2 Direct solution of linear systems

In this section we will deal with the most basic and common problem in linear algebra: the solution of a linear system

$$A\mathbf{x} = \mathbf{b},$$

where $A \in \mathbb{C}^{n\times n}$, $\mathbf{x}, \mathbf{b} \in \mathbb{C}^n$. Linear systems are used in almost every computational task in science, for solving non-linear equations, optimization problems, partial differential equations and more. There are many algorithms to solve a linear system. In this section we will deal with a certain type of direct methods (as opposed to iterative methods that we will discuss later).

2.1 Gaussian elimination

Gaussian elimination is the standard way of solving a linear system, and is the way most of us use analytically when we solve such a system "by hand". The goal is to manipulate the rows of the matrix (add, subtract and multiply rows by scalars) such that we are left with a triangular system. We will assume that the matrix A is of full rank (non-singular) and that a solution to the system exists.

Consider the following linear system

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{2}$$

Our aim is to use elementary operations and transform the system $A\mathbf{x} = \mathbf{b}$ to a system $U\mathbf{x} = \hat{\mathbf{b}}$ that has the same solution $\mathbf{x}$. The matrix U will be upper-triangular: such that

$$U = \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}. \tag{3}$$

Now we can solve the triangular system by solving the 1-variable equation for x_3, then for x_2 using x_3, and then for x_1. This process of solving an upper triangular system is called **Backward Substitution** in which we start solving from the last variable x_n all the way to

the first one x_1. That is:

$$\mathbf{x} = \texttt{BwdSub}(U, \mathbf{b}): \quad x_i = \frac{1}{u_{ii}}\left(\hat{b}_i - \sum_{j=i+1}^{n} u_{ij}x_j\right) \quad i = n, n-1, ..., 1.$$

Example 2. *Solve the following linear system using 4 significant digits using cut-off rounding.*

$$\begin{cases} 3x_1 + -x_2 + 2x_3 = 12 \\ 1x_1 + 2x_2 + 3x_3 = 11 \\ 2x_1 + -2x_2 + -x_3 = 2 \end{cases} \quad or \begin{bmatrix} 3 & -1 & 2 \\ 1 & 2 & 3 \\ 2 & -2 & -1 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 12 \\ 11 \\ 2 \end{bmatrix} \tag{4}$$

At the first iteration, the pivot will be 3:

$$\begin{bmatrix} \boldsymbol{3} & -1 & 2 \\ 1 & 2 & 3 \\ 2 & -2 & -1 \end{bmatrix}, \begin{bmatrix} 12 \\ 11 \\ 2 \end{bmatrix} \underset{\substack{\mathbf{r}_2 \leftarrow \mathbf{r}_2 - \frac{1}{3}\mathbf{r}_1 \\ \mathbf{r}_3 \leftarrow \mathbf{r}_3 - \frac{2}{3}\mathbf{r}_1}}{\Rightarrow} \begin{bmatrix} 3 & -1 & 2 \\ 0 & 2.333 & 2.334 \\ 0 & -1.334 & -2.332 \end{bmatrix}, \begin{bmatrix} 12 \\ 7.004 \\ -5.992 \end{bmatrix} \tag{5}$$

At the second iteration the pivot will be 2.333:

$$\begin{bmatrix} 3 & -1 & 2 \\ 0 & \boldsymbol{2.333} & 2.334 \\ 0 & -1.334 & -2.332 \end{bmatrix}, \begin{bmatrix} 12 \\ 7.004 \\ -5.992 \end{bmatrix} \underset{\mathbf{r}_3 \leftarrow \mathbf{r}_3 - \frac{-1.334}{2.333}\mathbf{r}_2}{\Rightarrow} \begin{bmatrix} 3 & -1 & 2 \\ 0 & 2.333 & 2.334 \\ 0 & 0 & -1.000 \end{bmatrix}, \begin{bmatrix} 12 \\ 7.004 \\ -1.993 \end{bmatrix} \tag{6}$$

From here, we apply the backward substitution algorithm for $U\mathbf{x} = \hat{\mathbf{b}}$

$$U = \begin{bmatrix} 3 & -1 & 2 \\ 0 & 2.333 & 2.334 \\ 0 & 0 & -1.000 \end{bmatrix}, \hat{\mathbf{b}} = \begin{bmatrix} 12 \\ 7.004 \\ -1.993 \end{bmatrix} \Rightarrow \mathbf{x} = \begin{bmatrix} 3.007 \\ 1.008 \\ 1.993 \end{bmatrix}; \quad where \quad \mathbf{x}_{exact} = \begin{bmatrix} 3 \\ 1 \\ 2 \end{bmatrix}$$

We got a pretty good approximation of the solution.

The Gaussian elimination algorithm is summarized in Alg. 1.

Now we wish to count the number of multiplications and divisions in GE. In the inner most loop we have one multiplication, and in the second most inner loop we have an additional

```
Algorithm: Gaussian Elimination
# A ∈ ℂ^{n×n}
Initialize: U ← A, b̂ ← b.
for k = 1, ..., n − 1 do
    for i = k + 1, ..., n do
        t = u_{i,k} / u_{k,k}.
        # In the next loop we do r_i ← r_i − t · r_k
        for j = k, ..., n do
            u_{i,j} ← u_{i,j} − t · u_{k,j}.
        end
        b̂_i ← b̂_i − t · b̂_k
    end
end
```

Algorithm 1: Gaussian Elimination

two. Overall we have

$$\begin{aligned} Cost(GE) &= \sum_{k=1}^{n-1}\sum_{i=k+1}^{n}(2+\sum_{j=k}^{n}1) = \sum_{k=1}^{n-1}\sum_{i=k+1}^{n}(2+(n-k+1)) = \sum_{k=1}^{n-1}\sum_{i=k+1}^{n}(n-k+3) = \\ &= \sum_{k=1}^{n-1}(n-k)(n-k+3) = \sum_{p=1}^{n-1}p(p+3) = \cdots \approx \frac{1}{3}n^3 = O(n^3) \end{aligned}$$

In the backward substitution stage we have i multiplications for row $n-i+1$

$$Cost(BS) = \sum_{i=1}^{n} i = \frac{1+n}{2}n \approx \frac{1}{2}n^2 = O(n^2)$$

Overall for solving the linear system we get $O(n^3)$.

