Lecture 12: PLU Decomposition

For MATH3976 students: in the assignment, we will be using a bitstype. The following creates a new type of precisely 128 bits, that is a subtype of `AbstractFloat`:

In [39]:

```
bitstype 128 Float128 <: AbstractFloat
```

To create a `Float128`, we need to reinterpret another 128-bit type. The easiest case is `UInt128`:

In [43]:

```
u_int=rand(UInt128)
f=reinterpret(Float128,u_int);

typeof(f)
```

Out[43]:

```
Float128
```

We can manipulate `f` by reinterpreting it back to a `UInt128`:

In [44]:

```
reinterpret(UInt128,f)
```

Out[44]:

```
0x4488af451cad267a956f4096d9ef0c36
```

We see that it has exactly 128 bits:

In [45]:

```
bits(u_int)
```

Out[45]:

```
"01000100100010001010111101000101000111001010110100100110011110101
0010101011011110100000010010110110110011110111100001100000110110"
```

We will need to access subsequences of the bits. In the following example, we decompose a 32-bit unsigned integer into two 16-bit unsigned integers.

In [46]:

```
x=rand(UInt32)

bits(x)
```

Out[46]:

```
"01011101000011011101100100011001"
```

The syntax `x % UInt16` drops the first 16 bits, and returns the last 16 bits as a `UInt16`:

In [47]:

```
x_16 = x % UInt16   # drops the first 15 bits, and keep the last 16 bits

bits(x_16)  # same as the last 16 bits of x
```

Out[47]:

```
"1101100100011001"
```

To get at the first 16 bits, we will need to shift the bits right. This is equivalent to dividing by two and dropping the extra bits:

In [48]:

```
bits(UInt32(div(x,2)))
```

Out[48]:

```
"00101110100001101110110010001100"
```

But it is more convenient to use `x >> k`, which shifts the bits right by `k`:

In [49]:

```
x_shift = x >> 1  # shifts the bits of x by 1, dropping the rightmost bit
bits(x_shift)
```

Out[49]:

```
"00101110100001101110110010001100"
```

In [50]:

```
x_shift = x >> 2  # shifts the bits of x by 2, dropping the rightmost two bi
ts
bits(x_shift)
```

Out[50]:

```
"00010111010000110111011001000110"
```

We thus get the first and last 16 bits as follows:

In [51]:

```
x_shift = x >> 16
x_first = x_shift % UInt16
x_last = x % UInt16

bits(x)
```

Out[51]:

```
"01011101000011011101100100011001"
```

In [52]:

```
bits(x_first) * bits(x_last)
```

Out[52]:

```
"01011101000011011101100100011001"
```

In [54]:

```
bits(x_first),bits(x_last)
```

Out[54]:

```
("0101110100001101","1101100100011001")
```

PLU Decomposition

The LU Decomposition breaks down when there is a zero on the diagonal. The PLU Decomposition consists of permuting the rows to put the largest entry on the diagonal. For example, if we have the matrix

$$A = \begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix}$$

we can multiply it on the left by the permutation matrix that exchanges the rows 1 and 3:

$$P_1 = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

So that

$$P_1 A = \begin{pmatrix} 6 & 7 & 8 \\ 3 & 4 & 5 \\ 0 & 1 & 2 \end{pmatrix}$$

In [3]:

```
A=[0 1 2;
   3 4 6;
   6 7 8]
P1=[0 0 1;
    0 1 0;
    1 0 0]

P1*A
```

Out[3]:

```
3x3 Array{Int64,2}:
 6  7  8
 3  4  6
 0  1  2
```

We can now apply a lower triangular operation

$$L_1 = \begin{pmatrix} 1 & & \\ -\frac{3}{6} & 1 & \\ 0 & 0 & 1 \end{pmatrix}$$

to introduce zeros:

In [5]:

```
L1=[1 0 0;
    -3/6 1 0;
     0   0  1]

L1*P1*A
```

Out[5]:

```
3x3 Array{Float64,2}:
 6.0  7.0  8.0
 0.0  0.5  2.0
 0.0  1.0  2.0
```

