Name: Aneesh Panchal
SR No: 06-18-01-10-12-24-1-25223
Email ID: aneeshp@iisc.ac.in
Date: November 27, 2024

Homework No: Homework 6
Course Code: DS288
Course Name: Numerical Methods
Term: AUG 2024

Solution 1

Solving Initial Value Problem of the form,

$$y'(t) = \frac{dy}{dt} = f(t,y) = \frac{1}{t^2} - \frac{y}{t} - y^2 \qquad \text{on interval, } 1 \le t \le 2 \qquad \text{with initial condition, } y(1) = -1 \tag{1}$$

Exact Solution of the problem is given by, $y(t) = -1/t$

Euler Method

Numerical Solution to the IVP of the form (1) (when domain is divided in n sub-intervals) is given by,

$$w_{i+1} = w_i + hf(t_i, w_i) \qquad \text{where, } w_0 = y(t_0) = y_0 \qquad \text{and, } h = \frac{t_n - t_0}{n} \tag{2}$$

Analytical Order of Accuracy is $\mathbf{O(h^2)}$.

Modified Euler Method

Numerical Solution to the IVP of the form (1) (when domain is divided in n sub-intervals) is given by,

$$w_{i+1} = w_i + \frac{h}{2}\left(f(t_i, w_i) + f(t_{i+1}, w_i + hf(t_i, w_i))\right) \qquad \text{where, } w_0 = y(t_0) = y_0 \qquad \text{and, } h = \frac{t_n - t_0}{n} \tag{3}$$

Analytical Order of Accuracy is $\mathbf{O(h^3)}$.

Solving Problem (1)

Given values of h are $\Delta t = 0.1\,(2^{-n})$ for $n = 0, 1, 2, 3$ and 4.

h or, Δt	$y(2)$ using Euler	$y(2)$ using Modified Euler	Exact	Abs Error Euler	Abs Error Modified Euler
0.1	-0.443138	-0.495556	-0.5	0.05686122	0.004443497
0.05	-0.471219	-0.498855	-0.5	0.028780301	0.001144365
0.025	-0.485515	-0.499710	-0.5	0.014484155	0.000289731
0.0125	-0.492733	-0.499927	-0.5	0.007266528	7.28505e-05
0.00625	-0.496360	-0.499981	-0.5	0.003639504	1.82624e-05

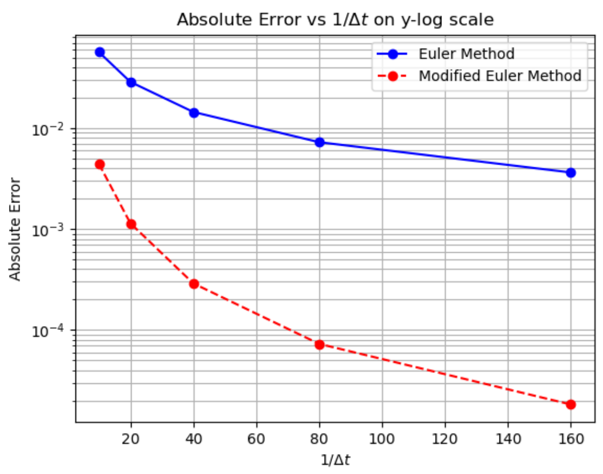


Figure 1: Absolute Error vs $1/\Delta t$ on **y-log scale** (keeping x axis simple)

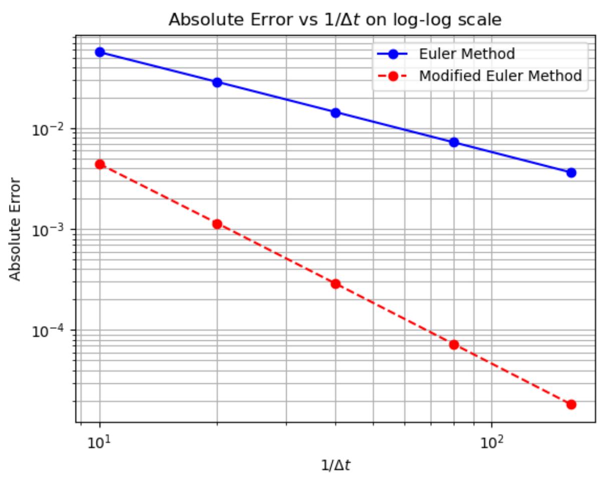


Figure 2: Absolute Error vs $1/\Delta t$ on **log-log scale** (log scaling on both axes)

Name:	Aneesh Panchal	**Homework No:**	Homework 6
SR No:	06-18-01-10-12-24-1-25223	**Course Code:**	DS288
Email ID:	aneeshp@iisc.ac.in	**Course Name:**	Numerical Methods
Date:	November 27, 2024	**Term:**	AUG 2024

Comparison of Errors: From Table we can see, for every value of h or, Δt, Absolute error for using Euler Method turns out to be larger than the Absolute error using Modified Euler Method. It can also be seen that,

$$\mathbf{O}(\textbf{Absolute Error for Modified Euler Method}) = \mathbf{O}(\mathbf{h} \times (\textbf{Absolute Error for Euler Method}))$$

which is true because, Euler Method accuracy is $\mathbf{O(h^2)}$ and for Modified Euler Method it is $\mathbf{O(h^3)}$.