2.2 LU decomposition without pivoting

The Gaussian elimination in the previous section can be used to get a powerful tool: LU decomposition. It turns out the the GE process can easily yield a decomposition of A into a product of lower and upper triangular matrices

$$A = LU,$$

where U is the same matrix that is computed in GE, and L is the lower triangular matrix that is decomposed out of the t variables in Alg. 1. That is, in our example

$$L = \begin{bmatrix} 1 & 0 & 0 \\ \frac{u_{2,1}^{(1)}}{u_{1,1}^{(1)}} & 1 & 0 \\ \frac{u_{3,1}^{(1)}}{u_{1,1}^{(1)}} & \frac{u_{3,2}^{(2)}}{u_{2,2}^{(2)}} & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0.333 & 1 & 0 \\ 0.666 & \frac{-1.334}{2.333} & 1 \end{bmatrix}$$

where $u_{i,j}^{(k)}$ is the i,j-th coefficient of U in the k-th iteration. In exact arithmetics we get for the previous example:

$$A = LU = \begin{bmatrix} 3 & -1 & 2 \\ 1 & 2 & 3 \\ 2 & -2 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{3} & 1 & 0 \\ \frac{2}{3} & -\frac{4}{7} & 1 \end{bmatrix} \cdot \begin{bmatrix} 3 & -1 & 2 \\ 0 & \frac{7}{3} & \frac{7}{3} \\ 0 & 0 & -1 \end{bmatrix}.$$

Algorithm: LU
$\# \; A \in \mathbb{C}^{n \times n}$
Initialize: $U = A$. $L = I$.
for $k = 1, ..., n-1$ **do**
  for $i = k+1, ..., n$ **do**
    $l_{i,k} = \frac{u_{i,k}}{u_{k,k}}$.
    for $j = k, ..., n$ **do**
      $u_{i,j} = u_{i,j} - l_{i,k} u_{k,j}$.
    end
  end
end

Algorithm 2: LU decomposition without pivoting

Algorithm 2 applies the process above to a general $n \times n$ matrix. Like the Gaussian Elimination, its running time complexity is $O(n^3)$ and its space complexity is $O(n^2)$.

In Julia code, the LU algorithm would be

```
"""
LU without pivoting
"""
function LUnp(A::Array)
n = size(A,1); # Assuming A is nxn.
U = copy(A);
L = Matrix{eltype(A)}(1.0I,n,n);
for k=2:n
        for i=k:n
                L[i,k-1] = U[i,k-1]/U[k-1,k-1];
                for j=k-1:n
                        U[i,j] = U[i,j] - L[i,k-1]*U[k-1,j];
                end
        end
end
return L,U
end
```

The advantage of having a decomposition: What did we gain out of this? Using the LU decompisition of A, we can solve any linear system $A\mathbf{x} = \mathbf{b}$ easily by two triangular matrix inversions

$$A\mathbf{x} = \mathbf{b} \Rightarrow LU\mathbf{x} = \mathbf{b} \Rightarrow_{\mathbf{y}=U\mathbf{x}} L\mathbf{y} = \mathbf{b}.$$

First by "forward substitution" we solve the lower triangular system

$$\mathbf{y} = \texttt{FwdSub}(L, \mathbf{b}) = L^{-1}\mathbf{b},$$

then, using backward substitution we solve the upper triangular system

$$U\mathbf{x} = \mathbf{y} \Rightarrow \mathbf{x} = \texttt{BwdSub}(U, \mathbf{y}) = U^{-1}\mathbf{y},$$

by "backward substitution" in $O(n^2)$ running time. This is most efficient when we have multiple linear systems to solve with the same matrix A but each time for a different right-hand-sides $\mathbf{b}$. In particular, the inverse of a matrix can be computed by solving $AX = I$

$$\begin{aligned} Y &= L^{-1}I \\ X &= U^{-1}Y \end{aligned}$$

and the result is $X = A^{-1}$. The process is computed in $O(n^3)$, like forming the LU decomposition itself.

Theorem 1 (The uniqueness of the LU decomposition)**.** *If A is non-singular, and $A = LU$ is an LU decomposition without pivoting such that $l_{ii} = 1$ for $i = 1, ..., n$, then this decomposition is unique.*

Proof. (Skipped in class) Assume by contradiction that there exist two different decompositions

$$A = L_1 U_1 \quad \text{and} \quad A = L_2 U_2,$$

such that $L_1 \neq L_2$ and $U_1 \neq U_2$. Since A is invertible, so are L_1 and L_2 and U_1 and U_2. Since $L_1 U_1 = L_2 U_2$, then so

$$L_2^{-1} L_1 = U_2 U_1^{-1}.$$

Since the inversion of a lower triangular matrix is also lower triangular, and so is a multipliplication of two lower triangular matrices, the matrix $L_2^{-1} L_1$ above is a lower triangular matrix. Similarly, the matrix $U_2 U_1^{-1}$, is an upper triangular matrix. The equality between the two means that both of the expressions are in fact diagonal matrices. By their structure, for every two lower or upper triangular matrices B, and C, we have that $(BC)_{ii} = B_{ii} C_{ii}$. Hence, for example, $(L_2^{-1} L_1)_{ii} = (L_2^{-1})_{ii} (L_1)_{ii} = (L_2)_{ii}^{-1} (L_1)_{ii}$. Since L_1 and L_2 both have 1's on the diagonal, we have that $(L_2)_{ii}^{-1} (L_1)_{ii} = 1$ for all i and $L_2^{-1} L_1 = U_2 U_1^{-1} = I$. This means $L_1 = L_2$, and $U_1 = U_2$. $\square$

Not every matrix has an LU decomposition. The following theorem shows special matrices for which LU decomposition can be carried out without pivoting.

