Problems

(1) A popular explicit Runge-Kutta method is defined by the following formulas:

$$\begin{aligned}
k_1 &= hf(x_n, y_n)\\
k_2 &= hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1)\\
k_3 &= hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_2)\\
k_4 &= hf(x_n + h, y_n + k_3)\\
y_{n+1} &= y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)
\end{aligned}$$

With this Runge-Kutta method in mind, we want to approximate the region of absolute stability acting on the standard test problem $y' = \lambda y$. So, let's just plug this test problem into our scheme as

$$\begin{aligned}
k_1 &= h\lambda y_n\\
k_2 &= h(\lambda y_n + \frac{1}{2}k_1)\\
k_3 &= h(\lambda y_n + \frac{1}{2}k_2)\\
k_4 &= h(\lambda y_n + k_3)\\
y_{n+1} &= y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)\\
&= y_n + h\lambda y_n + \frac{1}{2}(h\lambda)^2 y_n + \frac{1}{6}(h\lambda)^3 y_n + \frac{1}{24}(h\lambda)^4 y_n\\
&= \left(1 + h\lambda + \frac{1}{2}(h\lambda)^2 + \frac{1}{6}(h\lambda)^3 + \frac{1}{24}(h\lambda)^4\right) y_n.
\end{aligned}$$

If we let, $z = h\lambda$, then our final expression can be written as

$$y_{n+1} = \left(1 + z + \frac{1}{2}z^2 + \frac{1}{6}z^3 + \frac{1}{24}z^4\right) y_n$$

which shows us our scheme will converge if

$$p(z) = \left|1 + z + \frac{1}{2}z^2 + \frac{1}{6}z^3 + \frac{1}{24}z^4\right| < 1.$$

This equation is quite tricky to work with so we will find try to approximate the region of absolute stability. First, we will find the solutions of the equation $p(z) = 1$ for purely real z. Doing so, we find the two real roots to be at

$$z = 0$$

and

$$z = \frac{1}{3}\left(-10\sqrt[3]{\frac{2}{9\sqrt{29} - 43}} + 2^{2/3}\sqrt[3]{9\sqrt{29} - 43} - 4\right).$$

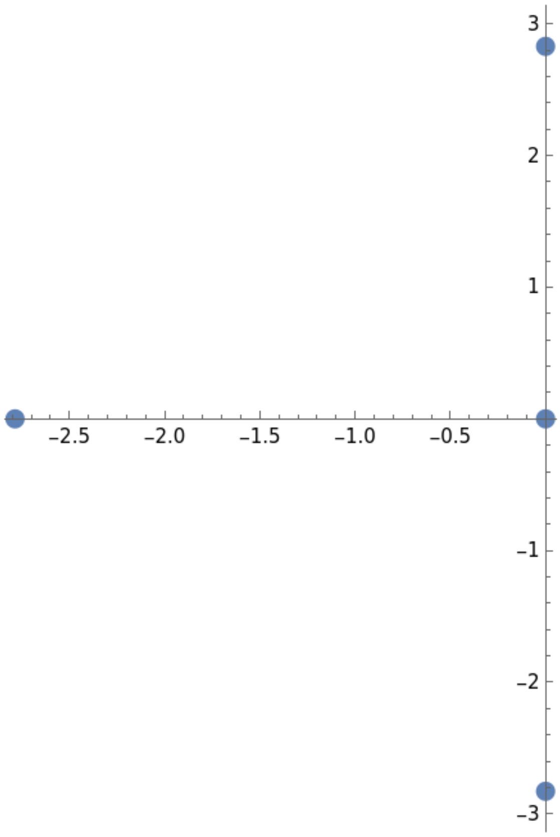
Next, we seek the purely imaginary solutions of $p(z) = 1$ which yields

$$z = -i2\sqrt{2}$$

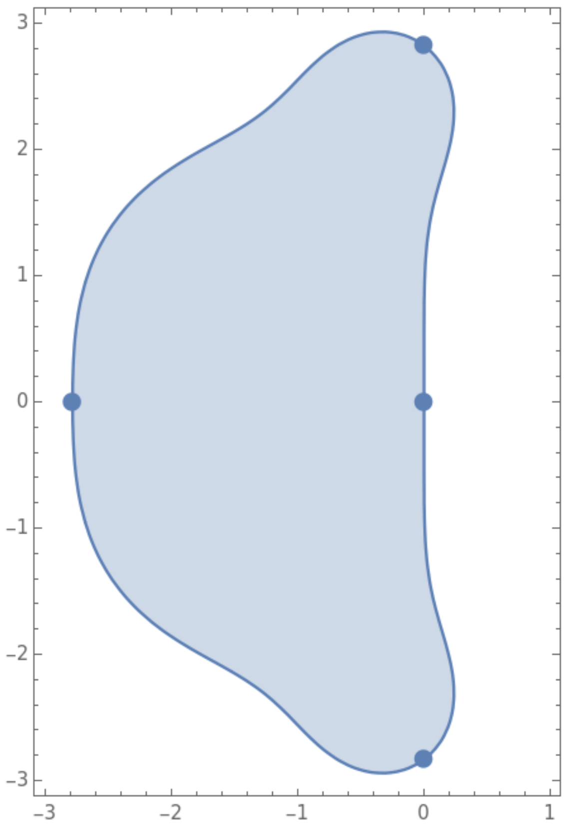
and

$$z = i2\sqrt{2}.$$

Plotting all of these points marks out the rough region below



We can compare this rough region with Mathematica's region plot as



(2) One seeks the solution of the eigenvalue problem

$$\frac{\mathrm{d}}{\mathrm{d}x}\left(\frac{1}{1+x}\frac{\mathrm{d}y}{\mathrm{d}x}\right) + \lambda y = 0$$

