Homework 1

AM213B

Kevin Silberberg

2025-04-09

Problem 1

Suppose E_n satisfies the recursive inequality

$$\begin{cases} E_{n+1} \leq (1 + Ch)\, E_n + h^3 \quad \text{for } n \geq 0 \\ E_0 = 0 \end{cases} \tag{1}$$

where $C > 0$ is a constant independent of h and n.

Derive that

$$E_N \leq \frac{e^{CT} - 1}{C} h^2 \quad \text{for } Nh \leq T \tag{2}$$

Solution

Starting from (1) we have:

$$E_{n+1} \leq (1 + Ch)\, E_n + h^3 \tag{3}$$

$$\frac{E_{n+1}}{h^3} \leq \frac{(1 + Ch)\, E_n + h^3}{h^3} \tag{4}$$

$$\frac{E_{n+1}}{h^3} \leq (1 + Ch)\, \frac{E_n}{h^3} + 1 \tag{5}$$

Let $\Lambda_n = \frac{E_n}{h^3}$ such that (5) becomes

$$\begin{cases} \Lambda_{n+1} \leq (1 + Ch)\, \Lambda_n + 1 \quad \text{for } n \geq 0 \\ \Lambda_0 = 0 \end{cases} \tag{6}$$

Let us solve the first 4 iterates of the recurrance relation.

$$\text{for } n = 0 : \tag{7}$$
$$\Lambda_0 = 0 \tag{8}$$
$$\text{for } n = 1 : \tag{9}$$
$$\Lambda_1 \leq (1 + Ch)\,\Lambda_0 + 1 = 1 \tag{10}$$
$$\text{for } n = 2 : \tag{11}$$
$$\Lambda_2 \leq (1 + Ch)\,\Lambda_1 + 1 = 2 + Ch \tag{12}$$
$$\text{for } n = 3 : \tag{13}$$
$$\Lambda_3 \leq (1 + Ch)\,(2 + Ch) + 1 = C^2h^2 + 3Ch + 3 \tag{14}$$

This suggests that Λ_n can be represented by the following series:

$$\Lambda_n \leq \frac{(1 + Ch)^n - 1}{Ch} \tag{15}$$

We need to prove that the series (15) is equivalent to the recurrance relation (6) for all $n \geq 0$

Let us verify that the first few iterates of the series match that of the recurrance relation. This will serve as our bases case in the proof by induction.

$$\text{for } n = 0 : \tag{16}$$
$$\Lambda_0 \leq \frac{(1 + Ch)^0 - 1}{Ch} = \frac{1 - 1}{Ch} = 0 \tag{17}$$
$$\text{for } n = 1 : \tag{18}$$
$$\Lambda_1 \leq \frac{(1 + Ch) - 1}{Ch} = \frac{Ch}{Ch} = 1 \tag{19}$$
$$\text{for } n = 2 : \tag{20}$$
$$\Lambda_2 \leq \frac{(1 + Ch)^2 - 1}{Ch} = \frac{C^2h^2 + 2Ch}{Ch} = 2 + Ch \tag{21}$$

For the inductive step we need to show that the recurrance relation representation is equivalent to the series representation. We can do this by simply plugging in the series representation into the $n + 1$ case and see if they are equivalent.

starting from the recurrance relation Λ_{n+1} and plugging in the proposed series representation Λ_n

$$\Lambda_{n+1} \leq (1 + Ch)\,\Lambda_n + 1 \tag{22}$$
$$\leq (1 + Ch)\,\frac{(1 + Ch)^n - 1}{Ch} + 1 \tag{23}$$
$$\leq \frac{(1 + Ch)^{n+1} - 1 - Ch}{Ch} + 1 \tag{24}$$
$$\leq \frac{(1 + Ch)^{n+1}}{Ch} - \frac{1}{Ch} - \frac{Ch}{Ch}^{0} + 1 \tag{25}$$
$$\leq \frac{(1 + Ch)^{n+1} - 1}{Ch} \tag{26}$$

Which is exactly equal to the original definition of the series representation of the recurrance relation.

Recall that $\Lambda_n = \frac{E_n}{h^3}$. Plugging this into the definition of (15) we have

$$\frac{E_N}{h^3} \leq \frac{(1+Ch)^N - 1}{Ch} \tag{27}$$

$$E_N \leq \frac{(1+Ch)^N - 1}{C} h^2 \tag{28}$$

We can express the value $e^x = 1 + x + \frac{x^2}{2!} + \cdots$, thus we have the inequality where $x > 0$:

$$1 + x \leq e^x \tag{29}$$

Plugging in $x = Ch$ we have

$$1 + Ch \leq e^{Ch} \tag{30}$$

Since $C > 0$ and $h > 0$ and $N > 0$ we can rewrite (28) as

$$E_N \leq \frac{(1+Ch)^N - 1}{C} h^2 \leq \frac{e^{ChN} - 1}{C} h^2 \tag{31}$$

$$E_N \leq \frac{e^{ChN} - 1}{C} h^2 \tag{32}$$

Let $Nh \leq T$ and we have arrived at our final derivation,

$$E_N \leq \frac{e^{CT} - 1}{C} h^2 \quad \text{for } Nh \leq T \tag{33}$$

Problem 2

Use the composite trapezoidal rule and the composite Simpson's rule, respectively, to approximate the integral

$$I = \int_1^3 \sqrt{2 + \cos^3(x) e^{\sin(x)}} dx \tag{34}$$

For each method, carry out simulations at a sequence of numerical resolutions:

$$N = 2^i \quad \text{for } i = 2, 3, 4, \cdots, 10$$

State and compare the numerical solutions of the two methods at $N = 2^{10}$.

For each method, use the numerical results to do numerical error estimations.

For each method, plot the estimated error (absolute value) as a function of h. Use log-log plot to accommodate the wide range of h and error.

Plot the two cureves in ONE figure to compare the performance of the two methods.

