

Chap 5: Initial Problems for ODE

Due on May, 2022

Yuchen Ge

1. Introduction

The chapter mainly deals with the problem of initial problems for ODE.

First we claim that all function and variable names are self-clear. And a/b is interpreted to be section a question b, e.g. 5.4/7.

2. Application of Euler's Method (5.2/7)

2.1 Program Running and Data Analysis

Question 5.2/7 a. requires the best approximation of the initial problem. First we calculate the real value

$$y(5) = e^{-5} + 5 = 5.006737946999086.$$

Then we apply Euler's Method with parameters $h = 0.2, 0.1, 0.05$. We run the program as follows. (Meanwhile we display the running time.)

```

1 import math
2 Real=math.exp(-5)+5
3 # euler's method
4 def f(t,y):
5     return -y*t+1
6 def euler(a,b,N,alpha):
7     h=(b-a)/N
8     t=a
9     w=alpha
10    for i in range(1,N+1):
11        w=w+h*f(t,w)
12        t=t+h
13    return w
14 print('The result applying Euler method with h=0.2 is',euler(0,5,25,1),'with error',Real-euler(0,5,25,1))
15 print('The result applying Euler method with h=0.1 is',euler(0,5,50,1),'with error',Real-euler(0,5,50,1))
16 print('The result applying Euler method with h=0.05 is',euler(0,5,100,1),'with error',Real-euler(0,5,100,1))
17 print('The result applying Euler method with h=0.0014 is',euler(0,5,3535,1),'with error',Real-euler(0,5,3535,1))
18

```

The result applying Euler method with h=0.2 is 5.00377893186297 with error 0.0029608538127887432
The result applying Euler method with h=0.1 is 5.0051537752073285 with error 0.0015841717917858655
The result applying Euler method with h=0.05 is 5.005920529228334 with error 0.0008174177787516612
The result applying Euler method with h=0.0014 is 5.0067141488252835 with error 2.3806173882843418e-05
[Finished in 50ms]

We write down the results in the table below, where we **reserve seven significant digits** for the result.

Table 1: Results of 5.2/7

h	Approximation of $y(5)$	Error
0.2	5.003778	0.002960
0.1	5.005154	0.001584
0.05	5.005921	0.000817

For question 5.2/7 b., we apply formula (5.14)

$$h = \sqrt{\frac{2\delta}{M}} = \left(\frac{2 \cdot 10^{-6}}{1}\right)^{1/2} = 0.001414213562373095 \approx 0.0014142.$$

Then we run the program as above and we have the approximation of $y(5) \approx 5.00671414$.

3. Application of Runge-Kutta's Method (5.4/4c,5c,6,7,8,9,11,12)

3.1 Program Running and Data Analysis of 5.4/4c,5c

Applying Modified Euler's method for $y' = -(y+1)(y+3)$ with initial value $y(0) = -2$ (5.4/4c), we run the program as follows: (Meanwhile we also display the running time as a byproduct.)

```

1 import math
2 # Modified Euler's method
3 def f(t,y):
4     return -(y+1)*(y+3)
5 def y(t):
6     return -3+2*(1-math.exp(-2*t))-1
7 def Modified_Euler(a,b,N,alpha):
8     h=(b-a)/N
9     t=a
10    w=a
11    k=0
12    for i in range(1,N+1):
13        w=h/2*f(t,w)+f(t+h,w+h*f(t,w))
14        t=a+ih
15        k=k+0.2
16        print("%f & %f & %f\\n" % (k,w,y(t)-w) )
17 def interpolation(t,a,b,c,d):
18     return (d-c)/(b-a)*(t-a)+c
19 Modified_Euler(0,2,10,-2)
20 # interpolation approximation
21 y1=interpolation(1,3,1,2,1.4,-1.172218612839859,-1.1200763382151344)
22 y2=interpolation(1.93,1.8,2,-1.057169897186425,-1.0391938189655483)
23 print()
24 print('The result y(1.3) applying Modified_Euler method with h=0.2 is %f with error %f' % (y1,y(1.3)-y1) )
25 print('The result y(1.93) applying Modified_Euler method with h=0.2 is %f with error %f' % (y2,y(1.93)-y2) )

```

0.2 & -1.80400000 & 0.00137532\\n
0.4 & -1.62292059 & 0.00286956\\n
0.6 & -1.46724025 & 0.00428981\\n
0.8 & -1.34132022 & 0.00535699\\n
1.0 & -1.24429031 & 0.00588447\\n
1.2 & -1.17221061 & 0.00586522\\n
1.4 & -1.12007633 & 0.00542798\\n
1.6 & -1.08307845 & 0.00474700\\n
1.8 & -1.05716989 & 0.00397590\\n
2.0 & -1.03919382 & 0.00322140\\n

The result y(1.3) applying Modified_Euler method with h=0.2 is -1.14614347 with error 0.00786663
The result y(1.93) applying Modified_Euler method with h=0.2 is -1.04548544 with error 0.00421885
[Finished in 103ms]

And therefore we give a table containing results and errors: (reserve nine significant digits)

Table 2: Results of 5.4/4c (Modified Euler's method)

t	Approximation of $y(t)$	Error
0.2	-1.80400000	0.00137532
0.4	-1.62292059	0.00286956
0.6	-1.46724025	0.00428981
0.8	-1.34132022	0.00535699
1.0	-1.24429031	0.00588447
1.2	-1.17221061	0.00586522
1.4	-1.12007633	0.00542798
1.6	-1.08307845	0.00474700
1.8	-1.05716989	0.00397590
2.0	-1.03919382	0.00322140

And we find that the error term grows until $t = 1.2$, which is contractdict to the earlier claim for Euler's method that the error monotonically increases with t increases. The reason behind this is that Euler's method requires only the term $f(t_i, w_i)$, while the other methods like the Modified Euler Method requires terms below which is much more complicated,

$$\frac{h}{2}[f(t_i, w_i) + f(t_{i+1}, w_i + hf(t_i, w_i))].$$

Then for 5.4/5c we use the approximation value of $y(1.2)$ and $y(1.4)$ to linear interpolate $y(1.3)$, which leads to

$$\tilde{y}(1.3) = \frac{\tilde{y}(1.4) - \tilde{y}(1.2)}{1.4 - 1.2}(1.3 - 1.2) + \tilde{y}(1.2) = -1.1461434715270966$$

where we denote the approximation value of $y(1.3)$ by $\tilde{y}(1.3)$. Similarly, we have $\tilde{y}(1.93) = -1.0454854440789212$. And thus gather the information for the table (reserve nine significant digits):

Table 3: Results of 5.4/5c (Modified Euler’s method)

t	Approximation of $y(t)$	Error
1.3	-1.14614347	0.00786663
1.93	-1.04548544	0.00421885

3.2 Program Running and Data Analysis of 5.4/6,7

Similarly we apply Heun method and run the program as follows:

```

1 import math
2 # Heun's method
3 def f(t,y):
4     return -(y-1)*(y+3)
5 def f(t):
6     return -3+2*(1-math.exp(-2*t))**(-1)
7 def Heun(t,w,alpha):
8     h=(b-a)/N
9     t=a
10    w=alpha
11    k=0
12    for i in range(1,N+1):
13        wwh=4*f(t,w)-3*f(t+2/3*h,w2-3/5*h*(t,w))
14        tw=a+h
15        kw=k+2
16        print("%1r % %8f % %8f" % (k,w,abs(w-y(t))))
17    def interpolation(t,w,d):
18        return (d-c)/(b-a)*(t-a)+c
19    Heun(0.2,10,-2)
20    # interpolation approximation
21    y1=interpolation(1.3,1.2,1.4,-1.1699932388972874,-1.1183617793296322)
22    y2=interpolation(1.93,1.62,-1.0562505811390515,-1.8391938196565483)
23    print()
24    print("The result y(1.3) applying Heun method with h=0.2 is %8f with error %8f" % (y1,(y1-y1_1) ) )
25    print("The result y(1.93) applying Heun method with h=0.2 is %8f with error %8f" % (y2,(y2-y2_1) ) )

```

And therefore we can give a table containing results and errors, where we reserve nine significant digits for the results.

