LU factorization...

$\boxed{LU = PA}$

using function lu(A) in Julia...

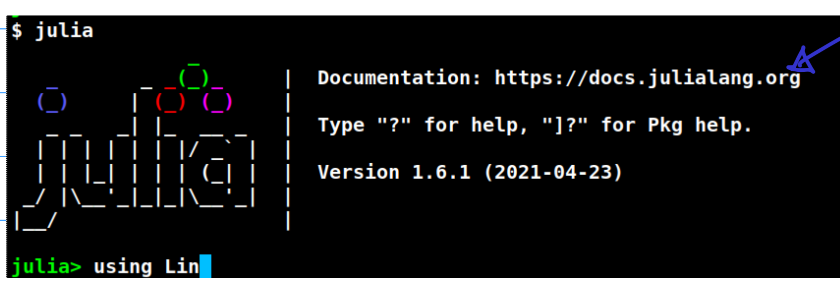
Note P is a permutation matrix and $P^{-1} = P^T$

- Not all matrices can you find the inverse by transposing them.
- The ones where this works are called orthogonal matrices
- The columns of an orthogonal matrix is a family of orthonormal vectors...

Gaussian Elimination

Gaussian Elimination with partial pivoting...

That's what lu does...



up to date documentation.

Note that stackoverflow and other websites sometimes contain obsolete information since Julia is being developed so quickly...

press tab key to ask the built-in AI to complete what you're typing..

```
julia> using LinearAlgebra
```

The linear algebra package includes lu factorization

```
julia> A=rand(3,3)
3×3 Matrix{Float64}:
 0.27815   0.468196  0.408535
 0.372304  0.145557  0.356158
 0.877041  0.777591  0.812321
```

create a random 3x3 matrix

```
julia> lu(A)
LU{Float64, Matrix{Float64}}
L factor:
3×3 Matrix{Float64}:
 1.0       0.0       0.0
 0.317146  1.0       0.0
 0.4245   -0.832771  1.0
U factor:
3×3 Matrix{Float64}:
 0.877041  0.777591  0.812321
 0.0       0.221586  0.150911
 0.0       0.0       0.137002
```

use lu to find the factorization

Note the built-in factorization routine uses partial pivoting (it swaps rows) to reduce rounding errors.

Choices from Math 330 linear algebra include

1. No pivoting
2. Partial pivoting (swap rows)
3. Full pivoting (swap columns and rows)

Lets assign the result of lu factorization to a variable...

```
julia> z=lu(A)
LU{Float64, Matrix{Float64}}
L factor:
3×3 Matrix{Float64}:
 1.0       0.0       0.0
 0.317146  1.0       0.0
 0.4245   -0.832771  1.0
U factor:
3×3 Matrix{Float64}:
 0.877041  0.777591  0.812321
 0.0       0.221586  0.150911
 0.0       0.0       0.137002
```

What's actually contained in z?

```
julia> typeof(z)
LU{Float64, Matrix{Float64}}

julia> propertynames(z)
(:L, :U, :p, :P)
```

The matrix L

The matrix U

permutation vector

permutation matrix

The type of z is an LU factorization — unsurprising, but what's that?

The propertynames function tells more...

Note different versions of Julia have changed the way this works over time — if in doubt, go to the documentation website

Examine the different parts of z ...

```
julia> z.p
3-element Vector{Int64}:
 3
 1
 2

julia> z.P
3x3 Matrix{Float64}:
 0.0  0.0  1.0
 1.0  0.0  0.0
 0.0  1.0  0.0

julia> z.U
3x3 Matrix{Float64}:
 0.877041  0.777591  0.812321
 0.0       0.221586  0.150911
 0.0       0.0       0.137002

julia> z.L
3x3 Matrix{Float64}:
 1.0       0.0       0.0
 0.317146  1.0       0.0
 0.4245   -0.832771  1.0
```

These two field give the same information in two different forms...

