Project 2

November 24, 2022

1 MAD4401 - Project 2

Consider the Initial Value Problem (IVP):

$$y'(t) = t^{-2}(\sin 2t - 2ty) \text{ for } 1 \le t \le 2,$$

$$y(1) = 2.$$

This IVP has an exact solution

$$y(t) = \frac{1}{2}t^{-2}(4 + \cos 2 - \cos 2t).$$

1.1 Problem 1

1.1.1 Part (a)

Approximate $y(t)$ using Euler's method with $h = \frac{1}{4}$ and complete the given table.

[1]:
```
"""
EulersMethod(f, a, b, N,  ) - implementation of Euler's method
for finding approximate solutions for initial value problems.
Arguments:
    - f := a function object representing first derivative
    - a := lower bound of interval
    - b := upper bound of interval
    - N := number of mesh points (doesn't include initial point)
    -   := the initial value given (64-bit floating-point preffered)
"""
function EulersMethod(f, a, b, N,  )
    h = (b-a)/N    # step size
    t = a    # set first mesh point to first point in interval
    w =      # set the initial value for w0
    vals = [w]    # vector to return approximations

    # compute approximations for remaining mesh points
```

```julia
    for i in 1:N
        w = w + h*f(t,w)    # compute w_i
        t = a + i*h    # compute t_i
        append!(vals, w)
    end
    return vals
end;
```

[2]:
```julia
# implementation of first derivative
function f(t,y)
    (t^-2)*(sin(2t)-2t*y)
end;
```

[3]:
```julia
# compute approximations
approximationsEM = EulersMethod(f, 1, 2, 4, 2.);
```

[4]:
```julia
# vector containing approximations of y(t) with h = 1/4
approximationsEM
```

[4]:
```
5-element Vector{Float64}:
 2.0
 1.2273243567064205
 0.8321501570804852
 0.5704467722825309
 0.37882661467612455
```

1.1.2 Part (b)

Plot the approximation.

[5]:
```julia
using Plots
```

[6]:
```julia
# exact function
function g(t)
    (1/2)*(t^-2)*(4+cos(2)-cos(2t))
end;
```

[7]:
```julia
# compute exact values
exactEM = g.(collect(1:.25:2));
```

[8]:
```julia
# vector containing exact values
exactEM
```

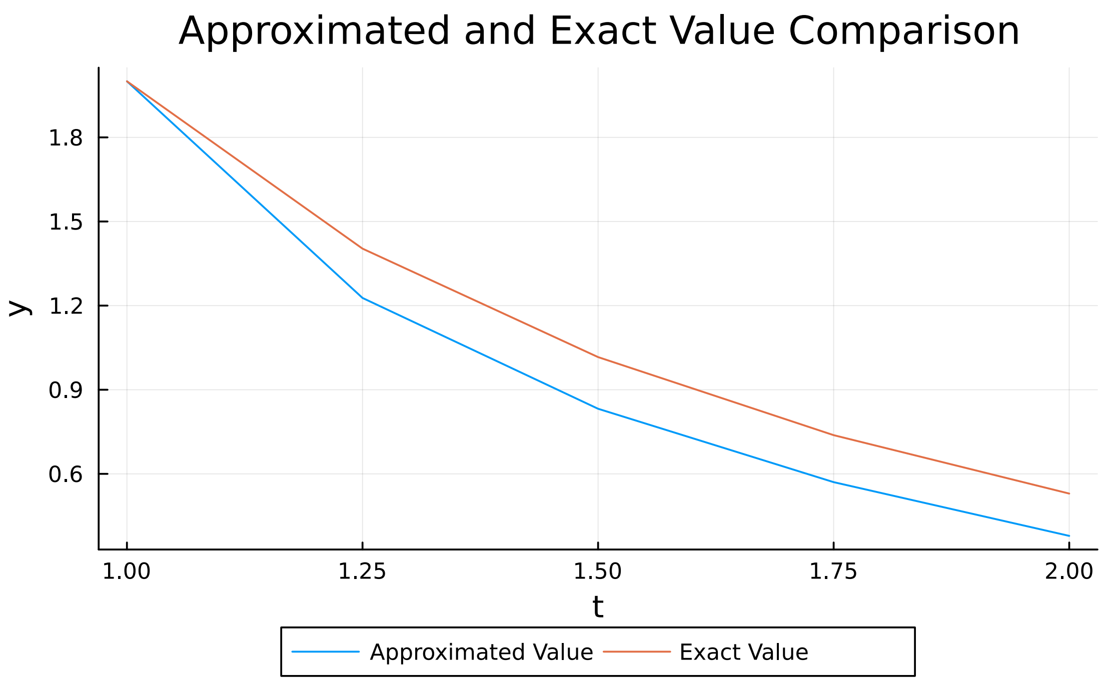
[8]:
```
5-element Vector{Float64}:
 2.0
 1.4031989692799334
 1.0164101466785118
```

```
0.7380097715499842
0.5296870980395587
```

Below is the graph comparing the exact value to the approximated values.