Table 4: Results of 5.4/6 (Heun method)

t	Approximation of $y(t)$	Error
0.2	-1.80266667	0.00004199
0.4	-1.62050375	0.00045271
0.6	-1.46425480	0.00130437
0.8	-1.33829391	0.00233068
1.0	-1.24158658	0.00318073
1.2	-1.16999323	0.00364784
1.4	-1.11836178	0.00371343
1.6	-1.08180537	0.00347392
1.8	-1.05625058	0.00305659
2.0	-1.03854252	0.00257010

Also we find that the error term grows until $t = 1.6$, which is contractdict to the earlier claim for Euler's method that the error monotonically increases with t increases. The reason behind this is the same.

Then for 5.4/7, similarly, we can interpolate in the same way as above and thus gives the table (reserve nine significant digits):

Table 5: Results of 5.4/7 (Heun method)

t	Approximation of $y(t)$	Error
1.3	-1.14417751	0.00590066
1.93	-1.04516369	0.00389709

3.3 Program Running and Data Analysis of 5.4/8,9

Similarly we apply Midpoint method and run the program as follows:

```

1 from numpy import *
2 from matplotlib import pyplot as plt
3 # Runge-Kutta method
4 def f(t, y):
5     return -0.5*(1+y**2)
6
7 def Runge_Kutta(f, t0, tf, h):
8     n = (tf-t0)/h
9     t = t0
10    y = y0
11    w = zeros(n+1)
12    for i in range(n):
13        k1 = f(t, y)
14        k2 = f(t+h/2, y+k1*h/2)
15        k3 = f(t+h, y+k1*h)
16        k4 = f(t+h, y+k1*h)
17        w[i+1] = y + h*(k1 + 2*k2 + 2*k3 + k4)/6
18        t = t+h
19        y = w[i+1]
20    return w
21
22 # Example
23 f = f
24 t0 = 0
25 tf = 2
26 h = 0.2
27 w = Runge_Kutta(f, t0, tf, h)
28
29 # Plot
30 t = linspace(t0, tf, 10)
31 y = w[t0:t0+h:t0+tf]
32 plt.plot(t, y)
33 plt.xlabel('t')
34 plt.ylabel('y')
35 plt.title('Runge-Kutta method')
36 plt.show()
37
38 # Error
39 error = 0
40 for i in range(n):
41     error = max(error, abs(w[i+1]-w[i]))
42
43 print('The result of Runge-Kutta method with h=0.2 is: %f with error %f' % (w[-1], error))
44
45 # Runge-Kutta method with h=0.1
46 h = 0.1
47 w = Runge_Kutta(f, t0, tf, h)
48 error = 0
49 for i in range(n):
50     error = max(error, abs(w[i+1]-w[i]))
51
52 print('The result of Runge-Kutta method with h=0.1 is: %f with error %f' % (w[-1], error))
53
54 # Runge-Kutta method with h=0.05
55 h = 0.05
56 w = Runge_Kutta(f, t0, tf, h)
57 error = 0
58 for i in range(n):
59     error = max(error, abs(w[i+1]-w[i]))
60
61 print('The result of Runge-Kutta method with h=0.05 is: %f with error %f' % (w[-1], error))
62
63 # Runge-Kutta method with h=0.025
64 h = 0.025
65 w = Runge_Kutta(f, t0, tf, h)
66 error = 0
67 for i in range(n):
68     error = max(error, abs(w[i+1]-w[i]))
69
70 print('The result of Runge-Kutta method with h=0.025 is: %f with error %f' % (w[-1], error))
71
72 # Runge-Kutta method with h=0.0125
73 h = 0.0125
74 w = Runge_Kutta(f, t0, tf, h)
75 error = 0
76 for i in range(n):
77     error = max(error, abs(w[i+1]-w[i]))
78
79 print('The result of Runge-Kutta method with h=0.0125 is: %f with error %f' % (w[-1], error))
80
81 # Runge-Kutta method with h=0.00625
82 h = 0.00625
83 w = Runge_Kutta(f, t0, tf, h)
84 error = 0
85 for i in range(n):
86     error = max(error, abs(w[i+1]-w[i]))
87
88 print('The result of Runge-Kutta method with h=0.00625 is: %f with error %f' % (w[-1], error))
89
90 # Runge-Kutta method with h=0.003125
91 h = 0.003125
92 w = Runge_Kutta(f, t0, tf, h)
93 error = 0
94 for i in range(n):
95     error = max(error, abs(w[i+1]-w[i]))
96
97 print('The result of Runge-Kutta method with h=0.003125 is: %f with error %f' % (w[-1], error))
98
99 # Runge-Kutta method with h=0.0015625
100 h = 0.0015625
101 w = Runge_Kutta(f, t0, tf, h)
102 error = 0
103 for i in range(n):
104     error = max(error, abs(w[i+1]-w[i]))
105
106 print('The result of Runge-Kutta method with h=0.0015625 is: %f with error %f' % (w[-1], error))
107
108 # Runge-Kutta method with h=0.00078125
109 h = 0.00078125
110 w = Runge_Kutta(f, t0, tf, h)
111 error = 0
112 for i in range(n):
113     error = max(error, abs(w[i+1]-w[i]))
114
115 print('The result of Runge-Kutta method with h=0.00078125 is: %f with error %f' % (w[-1], error))
116
117 # Runge-Kutta method with h=0.000390625
118 h = 0.000390625
119 w = Runge_Kutta(f, t0, tf, h)
120 error = 0
121 for i in range(n):
122     error = max(error, abs(w[i+1]-w[i]))
123
124 print('The result of Runge-Kutta method with h=0.000390625 is: %f with error %f' % (w[-1], error))
125
126 # Runge-Kutta method with h=0.0001953125
127 h = 0.0001953125
128 w = Runge_Kutta(f, t0, tf, h)
129 error = 0
130 for i in range(n):
131     error = max(error, abs(w[i+1]-w[i]))
132
133 print('The result of Runge-Kutta method with h=0.0001953125 is: %f with error %f' % (w[-1], error))
134
135 # Runge-Kutta method with h=0.00009765625
136 h = 0.00009765625
137 w = Runge_Kutta(f, t0, tf, h)
138 error = 0
139 for i in range(n):
140     error = max(error, abs(w[i+1]-w[i]))
141
142 print('The result of Runge-Kutta method with h=0.00009765625 is: %f with error %f' % (w[-1], error))
143
144 # Runge-Kutta method with h=0.000048828125
145 h = 0.000048828125
146 w = Runge_Kutta(f, t0, tf, h)
147 error = 0
148 for i in range(n):
149     error = max(error, abs(w[i+1]-w[i]))
150
151 print('The result of Runge-Kutta method with h=0.000048828125 is: %f with error %f' % (w[-1], error))
152
153 # Runge-Kutta method with h=0.0000244140625
154 h = 0.0000244140625
155 w = Runge_Kutta(f, t0, tf, h)
156 error = 0
157 for i in range(n):
158     error = max(error, abs(w[i+1]-w[i]))
159
160 print('The result of Runge-Kutta method with h=0.0000244140625 is: %f with error %f' % (w[-1], error))
161
162 # Runge-Kutta method with h=0.00001220703125
163 h = 0.00001220703125
164 w = Runge_Kutta(f, t0, tf, h)
165 error = 0
166 for i in range(n):
167     error = max(error, abs(w[i+1]-w[i]))
168
169 print('The result of Runge-Kutta method with h=0.00001220703125 is: %f with error %f' % (w[-1], error))
170
171 # Runge-Kutta method with h=0.000006103515625
172 h = 0.000006103515625
173 w = Runge_Kutta(f, t0, tf, h)
174 error = 0
175 for i in range(n):
176     error = max(error, abs(w[i+1]-w[i]))
177
178 print('The result of Runge-Kutta method with h=0.000006103515625 is: %f with error %f' % (w[-1], error))
179
180 # Runge-Kutta method with h=0.0000030517578125
181 h = 0.0000030517578125
182 w = Runge_Kutta(f, t0, tf, h)
183 error = 0
184 for i in range(n):
185     error = max(error, abs(w[i+1]-w[i]))
186
187 print('The result of Runge-Kutta method with h=0.0000030517578125 is: %f with error %f' % (w[-1], error))
188
189 # Runge-Kutta method with h=0.00000152587890625
190 h = 0.00000152587890625
191 w = Runge_Kutta(f, t0, tf, h)
192 error = 0
193 for i in range(n):
194     error = max(error, abs(w[i+1]-w[i]))
195
196 print('The result of Runge-Kutta method with h=0.00000152587890625 is: %f with error %f' % (w[-1], error))
197
198 # Runge-Kutta method with h=0.000000762939453125
199 h = 0.000000762939453125
200 w = Runge_Kutta(f, t0, tf, h)
201 error = 0
202 for i in range(n):
203     error = max(error, abs(w[i+1]-w[i]))
204
205 print('The result of Runge-Kutta method with h=0.000000762939453125 is: %f with error %f' % (w[-1], error))
