CIS 510 HW 3 Submission

Scott Lambert

Problem 1

Proposal emailed in advance.

Problem 2

Submitted as part of Homework 2.

Problem 3

Part a

See source code at end of file. Note that I modified my second derivative matrix slightly to allow for arbitrary boundary conditions; instead of the (N-1)x(N-1) matrix described in problem 2, the matrix is an (N-1)x(N+1) matrix obtained by appending the column $[1, 0, ..., 0]^T$ to the beginning of the matrix in problem 2 and appending the column $[0, 0, ..., 0, 1]^T$ to the end.

Part b

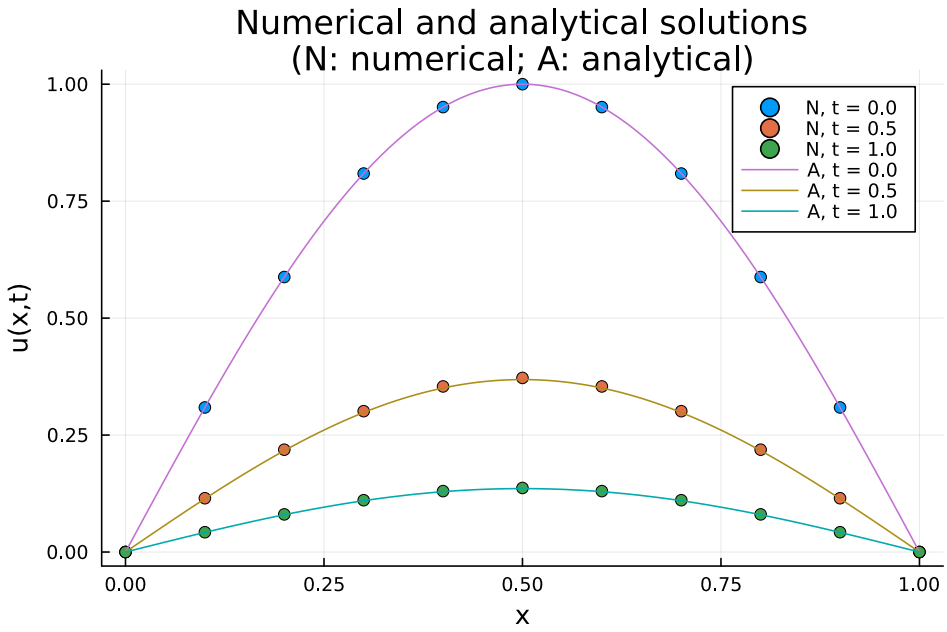

Figure 1: Numerical and analytical solutions to the heat equation with initial conditions.

Solution is above. I used only 11 nodes because solving the heat equation with forward Euler timestepping is so expensive due to the quadratic dependence of the maximum stable timestep on the spatial discretization length. The equation solved is

$$u_t(x,t) = \kappa u_{xx}(x,t) + F(x,t) \tag{1}$$

The parameters, source term, and boundary conditions are

$$\begin{aligned}
\kappa &= 2 \\
F(x,t) &= 2(\pi^2 - 1)\exp(-2t)\sin(\pi x) \\
f(x) &= \sin(\pi x) \\
u(0,t) &= 0 \\
u(1,t) &= 0 \\
\Delta x &= 0.1 \\
\Delta t &= \frac{1}{2}\frac{\Delta x^2}{2\kappa} = 0.00125 \text{ (Half the maximum stable timestep)}
\end{aligned}$$

Recall that the exact solution is

$$u(x,t) = \exp(-2t)\sin(\pi x) \tag{2}$$

Part c

To show the instability of the numerical solution a little more clearly, I chose to plot the numerical solution for $\Delta t/\Delta t_{\max} = 0.5$, 1, and 1.083 rather than $\Delta t = 1/10$, $1/100$, and $1/1000$. I also used $T_f = 1$ instead of 2. In the first two plots, the numerical solution is stable and accurately tracks the analytical solution. In the last plot, the numerical solution has already begun to diverge by t = 1.

