CIS 410/510 HW 3 - due Friday November 6th

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 (or you can write a short paragraph for each problem part) 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 (recall that you can do this all in Overleaf!)

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.

1. The 1-dimensional heat equation with an initial condition is given by

$$\begin{aligned} u_t &= ku_{xx} + F(x,t), \quad 0 \le x \le 1, \quad 0 \le t \le T \\ u(x,0) &= f(x). \end{aligned}$$

where $k > 0$ is the thermal diffusivity. Assume the temperature $u(x,t)$ is zero at the boundaries, i.e. $u(0,t) = u(1,t) = 0$.

Start by modifying myForwBack_Euler.jl to call forward/backward Euler on the ODE $y' = Ay + c(t)$. Next, open a new Julia script and call it heatl1D.jl (you may modify mine). To solve the 1D heat equation numerically, use the method of lines (MOL) by first discretizing in space using the standard (second-order accurate) centered difference approximation to u_{xx}, and arrive at an IVP given by the system of ODEs $u' = Au + c(t)$ and the initial condition $u(0) = f(x)$. Include the ODE solver functions by using include("myForwBack_Euler.jl") at the top of the script. Set $k = 2$, $F(x,t) = 2(\pi^2 - 1)e^{-2t}\sin(\pi x)$, $f(x) = \sin(\pi x)$.

(a) Set T = 0.1 and call your forward Euler method (from myForwBack_Euler.jl) on the IVP with $\lambda = 0.1$ (recall that $\lambda = \Delta t/\Delta x^2$). Note that the exact solution is given by $u(x,t) = e^{-2t}\sin(\pi x)$. Confirm that your numerical solution is converging in space at the correct rate (rate ≈ 2) by computing a convergence table, i.e. fill in the missing parts:

2nd Order			
Δx	$\text{error}_{\Delta x}$	ratio = $\text{error}_{\Delta x}/\text{error}_{\Delta x/2}$	rate = $\log_2$(ratio)
0.1		(empty entry)	(empty entry)
0.05			
0.025			
0.0125			

where the error is the discrete L^2-norm of difference $u(x_j, T) - u_j(T)$ (i.e. the exact solution $u(x,T)$ evaluated on the spatial grid at the final time and the numerical solution $u_j(T)$ evaluated at the final time). I will provide more details in class.

(b) Recall that stability for forward Euler for this problem requires the condition that $\lambda \le 1/(2k)$. Set $\Delta x = 0.1$ and T = 2, and confirm this stability result by trying out different time steps. i.e. take $\Delta t = 10^{-1}, 10^{-2}$ and 10^{-3} and report on how forward Euler performs (what I mean by this is, for each time step plot the exact solution and the numerical solution against space, on the same figure, and comment on whether or not the

numerical solution tracks the exact solution over time).

(c) Repeat part (b), but use backward Euler, which is unconditionally stable. You don't have to use the method of lines as done in part (a) (i.e. you don't have to develop heat1D.jl to call backward Euler, you can hard code it into heat1D.jl). You may want to do convergence tests with backward Euler (as in part (a) first). For backward Euler you may use a direct (LU) solve.

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()
```