Accuracy Trend Analysis: From Fig. 1 we can see that Modified Euler Method converges faster than Euler Method which agrees with our analytical knowledge of order of accuracy (for convergence, $O(h^4)$ converges faster than $O(h^3)$).
From Fig. 2 we can see that on log-log scale Modified Euler Method's slope (Analytically, it is 4) is higher than the slope of Euler Method (Analytically, it is 3) which is true.

Solution 2

Given system of equations forms 2 species competition model for species 1 and 2 with population at any time t are $N_1(t)$ and $N_2(t)$ and is given by,

$$\frac{dN_1(t)}{dt} = N_1(t)(A_1 - B_1N_1(t) - C_1N_2(t)); \qquad \frac{dN_2(t)}{dt} = N_2(t)(A_2 - B_2N_2(t) - C_2N_1(t)) \tag{4}$$

4^{th} **order Runge-Kutta method**

Numerical Solution to the IVP of the form (1) (when domain is divided in n sub-intervals) is given by,

$$w_{i+1} = w_i + \frac{h}{6}(k_1 + 2(k_2 + k_3) + k_4) \qquad \text{where, } w_0 = y(t_0) = y_0 \qquad \text{and, } h = \frac{t_n - t_0}{n} \tag{5}$$

$$k_1 = f(t_i, w_i) \qquad k_2 = f\left(t_i + \frac{h}{2}, w_i + \frac{k_1}{2}\right) \qquad k_3 = f\left(t_i + \frac{h}{2}, w_i + \frac{k_2}{2}\right) \qquad k_4 = f(t_i + h, w_i + k_3)$$

Analytical Order of Accuracy is $\mathbf{O(h^4)}$.

Empirical Experiment

Procedure popularly known as, **Double Mesh Technique** is used to carry out empirical experiment. In this procedure mesh is doubled at every iteration and when the error starts increasing or showing negligible performance then algorithm terminated. Algorithm adopted (Double Mesh Technique) for experiment is given as follows,

Algorithm 1 Double Mesh Technique

```
1: Input:
2:     $\Delta t = h_0 = b - a$
3:     $N_1(0), N_2(0)$
4: Output: $\Delta t = h_{m-1}$
5:
6: $h_1 = h_0/2$
7: $m = 1$
8: while True do
9:     $h_{m+1} = h_m/2$
10:    Compute $N_1^{h_m}(10), N_2^{h_{m+1}}(10)$ using RK4 method
11:    $Error_{old} = \|N_1^{h_m}(10) - N_1^{h_{m-1}}(10)\|_2$
12:    $Error_{new} = \|N_1^{h_{m+1}}(10) - N_1^{h_m}(10)\|_2$
13:    if $Error_{new} \geq Error_{old}$ then
14:        return $h_m$
15:    end if
16:    $m = m + 1$
17: end while
```

Hence, found time-step size, $\Delta t = h = 0.001953125$.

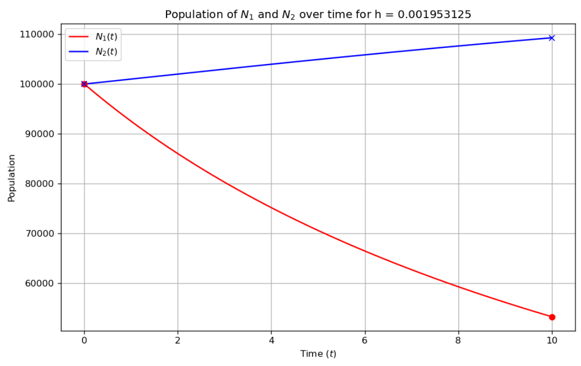


Figure 3: $N_1(t)$ and $N_2(t)$ plots for $\Delta t = h_0 = 0.001953125$

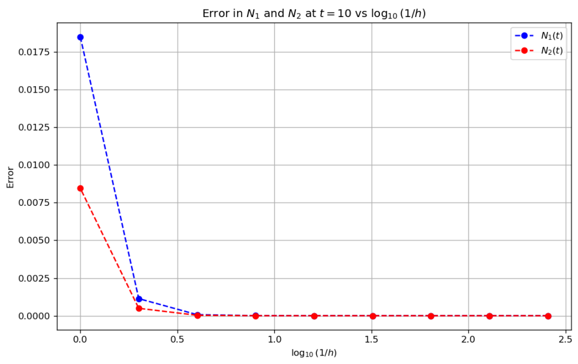


Figure 4: Absolute error for $N_1(10)$ and $N_2(10)$ for $\Delta t = 2^n h_0$ vs $\log_{10}(1/\Delta t)$