206
207 # Runge-Kutta method with h=0.0000003814697265625
208 h = 0.0000003814697265625
209 w = Runge_Kutta(f, t0, tf, h)
210 error = 0
211 for i in range(n):
212     error = max(error, abs(w[i+1]-w[i]))
213
214 print('The result of Runge-Kutta method with h=0.0000003814697265625 is: %f with error %f' % (w[-1], error))
215
216 # Runge-Kutta method with h=0.00000019073486328125
217 h = 0.00000019073486328125
218 w = Runge_Kutta(f, t0, tf, h)
219 error = 0
220 for i in range(n):
221     error = max(error, abs(w[i+1]-w[i]))
222
223 print('The result of Runge-Kutta method with h=0.00000019073486328125 is: %f with error %f' % (w[-1], error))
224
225 # Runge-Kutta method with h=0.000000095367431640625
226 h = 0.000000095367431640625
227 w = Runge_Kutta(f, t0, tf, h)
228 error = 0
229 for i in range(n):
230     error = max(error, abs(w[i+1]-w[i]))
231
232 print('The result of Runge-Kutta method with h=0.000000095367431640625 is: %f with error %f' % (w[-1], error))
233
234 # Runge-Kutta method with h=0.0000000476837158203125
235 h = 0.0000000476837158203125
236 w = Runge_Kutta(f, t0, tf, h)
237 error = 0
238 for i in range(n):
239     error = max(error, abs(w[i+1]-w[i]))
240
241 print('The result of Runge-Kutta method with h=0.0000000476837158203125 is: %f with error %f' % (w[-1], error))
242
243 # Runge-Kutta method with h=0.00000002384185791015625
244 h = 0.00000002384185791015625
245 w = Runge_Kutta(f, t0, tf, h)
246 error = 0
247 for i in range(n):
248     error = max(error, abs(w[i+1]-w[i]))
249
250 print('The result of Runge-Kutta method with h=0.00000002384185791015625 is: %f with error %f' % (w[-1], error))
251
252 # Runge-Kutta method with h=0.000000011920928955078125
253 h = 0.000000011920928955078125
254 w = Runge_Kutta(f, t0, tf, h)
255 error = 0
256 for i in range(n):
257     error = max(error, abs(w[i+1]-w[i]))
258
259 print('The result of Runge-Kutta method with h=0.000000011920928955078125 is: %f with error %f' % (w[-1], error))
260
261 # Runge-Kutta method with h=0.0000000059604644775390625
262 h = 0.0000000059604644775390625
263 w = Runge_Kutta(f, t0, tf, h)
264 error = 0
265 for i in range(n):
266     error = max(error, abs(w[i+1]-w[i]))
267
268 print('The result of Runge-Kutta method with h=0.0000000059604644775390625 is: %f with error %f' % (w[-1], error))
269
270 # Runge-Kutta method with h=0.00000000298023223876953125
271 h = 0.00000000298023223876953125
272 w = Runge_Kutta(f, t0, tf, h)
273 error = 0
274 for i in range(n):
275     error = max(error, abs(w[i+1]-w[i]))
276
277 print('The result of Runge-Kutta method with h=0.00000000298023223876953125 is: %f with error %f' % (w[-1], error))
278
279 # Runge-Kutta method with h=0.000000001490116119384765625
280 h = 0.000000001490116119384765625
281 w = Runge_Kutta(f, t0, tf, h)
282 error = 0
283 for i in range(n):
284     error = max(error, abs(w[i+1]-w[i]))
285
286 print('The result of Runge-Kutta method with h=0.000000001490116119384765625 is: %f with error %f' % (w[-1], error))
287
288 # Runge-Kutta method with h=0.0000000007450580596923828125
289 h = 0.0000000007450580596923828125
290 w = Runge_Kutta(f, t0, tf, h)
291 error = 0
292 for i in range(n):
293     error = max(error, abs(w[i+1]-w[i]))
294
295 print('The result of Runge-Kutta method with h=0.0000000007450580596923828125 is: %f with error %f' % (w[-1], error))
296
297 # Runge-Kutta method with h=0.00000000037252902984619140625
298 h = 0.00000000037252902984619140625
299 w = Runge_Kutta(f, t0, tf, h)
300 error = 0
301 for i in range(n):
302     error = max(error, abs(w[i+1]-w[i]))
303
304 print('The result of Runge-Kutta method with h=0.00000000037252902984619140625 is: %f with error %f' % (w[-1], error))
305
306 # Runge-Kutta method with h=0.000000000186264514923095703125
307 h = 0.000000000186264514923095703125
308 w = Runge_Kutta(f, t0, tf, h)
309 error = 0
310 for i in range(n):
311     error = max(error, abs(w[i+1]-w[i]))
312
313 print('The result of Runge-Kutta method with h=0.000000000186264514923095703125 is: %f with error %f' % (w[-1], error))
314
315 # Runge-Kutta method with h=0.0000000000931322574615478515625
316 h = 0.0000000000931322574615478515625
317 w = Runge_Kutta(f, t0, tf, h)
318 error = 0
319 for i in range(n):
320     error = max(error, abs(w[i+1]-w[i]))
321
322 print('The result of Runge-Kutta method with h=0.0000000000931322574615478515625 is: %f with error %f' % (w[-1], error))
323
324 # Runge-Kutta method with h=0.00000000004656612873077392578125
325 h = 0.00000000004656612873077392578125
326 w = Runge_Kutta(f, t0, tf, h)
327 error = 0
328 for i in range(n):
329     error = max(error, abs(w[i+1]-w[i]))
330
331 print('The result of Runge-Kutta method with h=0.00000000004656612873077392578125 is: %f with error %f' % (w[-1], error))
332
333 # Runge-Kutta method with h=0.000000000023283064365386962890625
334 h = 0.000000000023283064365386962890625
335 w = Runge_Kutta(f, t0, tf, h)
336 error = 0
337 for i in range(n):
338     error = max(error, abs(w[i+1]-w[i]))
339
340 print('The result of Runge-Kutta method with h=0.000000000023283064365386962890625 is: %f with error %f' % (w[-1], error))
341
342 # Runge-Kutta method with h=0.0000000000116415321826934814453125
343 h = 0.0000000000116415321826934814453125
344 w = Runge_Kutta(f, t0, tf, h)
345 error = 0
346 for i in range(n):
347     error = max(error, abs(w[i+1]-w[i]))
348
349 print('The result of Runge-Kutta method with h=0.0000000000116415321826934814453125 is: %f with error %f' % (w[-1], error))
350
351 # Runge-Kutta method with h=0.00000000000582076609134674072265625
352 h = 0.00000000000582076609134674072265625
353 w = Runge_Kutta(f, t0, tf, h)
354 error = 0
355 for i in range(n):
356     error = max(error, abs(w[i+1]-w[i]))
357
358 print('The result of Runge-Kutta method with h=0.00000000000582076609134674072265625 is: %f with error %f' % (w[-1], error))
359
360 # Runge-Kutta method with h=0.00000000000291038304567337036328125
361 h = 0.00000000000291038304567337036328125
362 w = Runge_Kutta(f, t0, tf, h)
363 error = 0
364 for i in range(n):
365     error = max(error, abs(w[i+1]-w[i]))
366
367 print('The result of Runge-Kutta method with h=0.00000000000291038304567337036328125 is: %f with error %f' % (w[-1], error))
368
369 # Runge-Kutta method with h=0.000000000001455191522836685181640625
370 h = 0.000000000001455191522836685181640625
371 w = Runge_Kutta(f, t0, tf, h)
372 error = 0
373 for i in range(n):
374     error = max(error, abs(w[i+1]-w[i]))
375
376 print('The result of Runge-Kutta method with h=0.000000000001455191522836685181640625 is: %f with error %f' % (w[-1], error))
377
378 # Runge-Kutta method with h=0.0000000000007275957614183425908203125
379 h = 0.0000000000007275957614183425908203125
380 w = Runge_Kutta(f, t0, tf, h)
381 error = 0
382 for i in range(n):
383     error = max(error, abs(w[i+1]-w[i]))
384
385 print('The result of Runge-Kutta method with h=0.0000000000007275957614183425908203125 is: %f with error %f' % (w[-1], error))
386