Note permutation matrices are a type of orthogonal matrix... That means the columns are orthonormal vectors... and that $P^{-1} = P^T$.

```
julia> z.P'*z.P
3x3 Matrix{Float64}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0
```

Check orthogonality property of P.

Since this is LU factorization with partial pivoting the rows have been swapped. Thus...

```
julia> z.L*z.U
3x3 Matrix{Float64}:
 0.877041  0.777591  0.812321
 0.27815   0.468196  0.408535
 0.372304  0.145557  0.356158
```

LU gives the original matrix, except the rows are out of order...

same

```
julia> A
3x3 Matrix{Float64}:
 0.27815   0.468196  0.408535
 0.372304  0.145557  0.356158
 0.877041  0.777591  0.812321

julia> z.P*A
3x3 Matrix{Float64}:
 0.877041  0.777591  0.812321
 0.27815   0.468196  0.408535
 0.372304  0.145557  0.356158
```

P tells how the rows were swapped...

Thus $LU = PA$ ← usual result from Math 330 ...

Even though there's a good lu factorization library built in to Julia we now write our own. Why? That's crazy...

Reasons: ① Remember Gaussian Elimination from Math 330

② Learn some details of writing Julia programs...

For simplicity — let's do Gaussian elimination No pivoting. again - this is not a practical thing to do, but a learning activity...

Recall Example

pivot

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & -2 & -5 \\ 1 & 0 & 8 \end{bmatrix}$$

Elimination steps...

$r_2 \leftarrow r_2 - 2\, r_1$

$r_3 \leftarrow r_3 - 1\, r_1$

next pivot

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & -6 & -11 \\ 0 & -2 & 5 \end{bmatrix}$$

$r_3 \leftarrow r_3 - \frac{-2}{-6} r_2$

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & -6 & -11 \\ 0 & 0 & ? \end{bmatrix}$$

Need to program this pattern of elimination steps...

For each column eliminate the rows under the pivot

store the size of the matrix in convenient variables...

```
julia> rows,cols=size(A)
(3, 3)

julia> for j=1:cols-1
           for i=j+1:row
               println("r$i <- r$i - ? r$j")
           end
       end
ERROR: UndefVarError: row not defined
Stacktrace:
 [1] top-level scope
   @ ./REPL[15]:2
```

oops misspelled rows...

Press up arrow and the back arrow to edit the block of code

```
julia> for j=1:cols-1
           for i=j+1:rows
               println("r$i <- r$i - ? r$j")
           end
       end
```

Press return to run it again...

```
julia> for j=1:cols-1
           for i=j+1:rows
               println("r$i <- r$i - ? r$j")
           end
       end
r2 <- r2 - ? r1
r3 <- r3 - ? r1
r3 <- r3 - ? r2
```

That's the pattern we want, now we need to fill in what the "?"'s are with the multipliers used for the elimination steps.

next pivot

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & \boxed{-6} & -11 \\ 0 & \boxed{-2} & 5 \end{bmatrix}$$

$$r_3 \leftarrow r_3 - \frac{-2}{-6} r_2$$

Before doing that, lets start a file with the program in it...

```
julia> pwd()
"/x/libb/ejolson/teach/466/2021/sep14"
```

Check what directory Julia is running from...
This will be different on your own computer or the lab computers...

I like to organize my directories for a class by date... that way if a student has a question, I can go back to the same day by date to see what code we wrote that day...

If Julia is in the wrong directory...you can change to the correct one within Julia by using

cd("newdirectory")

Open another window, change to the same directory that Julia is using and then start the editor... In the lab that would be

```
$ cd sep14
$ nano gauss.jl
```

Change to the correct directory

start the editor

let's call our program gauss.jl.

On my personal computer I'll use vi instead. You are free to use whatever program editor is convenient on your computer as well...

