and \( l \) terms in the “old” \( A \) matrix by overwriting \( a_{i,j} \) with \( l_{i,j} \) if \( i > j \) or \( u_{i,j} \) if \( i \leq j \) to give the matrix

\[
\begin{pmatrix}
u_{1,1} & u_{1,2} & \cdots & u_{1,n} \\
u_{2,1} & u_{2,2} & \cdots & u_{2,n} \\
\vdots & \vdots & \ddots & \vdots \\
u_{n,1} & \cdots & l_{n,n-1} & u_{n,n}\end{pmatrix}
\]

One drawback of overwriting \( A \) is that you lose your input data. If you do this, better make sure that you are working with a copy of your matrix, on the other hand, you will see in the next subsection that this makes the pivoting a lot easier. Even if you do not overwrite \( A \), there is no point returning \( L \) and \( U \) in two separate matrices. You only compute them to send them to your forward and backward substitution modules anyway...

The total number of steps in this factorization is about \( \frac{2}{3}n^3 \)

**Maple 3.** Implement the above algorithm as a Maple procedure.

As usual you will need to test your procedure thoroughly. In particular, since factorization is a “reversible” operation, you should verify that this is indeed the case with your procedure. In other words,

- If \( L \) and \( U \) are respectively a unit-diagonal lower triangular matrix and an upper-triangular matrix, then the LU factorization of the product \( LU \) should return you \( L \) and \( U \) again.

- If the LU factorization of a matrix \( A \) returns you matrices \( L \) and \( U \), then you should have \( LU = A \).

Note that to perform these verifications, you will probably need to write procedures to extract the \( L \) and \( U \) terms of the “composite” matrix \( LU \) returned by the LU factorization.

### 3.2 Solve a complete SLE

Assuming that the matrix \( A \) you choose for your SLE is suitable for the simplified (no pivoting) LU factorization algorithm, you should be able now to solve the SLE \( Ax = b \) by using your forward and backward substitution algorithms. This is where the fact that your procedures “trust” you to send matrices with the right shape come handy, since in fact you will be sending them the \( LU \) matrix, which is anything but triangular. So, how does it work? The backsubstitution procedure only uses the terms on or above the diagonal, which indeed corresponds to \( U \), while the forward substitution procedure uses subdiagonal elements while assuming that diagonal elements are all equal to 1.

**Maple 4.** Implement a Maple procedure \( \text{SolveSLE} \) that receives as parameters a square \( n \times n \) matrix \( A \) and an \( n \times 1 \) column vector \( b \), and returns the solution to the SLE \( Ax = b \).
4 Use the LU Factorization

4.1 Solve a general SLE

This is a pretty straightforward application of the factorization, merely an integration of all your procedures into a general “solve SLE” Maple procedure that takes as parameters a square \( n \times n \) matrix \( A \) and an \( n \times 1 \) column vector \( b \), and solve the SLE \( A \mathbf{x} = b \).

Of course, at this point you cannot solve just any SLE, so you will have to choose your \( A \) matrix carefully.

4.2 Compute the determinant of a matrix

This may come as a surprise to you, but the most efficient way to compute the determinant of a matrix is to compute its \( L \ U \) factorization first, rather than apply the recurrent expression you learned in your Linear Algebra course.

I will not explain how to get the determinant once you have the \( L \ U \) factorization: This one is almost embarrassingly easy. The only thing you have to be careful about is the sign of the determinant: each time you swap two lines, the sign of the determinant is inverted. In your C++/Java implementation, the simplest way to handle that is to use a local variable that is initially set to 1 and changes sign each time two rows are swapped.

4.3 Compute the inverse of a matrix

Having computed the \( L \ U \) factorization of a matrix \( A \), you shouldn’t have too many problems inverting it. The inversion computation can simply be posed as the resolution of \( n \) different SLEs with the same matrix, \( A \), but different right hand terms.

4.4 Compute the condition number of a matrix [4 pts extra credit]

An easy 4 points of extra credit here. Of course, for this I want both condition numbers (for the \( N_1 \) and \( N_\infty \) norms) and the C++/Java implementation.

5 Evaluation

5.1 What to hand in

Please read this part carefully. You will be penalized (see next subsection) if you don’t follow the rules and the TA ends up wasting time converting files just to be able to evaluate your work.

A: You should upload the following (the procedure to follow will be clarified by the end of this week):

1. Next week: the file of a well-commented Maple worksheet. Please make

2. Two weeks from now: A complete, cleaned-up CodeWarrior, gcc (mingw or cygwin), or Eclipse project folder.
Complete means that everything needed to compile and execute the project is there: source files, project file or make file, header files (.h files) and precompiled header source files (.pch files) if you program in C++. Of course, your test data files should be there as well, but the TA might use other data sets to test your code.

Cleaned-up means that files resulting from the compilation should be removed: .exe files, .sym files, .o object files (in C++), .class files (in Java), precompiled header binary files (if in C++). If you use CodeWarrior, the entire <name of project>.data folder should be deleted as well (close the project before you throw away this folder).

3. Two weeks from now: Your report as a Word, HTML, or Acrobat (.pdf) document.

**B:** You should hand in printouts of your (complete) Maple worksheet and of your report on the day the corresponding part of the assignment is due.
5.2 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:

**Maple Modelling**
- Accomplishes what was demanded: 20 pts
- Comments and analysis: 10 pts

**C++/Java Code**
- Accomplishes what was demanded: 20 pts
- Good class design: 10 pts
- General quality & readability: 10 pts

**Report**
- Discussion and analysis of the results: 20 pts
- General quality of the writing and presentation: 10 pts

5.3 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.

As I explained in class, I will post sometimes partial solutions to the assignments, to make sure that nobody gets too far behind. Obviously, it will be impossible to get any point on a part after a solution for it has been posted (this does not affect the late penalty count).