387 # Runge-Kutta method with h=0.00000000000036379788070917129541015625
388 h = 0.00000000000036379788070917129541015625
389 w = Runge_Kutta(f, t0, tf, h)
390 error = 0
391 for i in range(n):
392     error = max(error, abs(w[i+1]-w[i]))
393
394 print('The result of Runge-Kutta method with h=0.00000000000036379788070917129541015625 is: %f with error %f' % (w[-1], error))
395
396 # Runge-Kutta method with h=0.000000000000181898940354585592703125
397 h = 0.000000000000181898940354585592703125
398 w = Runge_Kutta(f, t0, tf, h)
399 error = 0
400 for i in range(n):
401     error = max(error, abs(w[i+1]-w[i]))
402
403 print('The result of Runge-Kutta method with h=0.000000000000181898940354585592703125 is: %f with error %f' % (w[-1], error))
404
405 # Runge-Kutta method with h=0.0000000000000909494701772927963515625
406 h = 0.0000000000000909494701772927963515625
407 w = Runge_Kutta(f, t0, tf, h)
408 error = 0
409 for i in range(n):
410     error = max(error, abs(w[i+1]-w[i]))
411
412 print('The result of Runge-Kutta method with h=0.0000000000000909494701772927963515625 is: %f with error %f' % (w[-1], error))
413
414 # Runge-Kutta method with h=0.0000000000000454747350886488178246875
415 h = 0.0000000000000454747350886488178246875
416 w = Runge_Kutta(f, t0, tf, h)
417 error = 0
418 for i in range(n):
419     error = max(error, abs(w[i+1]-w[i]))
420
421 print('The result of Runge-Kutta method with h=0.0000000000000454747350886488178246875 is: %f with error %f' % (w[-1], error))
422
41 # Runge-Kutta method with h=0.00000000000002273736754432440891234375
42 h = 0.00000000000002273736754432440891234375
43 w = Runge_Kutta(f, t0, tf, h)
44 error = 0
45 for i in range(n):
46     error = max(error, abs(w[i+1]-w[i]))
47
48 print('The result of Runge-Kutta method with h=0.00000000000002273736754432440891234375 is: %f with error %f' % (w[-1], error))
49
50 # Runge-Kutta method with h=0.000000000000011368683772162204456171875
51 h = 0.000000000000011368683772162204456171875
52 w = Runge_Kutta(f, t0, tf, h)
53 error = 0
54 for i in range(n):
55     error = max(error, abs(w[i+1]-w[i]))
56
57 print('The result of Runge-Kutta method with h=0.000000000000011368683772162204456171875 is: %f with error %f' % (w[-1], error))
58
59 # Runge-Kutta method with h=0.0000000000000056843418860811022280859375
60 h = 0.0000000000000056843418860811022280859375
61 w = Runge_Kutta(f, t0, tf, h)
62 error = 0
63 for i in range(n):
64     error = max(error, abs(w[i+1]-w[i]))
65
66 print('The result of Runge-Kutta method with h=0.0000000000000056843418860811022280859375 is: %f with error %f' % (w[-1], error))
67
68 # Runge-Kutta method with h=0.00000000000000284217094304055111404296875
69 h = 0.00000000000000284217094304055111404296875
70 w = Runge_Kutta(f, t0, tf, h)
71 error = 0
72 for i in range(n):
73     error = max(error, abs(w[i+1]-w[i]))
74
75 print('The result of Runge-Kutta method with h=0.00000000000000284217094304055111404296875 is: %f with error %f' % (w[-1], error))
76
77 # Runge-Kutta method with h=0.000000000000001421085471520275557021484375
78 h = 0.000000000000001421085471520275557021484375
79 w = Runge_Kutta(f, t0, tf, h)
80 error = 0
81 for i in range(n):
82     error = max(error, abs(w[i+1]-w[i]))
83
84 print('The result of Runge-Kutta method with h=0.000000000000001421085471520275557021484375 is: %f with error %f' % (w[-1], error))
85
86 # Runge-Kutta method with h=0.0000000000000007105427357601377785107421875
87 h = 0.0000000000000007105427357601377785107421875
88 w = Runge_Kutta(f, t0, tf, h)
89 error = 0
90 for i in range(n):
91     error = max(error, abs(w[i+1]-w[i]))
92
93 print('The result of Runge-Kutta method with h=0.0000000000000007105427357601377785107421875 is: %f with error %f' % (w[-1], error))
94
95 # Runge-Kutta method with h=0.00000000000000035527136788006888925537109375
96 h = 0.00000000000000035527136788006888925537109375
97 w = Runge_Kutta(f, t0, tf, h)
98 error = 0
99 for i in range(n):
100     error = max(error, abs(w[i+1]-w[i]))
101
102 print('The result of Runge-Kutta method with h=0.00000000000000035527136788006888925537109375 is: %f with error %f' % (w[-1], error))
103
104 # Runge-Kutta method with h=0.000000000000000177635683940034444627685546875
105 h = 0.000000000000000177635683940034444627685546875
106 w = Runge_Kutta(f, t0, tf, h)
107 error = 0
108 for i in range(n):
109     error = max(error, abs(w[i+1]-w[i]))
110
111 print('The result of Runge-Kutta method with h=0.000000000000000177635683940034444627685546875 is: %f with error %f' % (w[-1], error))
112
113 # Runge-Kutta method with h=0.0000000000000000888178419700172223138427734375
114 h = 0.0000000000000000888178419700172223138427734375
115 w = Runge_Kutta(f, t0, tf, h)
116 error = 0
117 for i in range(n):
118     error = max(error, abs(w[i+1]-w[i]))
119
120 print('The result of Runge-Kutta method with h=0.0000000000000000888178419700172223138427734375 is: %f with error %f' % (w[-1], error))
121
122 # Runge-Kutta method with h=0.00000000000000004440892098500861115692138671875
123 h = 0.00000000000000004440892098500861115692138671875
124 w = Runge_Kutta(f, t0, tf, h)
125 error = 0
126 for i in range(n):
127     error = max(error, abs(w[i+1]-w[i]))
128
129 print('The result of Runge-Kutta method with h=0.00000000000000004440892098500861115692138671875 is: %f with error %f' % (w[-1], error))
130
131 # Runge-Kutta method with h=0.0000000000000000222044604925043055784606934375
132 h = 0.0000000000000000222044604925043055784606934375
133 w = Runge_Kutta(f, t0, tf, h)
134 error = 0
135 for i in range(n):
136     error = max(error, abs(w[i+1]-w[i]))
137
138 print('The result of Runge-Kutta method with h=0.0000000000000000222044604925043055784606934375 is: %f with error %f' % (w[-1], error))
139
140 # Runge-Kutta method with h=0.000000000000000011102230246252152789230346875
141 h = 0.000000000000000011102230246252152789230346875
142 w = Runge_Kutta(f, t0, tf, h)
143 error = 0
144 for i in range(n):
145     error = max(error, abs(w[i+1]-w[i]))
146
147 print('The result of Runge-Kutta method with h=0.000000000000000011102230246252152789230346875 is: %f with error %f' % (w[-1], error))
148
149 # Runge-Kutta method with h=0.0000000000000000055511151231260763946151734375
150 h = 0.0000000000000000055511151231260763946151734375
151 w = Runge_Kutta(f, t0, tf, h)
152 error = 0
153 for i in range(n):
154     error = max(error, abs(w[i+1]-w[i]))
155
156 print('The result of Runge-Kutta method with h=0.0000000000000000055511151231260763946151734375 is: %f with error %f' % (w[-1], error))
157
158 # Runge-Kutta method with h=0.00000000000000000277555756156303819730758671875
159 h = 0.00000000000000000277555756156303819730758671875
160 w = Runge_Kutta(f, t0, tf, h)
161 error = 0
162 for i in range(n):
163     error = max(error, abs(w[i+1]-w[i]))
164
165 print('The result of Runge-Kutta method with h=0.00000000000000000277555756156303819730758671875 is: %f with error %f' % (w[-1], error))
166
167 # Runge-Kutta method with h=0.000000000000000001387778780781519098653
```