Theorem 2. *If all the leading minors of a matrix $A \in \mathbb{C}^{n\times n}$ are non-singular, then A has an LU decomposition.*

In particular, the following two classes of matrices satisfy this condition

Definition 3 (**Positive (semi-)definite matrices**)**.** *A matrix $A \in \mathbb{C}^{n\times n}$ is called positive definite (PD) if*

$$\mathbf{x}^* A\mathbf{x} > 0 \quad \forall \mathbf{x} \neq 0.$$

A is called positive semi-definite if equality is also allowed in the equation above, i.e., $\mathbf{x}^* A\mathbf{x} \geq 0 \quad \forall \mathbf{x} \in \mathbb{C}^n$.

It is common to refer to combine positive definiteness and symmetry or conjugate-symmetry. A matrix is called Hermitian (symmetric) positive definite (HPD or SPD) if it is both positive definite and Hermitian (symmetric). HPD matrices are of high importance as they appear in many realistic applications. They are also much easier to treat numerically (for example, they have an LU decomposition). In this course we will assume for simplicity that every positive definite matrix is Hermitian (symmetric).

Lemma 1. *If A is HPD, then Gauss elimination can be carried out without pivoting and A has an LU decomposition.*

Definition 4 (**Strictly diagonally dominant matrices (SDD)**)**.** *A matrix A is strictly diagonally dominant if for every row i*

$$|a_{ii}| > \sum_{j\neq i} |a_{ij}|$$

or for every column j

$$|a_{jj}| > \sum_{i\neq j} |a_{ij}|$$

Lemma 2. *If A is strictly diagonally dominant, then Gauss elimination can be carried out without pivoting, and A has an LU decomposition.*

The proof for both of these lemmas is achieved by the fact that all the principal submatrices of A in both cases are non-singular.

2.3 Partial pivoting and row equilibrium

Motivation The process in Algorithm 2 may fail if one of the pivots $u_{i,i}$ becomes zero. For example, if $a_{11} = 0$. Consider the following case.

$$A\mathbf{x} = \mathbf{b} : \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}.$$

The naive LU algorithm will divide a_{21} by zero in the first step and an error will occur, even though this matrix is invertible. There is a solution, though. The obvious thing to do is to replace the first row with second one that doesn't have a 0 in the first entry. In other words, we should pick a new pivot by permuting either the rows or the columns of the matrix.

Let us look at a similar case where there is no division by zero. Consider the following case where $\epsilon \approx 0$ but $\epsilon \neq 0$.

$$A\mathbf{x} = \mathbf{b} : \begin{bmatrix} \epsilon & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}.$$

The exact solution is $x_1 = \frac{1}{1-\epsilon} \approx 1 + \epsilon$, and $x_2 = 1 - \frac{\epsilon}{1-\epsilon} \approx 1 - \epsilon$.

Applying the GE process will yield

$$\begin{bmatrix} \epsilon & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 2 \end{bmatrix} \quad \underset{\mathbf{r}_2 \leftarrow \mathbf{r}_2 - \frac{1}{\epsilon}\mathbf{r}_1}{\Rightarrow} \quad \begin{bmatrix} \epsilon & 1 \\ 0 & 1 - \frac{1}{\epsilon} \end{bmatrix}, \begin{bmatrix} 1 \\ 2 - \frac{1}{\epsilon} \end{bmatrix} \tag{7}$$

and then $x_2 = \frac{2-\epsilon^{-1}}{1-\epsilon^{-1}} \approx 1$, and $x_1 = \frac{1-x_2}{\epsilon}$. In the computation for x_1 we will get a severe loss of significance and we will loose accuracy. Moreover, the error in the rounded $1 - x_2$ will be multiplied by $\frac{1}{\epsilon}$.

The example above shows that the problem was not even in $\epsilon \approx 0$. The problem was that ϵ was small *compared to the other elements at the same row.* If we apply the same process for the equivalent system (multiply the first row by $\frac{1}{\epsilon}$)

$$\begin{bmatrix} 1 & \frac{1}{\epsilon} \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} \frac{1}{\epsilon} \\ 2 \end{bmatrix} \tag{8}$$

we will get the same answer as before using GE and encounter the same loss of significance.

We will now see a numerical example.

Example 3. *Solve the following system using Gaussian Elimination using 4 significant digits of accuracy*

$$A\mathbf{x} = \mathbf{b} : \begin{bmatrix} 1 & 999 \\ 333 & -212 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 7096 \\ -472.9 \end{bmatrix}.$$

The exact solution is $x_1 = 3.1$, *and* $x_2 = 7.1$.

The first step of Gaussian Elimination will

$$\begin{bmatrix} 1 & 999 \\ 333 & -212 \end{bmatrix}, \begin{bmatrix} 7096 \\ -472.9 \end{bmatrix} \underset{\mathbf{r}_2 \leftarrow \mathbf{r}_2 - 333\mathbf{r}_1}{\Rightarrow} \begin{bmatrix} 1 & 999 \\ 0 & -332900 \end{bmatrix}, \begin{bmatrix} 7096 \\ -236300 \end{bmatrix} \tag{9}$$

That is because, for example, to get $u_{2,2}$

$$round(-212 - round(333 * 999)) = round(-212 - round(332667)) =$$

$$= round(-212 - 332700) = -332900.$$

We get $x_2 = 7.098$, *which is pretty accurate, but*

$$x_1 = 7096 - 999x_2 = 7096 - 7091 = 5,$$

which is far from the right answer 3.1 (60% error).