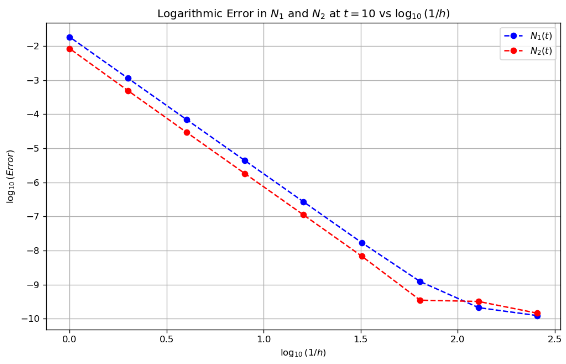


Figure 5: $\log_{10}$(Absolute error) for $N_1(10)$ and $N_2(10)$ for $\Delta t = 2^n h_0$ vs $\log_{10}(1/\Delta t)$

$\mathbf{\Delta t = h}$	$\mathbf{N_1(10)}$	$\mathbf{N_2(10)}$	**Absolute Error** $N_1(10)$	**Absolute Error** $N_2(10)$
0.001953125	53317.793617246	109284.010840000	-	-
0.00390625	53317.793617246	109284.010840000	2.319887434e-15	1.331568554e-15
0.0078125	53317.793617246	109284.010840000	3.957455035e-15	2.929450819e-15
0.015625	53317.793617247	109284.010840000	2.306241038e-14	3.195764529e-15
0.03125	53317.793617263	109284.010840000	3.200080020e-13	6.258372204e-14
0.0625	53317.793617517	109284.010840097	5.097202085e-12	1.027438296e-12
0.125	53317.793621604	109284.010841801	8.174300778e-11	1.661584504e-11
0.25	53317.793687380	109284.010869581	1.315395553e-09	2.708148120e-10
0.5	53317.794751717	109284.011331219	2.127753661e-08	4.495019244e-09
1.0	53317.812122643	109284.019293308	3.470773348e-07	7.735187350e-08

Convergence Analysis:
Value of time-step $\Delta t = h_0$ obtained from empirical experiment is $\mathbf{0.001953125}$.
For we have plotted graphs for $\Delta t = 2^n h_0$ which is equivalent to $2h_0, 4h_0, 8h_0$ and so on. We can clearly see that when Δt becomes large enough we can choose any 2 points and find the slope which turns out to be nearly, 4.

For example,
we choose for $N_1(t)$, $(1.50515, -7.76797)$ and $(0.0, -1.7327)$ which gives slope $=$ $\mathbf{4.0097465}$ (analytically, it must be 4)
we choose $N_2(t)$, $(1.50515, -8.16498)$ and $(0.0, -2.07297)$ which gives slope $=$ $\mathbf{4.047444}$ (analytically, it must be 4)
Hence, our observations and experiment agree with what we assume analytically i.e. order of accuracy of RK4 method must be $O(h^4)$ (which can be seen from above).

NOTE: Irregularity at very low level of Δt is due to the reason that our base solution also comprise of the error of the same order due to which it is difficult to analyze the behaviour of the solution at same order of accuracy.