with boundary conditions $y(0) = y(1) = 0$. We want to find the λ such that the initial conditions $y(0) = 0$ and $y'(0) = 1$ leads to the boundary condition $y(1) = 0$ being met. Using my Richardson's extrapolation and trapezoidal method code from the last assignment, we can introduce the bisection method to find the desired λ. The bisection method is super easy to implement (*my code is attached at the end*), we just need to choose a starting search range. In this case, we will start with the range $[6.7, 6.8]$. Then, because our ODE is second order, we will need to reduce the ODE into a system of first order ODEs. Reducing this ODE yields the system

$$\begin{aligned} y' &= u \\ u' &= \frac{1}{1+x}u - (1+x)y \end{aligned}$$

with initial conditions $y(0) = 0$ and $u(0) = 1$. Then, plugging this system and search range into my code finds λ to be

$$\lambda = 6.773873469310571$$

which, when compared to Mathematica's solution, has a relative error of

$$1.4423 \cdot 10^{-15}.$$

So, it appears my code is working well and is also quite fast!

Code Used

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

```
#=
# 2×2 First Order ODE Solver Using Trapezoidal rule
#
# Author: Caleb Jacobs
# Date last modified: 03-04-2022
=#

using Plots
using SpecialFunctions
using ForwardDiff
using LinearAlgebra

# First order Bessel Equation system
function f(t, y)
    # Use approximation if we are near the singularity t = 0
    if t < 1e-10
        return [y[2], -3*t/8] # Higher order terms + 5*(t^3)/96 - 7*(t^5)/3072]
    else
        return [y[2], ((1 - t^2)*y[1] - t*y[2]) / (t^2)]
    end
```

```
end

# Tricky eigenvalue system
h(x, y, ) = [y[2], y[2] / (1 + x) - (1 + x) *   * y[1]]

# Evaluate system at specified eigenvalue
function H()
    htmp(x, y) = h(x, y, )
    richTrap(htmp, 0, 1, [0, 1], n = 1, rn = 9)
end

# Newton method system solver
function newton(f; maxIts = 100,   = 1e-8, y0 = [0, 0])
    y = y0                      # Initial guess

    for i   1 : maxIts
        J = ForwardDiff.jacobian(f, y)  # Get jacobian
        ynew = y - (J \ f(y))           # Find next iterate

        # Check for convergence
        if norm(y - ynew) <=
            y = ynew

            return y
        end

        y = ynew                        # Pass to next iteration
    end

    return y
end

# Trapezoidal rule
function trapz(f, a, y0, h, n)
    yi = y0             # Initial conditions
    ti = a              # Initial time
    tf = a + h          # First time step

    # Run trapezoidal until desired time
    for i = 1 : n
        # Current trapezoidal equation
        g(y) = y - (yi + h * (f(ti, yi) + f(tf, y)) / 2)

        yi = newton(g)  # Solve trapezoidal equation

        ti = tf         # Store new time
        tf = ti + h     # Compute next time
    end

    return yi
end
```

```

# Trapezoidal rule with Richardson Extrapolation
function richTrap(f, a, b, y0; n = 1, rn = 1)
    h = (b - a) / n                 # Compute time step

    r = zeros(Float64, rn, rn)      # Initialize richardson matrix
    sol = trapz(f, a, y0, h, n)     # Get initial solution
    r[1, 1] = sol[1]                # Store initial solution

    # Begin Richardson exptrapolation
    for i = 1 : rn - 1
        h /= 2                      # Half time step size
        n *= 2                      # Double number of step to take

        sol = trapz(f, a, y0, h, n) # Get solution with current step size
        r[i + 1, 1] = sol[1]        # Store solution

        # Compute richardson exptrapolation with current data
        for j = 1 : i
            r[i + 1, j + 1] = ((4^j) * r[i + 1, j] - r[i, j]) / (4^j - 1)
        end
    end

    return r[rn, rn]
end

# Bisection method for root finding
function bisect(f, a, b; maxIts = 100,   = 1e-8)
    # Check required conditions
    if a > b
        return NaN
    end

    fa = f(a)       # Left function value
    fb = f(b)       # Right function value

    if (sign(fa) == sign(fb))
        return NaN
    end

    c = 0.0             # Initialize solution

    # Begin bisecting
    for n   1 : maxIts
        c = (a + b) / 2

        fc = f(c)

        # Check for convergence
        if abs(fc) <   || (b - a) / 2 <
            return c
```

```
        end

        # Check for which side to cut interval
        if sign(fc) == sign(fa)
            a  = c
            fa = fc
        else
            b  = c
            fb = fc
        end
    end

    display("Convergence never met,  found:")
    c
end

# besselJ = richTrap(f, 0, 3*, [0,1/2], n = 40, rn = 10)
# display(besselJ)
# display(besselJ - besselj(1, 3*))

sol = bisect(H, 6.7, 6.8,  = 1e-15)
tru = 6.773873469310561
display(sol)
display(abs(tru - sol) / tru)
```