In [1]:
```
using DrWatson;
@quickactivate "MATH361Lectures"
using LinearAlgebra, Latexify;
import MATH361Lectures;
```

# Partial Pivoting and Stability

To supplement this lecture, you might want to watch the video lecture on pivoting

## Introduction

Recall from the previous lecture that we can run into issues using LU factorization. For example, we looked at the matrix

$$\begin{bmatrix} 2.0 & 0.0 & 4.0 & 3.0 \\ -2.0 & 0.0 & 2.0 & -13.0 \\ 1.0 & 15.0 & 2.0 & -4.5 \\ -4.0 & 5.0 & -7.0 & -10.0 \end{bmatrix}$$

We can use Gaussian elimination to zero out all of the entries in the first column below the $(1,1)$ entry to get

$$\begin{bmatrix} 2 & 0 & 4 & 3 \\ 0 & 0 & 6 & -10 \\ 0 & 15 & 0 & -6 \\ 0 & 5 & 1 & -4 \end{bmatrix}$$

The next step in LU factorization would have us attempt to zero out the entries in the second column below the $(2,2)$ entry. However, this leads to a division by zero since the $(2,2)$ entry is zero. On the other hand, returning to Gaussian elimination as done in your linear algebra class, one would usually do a row exchange in the next step as follows:

$$\begin{bmatrix} 2 & 0 & 4 & 3 \\ 0 & 0 & 6 & -10 \\ 0 & 15 & 0 & -6 \\ 0 & 5 & 1 & -4 \end{bmatrix} \quad \mapsto \quad \begin{bmatrix} 2 & 0 & 4 & 3 \\ 0 & 5 & 1 & -4 \\ 0 & 15 & 0 & -6 \\ 0 & 0 & 6 & -10 \end{bmatrix}$$

## Partial Pivoting

As the last example shows, a row swap is necessary when the LU factorization algorithm would require division by zero. This happens exactly when a diagonal entry of the matrix is zero just before the elimination step. Suppose that entry $(j,j)$ is zero just before we need to zero out the entries in column $j$ below the diagonal. In this case, we call the $(j,j)$ entry a **pivot element**. Thus, when we have a zero pivot element, we need to swap row $j$ with another row, a process called **row pivoting** or **partial pivoting**. An important question is, which row should we swap with row $j$? First note that the only options from rows $j+1$ to $n$. However, we can not choose just any row from $j+1$ to $n$ because this can lead to stability issues. We illustrate the idea with a simple example.

Consider the linear system $A x = b$ with

$$A = \begin{bmatrix} -\epsilon & 1 \\ 1 & -1 \end{bmatrix}, \; b = \begin{bmatrix} 1-\epsilon \\ 0 \end{bmatrix}$$

where $\epsilon$ is a small positive number.

It is easy to see that $x = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ is the exact solution to this problem. Let's carry out Gaussian elimination to solve the system.

We start with an augmented matrix

$$\begin{bmatrix} -\epsilon & 1 & 1-\epsilon \\ 1 & -1 & 0 \end{bmatrix}$$

and use row operations to obtain

$$\begin{bmatrix} -\epsilon & 1 & 1-\epsilon \\ 0 & -1+\frac{1}{-\epsilon} & 1-1 \end{bmatrix}$$

which implies that $x_2 = 1, x_1 = \frac{(1-\epsilon)-1}{-\epsilon}$. Now in finite precision arithmetic, the expression $(1-\epsilon)-1$ is problematic due to subtractive cancellation. In other words, the computation of adding $\frac{1}{-\epsilon}$ times row 1 to row 2 is ill-conditioned.

Suppose that we swap rows 1 and 2 **before** the elimination step, even though it is not necessary to do so leading to

$$\begin{bmatrix} 1 & -1 & 0 \\ -\epsilon & 1 & 1-\epsilon \end{bmatrix}$$

now row operations produce

$$\begin{bmatrix} 1 & -1 & 0 \\ 0 & 1-\epsilon & 1-\epsilon \end{bmatrix}$$

with solution $x_2 = 1, x_1 = \frac{0-(-1)}{1}$.

The conclusion based on the last example is:

**Important:** When performing elimination in column $j$, swap row $j$ with the row below it whose entry in column $j$ is the largest in absolute value.

This is basically what we add to the LU algorithm in order to include partial pivoting.

## Algebra of Partial Pivoting

We have previously derived that Gaussian elimination (without pivoting) can be obtained by multiplying a matrix $A$ on the left by a sequence of elementary matrices that are lower triangular, resulting in a upper triangular matrix. In order to perform Gaussian elimination with partial pivoting, we need to introduce another type of elementary matrix. A **permutation matrix** is an $n \times n$ matrix with exactly one nonzero value of 1 in each row and column. Equivalently, a permutation matrix is a matrix obtained from the $n \times n$ identity matrix $I$ by permuting either the rows or columns of $I$. Two important facts about permutation matrices (that you will explore in the exercises) are:

1) If $P$ is a permutation matrix, then $P^{-1} = P^T$, so inverting permutation matrices is easy; and

2) The product of two permutation matrices is again a permutation matrix.

Let's illustrate these properties and the use of permutation matrices in Jula.