Partial pivoting In partial pivoting we change the order of the rows such that the pivot that we choose is maximal in absolute value. In our case, the maximum between $a_{11} = 1$ and $a_{21} = 333$ is 333. Meaning, we will replace the first row with the second.

$$\begin{bmatrix} 333 & -212 \\ 1 & 999 \end{bmatrix}, \begin{bmatrix} -472.9 \\ 7096 \end{bmatrix} \underset{\mathbf{r}_2 \leftarrow \mathbf{r}_2 - \frac{1}{333}\mathbf{r}_1}{\Rightarrow} \begin{bmatrix} 333 & -212 \\ 0 & 999.6 \end{bmatrix}, \begin{bmatrix} -472.9 \\ 7097 \end{bmatrix} \tag{10}$$

and the answer that we get now is $x_2 = \frac{7096}{999.7} = 7.1$ and $x_1 = \frac{-472.9+212\cdot 7.1}{333} = ... = 3.099$. The answer is quite accurate.

Remark 1. *(Row equilibrium or scaling) The partial pivoting approach suggested above aims at each iteration choose the largest pivot for numerical stability. The simplest way to increase the numerical stability, is to preceded the LU with a normalization of the rows of the matrix*

A, also known as "Row-equilibrium". That is, we replace the linear system $A\mathbf{x} = \mathbf{b}$ with an equivalent linear system

$$RA\mathbf{x} = R\mathbf{b}$$

where R is a diagonal matrix whose elements $R_{ii} = \frac{1}{\|\mathbf{a}_i\|}$, where $\mathbf{a}_i$ is the i-th row of A, and $\|\cdot\|$ is some vector norm, like the maximum norm. That is, RA should be given to the Gaussian Elimination or LU decomposition algorithms and not A. In the previous example we will initialize the process with:

$$\begin{bmatrix} 1 & \frac{-212}{333} \\ \frac{1}{999} & 1 \end{bmatrix}, \begin{bmatrix} \frac{-472.9}{333} \\ \frac{7096}{999} \end{bmatrix}, \quad \textit{that is, } R = \begin{bmatrix} \frac{1}{333} & 0 \\ 0 & \frac{1}{999} \end{bmatrix},$$

and in this example we've used the ℓ_∞ norm of the rows (maximum absolute value of each row). This will yield the same result as before in our example, but will generally yield better pivots in larger examples. However, in some examples this may lead to worse results than without scaling, and there are many other row-scaling options.

A different approach for row-scaling is to decide on the pivots based on normalized rows ***during the LU factorization****. That is, when we choose a pivot at iteration k from a set of rows, we will first find the norms of the relevant rows $d_i = \|\mathbf{a}_i\|$ (during the factorization), and then choose the maximal element for the pivot based on the scaled values $\frac{a_{i,k}}{d_i}$, instead of the raw values $a_{i,k}$. The scaling will only influence the choice of pivot, and we will not have to use a matrix R as before.*

We note that there are many other, more advanced scaling strategies in the literature. We will ignore the row scaling in these notes for the next examples, but note that such a process may be necessary. In addition, many software packages for computing LU return a row-scaling matrix R as above.

2.4 LU factorization with partial pivoting

The problems in the Gaussian Elimination algorithm will obviously appear in the LU factorization too. Suppose that we encounter a case where the pivot is zero, like

$$A = \begin{bmatrix} 0 & 4 & 2 \\ 10 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

we can permute the rows and get away with the division by 0. We will choose the best pivot (largest in magnitude) in every iteration. But - we need a way to keep track of these changes. For this, we will use "permutation matrices".

A permutation matrix P is a matrix that has exactly one non-zero entry 1.0 in every row and column. It replaces the locations of the entries of a vector. For example the matrix

$$P = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

replaces the first and second entries, and in this case,

$$PA = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 4 & 2 \\ 10 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 10 & 2 & 1 \\ 0 & 4 & 2 \\ 1 & 1 & 1 \end{bmatrix}.$$

The result is a fine matrix that can be LU-decomposed without additional pivoting. this matrix can also be denoted by a permutation vector $\mathbf{p} = [2, 1, 3]$. The following theorem says that any matrix A can be permuted to have an LU decomposition.

Theorem 3 (LU decomposition)**.** *Let $A \in \mathbb{C}^{n\times n}$, then there exists a permutation matrix P s.t*

$$PA = LU,$$

where L and U are lower and upper triangular matrices respectively.

How can we apply Algorithm 2 with partial pivoting? A "vanilla" solution will apply the algorithm as is, and each time the algorithm encounters a 0 pivot, it will define a permutation P_i and restart—in the first time it will be P_1, in the second it will be P_2 and so on. After each time the algorithm will be restarted for the permuted matrix

$$P \cdot A = P_i \cdots P_2 \cdot P_1 \cdot A$$

where P is the product of all permutation matrices, and is also a permutation matrix.

A better solution would be to apply the permutations to the relevant parts of U and L and not restart—this action does not change the lower/upper triangular structure of L and U, since we always perform a permutation of the rows ahead of us in the algorithm. For a numerical stability, we will always choose the largest pivot, so that we divide rows (which contain numerical errors) with the largest number possible. So, at step k we choose the entry with the largest absolute value out of $u_{k-1,k}, ..., u_{n,k}$.

