1. We solve the heat equation

$$u_t = ku_{xx}$$

on the spatial domain $(0, 1)$ and time domain $(0, T)$. We specify and initial position $u(x, 0) = f(x)$ and impose homogeneous Dirichlet conditions at the boundaries $x = 0, x = 1$. Convergence tests are run on the manufactured solution $u(x,t) = e^{-2t} \sin \pi x$ with $k = 2$, $T = 0.1$, and $\lambda = 0.1$. Figure 1 is a table containing the resulting convergence data as the mesh is sequentially refined. As expected, the numerical solution exhibits rate 2 convergence. hi

h	$\lVert u - u_h \rVert$	$\log_2 (e_h / e_{\frac{h}{2}})$
0.10000	0.000092	$\emptyset$
0.05000	0.000023	2.00765
0.02500	0.000006	2.00191
0.01250	0.000001	2.00048

Fig. 1: Convergence of the numerical solution for $T = 0.1, k = 2, \lambda = 0.1$.

2.

Set $T = 2$ and $\Delta x = 0.1$. Stability of forward Euler is dictated by the condition

$$k\frac{\Delta t}{\Delta x^2} \leq 1.$$

Then we must have that $200\Delta t \leq 1$. For the time steps $0.1, 0.01, 0.001$ we can see that only one of these steps ($\Delta t = 0.001$) actually satisfies the stability requirement. When plotting the solution using the other two time steps we see the numerical solution blow up.

3.

Set $T = 2$ and $\Delta x = 0.1$. Unlike forward Euler, the backward Euler method is unconditionally stable. So regardless of the choice for Δt, the backward Euler method gives an approximation to the exact solution at each time. Although I did not explore this I would be curious to look at the scenario in which backward Euler's computation cost hinder it and how fast it runs in comparison to forward Euler.

heat1D.jl

```julia
using SparseArrays
using Plots
include("myForwBack_Euler.jl")
include("/home/cody/github/Finite_Element_Methods/convergence_rates.jl")

# source function
function F(x,t)
    return 2*(pi^2 -1)*exp(-2t)*sin.(pi*x)
end

# initial value function
function f(x)
    return sin.(pi*x)
end

# exact solution
function exact(x,t)
    return exp(-2t)*sin.(pi*x)
    #return exp(-pi^2 * t)*sin.(pi*x)
end

function time_dependent_heat(k, Δx, Δt, T, my_source, my_initial, my_exact)
    N  = Integer(ceil((1-0)/Δx)) # N+1 total nodes, N-1 interior nodes
    x = 0:Δx:1
    t = 0:Δt:T
    M = Integer(ceil((T-0)/Δt)) # M+1 total temporal nodes

    λ = Δt/Δx^2

    # A is N-1 by N-1 matrix for forward Euler
    A = (1-2*λ*k)*sparse(1:N-1,1:N-1,ones(N-1), N-1, N-1) +
            (λ*k)*sparse(2:N-1,1:N-2,ones(N-2),N-1,N-1) +
            (λ*k)*sparse(1:N-2,2:N-1,ones(N-2),N-1,N-1)

        # A2 is the appropriate matrix for backward Euler
    A2 = (1+2*λ*k)*sparse(1:N-1,1:N-1,ones(N-1), N-1, N-1) -
             (λ*k)*sparse(2:N-1,1:N-2,ones(N-2),N-1,N-1) -
             (λ*k)*sparse(1:N-2,2:N-1,ones(N-2),N-1,N-1)

    u = Array{Float64}(undef,N-1)
    u .= my_initial(x[2:N])  # setting the initial condition for interior nodes
    u2 = copy(u)
    Y = zeros(N+1,M)         # initializing solution matrix
```

```
    Z = copy(Y)

    my_forward_Euler!(Y, Δt, t, A, F, u, x, N, M)
    my_backward_Euler!(Z, Δt, t, A2, F, u2, x, N, M)


    uₑ₁ = my_exact(x[:], t[M]) - Y[:, M]
    uₑ₂ = my_exact(x[:], t[M-1]) - Z[:, M-1]

    e1 = sqrt(Δx * uₑ₁' * uₑ₁)
    e2 = sqrt(Δx * uₑ₂' * uₑ₂)

    return Y, Z, e1, e2
end

# repackage the solution as a function of grid spacing which returns the discrete
# L²-error
function R(Δx)
    k = 2
    #Δx = 0.1

    T = 2
    λ = 0.2
    Δt = λ*Δx^2 / k

    x = 0:Δx:1
    t = 0:Δt:T

    U1, U2, e1, e2 = time_dependent_heat(k, Δx, Δt, T, F, f, exact)
return U1, U2, e1, e2
end
```

your_ForwBack_Euler

(I just modded your code)

```
using LinearAlgebra
using Plots

function my_forward_Euler!(Y, Δt, t, A, F, y, x, N, M)
    #M = length(t)
    #t = t1:Δt:tf
    #M = Integer(ceil((tf-t1)/Δt))
    Y[2:N,1] = y[:]

    for n = 1:M
        b = Δt * F(x[2:N],t[n])
        y[:] = A * y[:] + b
        Y[2:N,n] .= y[:]

    end

    #return (t, Y)

end


function my_backward_Euler!(Y, Δt, t, A, F, y, x, N, M)


    Y[2:N,1] = y[:]

    #Exact = Matrix{Float64}(undef,2,N+1)
    #Exact[:,1] = y[:]

    Id = I(N-1) #Matrix{Float64}(I,N+1,N+1)

    for n = 2:M-1
        y[:] = A\(y[:] .+ Δt*F(x[2:N], t[n]))
        Y[2:N,n] = y[:]
        #Exact[:,n] = exact(t[n])
    end
end
```