(1) Implement the Crank-Nicolson scheme for the heat equation

$$\begin{cases} u_t = \partial_x(a(x)u_x) + f(x,t), & t > 0, x \in (0,1) \\ u(x,0) = u_0(x), & x \in [0,1] \\ u(0,t) = u(1,t) = 0, & t > 0 \end{cases}$$

To implement Crank-Nicolson, we need to find our $F(x,t,u,u_x,u_{xx})$ operator. For this PDE, we simply have

$$F(x,t,u,u_x,u_{xx}) = \partial_x(a(x)u_x) + f(x,t) = a(x)u_{xx} + a'(x)u_x + f(x,t).$$

Then, our Crank-Nicolson scheme is given by

$$\frac{u_i^{n+1} - u_i^n}{h_t} = \frac{1}{2}\left(F_i^{n+1}(u,x,t,u_x,u_{xx}) + F_i^n(u,x,t,u_x,u_{xx})\right)$$

where F_i^n represents the second order finite difference version of F (I use central differences). Plugging in the finite differences, we have

$$F_i^n = a(x_i)\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{h_x^2} + a'(x_i)\frac{u_{i+1}^n - u_{i-1}^n}{2h_x} + f(x_i,t_n).$$

So, taking this expression for F_i^n and it plugging into our Crank-Nicolson scheme yields the linear tridiagonal system that I wrote my code to solve (Code attached at the end of the document).

To check that my code is working, I ran it on the test cases below:

(a) Standard heat equation

$$\begin{cases} u_t = u_{xx}, & t > 0, x \in (0,1) \\ u(x,0) = -4x(x-1), & x \in [0,1] \\ u(0,t) = u(1,t) = 0, & t > 0 \end{cases}$$

(b) Simple forced heat equation

$$\begin{cases} u_t = u_{xx} + x^2 t, & t > 0, x \in (0,1) \\ u(x,0) = -4x(x-1), & x \in [0,1] \\ u(0,t) = u(1,t) = 0, & t > 0 \end{cases}$$

(c) Spatially variable conductivity

$$\begin{cases} u_t = \partial_x(x u_x) & t > 0, x \in (0,1) \\ u(x,0) = -4x(x-1), & x \in [0,1] \\ u(0,t) = u(1,t) = 0, & t > 0 \end{cases}$$

(d) Ill-posed heat equation

$$\begin{cases} u_t = -u_{xx} & t > 0, x \in (0,1) \\ u(x,0) = -4x(x-1), & x \in [0,1] \\ u(0,t) = u(1,t) = 0, & t > 0 \end{cases}$$

In every case except case d, my code ran stably and accurately for many different spatial and temporal step sizes. However, for case (d), my code leads to an "exploding" solution which is expected because the problem doesn't have continuous dependence on the initial data and is thus ill-posed.

(2) Implement the second-order central difference scheme for the wave equation

$$\begin{cases} u_{tt} = \partial_x(a(x)u_x) + f(x,t), \\ u(x,0) = u_0(x), \\ u_t(x,0) = u_1(x) \end{cases}$$

where all functions are periodic in x with period 1.

To turn this into a finite difference problem, let's first expand the PDE as

$$u_{tt} = a(x)u_{xx} + a'(x)u_x + f(x,t).$$

Then, plugging in our central differences, we have

$$\frac{u_i^{n+1} - 2u_i^n + u_i^{n-1}}{h_t^2} = a(x_i)\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{h_x^2} + a'(x_i)\frac{u_{i+1}^n - u_{i-1}^n}{2h_x} + f(x_i, t_n)$$

which yields the explicit time stepping scheme

$$u_i^{n+1} = (r+k)u_{i+1}^n + (2-2r)u_i^n + (r-k)u_{i-1}^n - u_i^{n-1} + h_t^2 f(x_i, t_n)$$

where

$$r = a(x_i)\frac{h_t^2}{h_x^2} \quad \text{and} \quad k = a'(x_i)\frac{h_t^2}{2h_x}.$$

Now, this scheme works great on the interior for $t > 0$. However, at the very first step, we aren't directly given the u_i^{n-1} data and so we need to rely on our initial data. In this case, we can use the condition $u_t(x,0) = u_1(x)$ along with finite differences to get

$$\frac{u_i^2 - u_i^0}{2h_t} = u_1(x_i) \implies u_i^0 = u_i^2 - 2h_t u_1(x_i).$$

This expression can then be plugged into our explicit scheme to yield a slightly modified time stepping scheme for the first time step. My code at the end of the document implements this exact scheme. Do note, I added periodic boundary conditions to my code to make it well posed over the interval for x from 0 to 1.