Example 4 (factorization with partial pivoting)**.** *We perform an LU factorization for the matrix A, starting from the L, U, and* $\mathbf{p}$

$$\textit{Init.:} \quad A, U = \begin{bmatrix} 3 & 0 & 2 \\ -10 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \mathbf{p} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

It is clear that in the second step the third row will have to be a pivot because is is the only row with a non-zero value in the second entry. At first - the second row is permuted with the first one, because $|-10|$ *is larger than 3:*

$$\textit{1st permut.:} \; U = \begin{bmatrix} -10 & 0 & 1 \\ 3 & 0 & 2 \\ 1 & 1 & 1 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad P = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \mathbf{p} = \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix}$$

Then, we will apply $\mathbf{r}_2 \leftarrow \mathbf{r}_2 + 0.3\mathbf{r}_1$ *and* $r_3 \leftarrow \mathbf{r}_3 + 0.1\mathbf{r}_1$*. After the first step we will get*

$$1^{st} \textit{ outer it.:} \; U = \begin{bmatrix} -10 & 0 & 1 \\ 0 & 0 & 2.3 \\ 0 & 1 & 1.1 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 \\ -0.3 & 1 & 0 \\ -0.1 & 0 & 1 \end{bmatrix} \quad P = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \mathbf{p} = \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix}$$

Now the second row has to be replaced with the third one (because 1 is greater than 0), and

we get:

$$2^{nd}\ \textit{permut.:}\ U = \begin{bmatrix} -10 & 0 & 1 \\ 0 & 1 & 1.1 \\ 0 & 0 & 2.3 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 \\ -0.1 & 1 & 0 \\ -0.3 & 0 & 1 \end{bmatrix} \quad P = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \quad \mathbf{p} = \begin{bmatrix} 2 \\ 3 \\ 1 \end{bmatrix}.$$

Note that the 1*'s in the diagonal of the* L *matrix are not permuted—we fix them throughout the whole algorithm. After the permutation we also get the result of the second iteration because the matrix* U *is already triangular:*

$$2^{nd}\ \textit{outer it:}\ U = \begin{bmatrix} -10 & 0 & 1 \\ 0 & 1 & 1.1 \\ 0 & 0 & 2.3 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 \\ -0.1 & 1 & 0 \\ -0.3 & 0 & 1 \end{bmatrix} \quad P = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \quad \mathbf{p} = \begin{bmatrix} 2 \\ 3 \\ 1 \end{bmatrix}$$

At the end of the algorithm we have:

$$\textit{Result:}\ PA = \begin{bmatrix} -10 & 0 & 1 \\ 1 & 1 & 1 \\ 3 & 0 & 2 \end{bmatrix} \quad LU = \begin{bmatrix} -10 & 0 & 1 \\ 1 & 1 & 1 \\ 3 & 0 & 2 \end{bmatrix} \quad P = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix},$$

where P *is achieved in code by* `P = eye(3,3)[p,:];`.
Also note that if `y = P*x`, *then* `y = x[p]`.

Solving a linear system with the permutated LU: We can solve any linear system $A\mathbf{x} = \mathbf{b}$ easily by writing

$$A\mathbf{x} = \mathbf{b} \Rightarrow PA\mathbf{x} = P\mathbf{b}. \quad PA = LU \Rightarrow LU\mathbf{x} = P\mathbf{b} \Rightarrow_{\mathbf{y}=U\mathbf{x}} L\mathbf{y} = P\mathbf{b}.$$

and performing one permutation and two triangular matrix inversions

$$\begin{aligned} L\mathbf{y} = P\mathbf{b} \Rightarrow \mathbf{y} &= L^{-1}(P\mathbf{b}) = \texttt{FwdSub}(L, P\mathbf{b}) \\ U\mathbf{x} = \mathbf{y} \Rightarrow \mathbf{x} &= U^{-1}\mathbf{y} = \texttt{BwdSub}(U, \mathbf{y}). \end{aligned}$$

Note that any permutation matrix P is also an orthogonal matrix, that is $P^T P = I$ (or $P^{-1} = P^T$).

Below is a code for partial pivoted LU, possibly with row equilibrium.

```
"""
LU with partial pivoting
"""
function LUp(A::Array,rowscale::Bool = false)
n = size(A,1); # Assuming A is nxn.
r = [];
if rowscale
        r = sqrt.(1.0./(sum(A.^2,dims=2)[:]));
        U = r.*A;
else
        U = copy(A);
end
L = Matrix{eltype(A)}(1.0I,n,n);
p = collect(1:n);
for k=2:n
        for i=k:n
                rowkk = (U[(k-1):n,k-1])[:];
                pivot = findmax(abs.(rowkk))[2] + k - 2;
                if U[pivot,k-1] == 0.0
                        break;
                end
                if pivot != k-1
                        # Permute rows k-1 and pivot in U
                        t = p[pivot];
                        p[pivot] = p[k-1];
                        p[k-1] = t;
                        t = U[k-1,(k-1):n];
                        U[k-1,(k-1):n] = U[pivot,(k-1):n];
                        U[pivot,(k-1):n] = t;
                        t = L[k-1,1:(k-2)];
                        L[k-1,1:(k-2)] = L[pivot,1:(k-2)];
                        L[pivot,1:(k-2)] = t;
                end
                L[i,k-1] = U[i,k-1]/U[k-1,k-1];
                for j=k-1:n
                        U[i,j] = U[i,j] - L[i,k-1]*U[k-1,j];
                end
        end
end
return L,U,p,r
end
```