t	Approximation of $y(t)$	Error
1.3	-1.14320704	0.00493020
1.93	-1.04437431	0.00310771

And similarly we apply Runge-Kutta's method method and run the program as follows:

And therefore we give a table containing results and errors (reserve nine significant digits):

t	Approximation of y(t)	Error
0.2	-1.80262739	0.00000271
0.4	-1.62005764	0.00000660
0.6	-1.46296284	0.00001241
0.8	-1.33598238	0.00001915
1.0	-1.23843074	0.00002489
1.2	-1.16637354	0.00002815
1.4	-1.11467694	0.00002859
1.6	-1.07835821	0.00002676
1.8	-1.05321755	0.00002356
2.0	-1.03599222	0.00001980

$$\tilde{y}(1.3) = H_3(1.3) = -1.13830364$$

6

```

1  def hermite(p,x, y, dy):
2      f = 0
3      n = len(x)
4      for i in range(n):
5          la = 1
6          lp = 0
7          for j in range(n):
8              if j != i:
9                  la = la*(p - x[j])/(x[i] - x[j])
10                 lp = lp + 1/(x[i] - x[j])
11             temp1 = 1 - 2 * (p - x[i])*lp
12             temp2 = y[i] * temp1 * la * la
13             temp3 = dy[i] * (p - x[i]) * la * la
14             f = f + temp2 + temp3
15     return f

```

Similarly, we have $\tilde{y}(1.93) = -1.04128621$. And thus gather the information for the table (reserve nine significant digits)

Table 9: Results of 5.4/12 (Runge-Kutta's method)

t	Approximation of y(t)	Error
1.3	-1.13830364	0.00002680
1.93	-1.04128621	0.00001961

3.5 Conclusion of Above Methods

Generally speaking, by comparing the results we see that:

1. the Runge-Kutta method gives the best approximation but it has the most calculations.
2. Heun method gives good approximations near the initial values but gives no approximations further.
3. Midpoint/Modified-Euler method gives good approximations everywhere and has mild calculations.
4. Euler's Method has the smallest calculations but sacrifices its accuracy.

Turn to the next page for section 4.

4.1 Program Running and Data Analysis of 5.6/3c

[illegible]

Table 10: Results of 5.6/3c (Adams-Bashforth method)

t	Real value	2-step	3-step	4-step	5-step
0.1	-1.90033201	-1.90033209	-1.90033209	-1.90033209	-1.90033209
0.2	-1.80262468	-1.80182214	-1.80262486	-1.80262486	-1.80262486
0.3	-1.70868739	-1.70721663	-1.70876712	-1.70868768	-1.70868768
0.4	-1.62005104	-1.61811122	-1.62024327	-1.62008995	-1.62005146
0.5	-1.53788284	-1.53570097	-1.53819884	-1.53793718	-1.53786761
0.6	-1.46295043	-1.46074506	-1.46337907	-1.46300534	-1.46292261
0.7	-1.39563222	-1.39358576	-1.39614605	-1.39567144	-1.39559281
0.8	-1.33596323	-1.33420670	-1.33652621	-1.33597925	-1.33592042
0.9	-1.28370213	-1.28231190	-1.28427720	-1.28369238	-1.28365952
1.0	-1.23840584	-1.23740929	-1.23896052	-1.23837336	-1.23836930
1.1	-1.19950098	-1.19888717	-1.20001057	-1.19945122	-1.19947117
1.2	-1.16634539	-1.16607721	-1.16679398	-1.16628513	-1.16632443
1.3	-1.13827684	-1.13830220	-1.13865663	-1.13821228	-1.13826257
1.4	-1.11464835	-1.11490931	-1.11495817	-1.11458465	-1.11464097
1.5	-1.09485175	-1.09529098	-1.09509518	-1.09479250	-1.09484815
1.6	-1.07833145	-1.07889647	-1.07851510	-1.07827891	-1.07833179
1.7	-1.06459093	-1.06523630	-1.06472296	-1.06454612	-1.06459229
1.8	-1.05319399	-1.05388219	-1.05328300	-1.05315708	-1.05319726
1.9	-1.04376254	-1.04446387	-1.04381686	-1.04373311	-1.04376513