I ran my code on the test cases below:

(a) Standard wave equation

$$\begin{cases} u_{tt} = u_{xx}, \\ u(x,0) = \sin(2\pi x), \\ u_t(x,0) = 0 \end{cases}$$

(b) Resonant wave equation

$$\begin{cases} u_{tt} = u_{xx} + \sin(2\pi x)\cos(2\pi t), \\ u(x,0) = 0, \\ u_t(x,0) = 0 \end{cases}$$

(c) Plucked string

$$\begin{cases} u_{tt} = u_{xx}, \\ u(x,0) = 0, \\ u_t(x,0) = -4x(x-1) \end{cases}$$

(d) Variable tension string

$$\begin{cases} u_{tt} = \partial_x(xu_x), \\ u(x,0) = \sin(2\pi x), \\ u_t(x,0) = 0 \end{cases}$$

(e) Ill-posed wave equation

$$\begin{cases} u_{tt} = -u_{xx}, \\ u(x,0) = \sin(2\pi x), \\ u_t(x,0) = 0 \end{cases}$$

In all cases, aside from case (e), my code runs stably so long as we have r and k defined above less than 1 which gives us constraints on the relative sizes of h_x and h_t. So, as long as we pick h_x and h_t such that $|r| < 1$ and, $|k| < 1$, our scheme will be stable. For accuracy however, we desire a smaller h_t as well.

For case (e), our scheme leads to many growing amplitude, fast oscillation solutions which comes from the ill-posedness of the problem similar to the ill-posed heat equation.

Code Used

Note: some of the symbols are missing in my code snippet because LaTeX does not support all unicode characters.

Crank-Nicolson Code

```julia
#=
# Crank-Nicolson scheme for solving
#    t(u) = x(a(x) x(u)) + f(x,t)
#
# Author: Caleb Jacobs
# DLM: 12-04-2022
=#

using ForwardDiff
using LinearAlgebra
using Plots

default(xlims = (0, 1), ylims = (-1, 1))

"""
    getCRMat(a, x, ht)

Construct Crank-Nicolson matrix given function `a(x)`, uniform grid
data `x`, and time step size `ht`.
"""
function getCRMat(a, x, ht)
    hx = x[2] - x[1]                        # Spatial step size

    ax  = a.(x)                             # a(x)  evaluated at x
    adx = ForwardDiff.derivative.(a, x)     # a'(x) evaluated at x

    dl = ht * (-ax[2:end] / (2hx^2)
       + adx[2:end] / (4hx))                # Lower diagonal

    d  = 1 .+ ht * ax / hx^2                # Diagonal

    du = ht * (-ax[1:end - 1] / (2hx^2)
       - adx[1:end - 1] / (4hx))            # Upper diagonal

    A = Tridiagonal(dl, d, du)              # Tridiagonal Crank-Nicolson matrix
end

"""
    getRHS(a, f, x, u, t, ht)

Construct right hand side of Crank-Nicolson scheme given functions `a(x)` and
`f(x,t)`, and data (`x`,`u`) at time `t` with time step size `ht`.
"""
function getRHS(a, f, x, u, t, ht)
    hx = x[2] - x[1]                        # Spatial step size
```

```julia

    ax  = a.(x)                              # a(x)  evaluated at x
    adx = ForwardDiff.derivative.(a, x)      # a'(x) evaluated at x

    l = [0; ht * (ax[2:end] / (2hx^2) - adx[2:end] / (4hx)) .* u[1:end - 1]]          # Left node contribution
    m = (1 .- ht * ax / (hx^2)) .* u                                 # Center node contribution
    r = [ht * (ax[1:end - 1] / (2hx^2) + adx[1:end - 1] / (4hx)) .* u[2:end]; 0]    # Right node contribution

    return l + m + r + ht * (f.(x, t) + f.(x, t + ht)) / 2
end

"""
    driver(a, f, u0, hx, ht, tf)

Run Crank-Nicolson method for solving model problem.
"""
function driver(a, f, u0, hx, ht, tf)
    x = range(0 + hx, 1 - hx, step = hx)  # Spatial nodes
    X = [0; x; 1]
    t = 0                        # Initialize time
    u = u0.(x)                   # Initialize solution

    A = getCRMat(a, x, ht)
    F(t, u) = getRHS(a, f, x, u, t, ht)

    display(plot(X, [0;u;0]))

    while t < tf
        u = A \ F(t, u)

        display(plot(X, [0;u;0]))

        t += ht
    end

    display(plot(X, [0;u;0]))

    return u
end
```