Solution

The composite trapezoidal rule uses a constant spline and applies the trapezoidal rule to each subinterval. For $n+1$ equally spaced nodes:

$$\int_a^b f(x)dx \approx h\left[\frac{1}{2}f(a) + \sum_{i=1}^{n-1} f(a+ih) + \frac{1}{2}f(b)\right] \tag{35}$$

where $h = \frac{b-a}{n}$

The error for the composite trapezoidal rule is bounded by $\frac{1}{12}(b-a)^3\frac{|f^{(2)}(\xi)|}{n^2}$ for some $\xi \in (a,b)$.

The composite simpson's rule uses a quadratic spline. For equally spaced nodes,

$$\int_a^b f(x)dx \approx \frac{h}{3}\left(f(x_0) + 4\sum_{i=1}^{n/2} f(x_{2i-1}) + 2\sum_{i=2}^{n/2} f(x_{2i-2}) + f(x_n)\right) \tag{36}$$

The error of the composite simpson's rule is boundedc by $\frac{1}{180}(b-a)^5\frac{f^{(4)}(\xi)}{n^4}$ for some ξ over an interval (a,b).

Let us write two functions that implements the composite trapezoidal rule and the composite Simpson's rule.

```
using StaticArrays
# the function we are interested in integrating
f(x::Float64) = sqrt(2 + cos(x)^3 * exp(sin(x)))
# the support of the function
support = SVector{2, Float64}(1.0, 3.0)

function trapz(func::Function, sup::SVector{2, Float64}, n::Int)
    sum = 0.0
    a, b = sup
    h = (b-a)/n
    # precompute the summation
    for j in 1:n-1
        sum += func(a + j*h)
    end
    return h/2.0 * (func(a) + 2.0*sum + func(b))
end

function simpsons(func::Function, sup::SVector{2, Float64}, n::Int)
    a, b = sup
    h = (b-a)/n
    sum1 = 0.0
    sum2 = 0.0
    for i = 1:n/2
        sum1 += func(a + (2*i - 1)*h)
    end
    for i = 1:n/2-1
        sum2 += func(a + 2*i*h)
    end
    return h/3.0 * (func(a) + 4.0*sum1 + 2.0*sum2 + func(b))
end;
```

Let us approximate the integral for each method at spatial resolutions $N = 2^i$ for $i = 2, 3, \cdots, 10$ and save the data in a 9×2 matrix data structure where the rows correspond to each spatial resolutions and the columns corespond to each method.

```
# spatial resolutions
r = [2^i for i in 2:10]
# data structure for storing approximate solutions
data = Matrix{Float64}(undef, 9, 2)
for i in eachindex(r)
    data[i, 1] = trapz(f, support, r[i])
    data[i, 2] = simpsons(f, support, r[i])
end
```

Let us print the results of the approximate integral for the trapezoidal and simpson's method at spatial resolution $N = 2^{10}$.

```
println("Spatial resolution: N = $(r[end])")
println("Trapezoidal Method: $(data[end, 1])")
println("Simpson's Method:   $(data[end, 2])")
println("Relative error: $(abs(data[end, 1] - data[end, 2]))")
```

```
Spatial resolution: N = 1024
Trapezoidal Method: 2.4988361220951703
Simpson's Method:   2.498835859563374
Relative error: 2.6253179630231216e-7
```

The numerical solutions for $N = 2^{10}$ agree to within a relative error of approximately 10^{-7} using $N = 2^{10}$ equally spaced grid spaces.

We would like to plot the global truncation error for each method at different spatial resolutions. The integral has no analytical solution, so we use gauss kronrad-quadrature solver to solve the integral to machine precision and use this as our true solution for the calculation of the absolute error.

```
using QuadGK
using GLMakie
# wrapper function for using quadgk
integ(x::Function, sup::SVector{2}) = quadgk(x, sup[1], sup[2]; atol=1e-16, rtol=1e-16)[1]
y_true = integ(f, support)

fig = Figure()
ax = Axis(fig[1, 1],
          title = "numerical error for varying spatial resolution",
          xscale = log10,
          yscale = log10,
          xlabel = L"$N$",
          ylabel = L"$|y - y^*|$")
errors = abs.(y_true .- data)
scatterlines!(ax, r, errors[:, 1], label = "trapezoidal", color = :red, marker=:rect)
scatterlines!(ax, r, errors[:, 2], label = "simpson", color = :blue, marker=:rect)
Legend(fig[1, 2], ax, "composite method")
fig
```

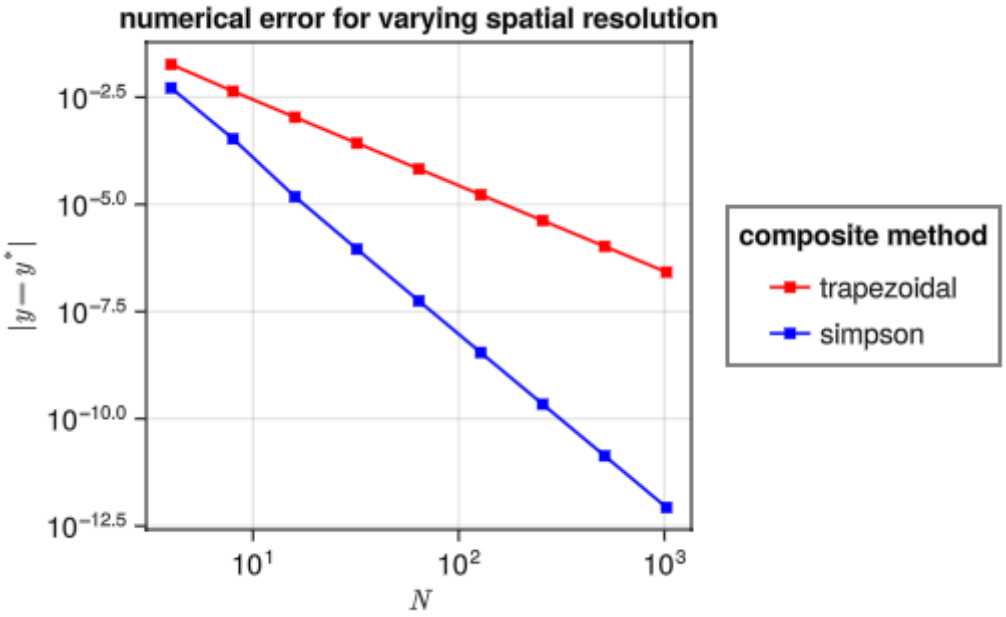

Figure 1: log-log plot of the global truncation error for varying spatial resolution N grid points for the composite trapezoidal method in red, and the composite simpson's method in blue.