Full pivoting There are a lot of calculations throughout the LU factorizaion process, and since our computer performs calculations inaccurately, error accumulate along the way. It

can be shown that to get the highest numerical stability, the pivots $u_{k,k}$ should be as large as possible in absolute value. To get that, we can also permute the columns in the same way we permute the rows and get the largest possible pivot (in absolute value) at every step. After all, the linear system

$$\begin{cases} 3x_1 + 4x_2 + 2x_3 = 21 \\ 10x_1 + 2x_2 + x_3 = 53 \\ 1x_1 + x_2 + x_3 = 7 \end{cases} \quad \text{or} \quad \begin{bmatrix} 3 & 4 & 2 \\ 10 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 5 \\ 15 \\ 1 \end{bmatrix} \tag{11}$$

is equivalent to

$$\begin{cases} 3x_1 + 2x_2 + 4x_3 = 21 \\ 10x_1 + x_2 + 2x_3 = 53 \\ 1x_1 + x_2 + x_3 = 7 \end{cases} \quad \text{or} \quad \begin{bmatrix} 3 & -2 & 4 \\ 10 & 1 & 2 \\ 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 5 \\ 15 \\ 1 \end{bmatrix} \tag{12}$$

because all that happened is that x_2 and x_3 switched locations, and the answer remains identical besides that. We will not demonstrate full pivoting in this course and note that in most of the cases, partial pivoting is sufficient for getting an accurate result.

2.5 Other roles of LU factorization and its variants

The LU factorization plays an important role in calculating the determinant of a matrix or determining whether it is singular or not. It is known that for any two matrices A, B

$$\det(AB) = \det(A)\det(B).$$

Hence, if $PA = LU$ then

$$\det(PA) = \det(L)\det(U).$$

It is easy to show that for such LU decomposition we have $\det(P) = \pm 1$, $\det(U) = \Pi_i u_{ii}$, $\det(L) = \Pi_i l_{ii} = 1$, so $\det(A)$ can be easily computed. Similarly, if U has a zero diagonal entry - then A is singular.

Example in code:

```
## Computation of the determinant using LU
A = [3.0 0.0 2 ; -10.0 0.0 1 ; 1 1 1];
(L,U,p) = LUp(A);
I3 = Matrix(1.0I,n,n);
P = I3[p,:];
println("A:");display(A);println();println()
println("L:");display(L); println();println()
println("U:");display(U); println();println()
println("p:");display(p); println();println()
println("det(A) is: ",det(A))
println("det(P) is: ",det(P)) # easy to compute by definition, either -1 or 1.
println("det(U) is: ",det(U)," which is also: ",prod(diag(U))) # multiplication of the diagonal elements
println("det(L) is: ",det(L)); # essentially 1.
det(U)
det(L) # is 1 by definition - L has 1 on diagonal
## Output: ####
# A:
# 3x3 Array{Float64,2}:
#   3.0 0.0  2.0
# -10.0 0.0  1.0
#   1.0 1.0  1.0

# L:
# 3x3 Array{Float64,2}:
#  1.0  0.0  0.0
# -0.1  1.0  0.0
# -0.3  0.0  1.0

# U:
# 3x3 Array{Float64,2}:
# -10.0  0.0  1.0
#  0.0  1.0  1.1
#  0.0  0.0  2.3

# p:
# 3-element Array{Int64,1}:
 # 2
 # 3
 # 1

# det(A) is: -23.0
# det(P) is: 1.0
# det(U) is: -23.0 which is also: -23.0
# det(L) is: 1.0
```

2.6 More examples for LU factorization

Example LU using code Consider the following linear system

$$\begin{cases} 3x_1 & + & 4x_2 & + & 2x_3 & = & 21 \\ 10x_1 & + & 2x_2 & + & x_3 & = & 53 \\ 1x_1 & + & x_2 & + & x_3 & = & 7 \end{cases} \quad \text{or} \quad \begin{bmatrix} 3 & 4 & 2 \\ 10 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 5 \\ 15 \\ 1 \end{bmatrix} \tag{13}$$

The matrix in this example is non-singular, hence there exists a solution to the system. We will follow this example by the following Julia code. First we define the following linear system, and show its solution using Julia's operator that solves it.

```
julia> A = [3.0 4 2 ; 10 2 1 ; 1 1 1]; b = [21.0;53;7];

julia> A
3x3 Array{Float64,2}:
  3.0  4.0  2.0
 10.0  2.0  1.0
  1.0  1.0  1.0

julia> b
3-element Array{Float64,1}:
 21.0
 53.0
  7.0

julia> A\b # solves the system
3-element Array{Float64,1}:
 5.0
 1.0
 1.0
```

Now we multiply the first row by $\frac{10}{3}$ and $\frac{1}{3}$ (or by $\frac{a_{21}}{a_{11}}$ and $\frac{a_{31}}{a_{11}}$), and subtract the answers from the second and third rows respectively.

```
julia> p1 = A[2,1]/A[1,1];  A[2,:] = A[2,:] - A[1,:]*p1;  b[2] = b[2] - b[1]*p1;
julia> p2 = A[3,1]/A[1,1];  A[3,:] = A[3,:] - A[1,:]*p2;  b[3] = b[3] - b[1]*p2;

julia> A
3x3 Array{Float64,2}:
 3.0    4.0        2.0
 0.0  -11.3333    -5.66667
 0.0   -0.333333   0.333333

julia> b
3-element Array{Float64,1}:
  21.0
 -17.0
   0.0
```

Finally, we multiply the first row by $\frac{a_{32}}{a_{22}}$ and subtract it from the third row:

```
julia> p3 = A[3,2]/A[2,2];  A[3,:] = A[3,:] - A[2,:]*p3;  b[3] = b[3] - b[2]*p3;

julia> A
3x3 Array{Float64,2}:
 3.0    4.0      2.0
 0.0  -11.3333  -5.66667
 0.0    0.0      0.5

julia> b
3-element Array{Float64,1}:
  21.0
 -17.0
   0.5
```