In [2]:
```
Ifour = Matrix{Float64}(I,4,4) # the four by four identity matrix
```

Out[2]:
```
4×4 Matrix{Float64}:
 1.0  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  1.0
```

In [3]:
```
# construct a permutation matrix that interchanges the first and last rows
P14 = Ifour[[4,2,3,1],:]
```

Out[3]:
```
4×4 Matrix{Float64}:
 0.0  0.0  0.0  1.0
 0.0  1.0  0.0  0.0
 0.0  0.0  1.0  0.0
 1.0  0.0  0.0  0.0
```

In [4]:
```
# illustrate its use
M = [1 1 1 1;2 2 2 2;3 3 3 3;4 4 4 4]
P14*M
```

Out[4]:
```
4×4 Matrix{Float64}:
 4.0  4.0  4.0  4.0
 2.0  2.0  2.0  2.0
 3.0  3.0  3.0  3.0
 1.0  1.0  1.0  1.0
```

In [5]:
```
#notice that transpose(P14)=inverse(P14)
P14'*P14
```

Out[5]:
```
4×4 Matrix{Float64}:
 1.0  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  1.0
```

In [6]:
```
# the matrix which should interchange the second and third rows
P23 = Ifour[[1,3,2,4],:]
```

Out[6]:
```
4×4 Matrix{Float64}:
 1.0  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  1.0
```

In [7]:
```
P23*M
```

Out[7]:
```
4×4 Matrix{Float64}:
 1.0  1.0  1.0  1.0
 3.0  3.0  3.0  3.0
 2.0  2.0  2.0  2.0
 4.0  4.0  4.0  4.0
```

In [8]:
```
# notice that P14 times P23 is again a permutation matrix
P14*P23
```

Out[8]:
```
4×4 Matrix{Float64}:
 0.0  0.0  0.0  1.0
 0.0  0.0  1.0  0.0
 0.0  1.0  0.0  0.0
 1.0  0.0  0.0  0.0
```

In [9]:
```
# which permutation matrix is it?
P14*P23*M
```

Out[9]:
```
4×4 Matrix{Float64}:
 4.0  4.0  4.0  4.0
 3.0  3.0  3.0  3.0
 2.0  2.0  2.0  2.0
 1.0  1.0  1.0  1.0
```

## LU Factorization with Partial Pivoting

LU factorization with partial pivoting is implemented in the base Julia package `LinearAlgebra.jl`. For the sake of completeness we illustrate the LU factorization with partial pivoting algorithm by coding it from scratch and comparing it's use with the base Julia implementation.

In [10]:
```
function luppfact(A)
    m,n = size(A);              # number of rows and columns
    P = Matrix{Float64}(I,n,n); # initialize P
    U = Matrix{Float64}(A);     # initialize U
    L = Matrix{Float64}(I,n,n); # initialize L
    for k=1:m-1
        ind = k;
        pivot=maximum(abs.(U[k:m,k]));
        for j=k:m
            if(abs(U[j,k])==pivot)
                ind=j;
                break
            end
        end
        U[[k,ind],k:m]=U[[ind,k],k:m];
        L[[k,ind],1:k-1]=L[[ind,k],1:k-1]
        P[[k,ind],:]=P[[ind,k],:]
        for j=k+1:m
            L[j,k]=U[j,k]/U[k,k];
            U[j,k:m]=U[j,k:m] - L[j,k]*U[k,k:m];
        end
    end
    return L, U, P
end
```

Out[10]:
```
luppfact (generic function with 1 method)
```

In [11]:
```
A = [2 0 4 3; -2 0 2 -13; 1 15 2 -4.5; -4 5 -7 -10];
latexify(A)
```

Out[11]:

$$\begin{bmatrix} 2.0 & 0.0 & 4.0 & 3.0 \\ -2.0 & 0.0 & 2.0 & -13.0 \\ 1.0 & 15.0 & 2.0 & -4.5 \\ -4.0 & 5.0 & -7.0 & -10.0 \end{bmatrix} \tag{1}$$

In [12]:
```
L,U,P = luppfact(A);
```

In [13]:
```
L1,U1,P1 = lu(A);
```

In [14]:
```
latexify(L) |> display
latexify(L1)|> display
```

$$\begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ -0.25 & 1.0 & 0.0 & 0.0 \\ 0.5 & -0.15384615384615385 & 1.0 & 0.0 \\ -0.5 & 0.15384615384615385 & 0.08333333333333334 & 1.0 \end{bmatrix} \tag{2}$$

$$\begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ -0.25 & 1.0 & 0.0 & 0.0 \\ 0.5 & -0.15384615384615385 & 1.0 & 0.0 \\ -0.5 & 0.15384615384615385 & 0.08333333333333336 & 1.0 \end{bmatrix} \tag{3}$$

In [15]:
```
latexify(U) |> display
latexify(U1)|> display
```

$$\begin{bmatrix} -4.0 & 5.0 & -7.0 & -10.0 \\ 0.0 & 16.25 & 0.25 & -7.0 \\ 0.0 & 0.0 & 5.538461538461538 & -9.076923076923077 \\ 0.0 & 0.0 & 0.0 & -0.1666666666666664 \end{bmatrix} \tag{4}$$

$$\begin{bmatrix} -4.0 & 5.0 & -7.0 & -10.0 \\ 0.0 & 16.25 & 0.25 & -7.0 \\ 0.0 & 0.0 & 5.538461538461538 & -9.076923076923077 \\ 0.0 & 0.0 & 0.0 & -0.16666666666666607 \end{bmatrix} \tag{5}$$

In [16]:
```
latexify(P) |> display
latexify(Matrix{Float64}(I,4,4)[P1,:]) |> display
```

$$\begin{bmatrix} 0.0 & 0.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 1.0 & 0.0 & 0.0 & 0.0 \end{bmatrix} \tag{6}$$

$$\begin{bmatrix} 0.0 & 0.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 1.0 & 0.0 & 0.0 & 0.0 \end{bmatrix} \tag{7}$$

In order to anlayse the stability and conditioning for linear algebra problems and numerical linear algebra algorithms, we need a way to measure error and the magnitude of perturbations. The mathematical tools used to do this are matrix norms, the topic of our next lecture. In preparation for the next lecture, you might want to watch this video.