Notice how in Figure 1 the slope of the line for each composite method corresponds to the order of the method. Since the composite trapezoidal method is a second-order method and is bounded in error by $O(h^2)$, the error decreases proportionally to h as $h \to 0$ or as $n \to \infty$. Since the plot is on a logarithmic scale, we expect the slope of the line to be -2. The composite Simpson's method is bounded in error by $O(h^4)$, so we expect the slope of the line as n increases to be -4, which is precisely what we see in Figure 1.

Problem 3

Implement Newton's method to solve the non-linear equation of x given below.

$$x - \alpha + \beta \sinh\left(x - \cos\left(s - 1\right)\right) = 0 \tag{37}$$

Solve the equation for each value of $s = [0 : 0.1 : 20]$. Use $\alpha = 0.9$ and $\beta = 50000$.

Plot x vs s and $\cos\left(s - 1\right)$ vs s. Plot the two curves in ONE figure for comparison.

Hint: Look at the sample code on how to implement Newton's method.

Solution

Newton's method is a root finding algorithm defined by the following recurrance relation,

Given a function f, it's derivative, f', and an initial value x_1, iteratively define

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \quad k = 1, 2, 3, \cdots \tag{38}$$

Let us write a code that implements newton's method.

```
# newton's method central difference approximation
function newton(f::Function; x_init::Float64 = 1.0, maxiter = 100, h = 1e-8)
    x_current = x_init
    x_next = x_current
    y = f(x_current)
    Δx = Inf
    k = 1
    while (abs(Δx) > 10*eps()) && (abs(y) > 10*eps()) && (k < maxiter)
        df = (f(x_current + h) - f(x_current - h)) / (2*h)
        Δx = -y/df
        x_next = x_current + Δx
        k += 1
        y = f(x_next)
        x_current = x_next
    end
    x_current
end
```

```
newton (generic function with 1 method)
```

The above function takes in a function $g(x)$ and an optional keyword arguments `x_init`, the initial value with default value 1, and `maxiter` with default argument set to 100 iterations. The algorithm continues until the maximum number of iterations has been met, or the change in the previous step and the current step has fallen below close to machine precision.

Let us plot the solution for (37) using newton's method for varying s against the value $\cos(s-1)$ for all s.

```
# function we are interested in solving
g(x, α, β, s) = x - α + β*sinh(x - cos(s - 1))
# parameters
α = 0.9
β = 50000
r = collect(0:0.1:20)
# figure
fig = Figure()
ax = Axis(fig[1, 1], title = L"Newton's Method solutions for varying $s$", xlabel = L"$s$")
data = [newton(x->g(x, α, β, s)) for s in r]
scatterlines!(ax, r, data, label = L"$x$")
data = [cos(s - 1) for s in r]
scatterlines!(ax, r, data, label = L"$\cos{(s - 1)}$")
Legend(fig[1,2], ax, "y-axis")
fig
```

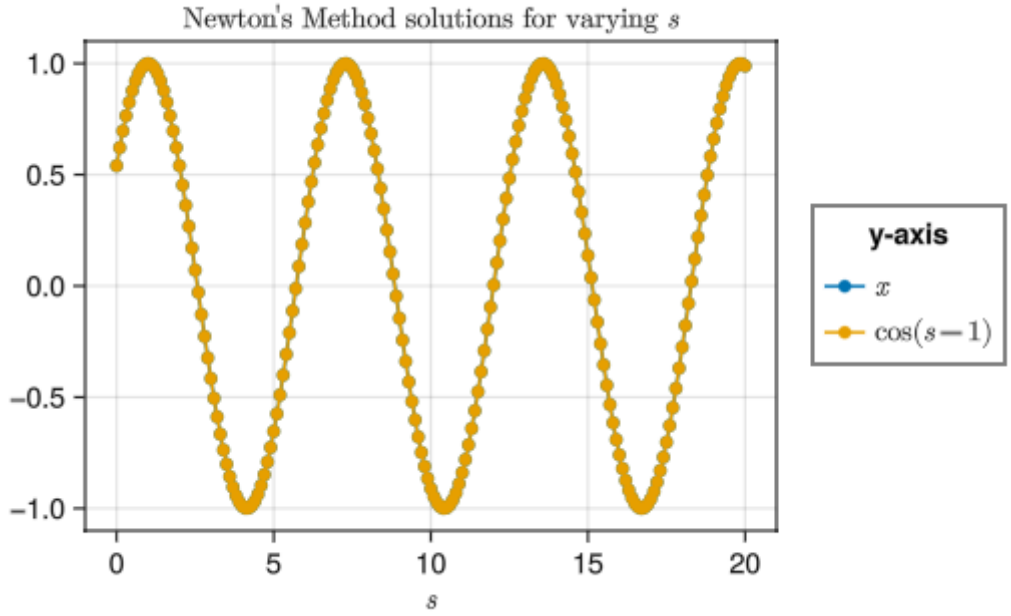

Figure 2: The solution x in blue and the value $cos(s-1)$ for the function at varying values of s.