Now we can solve the triangular system by solving the 1-variable equation for x_3, then for x_2 using x_3, and then for x_1 (this process of solving an upper triangular system from below upwards is called "backward substitution"). As expected, if we solve the triangular system we get the same answer as before.

```
julia> A\b
3-element Array{Float64,1}:
 5.0
 1.0
 1.0
```

In the process above we manipulated the rows of A by a scalars p, which where based

on the diagonal values a_{ii} along the algorithm. Those diagonal values are called *pivots*. Interestingly, the process above is equivalent to "dividing" A by a lower triangular system:

```
julia> A = [3.0 4 2 ; 10 2 1 ; 1 1 1];

julia> L = [1.0 0.0 0.0; p1 1.0 0.0; p2 p3 1.0]
3x3 Array{Float64,2}:
 1.0       0.0        0.0
 3.33333   1.0        0.0
 0.333333  0.0294118  1.0

julia> L\A
3x3 Array{Float64,2}:
 3.0    4.0      2.0
 0.0  -11.3333  -5.66667
 0.0    0.0      0.5
```

Example for an LU factorization of a singular matrix:

In this example we will see how both algorithms behave when a singular matrix A is given to them. It is easy to show that if only the last few rows of A are linearly dependent of the others, then we will just have a few zero lines at the bottom of U. However, if there is a linearly independent row after a linearly dependent row - then the LU without pivoting is expected to fail (have zero pivot). We will construct a 4×4 matrix from 3 random vectors.

$$\text{Initialization:} \quad A, U = \begin{bmatrix} 0.358 & 0.085 & 0.009 & 0.529 \\ 0.057 & 0.481 & 0.328 & 0.748 \\ 0.415 & 0.566 & 0.337 & 1.277 \\ 0.369 & 0.108 & 0.555 & 0.062 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Note that the third row in this matrix is a sum of the first two, hence the matrix is singular. First, we will run a pivoting-less LU decomposition.

$$\text{1st iter:} \quad U = \begin{bmatrix} 0.359 & 0.085 & 0.009 & 0.529 \\ -6.9\text{e-}18 & 0.468 & 0.327 & 0.664 \\ 0.0 & 0.468 & 0.327 & 0.664 \\ 5.5\text{e-}17 & 0.020 & 0.546 & -0.483 \end{bmatrix} \quad L = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.16 & 1.0 & 0.0 & 0.0 \\ 1.16 & 0.0 & 1.0 & 0.0 \\ 1.03 & 0.0 & 0.0 & 1.0 \end{bmatrix}.$$

$$\text{2nd iter:} \quad U = \begin{bmatrix} 0.359 & 0.085 & 0.009 & 0.529 \\ -6.9\text{e-}18 & 0.468 & 0.327 & 0.664 \\ 0.0 & 0.0 & 0.0 & 0.0 \\ 5.5\text{e-}17 & 0.0 & 0.532 & -0.512 \end{bmatrix} \quad L = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.160 & 1.0 & 0.0 & 0.0 \\ 1.16 & 1.0 & 1.0 & 0.0 \\ 1.03 & 0.044 & 0.0 & 1.0 \end{bmatrix}.$$

$$\text{3rd iter:} \quad U = \begin{bmatrix} 0.359 & 0.085 & 0.009 & 0.529 \\ -6.9\text{e-}18 & 0.468 & 0.327 & 0.664 \\ 0.0 & 0.0 & 0.0 & 0.0 \\ 5.5\text{e-}17 & 0.0 & NaN & NaN \end{bmatrix} \quad L = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.160 & 1.0 & 0.0 & 0.0 \\ 1.16 & 1.0 & 1.0 & 0.0 \\ 1.03 & 0.044 & \text{Inf} & 1.0 \end{bmatrix}.$$

The algorithm failed. It reached a zero row (and pivot), and then divided by zero. With pivoting, the LU factorization will have the following results:

$$\text{1st:} \quad U = \begin{bmatrix} 0.415 & 0.567 & 0.338 & 1.278 \\ 0.0 & 0.403 & 0.282 & 0.572 \\ 0.0 & -0.403 & -0.282 & -0.572 \\ 0.0 & -0.395 & 0.256 & -1.073 \end{bmatrix} \quad L = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.138 & 1.0 & 0.0 & 0.0 \\ 0.862 & 0.0 & 1.0 & 0.0 \\ 0.888 & 0.0 & 0.0 & 1.0 \end{bmatrix} \quad \mathbf{p} = \begin{bmatrix} 3 \\ 2 \\ 1 \\ 4 \end{bmatrix},$$

$$\text{2nd:} \quad U = \begin{bmatrix} 0.415 & 0.567 & 0.338 & 1.278 \\ 0.0 & 0.403 & 0.282 & 0.572 \\ 0.0 & 0.0 & 0.0 & 1.11\text{e-}16 \\ 0.0 & 0.0 & 0.532 & -0.512 \end{bmatrix} \quad L = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.862 & 1.0 & 0.0 & 0.0 \\ 0.138 & -1.0 & 1.0 & 0.0 \\ 0.888 & 0.979 & 0.0 & 1.0 \end{bmatrix} \quad \mathbf{p} = \begin{bmatrix} 3 \\ 1 \\ 2 \\ 4 \end{bmatrix},$$

$$\text{3rd:} \quad U = \begin{bmatrix} 0.415 & 0.567 & 0.338 & 1.278 \\ 0.0 & -0.403 & -0.282 & -0.572 \\ 0.0 & 0.0 & 0.532 & -0.512 \\ 0.0 & 0.0 & 0.0 & 1.1\text{e-}16 \end{bmatrix} \quad L = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.862 & 1.0 & 0.0 & 0.0 \\ 0.888 & 0.979 & 1.0 & 0.0 \\ 0.138 & -1.0 & 0.0 & 1.0 \end{bmatrix} \quad \mathbf{p} = \begin{bmatrix} 3 \\ 1 \\ 4 \\ 2 \end{bmatrix}.$$

2.7 The Cholesky factorization for HPD matrices

We mentioned before that there is an important class of matrices called Hermitian or symmetric positive definite matrices. For these matrices, we have a special LU decomposition that is easier to perform is practice.