8

4.2 Program Running and Data Analysis of 5.6/5,6

We will approximate the solutions to 5.6/3c by applying **Adams Fourth-Order Predictor-Corrector Method**. We run the program as below:

In which we change the algorithm (iterate the formula in the algorithm) so that it can make the corrector be iterated for p iterations. ($p=2,3,4$)

And we shall gather the results in a table as below

Table 11: Results of 5.6/5,6 (Adams Fourth-Order Predictor-Corrector Method)

t	Real value	1-iteration	2-iteration	3-iteration	4-iteration
0.0	-2.00000000	-2.00000000	-2.00000000	-2.00000000	-2.00000000
0.5	-1.53788284	-1.53787884	-1.53787967	-1.53787964	-1.53787964
1.0	-1.23840584	-1.23841344	-1.23841175	-1.23841185	-1.23841184
1.5	-1.09485175	-1.09486090	-1.09486040	-1.09486043	-1.09486043
2.0	-1.03597242	-1.03597573	-1.03597587	-1.03597586	-1.03597586

and with the errors also calculated in the table below.

Table 12: Results of 5.6/5,6 (Adams Fourth-Order Predictor-Corrector Method)

t	Errors of 1-iteration	Errors of 2-iteration	Errors of 3-iteration	Errors of 4-iteration
0.0	0.00000000	0.00000000	0.00000000	0.00000000
0.5	0.00000400	0.00000317	0.00000320	0.00000320
1.0	0.00000760	0.00000591	0.00000600	0.00000600
1.5	0.00000915	0.00000865	0.00000869	0.00000868
2.0	0.00000331	0.00000345	0.00000344	0.00000344

From the above table 12 we find that for almost all values of t , the best approximation is given by 2-iteration. So the best p is 2.

5 Application Runge-Kutta for Systems Algorithms (5.9/2c)

5.1 Program Running and Data Analysis of 5.9/2c Runge-Kutta for Systems Algorithms

We will approximate the solutions to 5.9/2c by applying **Runge-Kutta Method for Systems of Differential Equations**. First observe in chapter 5.9 we can only deal with the numerical approximation of the m-th order system in the form of (5.44). But in 5.9/2c, we will approximate the solutions of

$$y''' + 2y'' - y' - 2y = e^t$$

s.t. the initial conditions: $y(0) = 1, y'(0) = 2, y''(0) = 0$.

As a result of the above argument we can't directly apply the algorithm. But if we first magically suppose

$$y_1 := y \quad y_2 := y' \quad y_3 := y''$$

Then the form of 5.9/2c will be transformed to

$$\begin{cases} y_1' = y_2 \\ y_2' = y_3 \\ y_3' = 2y_1 + y_2 - 2y_3 + e^t \end{cases} \quad (1)$$

s.t. the initial conditions: $y_1(0) = 1, y_2(0) = 2, y_3(0) = 0$.

First and foremost, we run the program as below (Meanwhile we also display the running time as a byproduct.)

```

1 import math
2 import sys
3 from math import exp
4 from math import sqrt
5 # Adams Fourth-Order Predictor-Corrector method
6 def f(t, y):
7     y1, y2, y3 = y
8     return [y2, y3, 2*y1 + y2 - 2*y3 + exp(t)]
9
10 def AdamsFourthOrder(f, t0, tf, h):
11     n = (tf - t0) / h
12     t = t0
13     y = [1, 2, 0]
14     y1, y2, y3 = y
15     for i in range(n):
16         k1 = f(t, y)
17         k2 = f(t + h/4, y + h/4*k1)
18         k3 = f(t + h/2, y + h/2*k1)
19         k4 = f(t + 3*h/4, y + 3*h/4*k1)
20         y = y + h*(k1 + 3*k2 + 3*k3 + k4)/16
21         t = t + h
22         y1, y2, y3 = y
23     return y1, y2, y3
24
25 # Runge-Kutta method
26 def RK4(f, t0, tf, h):
27     n = (tf - t0) / h
28     t = t0
29     y = [1, 2, 0]
30     y1, y2, y3 = y
31     for i in range(n):
32         k1 = f(t, y)
33         k2 = f(t + h/2, y + h/2*k1)
34         k3 = f(t + h/2, y + h/2*k2)
35         k4 = f(t + h, y + h*k3)
36         y = y + h*(k1 + 2*k2 + 2*k3 + k4)/6
37         t = t + h
38         y1, y2, y3 = y
39     return y1, y2, y3
40
41 # Main program
42 t0 = 0
43 tf = 1
44 h = 0.2
45 y1, y2, y3 = AdamsFourthOrder(f, t0, tf, h)
46 print("Adams Fourth-Order [t, y1, y2, y3]:")
47 for i in range(n+1):
48     t = t0 + i*h
49     y1, y2, y3 = RK4(f, t0, t, h)
50     print(t, y1, y2, y3)
51
52 # Execution time
53 start = time.time()
54 AdamsFourthOrder(f, t0, tf, h)
55 end = time.time()
56 print("Execution time: %.2f seconds" % (end - start))

```

We see that approximated values of $y(t)$, $y'(t)$ and $y''(t)$ are all showed in the display table separated by & and \\ finished in 2.0s. And the errors represent the difference between real values of $y(t)$ and approximation values of $y(t)$ in a step of $h = 0.2$ as requested by the question text.

And therefore we shall gather the results into the table below which can represent the results more clearly, where we **reserve nine significant digits**.

Observe that the error accumulates with t grows as usual. But as a windfall we can approximate the solutions of $y'(t)$ and $y''(t)$. Still the details are gathered in the table below.