In Figure 2 we can see that the value of $\cos(s-1)$ and the value of the solution x computed using newton's method are identical to machine precision.

Problem 4

Implement the Euler method and the backward Euler method to solve the IVP below.

$$\begin{cases} \dot{u} = -\lambda \sinh\left(u - \cos(t-1)\right) & \lambda = 10^6 \\ u(0) = 0 \end{cases} \tag{39}$$

Part 1

For the Euler method, solve the IVP to $T = 2^{-10}$. Try $h = 2^n \quad \forall n \in \{-18, -19, -20, \cdots\}$

At what time step size, the numerical solution remains bounded?

Plot one representative figure showing the behavior of numerical solution when the time step is not small enough.

Plot another representative figure when the time step is small enough.

Solution

Let us write the explicit Euler Method.

$$y_{n+1} = y_n + hf(y_n) \tag{40}$$

where $h = \frac{b-a}{n}$ and n is the number of time steps.

```
function euler(f::Function, u0::Float64, tspan::SVector{2, Float64}, h::Float64)
    t0, tf = tspan
    N = Int(floor((tf - t0)/h))
    t = Vector{Float64}(undef, N+1)
    u = Vector{Float64}(undef, N+1)
    t[1] = t0
    u[1] = u0
    for i = 1:N
        u[i+1] = u[i] + h * f(u[i], t[i])
        t[i+1] = t[i] + h
    end
    return t, u
end
```

```
euler (generic function with 1 method)
```

Let us solve the IVP from $T = 0$ to $T = 2^{-10}$ for varying temporal resolutions.

```
u̇(u, t, λ) = -λ * sinh(u - cos(t - 1))
ts = SVector{2, Float64}(0.0, 2^(-10))
hs = [1/(2^18), 1/(2^19), 1/(2^20), 1/(2^21)]
lbs = [L"h = 2^{-18}", L"h = 2^{-19}", L"h = 2^{-20}", L"h = 2^{-21}"]
fig = Figure()
ax = Axis(
    fig[1, 1],
    title = "Bounded and Unbounded solutions for varying timestep h",
    limits = ((0, 0.0001), (-6, 6))
)
for i in eachindex(hs)
    h = hs[i]
    t, u = euler((u, t) -> u̇(u, t, 1e6), 0.0, ts, h)
    lines!(ax, t, u, label = lbs[i])
end
Legend(fig[1, 2], ax)
fig
```

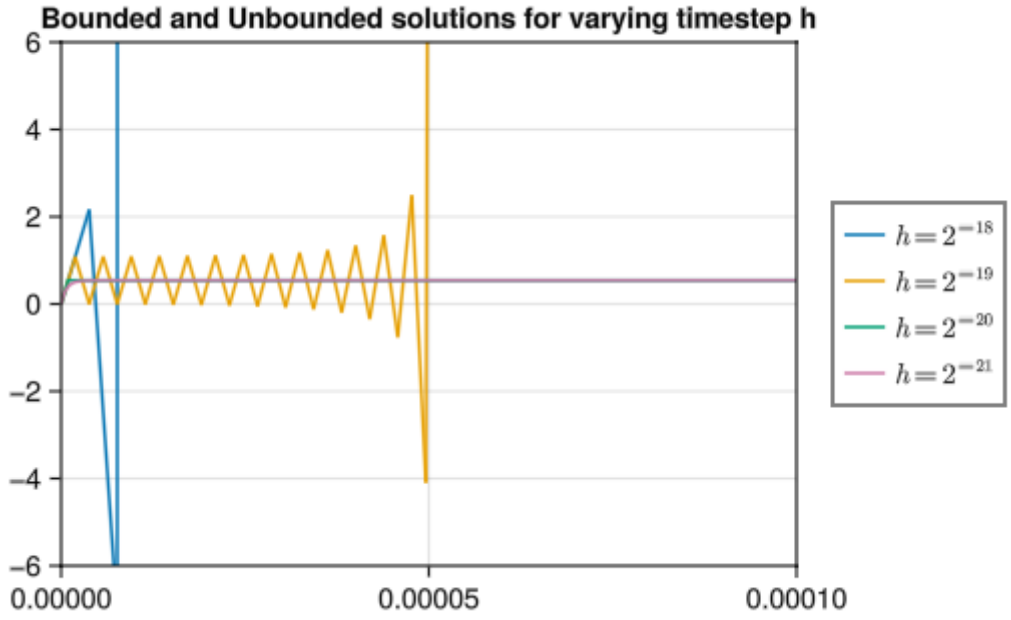

Figure 3: Plot of the Unbounded solutions at $h = 2^{-18}$ in blue and $h = 2^{-19}$ in yellow, and Bounded solutions at $h = 2^{-20}$ in green and $h = 2^{-21}$ in pink

The numerical solution remains bounded for all $h > 2^{20}$. You can clearly see this in Figure 3.

Part 2

For the backward Euler method, solve the IVP to $T = 10$. Use Newton's method to solve the non-linear equation in each time step. Use $h = 0.1$ in your simulations.

Plot the numerical solution vs t for the backward Euler method.