[9]:
```
x = collect(1:.25:2)
y1 = approximationsEM
y2 = exactEM
plot(x,[approximationsEM,exactEM],label=["Approximated Value" "Exact Value"])
plot!(legend=:outerbottom, legendcolumns=2)
xlabel!("t")
ylabel!("y")
title!("Approximated and Exact Value Comparison")
```

[9]:



1.1.3 Part (c)

Approximate $y(t)$ using Euler's method with $h = \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{32}$ and complete the given table.

[10]:
```
# compute exact values for given y(2)
exactValEM = g(2)
```

[10]: `0.5296870980395587`

[11]:
```julia
# compute approximated values at each given N
approximationsN = [];
i = 2
while i <= 32
    vec = EulersMethod(f,1,2,i,2.)
    append!(approximationsN, vec[end]) # only take the last approximation
    i = i*2
end
```

[12]:
```julia
# computed approximations at each N
approximationsN
```

[12]:
```
5-element Vector{Any}:
 0.18290957292869525
 0.37882661467612455
 0.45878660562608714
 0.495261988720689
 0.5127188700310937
```

[13]:
```julia
# compute errors
errorsN = []
for j in 1:5
    append!(errorsN,(abs.(approximationsN[j]-exactValEM)))
end
```

[14]:
```julia
# vector of the computed errors
errorsN
```

[14]:
```
5-element Vector{Any}:
 0.34677752511086346
 0.15086048336343416
 0.07090049241347157
 0.03442510931886972
 0.01696822800846498
```

1.1.4 Part (d)

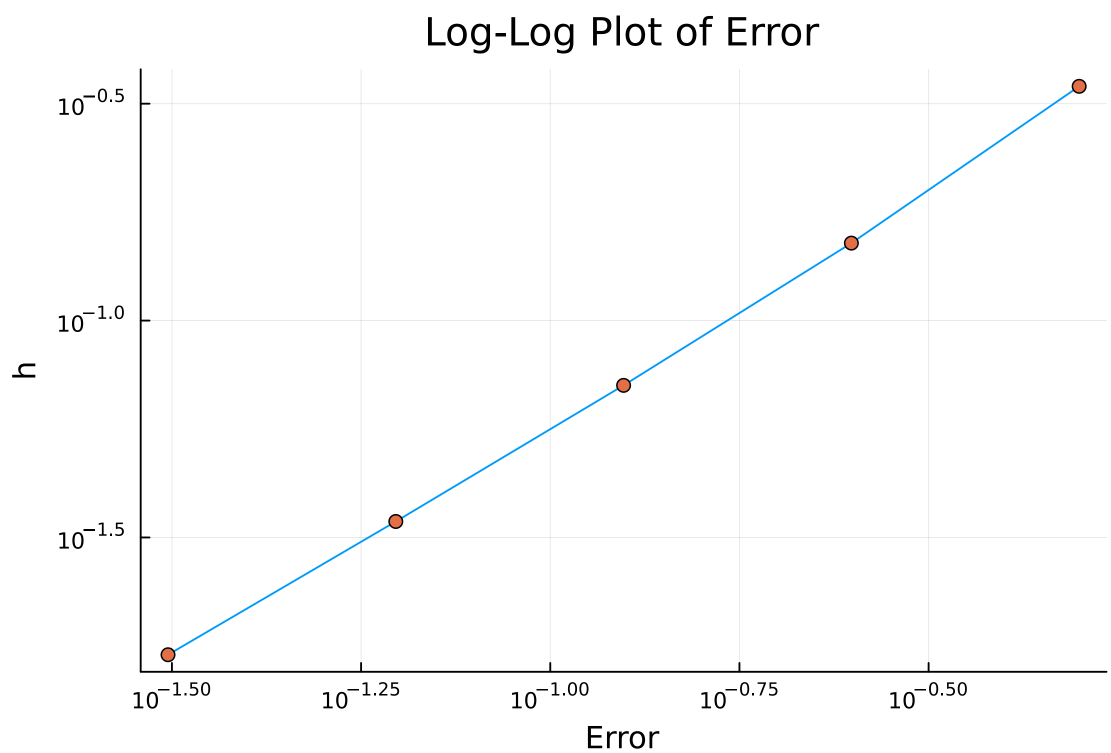
Use the loglog in Matlab to plot and verity the error is converging at the rate O(h). (Experimenting with Julia for this project, hopefully the log-log plot came out correctly)

[15]:
```julia
using Plots
```

[16]:
```julia
# generate a log-log plot of the errors
h = [0.5, 0.25, 0.125, 0.0625, 0.03125]
plot(h,errorsN) # plot h values against errors
plot!(xscale=:log10, yscale=:log10, minorgrid=true) # change scaling
plot!(h, errorsN, seriestype=:scatter, legend=false)
```

```julia
# add plot attributes
xaxis!("Error")
yaxis!("h")
title!("Log-Log Plot of Error")
```

[16]:



1.2 Problem 2

1.2.1 Part (a)

Approximate $y(t)$ using the Runge-Kutta (order 4) method with $h = \frac{1}{4}$.

