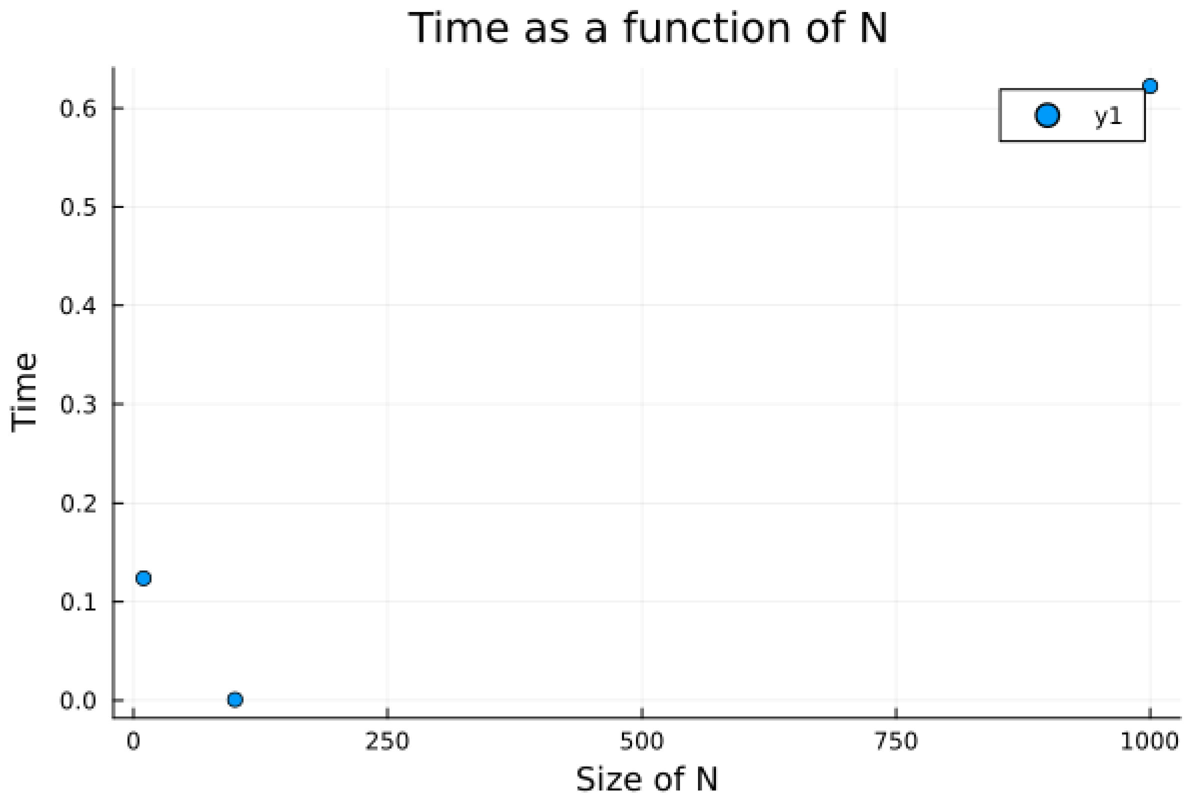
CIS 410/510 HW 1 - Brett Sumser

For this homework assignment we had to use the LU-factorization algorithm in order to decompose a matrix into upper and lower triangular forms, and use back/forward substitution to solve for x in $Ax = b$. I had issues with sickness and difficulties with the LU decomposition algorithm, but was able to implement most of the other functions. I also implemented a different algorithm called the Doolittle algorithm for LU matrix decomposition. Some of my difficulties were do to using the julia language for the first time, I am expecting to have less difficulties with it as the term continues.

Here are the functions I was able to successfully implement:

- find_pivot()-This function uses the built in method findmax to slice the matrix and find the max entry in a column. It returns the value of the entry and the index.
- swap()-This function swaps two rows in a matrix by constructing the proper identity matrix and multiplying it by the input matrix.