Solution

The backward Euler Method is as follows:

$$y_{n+1} = y_n + hf(y_{n+1}, t_{n+1}) \tag{41}$$

$$y_{n+1} - hf(y_{n+1}) - y_n = 0 \tag{42}$$

where we need to find the roots of the polynomial for equation (42).

```
function backwardEuler(f::Function, u0::Float64, tspan::SVector{2, Float64}, h::Float64)
    t0, tf = tspan
    N = Int(floor((tf-t0)/h))
    t = Vector{Float64}(undef, N+1)
    u = Vector{Float64}(undef, N+1)
    t[1] = t0
    u[1] = u0
    for i = 1:N
        t[i+1] = t[i] + h
        u[i+1] = newton(x -> x - h*f(x, t[i+1]) - u[i], h = 1e-14)
```

```
    end
    return t, u
end
```

```
backwardEuler (generic function with 1 method)
```

Let us solve the IVP (39) using backward euler from $T_0 = 0$ to $T_f = 10$ using a temporal resolution of $h = 0.1$.

```
ts = SVector{2, Float64}(0.0, 10.0)
h = 0.1
u0 = 0.0
t, u = backwardEuler((u, t) -> u̇(u, t, 1e6), u0, ts, h)
fig = Figure()
ax = Axis(
    fig[1, 1],
    title = "Numerical solution vs t for the backward Euler method.",
    xlabel = "t"
)
lines!(ax, t, u)
fig
```

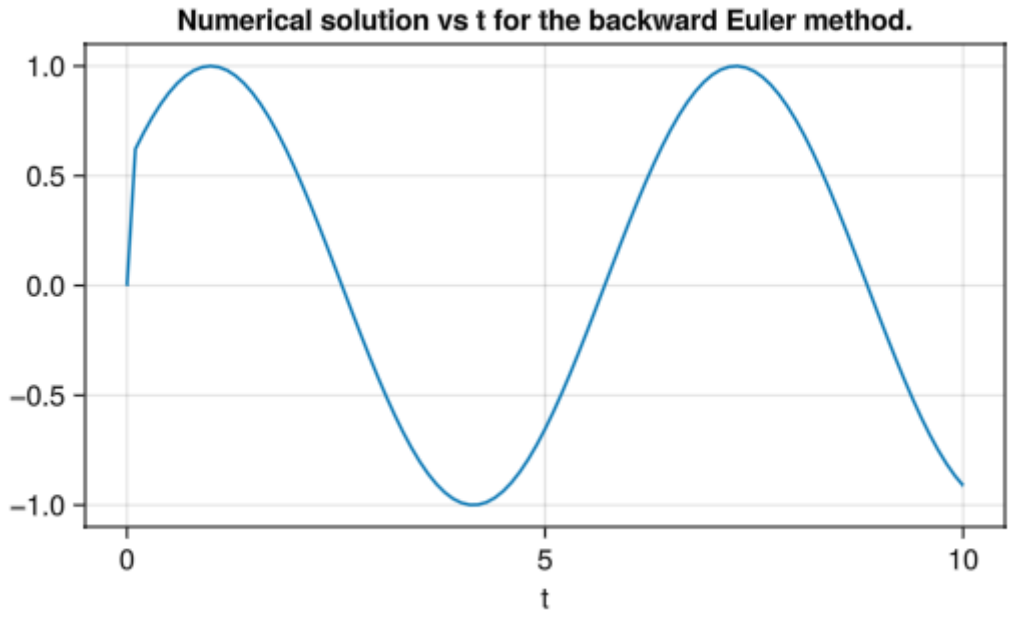

Figure 4: Plot of the numerical solution in blue over time for $t_0 = 0$ to $t_f = 10$ for temporal resolution $h = 0.1$.

Problem 5

Implement the trapezoidal method to solve the IVP in Problem 4.

Use Newton's method to solve the non-linear equation in each time step.

Part 1

Solve the IVP to $T = 10$. Use $h = 0.1$ in your simulations.

Plot the numerical solution vs t. Is the numerical solution bounded?

Do you observe any oscillation in the numerical solution with $h = 0.1$?

Solution

The trapezoidal method for solving IVP's is as follows:

$$y_{n+1} = y_n + \frac{1}{2}h\left(f(y_n, t_n) + f(y_{n+1}, t_{n+1})\right) \tag{43}$$

Where we need to solve for the roots of the equation

$$\frac{1}{2}h\left(f(y_n, t_n) + f(y_{n+1}, t_{n+1})\right) - y_{n+1} + y_n = 0 \tag{44}$$

in order to find the next time step y_{n+1}.

Let us Solve the IVP from $T_0 = 0$ to $T_f = 10$ and use temporal resolution $h = 0.1$.

```
function trapezoidalMethod(f::Function, u0::Float64, tspan::SVector{2, Float64}, h::Float64)
    t0, tf = tspan
    N = Int(floor((tf - t0)/h))
    t = Vector{Float64}(undef, N + 1)
    u = Vector{Float64}(undef, N + 1)
    t[1] = t0
    u[1] = u0
    for i = 1:N
        t[i + 1] = t[i] + h
        u[i+1] = newton(x->0.5*h*(f(u[i], t[i]) + f(x, t[i+1])) - x + u[i], h = 1e-14)
    end
    return t, u
end
t, u = trapezoidalMethod((u, t) -> u̇(u, t, 1e6), u0, ts, h)
fig = Figure()
ax = Axis(
    fig[1, 1],
    title = "Numerical solution to the IVP with trapezoidal method",
    xlabel = "t"
)
lines!(ax, t, u, label = L"h = 0.1")
Legend(fig[1, 2], ax)
fig
```