Table 13: Results of 5.9/2c (Adams Fourth-Order Predictor-Corrector Method)

t	Real value	Approximation of $y(t)$	Error	Approximation of $y'(t)$	Approximation of $y''(t)$
0.0	1.00000000	1.00000000	0.00000000	2.00000000	0.00000000
0.2	1.40637383	1.40633678	0.00003705	2.09439538	0.91961556
0.4	1.84923495	1.84918146	0.00005350	2.36188926	1.74722932
0.6	2.36197037	2.36190903	0.00006134	2.79290064	2.56757686
0.8	2.97762424	2.97755643	0.00006781	3.39310710	3.45012070
1.0	3.73170445	3.73162695	0.00007749	4.18124911	4.45721868
1.2	4.66469806	4.66460440	0.00009366	5.18835555	5.65033178
1.4	5.82454694	5.82442783	0.00011912	6.45808521	7.09509448
1.6	7.26928830	7.26913151	0.00015679	8.04801932	8.86584062
1.8	9.07004289	9.06983275	0.00021014	10.03184298	11.05003348
2.0	11.31452924	11.31424573	0.00028351	12.50243368	13.75296416
2.2	14.11129305	14.11091055	0.00038250	15.57594327	17.10304019
2.4	17.59486416	17.59434982	0.00051435	19.39702265	21.25797699
2.6	21.93209017	21.93140177	0.00068840	24.14540203	26.41222100
2.8	27.32994449	27.32902779	0.00091670	30.04410928	32.80597197
3.0	34.04517155	34.04395688	0.00121468	37.36968748	40.73623289

6. Code Appendix

6.1 Euler's Method

```

1  import math
2  Real=math.exp(-5)+5
3  # euler's method
4  def f(t,y):
5      return -y+t+1
6  def euler(a,b,N,alpha):
7      h=(b-a)/N
8      t=a
9      w=alpha
10     for i in range(1,N+1):
11         w=w+h*f(t,w)
12         t=a+i*h
13     return w
14     print('The result applying Euler method with h=0.2 is',euler(0,5,25,1),'with ...
15         error',Real-euler(0,5,25,1))
16     print('The result applying Euler method with h=0.1 is',euler(0,5,50,1),'with ...
17         error',Real-euler(0,5,50,1))
18     print('The result applying Euler method with h=0.05 is',euler(0,5,100,1),'with ...
19         error',Real-euler(0,5,100,1))
20     print('The result applying Euler method with h=0.0014 is',euler(0,5,3535,1),'with ...
21         error',Real-euler(0,5,3535,1))

```

6.2 Modified Euler's Method

```

1  import math
2  # Modified_Euler's method

```

```

3  def f(t,y):
4      return -(y+1)*(y+3)
5  def y(t):
6      return -3+2*(1+math.exp(-2*t))**(-1)
7  def Modified_Eular(a,b,N,alpha):
8      h=(b-a)/N
9      t=a
10     w=alpha
11     k=0
12     for i in range(1,N+1):
13         w=w+h/2*(f(t,w)+f(t+h,w+h*f(t,w)))
14         t=a+i*h
15         k=k+0.2
16         print("%.1f & %.8f & %.8f\\\\" % (k,w,y(t)-w) )
17 def interpolation(t,a,b,c,d):
18     return (d-c)/(b-a)*(t-a)+c
19 Modified_Eular(0,2,10,-2)
20 # interpolation approximation
21 y1=interpolation(1.3,1.2,1.4,-1.172210612839059,-1.1200763302151344)
22 y2=interpolation(1.93,1.8,2,-1.0571698907180425,-1.0391938189655483)
23 print()
24 print('The result y(1.3) applying Modified_Eular method with h=0.2 is %.8f with error %.8f' ...
25       %(y1,y(1.3)-y1) )
26 print('The result y(1.93) applying Modified_Eular method with h=0.2 is %.8f with error %.8f' ...
27       %(y2,y(1.93)-y2) )

```

6.3 Modified Euler's Method

```

1  import math
2  # Heun's method
3  def f(t,y):
4      return -(y+1)*(y+3)
5  def y(t):
6      return -3+2*(1+math.exp(-2*t))**(-1)
7  def Heun(a,b,N,alpha):
8      h=(b-a)/N
9      t=a
10     w=alpha
11     k=0
12     for i in range(1,N+1):
13         w=w+h/4*(f(t,w)+3*f(t+2/3*h,w+2/3*h*f(t,w)))
14         t=a+i*h
15         k=k+0.2
16         print("%.1f & %.8f & %.8f\\\\" % (k,w,abs(w-y(t))) )
17 def interpolation(t,a,b,c,d):
18     return (d-c)/(b-a)*(t-a)+c
19 Heun(0,2,10,-2)
20 # interpolation approximation
21 y1=interpolation(1.3,1.2,1.4,-1.1699932308872074,-1.1183617793290332)
22 y2=interpolation(1.93,1.8,2,-1.0562505811390515,-1.0391938189655483)
23 print()
24 print('The result y(1.3) applying Heun method with h=0.2 is %.8f with error %.8f' ...
25       %(y1,y(1.3)-y1) )
26 print('The result y(1.93) applying Heun method with h=0.2 is %.8f with error %.8f' ...
27       %(y2,y(1.93)-y2) )

```

6.4 Heun's Method

```

1  import math
2  # Mid_Point method
3  def f(t,y):
4      return -(y+1)*(y+3)
5  def y(t):
6      return -3+2*(1+math.exp(-2*t))**(-1)
7  def Mid_Point(a,b,N,alpha):
8      h=(b-a)/N
9      t=a
10     w=alpha
11     k=0
12     for i in range(1,N+1):
13         w=w+h*f(t+h/2,w+h/2*f(t,w))
14         t=a+i*h
15         k=k+0.2
16         print("%.1f & %.8f & %.8f\\\\" % (k,w,abs(w-y(t))) )
17 def interpolation(t,a,b,c,d):
18     return (d-c)/(b-a)*(t-a)+c
19 Mid_Point(0,2,10,-2)
20 # interpolation approximation
21 y1=interpolation(1.3,1.2,1.4,-1.1688975879735313,-1.1175164873711958)
22 y2=interpolation(1.93,1.8,2,-1.0557987215668139,-1.038222697151529)
23 print()
24 print('The result y(1.3) applying Mid_Point method with h=0.2 is %.8f with error %.8f' ...
25       %(y1,y(1.3)-y1) )
26 print('The result y(1.93) applying Mid_Point method with h=0.2 is %.8f with error %.8f' ...
27       %(y2,y(1.93)-y2) )

```

6.5 Midpoint Method

```

1  import math
2  # Mid_Point method
3  def f(t,y):
4      return -(y+1)*(y+3)
5  def y(t):
6      return -3+2*(1+math.exp(-2*t))**(-1)
7  def Mid_Point(a,b,N,alpha):
8      h=(b-a)/N
9      t=a
10     w=alpha
11     k=0
12     for i in range(1,N+1):
13         w=w+h*f(t+h/2,w+h/2*f(t,w))
14         t=a+i*h
15         k=k+0.2
16         print("%.1f & %.8f & %.8f\\\\" % (k,w,abs(w-y(t))) )
17 def interpolation(t,a,b,c,d):
18     return (d-c)/(b-a)*(t-a)+c
19 Mid_Point(0,2,10,-2)
20 # interpolation approximation
21 y1=interpolation(1.3,1.2,1.4,-1.1688975879735313,-1.1175164873711958)
22 y2=interpolation(1.93,1.8,2,-1.0557987215668139,-1.038222697151529)
23 print()
24 print('The result y(1.3) applying Mid_Point method with h=0.2 is %.8f with error %.8f' ...
25       %(y1,y(1.3)-y1) )
26 print('The result y(1.93) applying Mid_Point method with h=0.2 is %.8f with error %.8f' ...
27       %(y2,y(1.93)-y2) )