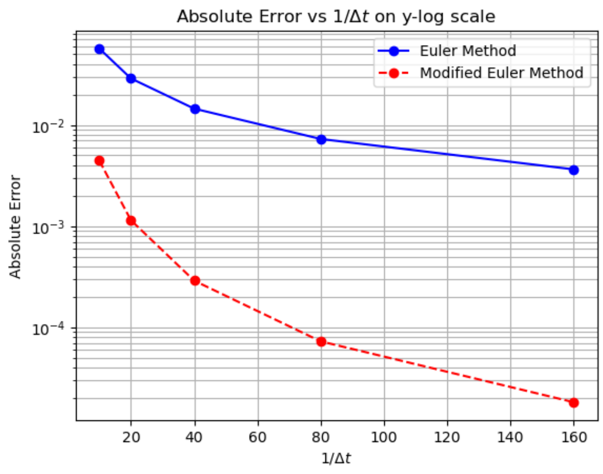
1 Appendix: Assignment 6 Programming

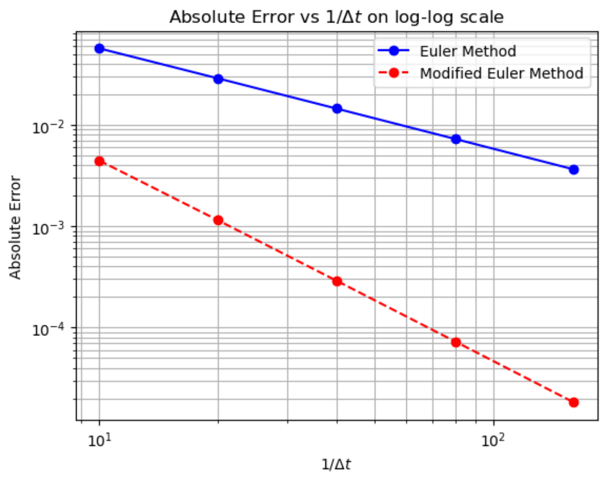
1.1 Ques 1

[1]:
```python
import numpy as np
import math
import matplotlib.pyplot as plt

t0 = 1
tn = 2
y0 = -1
h = []
n = []
for i in range(5):
    h.append(0.1*(2**(-i)))
    n.append(int((tn - t0)/h[i]))

def f(t,y):
    return (1/(t*t)) - (y/t) - (y*y)

def fexact(t):
    return (-1/t)

def euler(t0, tn, h, n, y0):
    y = [y0]
    for i in range(n):
        ti = t0 + i*h
        fi = f(ti, y[i])
        y.append(y[i] + h*fi)
    return y

def modifiedeuler(t0, tn, h, n, y0):
    y = [y0]
    for i in range(n):
        ti = t0 + i*h
        fi = f(ti, y[i])
        y.append(y[i] + (h/2)*(fi + f(ti + h, y[i] + h*fi)))
    return y

feuler = []
fmodeuler = []
for i in range(len(h)):
    feuler.append(euler(t0, tn, h[i], n[i], y0))
    fmodeuler.append(modifiedeuler(t0, tn, h[i], n[i], y0))
ftrue = []
for j in range(len(n)):
    for i in range(n[j]+1):
        ftrue.append(fexact(t0 + i*h[j]))

for i in range(len(h)):
    print("For h = ", h[i])
    print("Euler Method: ", feuler[i][n[i]])
    print("Modified Euler Method: ", fmodeuler[i][n[i]])
    print("Exact: ", ftrue[len(ftrue)-1])
    print("\n")

error_euler = []
for i in range(len(h)):
    error_euler.append(abs(ftrue[len(ftrue)-1] - feuler[i][n[i]]))
    print("Error for h = ", h[i], "is ", error_euler[i])
print("\n")
error_modeuler = []
for i in range(len(h)):
    error_modeuler.append(abs(ftrue[len(ftrue)-1] - fmodeuler[i][n[i]]))
    print("Error for h = ", h[i], "is ", error_modeuler[i])

invh = []
for i in range(len(h)):
    invh.append(1/h[i])

plt.plot(invh, error_euler, 'bo-')
plt.plot(invh, error_modeuler, 'ro--')
plt.legend(['Euler Method', 'Modified Euler Method'])
plt.xlabel(r'$1/\Delta t$')
plt.ylabel('Absolute Error')
plt.title(r'Absolute Error vs $1/\Delta t$ on y-log scale')
plt.yscale('log')
plt.grid(True, which="both", ls="-")
plt.savefig('errorvsinvh.png')
plt.show()

plt.plot(invh, error_euler, 'bo-')
plt.plot(invh, error_modeuler, 'ro--')
plt.legend(['Euler Method', 'Modified Euler Method'])
plt.xlabel(r'$1/\Delta t$')
plt.ylabel('Absolute Error')
plt.title(r'Absolute Error vs $1/\Delta t$ on log-log scale')
plt.xscale('log')
plt.yscale('log')
plt.grid(True, which="both", ls="-")
plt.savefig('errorvsinvhloglog.png')
plt.show()
```

```
For h =  0.1
Euler Method:  -0.4431387797315269
Modified Euler Method:  -0.49555650257013534
Exact:  -0.5


For h =  0.05
Euler Method:  -0.4712196988327847
Modified Euler Method:  -0.4988556346008795
Exact:  -0.5


For h =  0.025
Euler Method:  -0.4855158447937394
Modified Euler Method:  -0.49971026858569617
```

```
Exact:  -0.5


For h =  0.0125
Euler Method:  -0.49273347246793286
Modified Euler Method:  -0.49992714948285133
Exact:  -0.5


For h =  0.00625
Euler Method:  -0.4963604959538005
Modified Euler Method:  -0.49998173758752584
Exact:  -0.5


Error for h =  0.1 is  0.05686122026847312
Error for h =  0.05 is  0.028780301167215305
Error for h =  0.025 is  0.014484155206260618
Error for h =  0.0125 is  0.007266527532067135
Error for h =  0.00625 is  0.0036395040461995043


Error for h =  0.1 is  0.004443497429864662
Error for h =  0.05 is  0.001144365399120495
Error for h =  0.025 is  0.00028973141430382876
Error for h =  0.0125 is  7.285051714867041e-05
Error for h =  0.00625 is  1.8262412474157053e-05
```