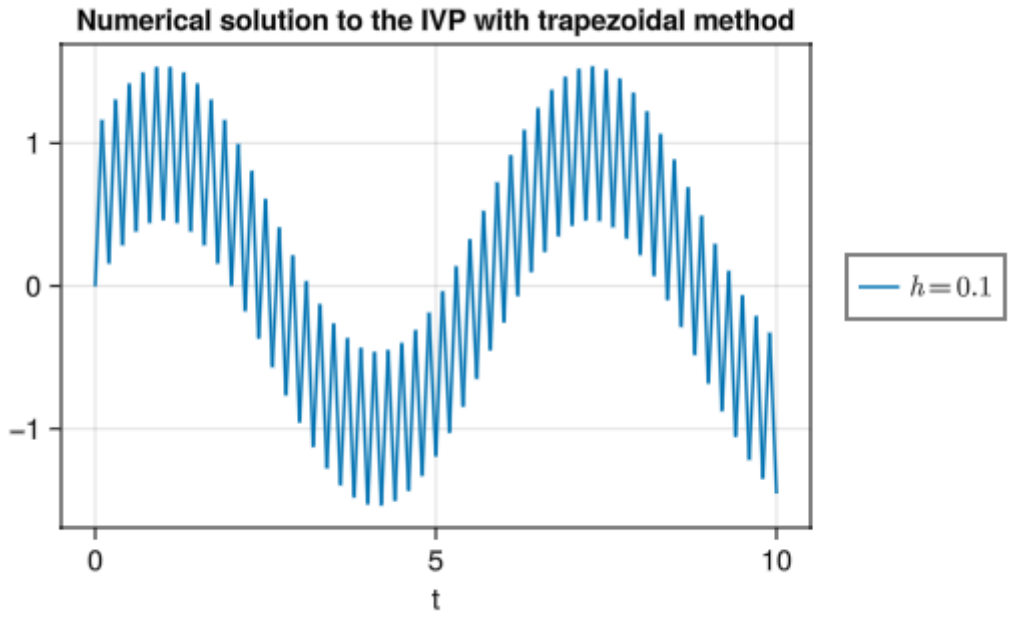

Figure 5: Plot of the numerical solution in blue for $h = 0.1$, the same as the previous figure, except solved with the trapezoidal method, using newtons root finding for every timestep. The Solution is bounded, but oscillates.

The solution is bounded, but it oscillates about the mean of the true solution?

Part 2

Reduce the time step to $h = 2^n \forall n \in \{-7, -8, -9, \cdots\}$

What happens to the oscillation when the time step is refined?

Solution

Let us refine the temporal resolution.

```
fig = Figure()
ax = Axis(fig[1, 1], title = "Trapezoidal method for varying h")
hs = [1/(2^7), 1/(2^8), 1/(2^9), 1/(2^10)]
lbs = [L"h = 2^{-7}", L"h = 2^{-8}", L"h = 2^{-9}", L"h = 2^{-10}"]
for i in eachindex(hs)
    t, u = trapezoidalMethod((u, t) -> u̇(u, t, 1e6), u0, ts, hs[i])
    lines!(ax, t, u, label = lbs[i])
end
Legend(fig[1, 2], ax)
fig
```

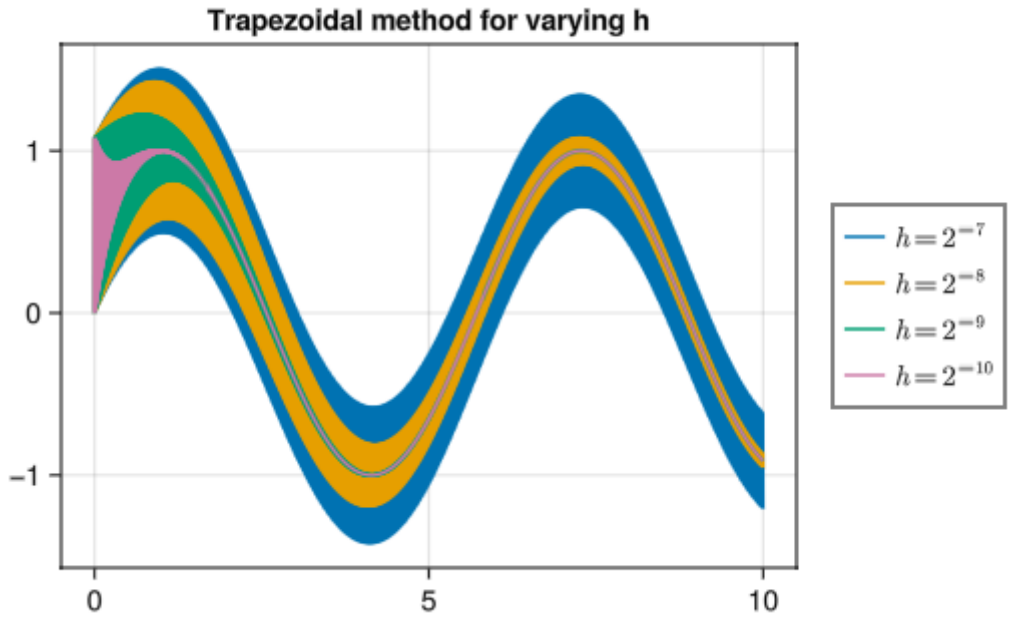

Figure 6: Plot of the numerical solutions for varying temporal resolutions $h = 2^{-7}$ in blue, $h = 2^{-8}$ in yellow, $h = 2^{-9}$ in green, and $h = 2^{-10}$ in pink. Notice that as h decreases the oscillations decrease with time.

As the time step is refined, the oscillations seem to decrease (exponentially) with time.

Problem 6

Use the Euler method and the 2-step midpoint method, respectively, to solve the IVP

$$\begin{cases} \dot{u} = -u \\ u(0) = 1 \end{cases} \tag{45}$$

The exact solution of the IVP is $u(t) = e^{-t}$. In the midpoint method, use the exact solution $u_1 = e^{-h}$ to get started. Use $h = 0.2$ for both methods.

Part 1

Solve the IVP to $T = 2$. Compare the numerical results of the two methods and the exact solution in ONE figure.

Is the midpoint method more accurate than the Euler in this time period?