- luDoolittleDecomp(A,N)-This function is an implementation of the Doolittle Algorithm for a LU matrix decomposition.
- LUPsolve()-I used my above implementation of the Doolittle algorithm to attempt to solve the LU decomposition. LUPsolve performs the decomposition, and then calls two different functions to perform the forward/backward substitution.

Based on the below figure, it seems that the time for the LU decomposition increases dramatically when the size of the matrix is increased. Comparing a 10 element matrix vs a 1000 element matrix, the time needed more than quintuples! That is quite an increase, and seems in line with the $O(n^3)$ complexity described in class.



```
using Plots # add Plots.jl from the package manager if you have not already done so.

# HW 1 starting script (if you want): contains function computeLU() - to compute an LU
# matrix A, namely, A = LU, where L and U are lower and upper triangular matrices.


"""
    computeLU(A)
Compute and return LU factorization `LU = A` of square matrix `A`.
Might not work on all matrices, since no pivoting is done!
# Examples (don't need examples, but fine to include)
'''
julia> A = [6 -2 2;12 -8 6;3 -13 3]
3 3  Array{Int64,2}:
  6   -2  2
 12   -8  6
  3  -13  3
julia> (L, U) = computeLU(A)
([1.0 0.0 0.0; 2.0 1.0 0.0; 0.5 3.0 1.0], [6.0 -2.0 2.0; 0.0 -4.0 2.0; 0.0 0.0 -4.0])
julia> norm(A - L*U)
0.0
'''
"""
function computeLU(A)

    N = size(A)[1]

    #Id = Matrix{Float64}(I, N, N) # N x N identity matrix
    Id = create_identity(N)

    L = copy(Id)   # initialize
    U = copy(Id)   # initialize
    A    = copy(A) # initialize. A   corresponds to A as it goes under elimination sta

    for k = 1:N-1 # march across columns

        (Lk, Lk_inv) = compute_Lk(A  , k)

        A   .= Lk * A
        L .= L * Lk_inv

    end

    U .= A

    return (L, U)

end


"""
    compute_Lk(A, k)
Compute Lk and its inverse from A, assuming first k-1 columns have undergone eliminati
"""
function compute_Lk(A, k)


    N = size(A)[1]
```

```
    Lk = create_identity(N) # Matrix{Float64}(I, N, N)       # initialize as identity
    Lk_inv = create_identity(N)# Matrix{Float64}(I, N, N)   # initialize as identity m

    # now modify column k, strictly below diagonal (i = k+1:N)
    for i = k+1:N
        Lk[i,k] = -A[i,k] / A[k,k]    # fill me in (compute elimination factors)
        Lk_inv[i,k] = A[i,k] / A[k,k]  # fill me in (compute elimination factors)
    end

    return (Lk, Lk_inv)

end

"""
    create_identity(N)
Given integer N, constructs a square identity matrix of size N.
"""
function create_identity(N)

    I = Matrix{Float64}(undef, N, N)
    I .= 0

    for i = 1:N
        I[i, i] = 1
    end

    return I
end

"""
    find_pivot(A, k)

Given matrix A and column k, find largest element in that column. Uses built in method
where A[] is the proper slice of the matrix for the column we need. findmax() returns
and a Cartesian Coordinate pair for the index of the element.

julia> A .= [6 -2 2;12 -8 6;3 -13 3]
3 3  Matrix{Float64}:
  6.0   -2.0  2.0
 12.0   -8.0  6.0
  3.0  -13.0  3.0

julia> A[:,1:1]
3 1  Matrix{Float64}:
  6.0
 12.0
  3.0

julia> A[:,2:2]
3 1  Matrix{Float64}:
  -2.0
  -8.0
 -13.0
"""
function find_pivot(A, k)
    return (value, index) = findmax(A[:,k:k])
end

"""
```

```
    swap(L, j, k)

Function that swaps rows j and k in all columns from 1:k-1 in matrix L by constructing
permutation matrix.