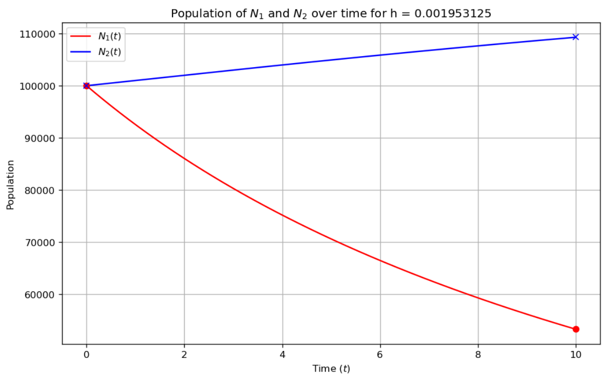




1.2 Ques 2

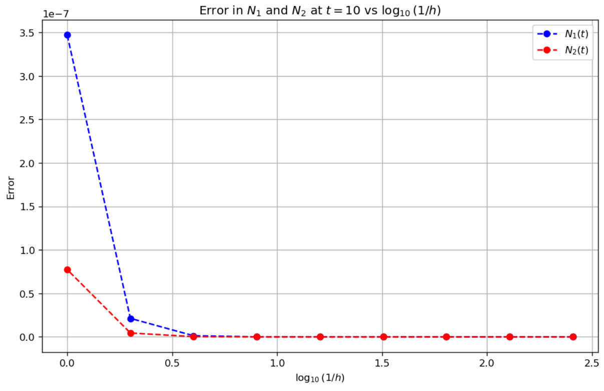
[2]:
```
def model(t, N):
    return np.array([0.1*N[0] - (8e-7)*N[0]**2 - (1e-6)*N[0]*N[1], 0.1*N[1] -␣
↪(8e-7)*N[1]**2 - (1e-7)*N[1]*N[0]])

def RK4(f, N0, t0, tf, h):
    t = np.arange(t0, tf+h, h)
    n = len(t)
    N = np.zeros((len(N0), n))
    N[:,0] = N0
    for i in range(n-1):
        k1 = h*f(t[i], N[:,i]);
        k2 = h*f(t[i] + h/2, N[:,i] + k1/2)
        k3 = h*f(t[i] + h/2, N[:,i] + k2/2)
        k4 = h*f(t[i] + h, N[:,i] + k3)
        k = (k1 + 2*(k2 + k3) + k4)/6
        N[:,i+1] = N[:,i] + k;
    return N, t

N0 = np.array([1e5, 1e5])
h = 1
t0 = 0
tf = 10

# Approximation for value of h
Nuse, tuse = RK4(model, N0, t0, tf, h)
Nbetter, tbetter = RK4(model, N0, t0, tf, h/2)
olderr = np.linalg.norm(Nuse[:,-1] - Nbetter[:,-1])
err = np.linalg.norm(Nuse[:,-1] - Nbetter[:,-1])
while True:
    Nuse, tuse = RK4(model, N0, t0, tf, h)
    Nbetter, tbetter = RK4(model, N0, t0, tf, h/2)
    err = np.linalg.norm(Nuse[:,-1] - Nbetter[:,-1])
    if err > olderr:
        break
    h = h/2
    olderr = err
print("h value to be independent: ", h)

# True solution and Plot
N, t = RK4(model, N0, t0, tf, h)
plt.figure(figsize=(10, 6), dpi=120)
plt.plot(t, N[0,:], "r-", label=r"$N_1(t)$")
plt.plot(t, N[1,:], "b-", label=r"$N_2(t)$")

# plot 1st and last points
plt.plot(t[0], N[0,0], "ro")
plt.plot(t[0], N[1,0], "bx")
plt.plot(t[-1], N[0,-1], "ro")
plt.plot(t[-1], N[1,-1], "bx")

plt.title(r"Population of $N_1$ and $N_2$ over time for " + f"h = {h}")
plt.xlabel(r"Time ($t$)")
plt.ylabel("Population")
plt.legend()
plt.grid()
plt.savefig(f'RK4.png')
plt.show()

# Error Analysis
factor = [2, 4, 8, 16]
for i in range(len(factor)):
    errN1 = [0]
    errN2 = [0]
    hv = [h]
    iter = 0
    print("h = ", hv[iter])
    print("N1(10) = ", N[0,-1])
    print("N2(10) = ", N[1,-1])
    print("\n")

    while hv[iter]*factor[i] <= 1:
        hv.append(hv[iter]*factor[i])
        iter = iter + 1
        Nv, tv = RK4(model, N0, t0, tf, hv[iter])
        errN1.append((abs(Nv[0,-1] - N[0,-1]))/(abs(N[0,-1])))