In general, we interchange the first row with the row with maximum entry:

In [ ]:

```
n=5

A=rand(n,n)
```

In [34]:

```
P=Array(Matrix{Float64},n)  # A vector of matrices that will hold our permut
ation matrices
L=Array(Matrix{Float64},n)  # A vector of matrices that will hold our lower
triangular matrices




mx=findmax(A[:,1])[2]   # max row
p=[mx;(2:mx-1);1;(mx+1:n)]  # the permutaton
P[1]=eye(n)[:,p]   # permutation matrix
B=P[1]*A  # has largest entry in first row
```

Out[34]:

```
5x5 Array{Float64,2}:
 0.932944   0.0274823  0.455683  0.2833     0.664665
 0.0288843  0.53984    0.904725  0.551841   0.497234
 0.67117    0.0728994  0.898272  0.0395353  0.518045
 0.599921   0.734369   0.159317  0.160247   0.402057
 0.491023   0.0518996  0.985967  0.0378612  0.935159
```

For $B = P_1 A$, we now create a lower triangular matrix that introduces zeros in the first column:

$$L_1 = \begin{pmatrix} 1 & & & & \\ -\frac{B_{2,1}}{B_{1,1}} & 1 & & & \\ -\frac{B_{3,1}}{B_{1,1}} & & 1 & & \\ \vdots & & & \ddots & \\ -\frac{B_{n,1}}{B_{1,1}} & & & & 1 \end{pmatrix}$$

In [35]:

```
L[1]=eye(n)
L[1][2:end,1]=-B[2:end,1]/B[1,1]
L[1]
```

Out[35]:

```
5x5 Array{Float64,2}:
  1.0         0.0  0.0  0.0  0.0
 -0.0309604   1.0  0.0  0.0  0.0
 -0.719411    0.0  1.0  0.0  0.0
 -0.643041    0.0  0.0  1.0  0.0
 -0.526316    0.0  0.0  0.0  1.0
```

In [36]:

```
L[1]*P[1]*A
```

Out[36]:

```
5x5 Array{Float64,2}:
 0.932944  0.0274823   0.455683   0.2833     0.664665
 0.0       0.538989    0.890616   0.543069   0.476655
 0.0       0.0531283   0.570448  -0.164274   0.0398784
 0.0       0.716696   -0.133706  -0.021926  -0.0253499
 0.0       0.0374352   0.746134  -0.111244   0.585336
```

We now proceed with remaining columns, always leaving the first column alone. First find the largest entry in rows 2:n

In [37]:

```
C=L[1]*P[1]*A

mx=findmax(C[2:end,2])[2]+1   # max row
p=[1;mx;(3:mx-1);2;(mx+1:n)]  # the permutaton
P[2]=eye(n)[:,p]   # permutation matrix
P[2]*L[1]*P[1]*A
```

Out[37]:

```
5x5 Array{Float64,2}:
 0.932944  0.0274823   0.455683   0.2833     0.664665
 0.0       0.716696   -0.133706  -0.021926  -0.0253499
 0.0       0.0531283   0.570448  -0.164274   0.0398784
 0.0       0.538989    0.890616   0.543069   0.476655
 0.0       0.0374352   0.746134  -0.111244   0.585336
```

Now introduce zeros to $D = P_2 L_1 P_1 A$ using

$$L_2 = \begin{pmatrix} 1 & & & & & \\ & 1 & & & & \\ & -\frac{D_{3,2}}{D_{2,2}} & 1 & & & \\ & -\frac{D_{4,2}}{D_{2,2}} & & 1 & & \\ & \vdots & & & \ddots & \\ & -\frac{D_{n,2}}{D_{2,2}} & & & & 1 \end{pmatrix}$$

In [38]:

```
D=P[2]*L[1]*P[1]*A

L[2]=eye(n)
L[2][3:end,2]=-D[3:end,2]/D[2,2]
L[2]
```