Central Difference Wave Code

```
#=
# Central Difference Based Solver for
#    tt(u) = x(a(x) x(u)) + f(x,t)
#
# Author: Caleb Jacobs
# DLM: 12-04-2022
=#

using Plots
using ForwardDiff

default(xlims = (0,1), ylims = (-1, 1))

"""
    solveInitial(x, hx, ht, u0, u1)

Solve for first time step incorporating boundary data `u0` and `u1`.
"""
function solveInitial(a, f, x, u0, u1, ht)
    n = size(x, 1)                                          # Number of nodes
    t = 0                                                   # Initial time
    hx = x[2] - x[1]                                        # Spatial stepsize

    r = (ht^2 / hx^2) * a.(x)                               # a(x)  evaluated at x
    k = (ht^2 / (2hx)) * ForwardDiff.derivative.(a, x)      # a'(x) evaluated at x
    u1x = u1.(x)                                            # u1(x) evaluated at x

    u = zeros(n, 2)                                         # Initialize solution
    u[:, 1] = u0.(x)                                        # Initial condition

    inr = 2:(n - 1)                                         # Inner range
    otr = [1, n]                                            # Outer range

    display(plot(x, u[:, 1]))
    sleep(1)

    # Compute inner node step
    u[inr, 2] .= ((r[inr] + k[inr]) .* u[inr .+ 1, 1] +
                  (2 .- 2r[inr])    .* u[inr, 1]        +
                  (r[inr] - k[inr]) .* u[inr .- 1, 1]   +
                   2ht * u1x[inr] + ht^2 * f.(x[inr], t)) / 2

    # Compute boundary node step using period BCs
    u[otr, 2] .= ((r[otr] + k[otr])  * u[2, 1]     .+
                  (2 .- 2r[otr])    .* u[otr, 1]   .+
                  (r[otr] - k[otr])  * u[n - 1, 1]  +
                   2ht * u1x[otr] + ht^2 * f.(x[otr], t)) / 2

    return u
end
```

```
"""
    solveFD(a, f, hx, ht, u0, u1, tf)

Solve wave-like problem given standard constraints.
"""
function solveFD(a, f, hx, ht, u0, u1, tf)
    x = range(0, 1, step = hx)                          # Spatial nodes
    n = size(x, 1)                                      # Number of nodes
    t = 0                                               # Initialize time

    u = solveInitial(a, f, x, u0, u1, ht)               # Initial solution
    uNew = zeros(n)                                     # Initialize solution vector

    r = (ht^2 / hx^2) * a.(x)                           # a(x)  evaluated at x
    k = (ht^2 / (2hx)) * ForwardDiff.derivative.(a, x)  # a'(x) evaluated at x

    inr = 2:(n - 1)                                     # Inner range
    otr = [1, n]                                        # Outer range

    display(plot(x, u[:,1]))

    while t < tf
        # Compute inner node step
        uNew[inr] .= (r[inr] + k[inr]) .* u[inr .+ 1, 2] +
                     (2 .- 2r[inr])    .* u[inr, 2]      +
                     (r[inr] - k[inr]) .* u[inr .- 1, 2] -
                     u[inr, 1]  +  ht^2 * f.(x[inr], t)

        # Compute boundary node step using period BCs
        uNew[otr] .= (r[otr] + k[otr])  * u[2, 2]     .+
                     (2 .- 2r[otr])    .* u[otr, 2]   .+
                     (r[otr] - k[otr])  * u[n - 1, 2]  -
                     u[otr, 1]  +  ht^2 * f.(x[otr], t)

        u[:, 1] = u[:, 2]   # Move current nodes back
        u[:, 2] = uNew      # Move new nodes into current
        t += ht             # Update time

        display(plot(x, uNew))
    end

    return uNew
end

function driver(a, f, hx, ht, u0, u1, tf)
    sol = solveFD(a, f, hx, ht, u0, u1, tf)
end
```