        errN2.append((abs(Nv[1,-1] - N[1,-1]))/(abs(N[1,-1])))
        print("h = ", hv[iter])
        print("N1(10) = ", Nv[0,-1])
        print("N2(10) = ", Nv[1,-1])
        print("err N1(10) = ", errN1[iter])
        print("err N2(10) = ", errN2[iter])
        print("\n")

    # Plots for Error vs log_{10}(1/h)
    invh = []
    for j in range(1,len(hv)):
        invh.append(math.log10(1/hv[j]))
    plt.figure(figsize=(10, 6), dpi=120)
    plt.plot(invh, errN1[1:], 'bo--')
    plt.plot(invh, errN2[1:], 'ro--')
    plt.legend([r"$N_1(t)$", r"$N_2(t)$"])
    plt.xlabel(r"$\log_{10}(1/h)$")
    plt.ylabel("Error")
    plt.title(r"Error in $N_1$ and $N_2$ at $t=10$ vs $\log_{10}(1/h)$")
    plt.grid()
    plt.savefig(f'errorvsinvh{factor[i]}.png')
    plt.show()

factor = [2, 4, 8, 16, 32, 64, 128, 256, 512]
hlatest = [h]
errorN1 = [0]
errorN2 = [0]
errorlN1 = [0]
errorlN2 = [0]
for i in factor:
    hlatest.append(hlatest[0]*i)
    Nlatest, tlatest = RK4(model, N0, t0, tf, hlatest[-1])
    errorlN1.append(math.log10(abs(Nlatest[0,-1] - N[0,-1])))
    errorlN2.append(math.log10(abs(Nlatest[1,-1] - N[1,-1])))
    errorN1.append(abs(Nlatest[0,-1] - N[0,-1]))
    errorN2.append(abs(Nlatest[1,-1] - N[1,-1]))
    invh = []
    for j in range(1,len(hlatest)):
        invh.append(math.log10(1/hlatest[j]))

plt.figure(figsize=(10, 6), dpi=120)
plt.plot(invh, errorlN1[1:], 'bo--')
plt.plot(invh, errorlN2[1:], 'ro--')
plt.legend([r"$N_1(t)$", r"$N_2(t)$"])
plt.xlabel(r"$\log_{10}(1/h)$")
plt.ylabel(r"$\log_{10}(Error)$")
plt.title(r"Logarithmic Error in $N_1$ and $N_2$ at $t=10$ vs $\log_{10}(1/h)$")
plt.grid()
plt.savefig(f'logerrorvsinvh_allfactors.png')
plt.show()

plt.figure(figsize=(10, 6), dpi=120)
plt.plot(invh, errorN1[1:], 'bo--')
plt.plot(invh, errorN2[1:], 'ro--')
plt.legend([r"$N_1(t)$", r"$N_2(t)$"])
plt.xlabel(r"$\log_{10}(1/h)$")
plt.ylabel("Error")
plt.title(r"Error in $N_1$ and $N_2$ at $t=10$ vs $\log_{10}(1/h)$")
plt.grid()
plt.savefig(f'errorvsinvh_allfactors.png')
plt.show()
```

```
h value to be independent:  0.001953125
```