Solution

The midpoint method is given by the following,

$$y_{n+1} = y_{n-1} + 2hf(y_n, t_n) \tag{46}$$

Let us solve the IVP given by (45), with $h = 0.2$.

```julia
# midpoint method
function midpoint(f::Function, u0::Float64, u1::Float64, tspan::SVector{2, Float64}, h::Float64)
    t0, tf = tspan
    N = Int(floor((tf - t0)/h))
    t = Vector{Float64}(undef, N + 1)
    u = Vector{Float64}(undef, N + 1)
    t[1] = t0
    u[1] = u0
    u[2] = u1
    t[2] = t[1] + h
    for i = 2:N
        t[i + 1] = t[i] + h
        u[i + 1] = u[i - 1] + 2*h*f(u[i], t[i])
    end
    return t, u
end
λ(t) = exp(-t)
ts = SVector{2, Float64}(0.0, 2.0)
h = 0.2
fig = Figure()
ax = Axis(fig[1, 1], title = "comparison of euler and midpoint", xlabel = L"t")
t, u = midpoint((u, t) -> -u, 1.0, exp(-h), ts, h)
lines!(ax, t, u, label = "midpoint")
t, u = euler((u, t) -> -u, 1.0, ts, h)
lines!(ax, t, u, label = "euler")
lines!(ax, t, λ.(t), color = :red, linestyle = :dash, label = "exact")
Legend(fig[1, 2], ax)
fig
```

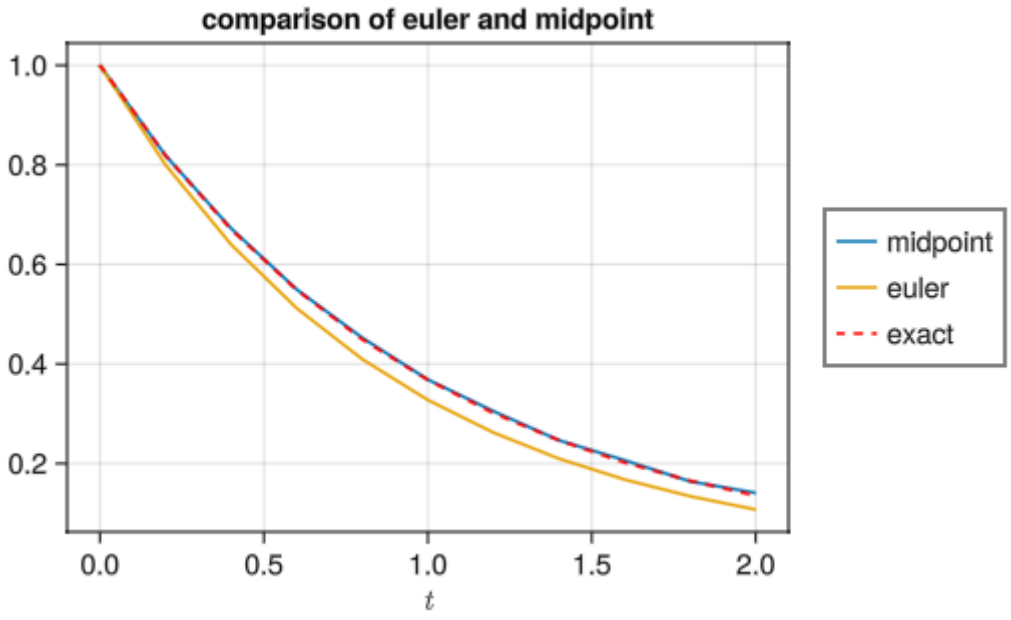

Figure 7: Comparison of the Euler method in yellow, and the 2-step midpoint method in blue. The exact solution is a dashed red line. The midpoint method seems to be more accurate than the euler method.

The midpoint method is more accurate than the euler method.

Part 2

Solve the IVP to $T = 20$. Compare the numercial results of the two methods and the exact solution in ONE figure.

Is the result of the midpoint method well behaved over this longer period?

Solution

Let us compute the same as above but for $T = 20$

```
ts = SVector{2, Float64}(0.0, 20.0)
h = 0.2
fig = Figure()
ax = Axis(
    fig[1, 1],
    title = "comparison of euler and midpoint methods",
    xlabel = "t",
    limits = ((10.0, 20.0), nothing)
)
t, u = midpoint((u, t) -> -u, 1.0, exp(-h), ts, h)
lines!(ax, t, u, label = "midpoint")
t, u = euler((u, t) -> -u, 1.0, ts, h)
lines!(ax, t, u, label = "euler")
lines!(ax, t, λ.(t), color = :red, linestyle = :dash, label = "exact")
Legend(fig[1, 2], ax)
fig
```

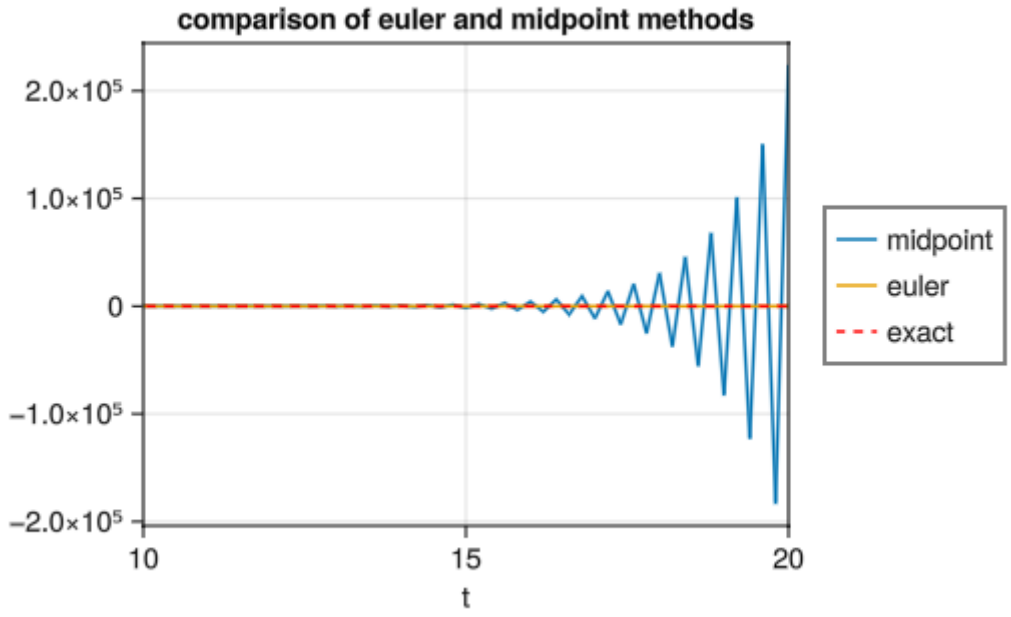

Figure 8: Comparison of the numerical solution to the IVP using the euler (yellow) and 2-step midpoint method (blue) and exact solution in dashed red line from $t = 10$ to $t = 20$.