[17]:
```julia
"""
RungeKutta4(f, a, b, N,  ) - Implementation of Runge-Kutta method (order 4)
Arguments:
    - f := function object
    - a := lower bound of interval
    - b := upper bound of interval
    - N := number of mesh points
    -   := initial value (64-bit floating-point preffered)
"""
function RungeKutta4(f, a, b, N,  )
```

```
    h = (b-a)/ N
    t = a
    w =
    vals = [w]

    # compute approximations for remaining mesh points
    for i in 1:N
        k1 = h * f(t,w)
        k2 = h * f(t+(h/2), w + (k1/2))
        k3 = h * f(t+(h/2), w + (k2/2))
        k4 = h * f(t+h, w + k3)
        w = w + ((k1+2k2+2k3+k4)/6)
        append!(vals, w)    # add approximation to approximation vector
        t = a + i*h
    end
    return vals
end;
```

[18]:
```
# vector of computed approximations using given parameters
approximationsRK = RungeKutta4(f, 1, 2, 4, 2.)
```

[18]:
```
5-element Vector{Float64}:
 2.0
 1.403356615429061
 1.0165585859136061
 0.7381316836791276
 0.5297855647836995
```

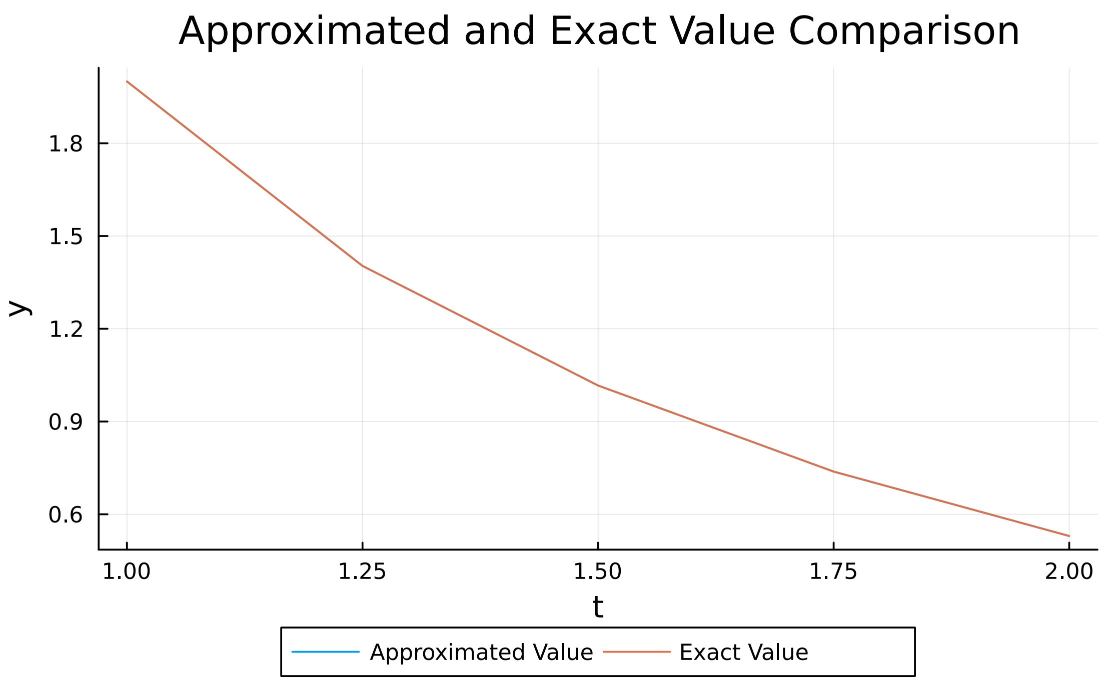
1.2.2 Part(b)

Plot the approximations.