"""
function swap(L, j, k)
    print("swap called")
    N = size(L)[1]
    I = Matrix{Float64}(undef, N, N)
    I .= 0

    for i = 1:N
        I[i,i] = 1
    end

    for i = 1:N
        I[j,i], I[k,i] = I[k,i], I[j,i]
    end

    L .= I * L

    return L
end

"""
    luDoolittleDecomp(A, N)

Function that performs an LU decomposition using the doolittle algorithm.
"""
function luDoolittleDecomp(A,N)
    U = Matrix{Float64}(undef, N, N)
    U .= 0

    L = Matrix{Float64}(undef, N, N)
    L .= 0

    for i = 1:N
        for k = i:N
            sum = 0
            for j = 1:i
                sum += (L[i,j] * U[j,k])

            end
            U[i,k] = A[i,k] - sum
        end
        for k = i:N
            if (i == k)
                L[i,i] = 1
            else
                sum = 0
                for j = 1:i
                    sum += (L[k,j] * U[j,i])
                end
                L[k,i] = (A[k,i] - sum) / U[i,i]
            end
        end
    end
    #display(U)
    #display(L)
```

```
    return(L,U)
end

"""
    LUPsolve(A)

Function that solves Ax=b by computing LUP-factorization and performs forward/backward
"""
function LUPsolve(A, b)
    # test matrix to check for accuracy in solving
    #L = Matrix{Float64}(undef, 3, 3)
    #L .= [1 0 0;4 1 0;4 0.5 1]
    #U = Matrix{Float64}(undef, 3, 3)
    #U .= [1 2 2;0 -4 -6;0 0 -1]

    #size of matrix working
    N = size(A)[1]

    (L,U) = luDoolittleDecomp(A,N)
    #display(U)
    #display(L)

    y = forward_sub(L, b)
    x = backward_sub(U, y)

    #display(x)
    return x
end

"""
    forward_sub(L, b)

Give lower triangular matrix L and vector b, perform the forward substitution to solve
"""
function forward_sub(L, b)
    N = size(L)[1]
    x = similar(L)
    x .= 0

    for i = 1:N
        temp = b[i]
        for j = 1:i-1
            temp -= L[i,j] * x[j]
        end
        x[i] = temp / L[i,i]
    end
    return x
end


"""
    backward_sub(U, b)

Give upper triangular matrix U and vector b, perform the forward substitution to solve
(Backward version of forward substitution)
"""
function backward_sub(U, b)
    N = size(U)[1]
    x = similar(U)
```

```
    x .= 0

    for i = N:-1:1
        temp = b[i]
        for j = i+1:N
            temp -= U[i,j] * x[j]
        end
        x[i] = temp / U[i,i]
    end
    return x
end


testSizes = [10, 100, 1000]
y_time = []

A_10 = rand([1,10], testSizes[1], testSizes[1])
b_10 = rand(10,1)

A_100 = rand([1,10], testSizes[2], testSizes[2])
b_100 = rand(100,1)

A_1000 = rand([1,10], testSizes[3], testSizes[3])
b_1000= rand(1000,1)

A = Matrix{Float64}(undef, 3, 3)
A .= [2 -1 -2;-4 6 3;-4 -2 8]


temp = @timed LUPsolve(A_10, b_10)
push!(y_time, temp[2])
display(temp[2])

temp = @timed LUPsolve(A_100, b_100)
push!(y_time, temp[2])
display(temp[2])

temp = @timed LUPsolve(A_1000, b_1000)
push!(y_time, temp[2])
display(temp[2])

#swap(A,3,1)
#print(A_1000)
#A_1000_time = @time luDoolittleDecomp(A_1000, 1000)
#display(A_1000_time)


scatter(testSizes, y_time, xlabel="Size of N", ylabel="Time", title = "Time as a funct
savefig("testPlot.png")
#b = rand(3, 1)
#
#(L, U) = computeLU(A)
#@assert L*U     A
```