Out[38]:

```
5x5 Array{Float64,2}:
 1.0   0.0         0.0  0.0  0.0
 0.0   1.0         0.0  0.0  0.0
 0.0  -0.0741294  1.0  0.0  0.0
 0.0  -0.752047   0.0  1.0  0.0
 0.0  -0.052233   0.0  0.0  1.0
```

In [39]:

```
L[2]*P[2]*L[1]*P[1]*A
```

Out[39]:

```
5x5 Array{Float64,2}:
  0.932944     0.0274823     0.455683   0.2833     0.664665
  0.0          0.716696     -0.133706  -0.021926  -0.0253499
 -1.11022e-16  0.0           0.58036   -0.162648   0.0417575
  0.0          0.0           0.991169   0.559559   0.49572
  5.55112e-17  3.46945e-18   0.753118  -0.110099   0.58666
```

We continue on with the remaining columns:

In [46]:

```
j=3  # 3rd column



E=A
for l=1:j-1
    E=L[l]*P[l]*E
end

E  # the current updated matrix L[j-1]*P[j-1]*…*L[1]*P[1]*A
```

Out[46]:

```
5x5 Array{Float64,2}:
 0.932944  0.0274823   0.455683   0.2833     0.664665
 0.0       0.716696   -0.133706  -0.021926  -0.0253499
 0.0       0.0         0.58036   -0.162648   0.0417575
 0.0       0.0         0.991169   0.559559   0.49572
 0.0       0.0         0.753118  -0.110099   0.58666
```

In [57]:

```
mx=findmax(E[j:end,j])[2]+j-1   # max row
p=[1:j-1;mx;(j+1:mx-1);j;(mx+1:n)]  # the permutaton
P[j]=eye(n)[:,p]   # permutation matrix


F=P[j]*E  # has max entry in the third column on diagonal
```

Out[57]:

```
5x5 Array{Float64,2}:
 0.932944  0.0274823   0.455683   0.2833     0.664665
 0.0       0.716696   -0.133706  -0.021926  -0.0253499
 0.0       0.0         0.991169   0.559559   0.49572
 0.0       0.0         0.58036   -0.162648   0.0417575
 0.0       0.0         0.753118  -0.110099   0.58666
```

In [58]:

```
L[j]=eye(n)
L[j][j+1:end,j]=-F[j+1:end,j]/F[j,j]
L[j]  # introduces zeros in the jth column
```

Out[58]:

```
5x5 Array{Float64,2}:
 1.0  0.0   0.0       0.0  0.0
 0.0  1.0   0.0       0.0  0.0
 0.0  0.0   1.0       0.0  0.0
 0.0  0.0  -0.58553   1.0  0.0
 0.0  0.0  -0.759828  0.0  1.0
```

In [59]:

```
L[j]*P[j]*E
```

Out[59]:

```
5x5 Array{Float64,2}:
 0.932944  0.0274823   0.455683   0.2833     0.664665
 0.0       0.716696   -0.133706  -0.021926  -0.0253499
 0.0       0.0         0.991169   0.559559   0.49572
 0.0       0.0         0.0       -0.490287  -0.248501
 0.0       0.0         0.0       -0.535267   0.209998
```

We thus have the following for loop to calculate the decomposition

$$U = L_{n-1} P_{n-1} \cdots L_1 P_1 A$$

In [86]:

```
E=A

P=Array(Matrix{Float64},n-1)  # A vector of matrices that will hold our perm
utation matrices
L=Array(Matrix{Float64},n-1)  # A vector of matrices that will hold our lowe
r triangular matrices




for j=1:4
    # create P[j]
    mx=findmax(E[j:end,j])[2]+j-1   # max row
    if mx == j
        P[j]=eye(n)
    else
        p=[1:j-1;mx;(j+1:mx-1);j;(mx+1:n)]  # the permutaton
        P[j]=eye(n)[:,p]   # permutation matrix
    end

    F=P[j]*E  # has max entry in the third column on diagonal

    # create L[j]
    L[j]=eye(n)
    L[j][j+1:end,j]=-F[j+1:end,j]/F[j,j]

    E=L[j]*F
end

U=E
```