Theorem 4. *Let $A \in \mathbb{C}^{n\times n}$ be a Hermitian positive definite then there exists a lower triangular matrix $L \in \mathbb{C}^{n\times n}$ with a positive diagonal such that*

$$A = LL^*.$$

This decomposition is called the Cholesky *factorization and it is unique.*

Remark 2. *If a matrix A is symmetric positive definite, then A can be decomposed as $A = LL^T$, where $L \in \mathbb{R}^{n\times n}$.*

Proof. Because A is HPD, it has a spectral decomposition $A = U\Lambda U^*$, where $U^*U = I$ and $\Lambda = \mathrm{diag}(\lambda_1, ..., \lambda_n)$ for $\lambda_i > 0$. Such a matrix A can be decomposed to

$$A = U(\Lambda)^{\frac{1}{2}}(\Lambda)^{\frac{1}{2}}U^* = U(\Lambda)^{\frac{1}{2}}U^*U(\Lambda)^{\frac{1}{2}}U^* = A^{\frac{1}{2}}A^{\frac{1}{2}} = (A^*)^{\frac{1}{2}}A^{\frac{1}{2}} = (A^{\frac{1}{2}})^*A^{\frac{1}{2}}.$$

It is clear that $A^{\frac{1}{2}}$ is also HPD. We will see later that any HPD matrix B can be decomposed to $B = QR$ where Q is a unitary matrix ($Q^*Q = I$) and R is an upper triangular matrix with a positive diagonal, and they are unique if B is non-singular. Therefore - let $A^{\frac{1}{2}} = QR$ for such Q and R.

$$A = (A^{\frac{1}{2}})^*A^{\frac{1}{2}} = (QR)^*QR = R^*Q^*QR = R^*R.$$

Now define $L = R^*$ and the Cholesky factorization is achieved.

The uniqueness of the Cholesky factorization is proved similarly to the proof of the uniqueness of pivoting-less LU factorization. □

The Cholesky factorization is very important - in some sense L is the square root of the matrix A - and it exists only because A is positive. The advantage of the Cholesky factorization over LU decomposition is straightforward - it requires half of the memory and it does not require pivoting.

The algorithm for the Cholesky factorization is built on the equation for a_{ij}

$$a_{ij} = \sum_{k=1}^{n} l_{ik}\bar{l}_{kj} = \sum_{k=1}^{\min(i,j)} l_{ik}\bar{l}_{kj}. \tag{14}$$

In the first iteration we set $l_{11} = \sqrt{a_{11}}$, and then set the whole first column of L according to Eq. (14) to be $l_{i1} = \frac{a_{i1}}{l_{11}}$. In the i-th row, we assume that all the entries that we've determined so far (columns $1, ..., i-1$ of L) are given and set according to Eq. (14) for $a_{i,i}$

$$l_{ii} = \left(a_{ii} - \sum_{k=1}^{i-1} l_{ik}\bar{l}_{ki}\right)^{\frac{1}{2}}.$$

Following that we set the i-th column of L (set l_{ij} for $j = i+1, ..., n$) as

$$l_{ij} = \frac{1}{l_{ii}} \left(a_{ij} - \sum_{k=1}^{i-1} l_{ik} \bar{l}_{kj} \right),$$

which is again because of Eq. (14)

Algorithm: Cholesky
$\# \; A \in \mathbb{C}^{n \times n}$
Initialize: $L = 0^{n \times n}$.
for $i = 1, ..., n$ **do**
  $l_{ii} = \left(a_{ii} - \sum_{k=1}^{i-1} l_{ik} \bar{l}_{ki} \right)^{\frac{1}{2}}$.
  for $j = i+1, ..., n$ **do**
    $l_{ij} = \frac{1}{l_{ii}} \left(a_{ij} - \sum_{k=1}^{i-1} l_{ik} \bar{l}_{kj} \right)$,
  end
end

Algorithm 3: Cholesky decomposition

Given an LU decomposition $A = LU$, one can extract the diagonal of U, and write $U = D\hat{U}$, where $\hat{U}$ is an upper triangular system with ones on the diagonal and D is a diagonal matrix such that $D_{ii} = U_{ii}$. We obtain

$$A = LD\hat{U}.$$

This factorization is also called LDU factorization. To get a Cholesky factorization, we will denote $\bar{L} = LD^{\frac{1}{2}}$, and $\bar{U} = D^{\frac{1}{2}}\hat{U}$. We obtain $A = \bar{L}\bar{U}$, where both matrices have identical entries on the diagonal. If A is Hermitian positive definite, then $A = A^*$, and hence $\bar{L}\bar{U} = \bar{U}^*\bar{L}^*$. From here, it is easy to show that $\bar{L} = \bar{U}^*$, and we get the Cholesky factorization.

Computing Determinant. Similarly to the LU factorization, the Cholesky factorization can also be used to calculate the determinant of an Hermitian A. If $A = LL^*$, $\det(L) = \Pi_i l_{ii}$, and $\det(A) = \det(L)^2$. Also, if the Cholesky factorization has only positive values on its diagonal - the matrix is HPD. If the factorization fails - the matrix is not positive definite.