Some people prefer an integrated development environment such as Atom or similar. Since the REPL (read evaluate print loop) in Julia has such a good AI, I prefer to use an editor in one windows and a copy of Julia in another window... do what you like best...

```
vi gauss.jl
rows,cols=size(A)
for j=1:cols-1
    for i=j+1:rows
        println("r$i <- r$i - ? r$j")
    end
end
~
6,1                 All
```

use the mouse to copy what we've done so far into the editor and save it...

```
julia> A=rand(4,4)
4x4 Matrix{Float64}:
 0.459076  0.201025  0.192312  0.464086
 0.783558  0.544086  0.450489  0.0602417
 0.127198  0.916158  0.272742  0.535181
 0.432221  0.220349  0.315929  0.373361

julia> readdir()
2-element Vector{String}:
 ".gauss.jl.swp"
 "gauss.jl"

julia> include("gauss.jl")
r2 <- r2 - ? r1
r3 <- r3 - ? r1
r4 <- r4 - ? r1
r3 <- r3 - ? r2
r4 <- r4 - ? r2
r4 <- r4 - ? r3
```

create a bigger matrix...

Check the file gauss.jl in in the directory and ready to run...

run it by using the include command.

The correct pattern for gaussian elimination on a 4x4 matrix...

Since we are making an LU factorization, we need to create a matrix L and a matrix in the program...

```
rows,cols=size(A)

U=copy(A)
L=Matrix{Float16}(I,rows,cols)

for j=1:cols-1
    for i=j+1:rows
        println("r$i <- r$i - ? r$j")
    end
end
```

note that we need a copy of the matrix A on which to perform the gaussian elimination. Since julia work with references wherever possible for speed, use the copy function to copy the values of A in memory — rather than simply making another name for the same memory locations...

I is a built-in part of the linear algebra package... type using LinearAlgebra if it's not defined from before...

Test the changes...

```
julia> include("gauss.jl")
r2 <- r2 - ? r1
r3 <- r3 - ? r1
r4 <- r4 - ? r1
r3 <- r3 - ? r2
r4 <- r4 - ? r2
r4 <- r4 - ? r3

julia> L
4×4 Matrix{Float16}:
 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

julia> U
4×4 Matrix{Float64}:
 0.459076  0.201025  0.192312  0.464086
 0.783558  0.544086  0.450489  0.0602417
 0.127198  0.916158  0.272742  0.535181
 0.432221  0.220349  0.315929  0.373361
```

now we need to fill in the entries of L and actually perform the row operations...

next pivot

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & \boxed{-6} & -11 \\ 0 & -2 & 5 \end{bmatrix}$$

$$r_3 \leftarrow r_3 - \frac{-2}{-6} r_2$$

row

column

pivot on the bottom

entry being eliminated on top

In general $\underbrace{r_i \leftarrow r_i - \alpha r_j}$

```
rows,cols=size(A)

U=copy(A)
L=Matrix{Float16}(I,rows,cols)

for j=1:cols-1
    for i=j+1:rows
        alpha=U[i,j]/U[j,j]
        println("r$i <- r$i - $alpha * r$j")
        U[i,:]=U[i,:]-alpha*U[j,:]
    end
end
```

actual row operation...

the : means to do the same thing to all elements in the row... i.e. row operation...

Test the script again by running it...

```
julia> include("gauss.jl")
r2 <- r2 - 1.7068163853030605 * r1
r3 <- r3 - 0.27707479211268643 * r1
r4 <- r4 - 0.9415032749173596 * r1
r3 <- r3 - 4.281471147132609 * r2
r4 <- r4 - 0.15466435038201543 * r2
r4 <- r4 - -0.3815208380290779 * r3

julia> U
4×4 Matrix{Float64}:
 0.459076  0.201025      0.192312   0.464086
 0.0       0.200973      0.122247  -0.731868
 0.0       0.0          -0.30394    3.54007
 0.0       3.46945e-18   0.0        1.40023
```

upper triangular → almost except rounding error...

```
rows,cols=size(A)

U=copy(A)
L=Matrix{Float16}(I,rows,cols)

for j=1:cols-1
    for i=j+1:rows
        alpha=U[i,j]/U[j,j]
        println("r$i <- r$i - $alpha * r$j")
        U[i,:]=U[i,:]-alpha*U[j,:]
        U[i,j]=0.0
    end
end
```

Let's fix the rounding error by setting the things that are supposed to be zero explicitly to zero...