Out[86]:

```
5x5 Array{Float64,2}:
 0.932944  0.0274823   0.455683   0.2833     0.664665
 0.0       0.716696   -0.133706  -0.021926  -0.0253499
 0.0       0.0         0.991169   0.559559   0.49572
 0.0       0.0         0.0       -0.490287  -0.248501
 0.0       0.0         0.0        0.0        0.481298
```

Indeed:

In [92]:

```
norm(L[4]*P[4]*L[3]*P[3]*L[2]*P[2]*L[1]*P[1]*A-U)
```

Out[92]:

```
2.305180457824972e-16
```

Note that inverting each `L` is just negating its sub-diagonal entries, for example:

$$L_2^{-1} = \begin{pmatrix} 1 & & & & & \\ & 1 & & & & \\ & -\frac{D_{3,2}}{D_{2,2}} & 1 & & & \\ & -\frac{D_{4,2}}{D_{2,2}} & & 1 & & \\ & \vdots & & & \ddots & \\ & -\frac{D_{n,2}}{D_{2,2}} & & & & 1 \end{pmatrix}^{-1} = \begin{pmatrix} 1 & & & & & \\ & 1 & & & & \\ & \frac{D_{3,2}}{D_{2,2}} & 1 & & & \\ & \frac{D_{4,2}}{D_{2,2}} & & 1 & & \\ & \vdots & & & \ddots & \\ & \frac{D_{n,2}}{D_{2,2}} & & & & 1 \end{pmatrix}$$

We thus construct the inverses:

In [104]:

```
Li=Array(Matrix{Float64},n-1)

for j=1:n-1
    Li[j]=eye(n)
    Li[j][j+1:end,j]=-L[j][j+1:end,j]
end

Li[2]*L[2]
```

Out[104]:

```
5x5 Array{Float64,2}:
 1.0  0.0  0.0  0.0  0.0
 0.0  1.0  0.0  0.0  0.0
 0.0  0.0  1.0  0.0  0.0
 0.0  0.0  0.0  1.0  0.0
 0.0  0.0  0.0  0.0  1.0
```

We thus get the decomposition (using the fact that $P_j^\top = P_j = P_j^{-1}$)

$$A = P_1 L_1^{-1} P_2 \cdots P_{n-1} L_{n-1}^{-1} * U$$

In [106]:

```
norm(A-P[1]*Li[1]*P[2]*Li[2]*P[3]*Li[3]*P[4]*Li[4]*U)
```

Out[106]:

```
1.1102230246251565e-16
```

We now want to interchange the P_j and L_j^{-1} to get PLU. The key idea is that each `P[j]` leaves the first `j-1` rows alone, so satisfies `P[j][1:j-1,1:j-1]==eye(j-1)`. At the same time, L_j^{-1} satisfies and `L[j][j:end,j:end]==eye(n-j)`:

In [117]:

```
Li[1],P[2]
```

Out[117]:

```
(

5x5 Array{Float64,2}:
 1.0        0.0  0.0  0.0  0.0
 0.0309604  1.0  0.0  0.0  0.0
 0.719411   0.0  1.0  0.0  0.0
 0.643041   0.0  0.0  1.0  0.0
 0.526316   0.0  0.0  0.0  1.0,

5x5 Array{Float64,2}:
 1.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  1.0  0.0
 0.0  0.0  1.0  0.0  0.0
 0.0  1.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  1.0)
```