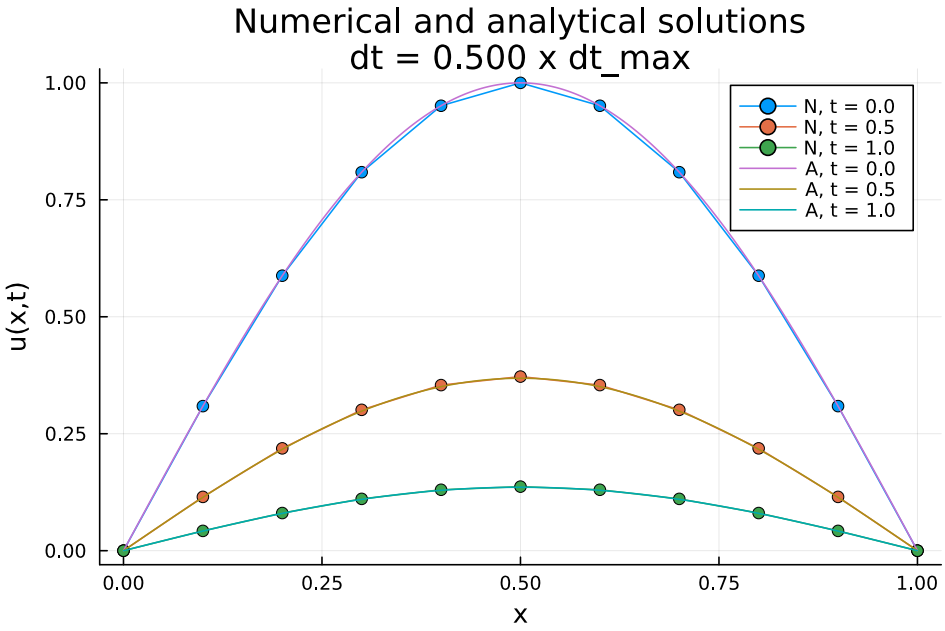

Figure 2

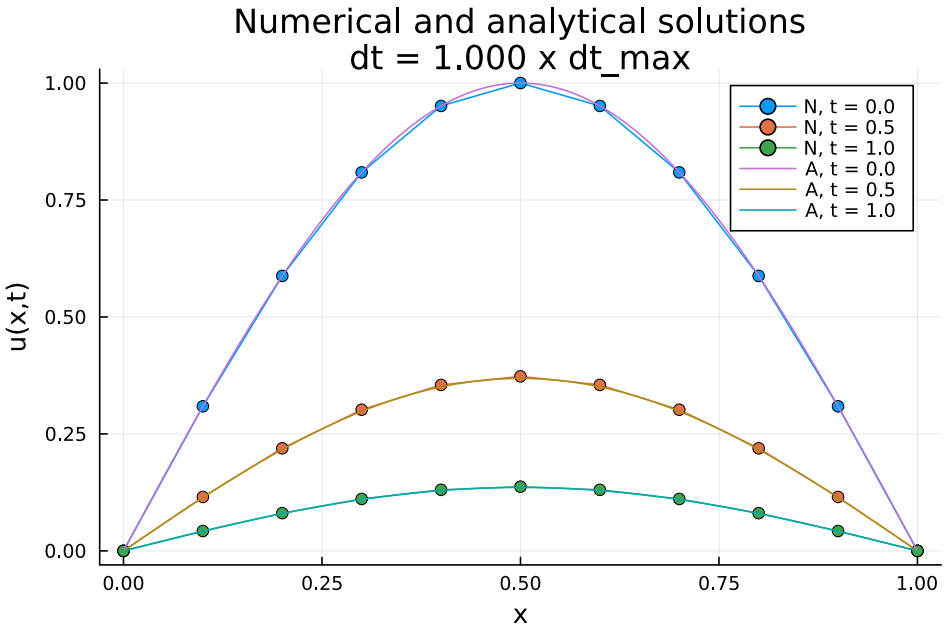

Figure 3

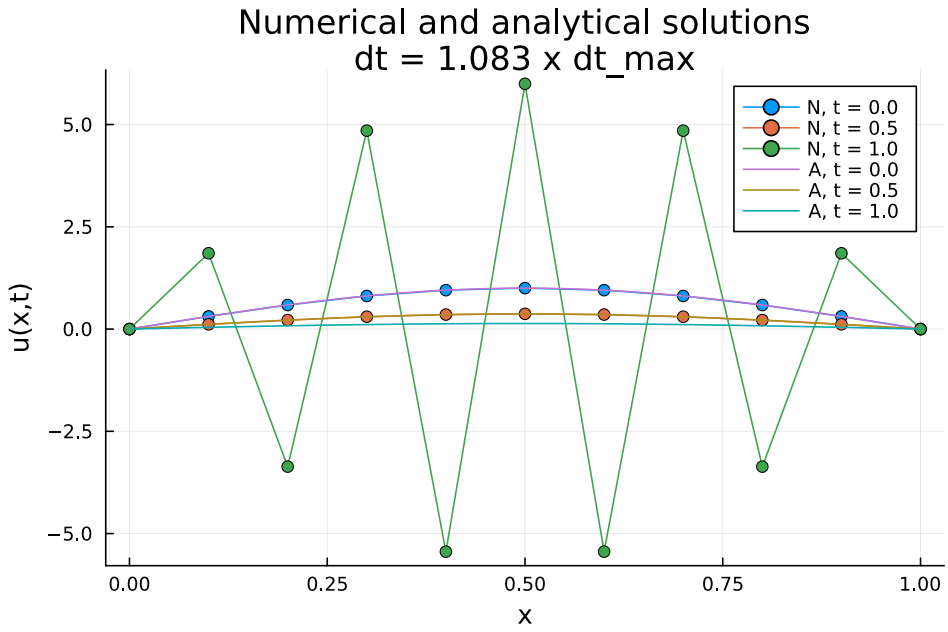

Figure 4

Part d

Following the instructions I get the following table:

Δx	$\text{error}_{\Delta x}$	$\text{ratio} = \text{error}_{\Delta x}/\text{error}_{\Delta x/2}$	$\text{rate} = \log_2(\text{ratio})$
1	4.373e-03	·	·
0.5	1.088e-03	4.019e+00	2.007e+00
0.25	2.717e-04	4.005e+00	2.002e+00
0.125	6.790e-05	4.001e+00	2.000e+00

Source code

Problem 3

Client script (hw03p3client.jl)

```
include("../euler_methods.jl")
include("../my_solvers.jl")
include("../heat_MOL.jl")

using Plots
using Printf

# Solve the heat equation with simple initial
# and boundary conditions and a simple source term
function F(x,t)
    return 2*(pi^2-1)*exp(-2*t).*sin.(pi.*x)
end

function f(x)
    return sin.(pi.*x)
end


function plotHeatSol(f,F,k,X,T,N)
    """
        plotHeatSol(f,F,k,X,T,N)
    Plot numerical and analytical solutions to the heat equation.
    """
    ul = 0
    ur = 0
    dx = X/N
    dt  = 0.5*dx^2/(2*k) # Half the maximum stable timestep
    n = Int(ceil(T/dt))
    x,t,u = heat1d(f,F,k,ul,ur,X,T,dt,N)

    p = scatter(x,u[:,1],label=@sprintf("N, t = %.1f",t[1]))
    midind = Int(ceil(0.5*n))
    scatter!(p,x,u[:,midind],label = @sprintf("N, t = %.1f",t[midind]))
    scatter!(p,x,u[:,n],label = @sprintf("N, t = %.1f",t[end]))

    ndense = 1000
    tdense = T.*(0:1/ndense:1)
    xdense = X.*(0:1/ndense:1)
    uexact = sin.(pi.*xdense)*exp.(-2 .* tdense)'
    plot!(p,xdense,uexact[:,1],label=@sprintf("A, t = %.1f",tdense[1]))
    midind_dense = Int(ceil(0.5*ndense))
    plot!(p,xdense,uexact[:,midind_dense],
        label=@sprintf("A, t = %.1f",tdense[midind_dense]))
    plot!(p,xdense,uexact[:,ndense],
        label=@sprintf("A, t = %.1f",tdense[end]))
    display(p)
    title!(p,"Numerical and analytical solutions\n" *
    "(N: numerical; A: analytical)")
    ylabel!(p,"u(x,t)")
    xlabel!(p,"x")
    savefig(p,"heat1dsol.pdf")
end
```

```
function plotTimesteps(f,F,k,X,T,N)
    """
        plotTimesteps(f,F,k,X,T,N)
    Generate plots of solutions of the heat equation for different
    values of the timestep.
    """
    ul = 0
    ur = 0
    dx = X/N
    tfacts = [0.5;1;1.083]
    for (i,tf) in enumerate(tfacts)
        dt  = tf*dx^2/(2*k) # Half the maximum stable timestep
        n = Int(ceil(T/dt))
        x,t,u = heat1d(f,F,k,ul,ur,X,T,dt,N)

        p = plot(x,u[:,1],label=@sprintf("N, t = %.1f",t[1]),marker=:circle)
        midind = Int(ceil(0.5*n))
        plot!(p,x,u[:,midind],label = @sprintf("N, t = %.1f",t[midind]),
            marker=:circle)
        plot!(p,x,u[:,n],label = @sprintf("N, t = %.1f",t[end]),
            marker=:circle)

        ndense = 1000
        tdense = T.*(0:1/ndense:1)
        xdense = X.*(0:1/ndense:1)
        uexact = sin.(pi.*xdense)*exp.(-2 .* tdense)'
        plot!(p,xdense,uexact[:,1],label=@sprintf("A, t = %.1f",tdense[1]))
        midind_dense = Int(ceil(0.5*ndense))
        plot!(p,xdense,uexact[:,midind_dense],
            label=@sprintf("A, t = %.1f",tdense[midind_dense]))
        plot!(p,xdense,uexact[:,ndense],
            label=@sprintf("A, t = %.1f",tdense[end]))
        display(p)
        title!(p,"Numerical and analytical solutions\n" *
        @sprintf("dt = %.3f x dt_max",tf))
        ylabel!(p,"u(x,t)")
        xlabel!(p,"x")
        savefig(p,@sprintf("tfact%d.pdf",i))
    end
end

function L2norm(v,dx)
    """
        Compute the discrete L2 norm with point spacing dx.
    """
    return sqrt(dx)*norm(v)
end

function errsdx(f,F,k)
    """
        errsdx(f,F,k,X,T,N)
    Compute the L2 error for different values of dx.
    """
    ul = 0
    ur = 0
    lambda = 0.1
    X = 1
    T = 0.1
    errs = zeros(4)
```

```
    Ns = [10,20,40,80]
    for (i,N) in enumerate(Ns)
        dx = X/N
        dt = lambda*dx^2
        x,t,u = heat1d(f,F,k,ul,ur,X,T,dt,N)
        uexact = exp(-2*T).*sin.(pi.*x)
        errs[i] = L2norm(u[:,end]-uexact,dx)
    end
    ratios = errs[1:end-1]./errs[2:end]
    rates = log2.(ratios)
    return errs, ratios, rates
end

# Function call for part b:
plotHeatSol(f,F,2,1,1,10)

# Function call for part c:
plotTimesteps(f,F,2,1,1,10)

# Function calls for part d:
errs,ratios,rates = errsdx(f,F,2)
display(map(x -> @sprintf("%.3e",x), errs))
display(map(x -> @sprintf("%.3e",x), ratios))
display(map(x -> @sprintf("%.3e",x), rates))
```