Test it again...

```
julia> include("gauss.jl")
r2 <- r2 - 1.7068163853030605 * r1
r3 <- r3 - 0.27707479211268643 * r1
r4 <- r4 - 0.9415032749173596 * r1
r3 <- r3 - 4.281471147132609 * r2
r4 <- r4 - 0.15466435038201543 * r2
r4 <- r4 - -0.3815208380290779 * r3

julia> U
4×4 Matrix{Float64}:
 0.459076  0.201025  0.192312   0.464086
 0.0       0.200973  0.122247  -0.731868
 0.0       0.0      -0.30394    3.54007
 0.0       0.0       0.0        1.40023
```

That looks better!

```
rows,cols=size(A)

U=copy(A)
L=Matrix{Float16}(I,rows,cols)

for j=1:cols-1
    for i=j+1:rows
        alpha=U[i,j]/U[j,j]
        println("r$i <- r$i - $alpha * r$j")
        U[i,:]=U[i,:]-alpha*U[j,:]
        U[i,j]=0.0
        L[i,j]=alpha
    end
end
```

Finally, we store the multipliers alpha from the elimination steps in L...

Test again...

```
julia> include("gauss.jl")
r2 <- r2 - 1.7068163853030605 * r1
r3 <- r3 - 0.27707479211268643 * r1
r4 <- r4 - 0.9415032749173596 * r1
r3 <- r3 - 4.281471147132609 * r2
r4 <- r4 - 0.15466435038201543 * r2
r4 <- r4 - -0.3815208380290779 * r3

julia> U
4×4 Matrix{Float64}:
 0.459076  0.201025   0.192312   0.464086
 0.0       0.200973   0.122247  -0.731868
 0.0       0.0       -0.30394    3.54007
 0.0       0.0        0.0        1.40023

julia> L
4×4 Matrix{Float16}:
 1.0     0.0     0.0     0.0
 1.707   1.0     0.0     0.0
 0.277   4.28    1.0     0.0
 0.9414  0.1547 -0.3816  1.0
```

The computer is giving an answer...

The thing that separates science and engineering from just making up answers is having bounds on the error in the answers...

It's not science unless you know to what degree the answer is correct...

That's the analysis of errors that we talked about in the previous chapter...

Okay... so there are a bunch of numbers... how do we know they are correct... or at least mostly correct?

Backwards error analysis... that is plug it in and check... in this case multiply LU to see if we get A back...

```
julia> L*U
4×4 Matrix{Float64}:
 0.459076  0.201025  0.192312  0.464086
 0.783656  0.544129  0.45053   0.0603414
 0.12721   0.916119  0.27272   0.535355
 0.432177  0.22033   0.315932  0.373066

julia> A
4×4 Matrix{Float64}:
 0.459076  0.201025  0.192312  0.464086
 0.783558  0.544086  0.450489  0.0602417
 0.127198  0.916158  0.272742  0.535181
 0.432221  0.220349  0.315929  0.373361
```

rounding error!

Looks good...sort of...

wait! to only 3 significant digits???

Remember, this factorization was done without partial pivoting... and partial pivoting is important to reduce rounding error...

Note there are some matrices where even partial pivoting is not enough to get a good answer... in that case one has to resort to full pivoting...

Fortunately, the matrices which require full pivoting "never" appear in practical applications... so partial pivoting is almost always good enough...

If you have time until next class, try modifying the code to include the pivoting operations

```
rows,cols=size(A)

U=copy(A)
L=Matrix{Float16}(I,rows,cols)

for j=1:cols-1
    for i=j+1:rows
        alpha=U[i,j]/U[j,j]
        println("r$i <- r$i - $alpha * r$j")
        U[i,:]=U[i,:]-alpha*U[j,:]
        U[i,j]=0.0
        L[i,j]=alpha
    end
end
```

insert the pivoting step here before doing the elimination steps...