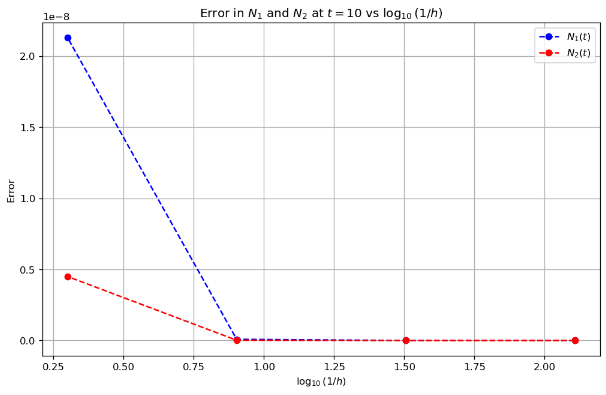
```
h =  0.001953125
N1(10) =  53317.79361724571
N2(10) =  109284.0108399848


h =  0.00390625
N1(10) =  53317.793617245836
N2(10) =  109284.01083998465
err N1(10) =  2.319887434372568e-15
err N2(10) =  1.3315685539464665e-15


h =  0.0078125
N1(10) =  53317.79361724592
N2(10) =  109284.01083998448
err N1(10) =  3.9574550351061455e-15
err N2(10) =  2.929450818682226e-15


h =  0.015625
N1(10) =  53317.79361724694
N2(10) =  109284.01083998515
err N1(10) =  2.3062410376997882e-14
err N2(10) =  3.1957645294715197e-15


h =  0.03125
N1(10) =  53317.793617262774
N2(10) =  109284.01083999164
err N1(10) =  3.200080019766866e-13
err N2(10) =  6.258372203548392e-14


h =  0.0625
N1(10) =  53317.793617517484
N2(10) =  109284.01084009708
err N1(10) =  5.0972020852167155e-12
err N2(10) =  1.0274382962250936e-12
```

```
h =  0.125
N1(10) =  53317.79362160407
N2(10) =  109284.01084180064
err N1(10) =  8.174300778168849e-11
err N2(10) =  1.6615845043565585e-11


h =  0.25
N1(10) =  53317.7936873797
N2(10) =  109284.01086958052
err N1(10) =  1.3153955531725214e-09
err N2(10) =  2.7081481197219854e-10


h =  0.5
N1(10) =  53317.79475171702
N2(10) =  109284.01133121853
err N1(10) =  2.1277536610916016e-08
err N2(10) =  4.49501924353509e-09


h =  1.0
N1(10) =  53317.812122643416
N2(10) =  109284.01929330778
err N1(10) =  3.4707733475794565e-07
err N2(10) =  7.735187349892723e-08
```



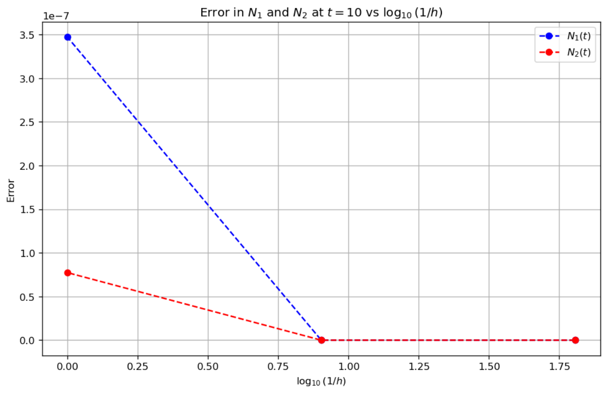
```
h =  0.001953125
N1(10) =  53317.79361724571
N2(10) =  109284.0108399848


h =  0.0078125
N1(10) =  53317.79361724592
```

```
N2(10) =  109284.01083998448
err N1(10) =  3.9574550351061455e-15
err N2(10) =  2.929450818682226e-15


h =  0.03125
N1(10) =  53317.793617262774
N2(10) =  109284.01083999164
err N1(10) =  3.200080019766866e-13
err N2(10) =  6.258372203548392e-14


h =  0.125
N1(10) =  53317.79362160407
N2(10) =  109284.01084180064
err N1(10) =  8.17430077816849e-11
err N2(10) =  1.6615845043565585e-11


h =  0.5
N1(10) =  53317.79475171702
N2(10) =  109284.01133121853
err N1(10) =  2.1277536610916016e-08
err N2(10) =  4.49501924353509e-09
```