```

6.6 Runge-Kutta's Method

```

1  import math
2  import numpy as np
3  from sympy import *
4  from matplotlib import pyplot as plt
5  # Runge-Kutta's method
6  def f(t,y):
7      return -(y+1)*(y+3)
8  def y(t):
9      return -3+2*(1+math.exp(-2*t))**(-1)
10 def Runge_Kutta(a,b,N,alpha):
11     h=(b-a)/N
12     t=a
13     w=alpha
14     k=0
15     for i in range(1,N+1):
16         K1=h*f(t,w)
17         K2=h*f(t+h/2,w+K1/2)
18         K3=h*f(t+h/2,w+K2/2)
19         K4=h*f(t+h,w+K3)
20         w=w+(K1+2*K2+2*K3+K4)/6
21         t=t+h
22         k=k+0.2
23         print("%.1f & %.8f & %.8f\\\\" % (k,w,abs(w-y(t))) )
24 def hermite(p,x, y, dy):
25     f = 0
26     n = len(x)
27     for i in range(n):
28         la = 1
29         lp = 0
30         for j in range(n):
31             if j != i:
32                 la = la*(p - x[j])/(x[i] - x[j])
33                 lp = lp + 1/(x[i] - x[j])
34         temp1 = 1 - 2 * (p - x[i])*lp
35         temp2 = y[i] * temp1 * la * la
36         temp3 = dy[i] * (p - x[i]) * la * la
37         f = f + temp2 + temp3
38     return f
39 Runge_Kutta(0,2,20,-2)
40 # interpolation approximation
41 y1=hermite(1.3,[1.2,1.4],[-1.16637354,-1.11467694],[f(1.2,-1.16637354),f(1.4,-1.11467694)])
42 y2=hermite(1.93,[1.8,2],[-1.05321755,-1.03599222],[f(1.2,-1.05321755),f(1.4,-1.03599222)])
43 print()
44 print('The result y(1.3) applying Runge_Kutta method with h=0.2 is %.8f with error %.8f' ...
45       % (y1,y(1.3)-y1) )
46 print('The result y(1.93) applying Runge_Kutta method with h=0.2 is %.8f with error %.8f' ...
47       % (y2,y(1.93)-y2) )ult y(1.93) applying ...
48       Mid_Point method with h=0.2 is %.8f with ...
49       error %.8f' % (y2,y(1.93)-y2) )

```

6.7 Adams Bashforth's method

```

1  import math
2  import numpy as np
3  from sympy import *

```

```

4  from matplotlib import pyplot as plt
5  # Adams_Bashforth's method
6  def f(t,y):
7      return -(y+1)*(y+3)
8  def y(t):
9      return -3+2*(1+math.exp(-2*t))**(-1)
10 def Adams_Bashforth(a,b,N,alpha):
11     w = [[0.5]*(N+10) for a in range(6)]
12     w=np.mat(w)
13     h=(b-a)/N
14     t=a
15     w[2,0]=alpha
16     w[2,1]=-1.90033209
17     w[3,0]=alpha
18     w[3,1]=-1.90033209
19     w[3,2]=-1.80262486
20     w[4,0]=alpha
21     w[4,1]=-1.90033209
22     w[4,2]=-1.80262486
23     w[4,3]=-1.70868768
24     w[5,0]=alpha
25     w[5,1]=-1.90033209
26     w[5,2]=-1.80262486
27     w[5,3]=-1.70868768
28     w[5,4]=-1.62005146
29     k=0
30     for i in range(1,N):
31         w[2,i+1]=w[2,i]+h/2*(3*f(t,w[2,i])-f(t-h,w[2,i-1]))
32         w[3,i+2]=w[3,i+1]+h/12*(23*f(t,w[3,i+1])-16*f(t-h,w[3,i])+5*f(t-2*h,w[3,i-1]))
33         w[4,i+3]=w[4,i+2]+h/24*(55*f(t,w[4,i+2])-59*f(t-h,w[4,i+1])
34             +37*f(t-2*h,w[4,i])-9*f(t-3*h,w[4,i-1]))
35         w[5,i+4]=w[5,i+3]+h/720*(1901*f(t,w[5,i+3])-2774*f(t-h,w[5,i+2])
36             +2616*f(t-2*h,w[5,i+1])-1274*f(t-3*h,w[5,i])+251*f(t-4*h,w[5,i-1]))
37         t=a+i*h
38         k=k+0.1
39         print("%.1f & %.8f & %.8f & %.8f & %.8f & %.8f\\\\" ...
                                                                %(k,y(k),w[2,i],w[3,i],w[4,i],w[5,i]) ...
                                                                )
40 Adams_Bashforth(0,2,20,-2)

```

6.8 Runge-Kutta Method for Systems of Differential Equations

```

1  import math
2  import numpy as np
3  from sympy import *
4  from matplotlib import pyplot as plt
5  # initialization
6  w=[0.5,0.5,0.5,0.5]
7  k=[[0.5]*5 for i in '12345']
8  k=np.matrix(k)
9  def y(t):
10     return 43/36*math.exp(t)+1/4*math.exp(-t)-4/9*math.exp(-2*t)+1/6*t*math.exp(t)
11 def f1(t,w1,w2,w3):
12     return w2
13 def f2(t,w1,w2,w3):
14     return w3
15 def f3(t,w1,w2,w3):
16     return 2*w1+w2-2*w3+math.exp(t)
17 # Runge-Kutta Method for Systems of Differential Equations

```

```

18 def Runge_Kutta_Systems(a,b,N,alpha):
19     h=(b-a)/N
20     t=a
21     for j in range(1,4):
22         w[j]=alpha[j]
23         print("%.1f & %.8f & %.8f & %.8f & %.8f & %.8f\\\\" ...
                                                    %(t,y(t),w[1],abs(y(t)-w[1]),w[2],w[3]) ...
                                                    )
24     for i in range(1,N+1):
25         # 1
26         k[1,1]=h*f1(t,w[1],w[2],w[3])
27         k[1,2]=h*f2(t,w[1],w[2],w[3])
28         k[1,3]=h*f3(t,w[1],w[2],w[3])
29         # 2
30         k[2,1]=h*f1(t+h/2,w[1]+k[1,1]/2,w[2]+k[1,2]/2,w[3]+k[1,3]/2)
31         k[2,2]=h*f2(t+h/2,w[1]+k[1,1]/2,w[2]+k[1,2]/2,w[3]+k[1,3]/2)
32         k[2,3]=h*f3(t+h/2,w[1]+k[1,1]/2,w[2]+k[1,2]/2,w[3]+k[1,3]/2)
33         # 3
34         k[3,1]=h*f1(t+h/2,w[1]+k[2,1]/2,w[2]+k[2,2]/2,w[3]+k[2,3]/2)
35         k[3,2]=h*f2(t+h/2,w[1]+k[2,1]/2,w[2]+k[2,2]/2,w[3]+k[2,3]/2)
36         k[3,3]=h*f3(t+h/2,w[1]+k[2,1]/2,w[2]+k[2,2]/2,w[3]+k[2,3]/2)
37         # 4
38         k[4,1]=h*f1(t+h,w[1]+k[3,1],w[2]+k[3,2],w[3]+k[3,3])
39         k[4,2]=h*f2(t+h,w[1]+k[3,1],w[2]+k[3,2],w[3]+k[3,3])
40         k[4,3]=h*f3(t+h,w[1]+k[3,1],w[2]+k[3,2],w[3]+k[3,3])
41         for j in range(1,4):
42             w[j]=w[j]+(k[1,j]+2*k[2,j]+2*k[3,j]+k[4,j])/6
43         t=a+i*h
44         print("%.1f & %.8f & %.8f & %.8f & %.8f & %.8f\\\\" ...
                                                    %(t,y(t),w[1],abs(y(t)-w[1]),w[2],w[3]) ...
                                                    )
45 Runge_Kutta_Systems(0,3,15,[0,1,2,0])

```