[19]:
```
# compute exact values for function
exactRK = g.(collect(1:.25:2));
```

[20]:
```
# generate graph of approximated vs. exact values
# they're REALLY close, can barely see them both
x = collect(1:.25:2)
y1 = approximationsRK
y2 = exactRK
plot(x,[approximationsRK,exactRK],label=["Approximated Value" "Exact Value"])
plot!(legend=:outerbottom, legendcolumns=2)
xlabel!("t")
ylabel!("y")
title!("Approximated and Exact Value Comparison")
```

[20]:



1.2.3 Part (c)

Approximate $y(t)$ using the Runge-Kutta (order 4) method with the same values of h as in problem 1.

[21]:
```
# compute exact values for given y(2)
exactValRK = g(2)
```

[21]: 0.5296870980395587

[22]:
```
# compute approximated values at each given N
approximationsRKN = [];
i = 2
while i <= 32
    vec = RungeKutta4(f,1,2,i,2.)
    append!(approximationsRKN, vec[end]) # only take the last approximation
    i = i*2
end
```

[23]:
```
# computed approximations at each N
approximationsRKN
```

[23]:

```
5-element Vector{Any}:
 0.5315587749650665
 0.5297855647836995
 0.5296925824351879
 0.5296874192459855
 0.5296871174398682
```

[24]:

```julia
# compute errors
errorsRKN = []
for j in 1:5
    append!(errorsRKN,(abs.(approximationsRKN[j]-exactValRK)))
end
```

[25]:

```julia
# output computed absolute errors
errorsRKN
```

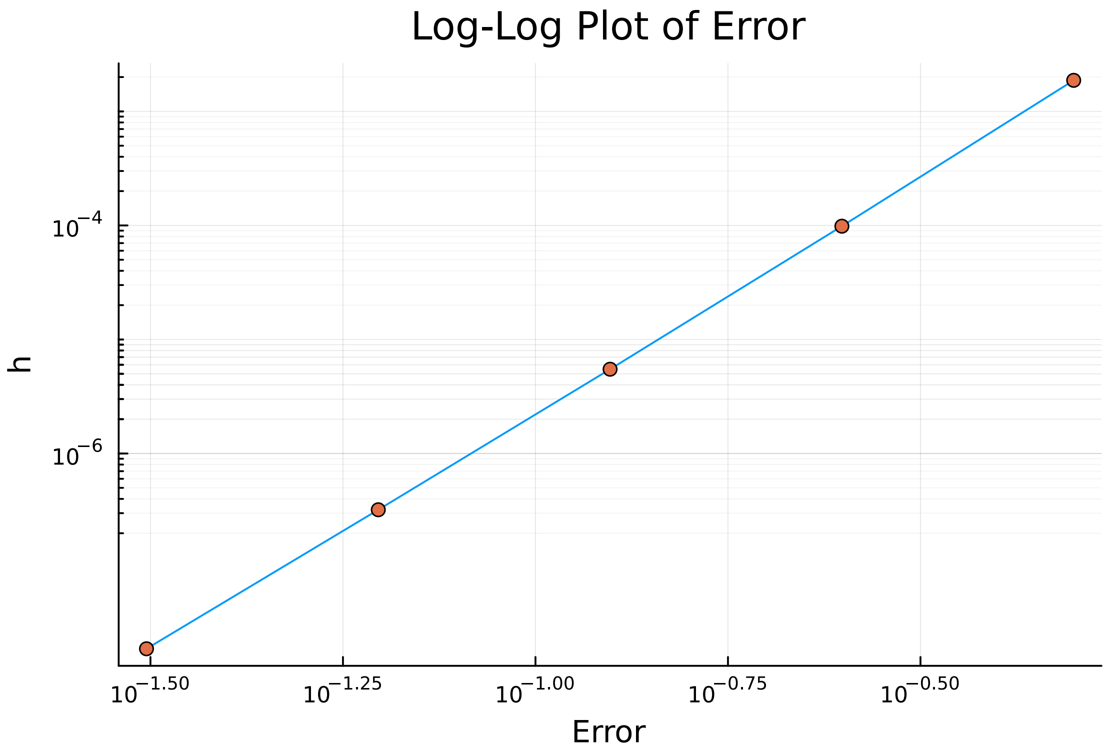
[25]:

```
5-element Vector{Any}:
 0.0018716769255078258
 9.846674414082379e-5
 5.484395629196115e-6
 3.212064267898995e-7
 1.9400309470007926e-8
```

[26]:

```julia
# generate a log-log plot of the errors
h = [0.5, 0.25, 0.125, 0.0625, 0.03125]
plot(h,errorsRKN)
plot!(xscale=:log10, yscale=:log10, minorgrid=true)
plot!(h, errorsRKN, seriestype=:scatter, legend=false)
xaxis!("Error")
yaxis!("h")
title!("Log-Log Plot of Error")
```

[26]:


Log-Log Plot of Error
h
10^−4
10^−6
10^−1.50
10^−1.25
10^−1.00
10^−0.75
10^−0.50
Error