The numerical solution using the midpoint method begins to oscillate and blow up after $t = 15$ and $t \to \infty$.

Part 3

With $T = 20$, reduce the time step to $h = \frac{0.2}{32}, \frac{0.2}{64}, \frac{0.2}{128}$. Does that reduce the growth of error in the midpoint method?

```
ts = SVector{2, Float64}(0.0, 20.0)
hs = [0.2/32,  0.2/64, 0.2/128]
lbs = [L"h = \frac{0.2}{32}", L"h = \frac{0.2}{64}", L"h = \frac{0.2}{128}"]
fig = Figure(resolution = (600, 520))
for i in eachindex(hs)
    ax = Axis(
        fig[i, 1],
        title = lbs[i],
        limits = ((15, 20), (-10, 10)),
        xlabel = "t"
    )
    t, u = midpoint((u, t) -> -u, 1.0, exp(-hs[i]), ts, hs[i])
    lines!(ax, t, u, label = "midpoint")
    t, u = euler((u, t) -> -u, 1.0, ts, hs[i])
    lines!(ax, t, u, label = "euler")
    lines!(ax, t, λ.(t), color = :red, linestyle = :dash, label = "exact")
end
Legend(fig[1, 2], ax)
fig
```

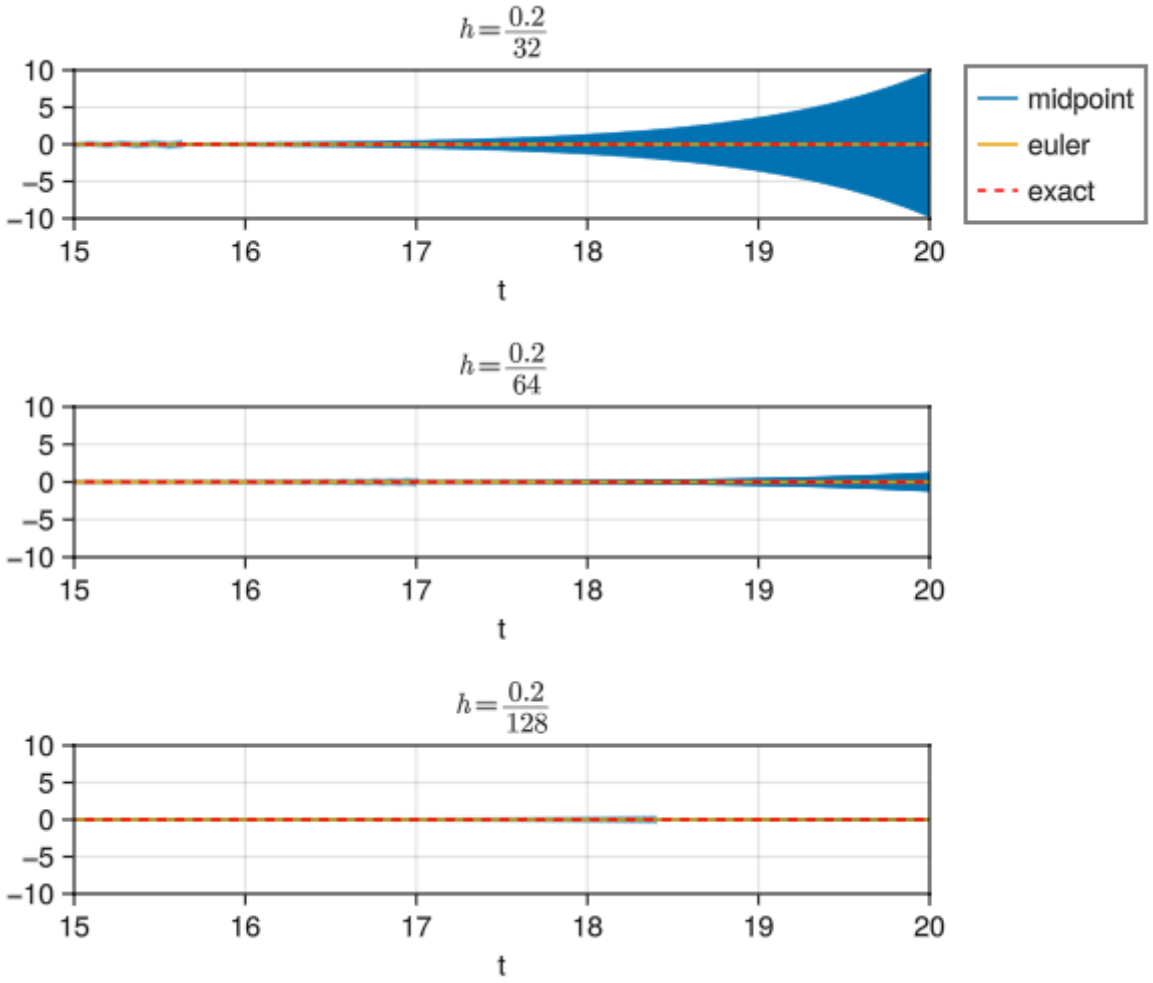

Figure 9: Comparison of Euler method in yellow and midpoint method in blue and the exact solution in dashed red for varying temporal resolutions h from time $T = 15$ to $T = 20$. As h descreases the error also decreases for this domain.

Increasing the temporal resolution results in a reduction in the growth of the error for the midpoint method.