Hint: How to choose a pivot... Need the number of largest magnitude in the column...

Recall Example

pivot

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & -2 & -5 \\ 1 & 0 & 8 \end{bmatrix}$$

better pivot

In our original example we just chose the upper left entry of the matrix to be the pivot

larger magnitude so swap rows 1 and 2 before doing the elimination steps...

How to find the index of the element of largest magnitude in a vector?

use the REPL of Julia to test some ideas...

read evaluate print loop

```
julia> v=rand(10)
10-element Vector{Float64}:
 0.2009007929864166
 0.4530657041980637
 0.1038271307702534
 0.8018961964018594
 0.840398773653612
 0.46924384852071954
 0.6926470634806525
 0.3677322230182025
 0.8987343985955654
 0.6106662117708499
```

index: 1 2 3 4 5 6 7 8 9 10

```
julia> findmax(v)
(0.8987343985955654, 9)
```

max value

index of max value

Since findmax returns a list... you can select which one using subscript notation

```
julia> findmax(v)[2]
9
```

just the index

```
julia> findmax(v)[1]
0.8987343985955654
```

just the value...

But wait... what if there were positive and negative values...then selecting the one with largest magnitude is more complicated...

```
julia> v=rand(10)-0.5
ERROR: MethodError: no method matching -(::Vect
For element-wise subtraction, use broadcasting
Closest candidates are:
  -(::Base.TwicePrecision, ::Number) at twicepr
  -(::Array, ::SparseArrays.AbstractSparseMatr
r/share/julia/stdlib/v1.6/SparseArrays/src/spar
  -(::UniformScaling, ::Number) at /builddir/ju
v1.6/LinearAlgebra/src/uniformscaling.jl:147
  ...
Stacktrace:
 [1] top-level scope
   @ REPL[28]:1
```

Try to create a vector of random numbers between -0.5 and 0.5 by subtracting 0.5...

didn't work because vector - scalar doesn't make any sense...

We don't want a difference of vectors but a vector of differences... for this we use the Julia dot notation...

This is a generalization of the same idea from Matlab...but Julia tries to do it better and make it more general...

dot

```
julia> v=rand(10).-0.5
10-element Vector{Float64}:
  0.17089836552187032
  0.28713581278423694
 -0.33308490617535846
 -0.06330287144113256
 -0.1263381199662299
  0.3060430415668278
  0.3921440437428063
 -0.15066928029944138
 -0.0773553708451491
 -0.17685041185825212
```

That work... now need to take absolute value of each element to find the maximum magnitude...

didn't work...

```
julia> abs(v)
ERROR: MethodError: no method mat
Closest candidates are:
  abs(::Bool) at bool.jl:79
  abs(::Unsigned) at int.jl:169
  abs(::Signed) at int.jl:170
  ...
Stacktrace:
 [1] top-level scope
   @ REPL[30]:1
```

That's the same problem as before... we didn't want the absolute value of a vector, but instead a vector of absolute values...

Use the Julia dot notation...

Note, being able to use the dot notation with <u>any</u> <u>function</u> is a convenient generalization of how the dot notation works in Matlab

dot

```
julia> abs.(v)
10-element Vector{Float64}:
 0.17089836552187032
 0.28713581278423694
 0.33308490617535846
 0.06330287144113256
 0.1263381199662299
 0.3060430415668278
 0.392144043742806
 0.15066928029944138
 0.0773553708451491
 0.17685041185825212
```

largest

That's better!

Now find the index of the entry with largest magnitude...

```
julia> findmax(abs.(v))[2]
7
```

Index of Largest absolute value...

Thus...

using the findmax and abs functions it should be possible to determine which rows to swap when doing partial pivoting...

Please try to write some code for next time that implements Gaussian elimination with partial pivoting...

Don't worry if you don't get it to work... you'll still learn quite a bit just from trying...