CIS 410/510 HW 2 - due Friday October 23rd at midnight

For all of the user-defined functions below, provide documentation according to the Julia documentation page `https://docs.julialang.org/en/v1/manual/documentation/index.html` - see sample code below for the most basic documentation that I expect.

Your report should be:
i. Written preferably in Latex, but Word (or any word processor) fine
ii. Composed of a single paragraph discussing your findings
iii. Include 1-2 illustrative tables or figures showing your results

Your code should be:
iv. Written in Julia
v. Attached at the end of your report

Upload your report in a single (.pdf, .doc etc.) file to Canvas. You will be graded on correctness of code, thoroughness in addressing the prompts and drawing conclusions from your findings.

Getting started: If you haven't already done so, organize your script *my_solvers.jl* from HW1 to include two separate functions, *computeLUP*() (that computes and returns an LUP-decomposition of an $N \times N$ square matrix A using partial pivoting) and $LUPsolve()$ that solves $Ax = b$ using an LUP-factorization via forward and backward substitution. Make sure these functions are tested on a system $Ax = b$ that actually has a solution (not every system does). You can guarantee this by setting $B = rand(N, N)$ and defining $A = I + B^T B$, where I is the $N \times N$ identity matrix. You can set $b = rand(N, 1)$. Remember that you should assert these two functions are correct by confirming that the vector x that is returned after calling these two functions satisfies $Ax \approx b$.

1. Include in *my_solvers.jl* a function *conj_grad*() that performs the conjugate gradient (CG) algorithm. *conj_grad*() should take in a positive definite matrix A, an initial guess, a right-hand-side vector b, a tolerance ϵ and a maximum number of iterations $\text{iter}_{\max}$, and return an approximate solution $\tilde{x}$ to $Ax = b$, such that the relative error $err_{\text{R}} \leq \epsilon$ where

$$err_{\text{R}} = \frac{||A\tilde{x} - b||}{||\tilde{x}||}$$

where $||\cdot||$ is the 2-norm, i.e. $||x|| = \sqrt{x^T x}$.
(Hint: one way to generate a positive definite $N \times N$ matrix is to define $A = I = B^T B$, as suggested above. Also, a common choice for initial condition is the vector of all zeros).

2. Consider the $N \times N$ matrix

$$A = \begin{bmatrix} -2 & 1 & 0 & 0 & & \dots & & 0 \\ 1 & -2 & 1 & 0 & & \dots & & 0 \\ 0 & 1 & -2 & 1 & 0 & & & \\ 0 & 0 & 1 & -2 & 1 & 0 & \dots & 0 \\ \vdots & & & \ddots & \ddots & \ddots & & \\ 0 & \dots & & 0 & 1 & -2 & 1 & 0 \\ 0 & \dots & & & 0 & 1 & -2 & 1 \\ 0 & \dots & & & & 0 & 1 & -2 \end{bmatrix}$$

Write a Julia script that: (i) defines A in dense array format. Use built-in (optimized) Julia functionalilty to compute an LUP factorization, and then solve the system $Ax = b$, where $N = 1,000$ and b is an $N \times 1$ vectors of 1's. Time the two computations. How do these compare with the timing obtained in question 1 for the $N = 1,000$ case?

(ii) Repeat i) but define A in sparse array format (do NOT call sparse(A) on dense A. why?). Comment on speed-up and memory usage (compared to (i)).

3 [510 only]. Invoke the (optimized) built-in Julia functionilty to perform a direct solve of $Ax = b$ (where A is a random, dense, positive definite matrix) using both an LUP-factorization and a CG solve (from the Julia IterativeSolvers package). Compute the time taken to perform each solve (i.e. for the direct solve, do not count the time it takes to obtain the factorization). For the CG solve, set $\epsilon = 10^-9$ and experiment with different initial conditions. For N = 10, 100, 1000, how do the two methods compare? Does one seem favorable over the other?

Below is the listing for the file *matrix_multiply.jl*.

```
using Printf
using LinearAlgebra

"""
    matmul_naive!(c, a, b)

Compute the product of matrices ‘a‘ and ‘b‘ and store in ‘c‘.

# Examples
‘‘‘jldoctest
julia> C = zeros(3,3); A = 7 .* ones(3,3); B = Matrix{Int64}(I, 3,3);
julia> matmul_naive!(C, A, B)
julia> C
3x3 Array{Float64,2}:
 7.0  7.0  7.0
 7.0  7.0  7.0
 7.0  7.0  7.0
‘‘‘
"""
function matmul_naive!(C, A, B)
    (N, M) = size(A)
    (P, R) = size(B)
    (n, r) = size(C)
    @assert M == P # check matrix size
    @assert n == N
    @assert r == R

    for i = 1:N
        for j = 1:R
            for k = 1:M
                C[i, j] += A[i, k] * B[k, j]
            end
        end
    end

end

N = 100
M = 200
L = 50

C = zeros(N, L)
A = rand(N, M)
B = rand(M, L)

#= Do some timing tests
   between Julia native
   and naive matrix multiply.
=#
println()
println("Julia's  Native A * B")
@time C = A * B
@time C = A * B
println("---------------")
println()

println("Using Naive! Function!")
```

```
D = zeros(N, L)
matmul_naive!(D, A, B)  #call once to compile
@time matmul_naive!(D, A, B)
@time matmul_naive!(D, A, B)
C += (A * B)
C += (A * B)
@printf "norm(C - D) = \x1b[31m %e \x1b[0m\n" norm(C - D)
println("----------------")
println()
```