1d Heat equation solver (heat_MOL.jl)

```
include("my_solvers.jl")
include("euler_methods.jl")

function heat1d(f,F,k,ul,ur,X,T,dt,N)
    """
        heat1d(F,k,N,dt)
    Solve the 1d heat equation

    u_t(x,t) = k*u_xx(x,t) + F(x,t)

    on the spatial interval
    [0,X] for the time interval [0,T] with fixed
    boundary conditions u(0,t) = ul and u(X,t) = ur
    and initial condition f(x)

    -Initial and boundary conditions are assumed to be
    compatible

    N = number of subdivisions of spatial interval
    N+1 = number of solution points
    dt = timestep

    """
    # space array
    dx = X/N
    x = dx.*(0:N)
    # Number of time intervals
    n = Int(ceil(T/dt))
    # Solution array
    u = zeros(N+1,n+1)
    # Initial condition
```

```
    u[:,1] = f.(x)
    # Boundary conditions
    u[1,:] .= ul
    u[end,:] .= ur

    # Centered difference matrix
    dd = ddMatrix(N)

    # Solve the equation with forward Euler time integration
    # Bracketing y between BCs enables arbitrary constants
    # to be chosen as boundary values
    fstep(t,y) = k.*dd*[ul;y;ur]./dx^2 .+ F(x[2:end-1],t)
    y0 = u[2:end-1,1]
    t,y = forward_euler(fstep,y0,T,dt)
    u[2:end-1,:] = y
    return x, t, u
end
```

Modified Forward Euler method (in euler_methods.jl)

```
function forward_euler(f,y0,T,dt)
    """
        forward_euler(f,y0,T,dt)
    solve the system y' = f(t,y) with the forward Euler method
    on the interval [0,T]. Assumes y is a vector. Generalization
    of my_forward_euler to multidimensional systems
    """
    N = size(y0)[1]
    n = Int(ceil(T/dt))
    y = zeros(N,n+1)
    t = dt.*(0:n)
    y[:,1] = y0
    for i = 1:n
        y[:,i+1] = y[:,i] + dt.*f(t[i],y[:,i])
    end
    return t,y
end
```

Modified second derivative matrix method (in my_solvers.jl)

```
function ddMatrix(N;make_sparse=true)
    """
        ddMatrix(N)
    compute the discrete second derivative matrix as either a dense
    array or a sparse array
    N+1 = number of nodes in spatial discretization
    """
    if !make_sparse
        # Create A in dense format
        A = zeros(N-1,N+1)
        for i = 1:N-1
            A[i,i:i+2] = [1 -2 1]
        end
    else
        # Create A in sparse format
        rowInds = [1:N-1;1:N-1;1:N-1]
        colInds = [1:N-1;2:N;3:N+1]
```

```
        vals = [ones(N-1); -2 .* ones(N-1); ones(N-1)]
        A = sparse(rowInds,colInds,vals)
    end
    return A
end
```

Norm method (in my_solvers.jl)

```
function norm(v)
"""
    norm(v)
Compute the norm of the vector (or array) v.
"""
    return sqrt(sum(v.^2))

end
```