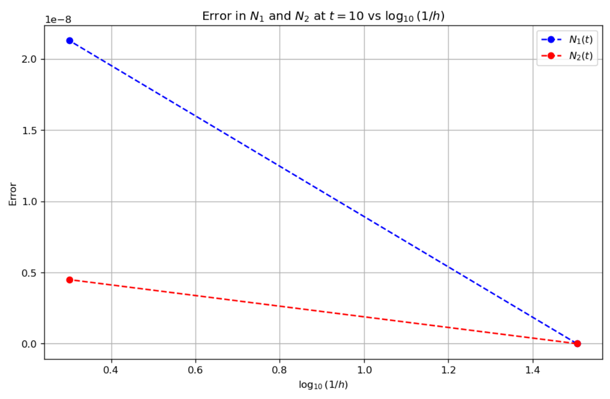


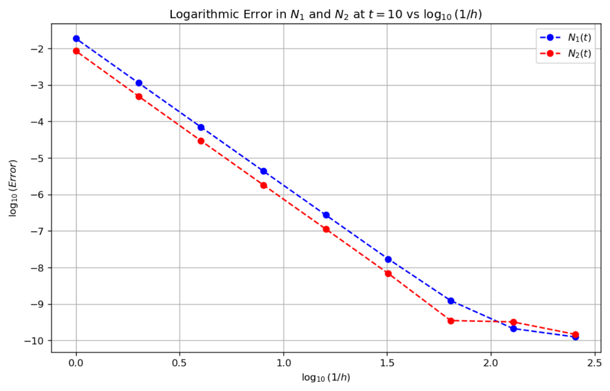
```
h =  0.001953125
N1(10) =  53317.79361724571
N2(10) =  109284.0108399848


h =  0.015625
N1(10) =  53317.79361724694
N2(10) =  109284.01083998515
err N1(10) =  2.3062410376997882e-14
err N2(10) =  3.1957645294715197e-15
```

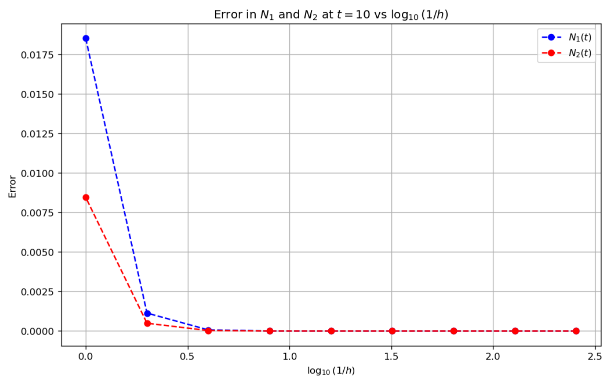
```
h =  0.125
N1(10) =  53317.79362160407
N2(10) =  109284.01084180064
err N1(10) =  8.17430077816849e-11
err N2(10) =  1.6615845043565585e-11


h =  1.0
N1(10) =  53317.812122643416
N2(10) =  109284.01929330778
err N1(10) =  3.4707733475794565e-07
err N2(10) =  7.735187349892723e-08
```



```
h =  0.001953125
N1(10) =  53317.79361724571
N2(10) =  109284.0108399848


h =  0.03125
N1(10) =  53317.793617262774
N2(10) =  109284.01083999164
err N1(10) =  3.200080019766866e-13
err N2(10) =  6.258372203548392e-14


h =  0.5
N1(10) =  53317.79475171702
N2(10) =  109284.01133121853
err N1(10) =  2.1277536610916016e-08
err N2(10) =  4.49501924353509e-09
```







1.3 Equillibrium Point Analysis

[3]:
```
a1 = 0.1
b1 = 8e-7
c1 = 1e-6
a2 = 0.1
```

```python
b2 = 8e-7
c2 = 1e-7

def analysis(a1, b1, c1, a2, b2, c2, x0, y0):
    a11 = a1 - 2*b1*x0 - c1*y0
    a12 = -c1*x0
    a21 = -c2*y0
    a22 = a2 - 2*b2*y0 - c2*x0
    eigval, eigvect = np.linalg.eig(np.array([[a11, a12], [a21, a22]]))
    print("N1: ", x0)
    print("N2: ", y0)
    print("EigenValues: ", eigval)
    if eigval[0] < 0 and eigval[1] < 0:
        print("Stable Point")
    else:
        print("Unstable Point")

x0 = [0, 0, a1/b1, (c1*a2 - b2*a1)/(c1*c2 - b1*b2)]
y0 = [0, a2/b2, 0, (c2*a1 - b1*a2)/(c1*c2 - b1*b2)]

for i in range(len(x0)):
    print(f"Case {i+1}:")
    analysis(a1, b1, c1, a2, b2, c2, x0[i], y0[i])
    print("\n")
```

```
Case 1:
N1:  0
N2:  0
EigenValues:  [0.1 0.1]
Unstable Point


Case 2:
N1:  0
N2:  125000.00000000001
EigenValues:  [-0.1   -0.025]
Stable Point


Case 3:
N1:  125000.00000000001
N2:  0
EigenValues:  [-0.1     0.0875]
Unstable Point


Case 4:
N1:  -37037.03703703703
N2:  129629.62962962966
EigenValues:  [ 0.02592593 -0.1       ]
Unstable Point
```