Thus we can interchange:

$$L_{j-1}^{-1}P_j = \begin{pmatrix} I_{j-2} & & \\ & 1 & \\ & \mathbf{l}_{j-1} & I_{n-j+1} \end{pmatrix} \begin{pmatrix} I_{j-1} & \\ & \tilde{P}_{n-j+1} \end{pmatrix} = \begin{pmatrix} I_{j-1} & \\ & \tilde{P}_{n-j+1} \end{pmatrix} \begin{pmatrix} I_{j-2} & & \\ & 1 & \\ & (\tilde{P}_{n-j+1}^\top \mathbf{l}_{j-1}) & I_{n-j} \end{pmatrix}$$

In [118]:

```
Li[1]*P[2]
```

Out[118]:

```
5x5 Array{Float64,2}:
 1.0        0.0  0.0  0.0  0.0
 0.0309604  0.0  0.0  1.0  0.0
 0.719411   0.0  1.0  0.0  0.0
 0.643041   1.0  0.0  0.0  0.0
 0.526316   0.0  0.0  0.0  1.0
```

Thus we have a modified L matrices:

In [150]:

```
L1=eye(n)
L1[2:n,1]=P[2][2:end,2:end]'*Li[1][2:n,1]

P[2]*L1-Li[1]*P[2]
```

Out[150]:

```
5x5 Array{Float64,2}:
 0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
```

We thus work inductively from the back: first interchange L_{n-2}^{-1} and P_{n-1}:

In [154]:

```
L̃=Array(Matrix{Float64},n-1)
L̃[n-1]=Li[n-1]

j=4

L̃[j-1]=eye(n)
L̃[j-1][j:n,j-1]=P[j][j:end,j:end]'*Li[j-1][j:n,j-1]

norm(A-P[1]*Li[1]*P[2]*Li[2]*P[3]*P[4]*L̃[3]*L̃[4]*U)
```

Out[154]:

```
1.1102230246251565e-16
```

Now interchange L_2^{-1} with $\tilde{P} = P_3 P_4$:

In [155]:

```
j=3

P̃=P[3]*P[4]

L̃[j-1]=eye(n)
L̃[j-1][j:n,j-1]=P̃[j:end,j:end]*Li[j-1][j:n,j-1]

norm(A-P[1]*Li[1]*P[2]*P̃*L̃[2]*L̃[3]*L̃[4]*U)
```

Out[155]:

```
1.1102230246251565e-16
```

Now interchange L_1^{-1} with $\tilde{P} = P_2 P_3 P_4$:

In [156]:

```
j=2


P̃=P[2]*P[3]*P[4]

L̃[j-1]=eye(n)
L̃[j-1][j:n,j-1]=P̃[j:end,j:end]'*Li[j-1][j:n,j-1]

norm(A-P[1]*P̃*L̃[1]*L̃[2]*L̃[3]*L̃[4]*U)
```

Out[156]:

```
1.1102230246251565e-16
```

We thus get the PLU Decomposition:

In [157]:

```
P=P[1]*P[2]*P[3]*P[4]
L=L̃[1]*L̃[2]*L̃[3]*L̃[4]

norm(P*L*U-A)
```

Out[157]:

```
1.1102230246251565e-16
```

Matrix norms

Just like vectors, matrices have norms that measure their "length". The simplest example is the Fr\"obenius norm, defined for an $n \times m$ real matrix A as

$$\|A\|_F = \|\text{vec}(A)\|_2 = \sqrt{\sum_{k=1}^{n} \sum_{j=1}^{m} A_{kj}^2}$$

This is using Julia's `vec` notation, which converts a matrix to a vector:

In [58]:

```
vec([1 2 3; 4 5 6; 7 8 9])
```

Out[58]:

```
9-element Array{Int64,1}:
 1
 4
 7
 2
 5
 8
 3
 6
 9
```

While this is the simplest norm, it is not the most useful. In a lecture we will describe which norm is used in Julia.

The important thing for us is that if $\|A\| = 0$ then $A = 0$, so it can be used to test if two matrices are equal: if $\|A - B\| \approx 0$, then $A \approx B$:

In [59]:

```
A=rand(5,5)

Q,R=qr(A)

norm(Q*R-A)
```

Out[59]:

```
5.507977739769666e-16
```