Math 151B Homework 2

Nick Monozon

5.4.3(a)

Consider the initial value problem given by

$$y' = y/t - (y/t)^2,\ 1 \le t \le 2,\ y(1) = 1$$

along with the actual solution $y(t) = t/(1 + \ln t)$. We want to use Modified Euler's Method with a step size of $h = 0.1$ to approximate the solutions. Hence, we will perform $N = (b - a)/h = (2 - 1)/0.1 = 10$ iterations. Using the code for Modified Euler's Method (modified_euler.py), we obtain as output the table below.

i	t_i	w_i	$y(t_i)$	$\lvert y(t_i) - w_i \rvert$
0	1.0	1.0000000	1.0000000	0.0000000
1	1.1	1.0041322	1.0042817	0.0001495
2	1.2	1.0147137	1.0149523	0.0002386
3	1.3	1.0295197	1.0298137	0.0002940
4	1.4	1.0472044	1.0475339	0.0003295
5	1.5	1.0669093	1.0672624	0.0003530
6	1.6	1.0880637	1.0884327	0.0003690
7	1.7	1.1102751	1.1106551	0.0003800
8	1.8	1.1332657	1.1336536	0.0003878
9	1.9	1.1568349	1.1572284	0.0003935
10	2.0	1.1808345	1.1812322	0.0003977

5.4.3(b)

We are given the initial value problem

$$y' = 1 + y/t + (y/t)^2,\ 1 \le t \le 3,\ y(1) = 0$$

and the corresponding actual solution $y(t) = t\tan(\ln t)$. To use Modified Euler's Method to approximate the solutions for a step size of $h = 0.2$, we will do $N = (3 - 1)/0.2 = 10$ iterations. We use the code modified_euler.py and obtain the table below as output.

i	t_i	w_i	$y(t_i)$	$\lvert y(t_i) - w_i \rvert$
0	1.0	0.0000000	0.0000000	0.0000000
1	1.2	0.2194444	0.2212428	0.0017983
2	1.4	0.4850495	0.4896817	0.0046322
3	1.6	0.8040116	0.8127527	0.0087411
4	1.8	1.1848560	1.1994386	0.0145827
5	2.0	1.6384229	1.6612818	0.0228589
6	2.2	2.1788772	2.2135018	0.0346246
7	2.4	2.8250651	2.8765514	0.0514863
8	2.6	3.6025247	3.6784753	0.0759506
9	2.8	4.5466136	4.6586651	0.1120515
10	3.0	5.7075699	5.8741000	0.1665301

5.4.7(a)

We will approximate solutions to the initial value problem from 5.4.3(a) using the Runge-Kutta midpoint method with a step size of $h = 0.1$, performing $N = 10$ iterations as found previously. Using the midpoint method code (midpoint_method.py), we obtain as output the following table.

i	t_i	w_i	$y(t_i)$	$\lvert y(t_i) - w_i \rvert$
0	1.0	1.0000000	1.0000000	0.0000000
1	1.1	1.0045351	1.0042817	0.0002534
2	1.2	1.0153257	1.0149523	0.0003734
3	1.3	1.0302470	1.0298137	0.0004333
4	1.4	1.0479982	1.0475339	0.0004643
5	1.5	1.0677427	1.0672624	0.0004804
6	1.6	1.0889214	1.0884327	0.0004887
7	1.7	1.1111478	1.1106551	0.0004928
8	1.8	1.1341481	1.1336536	0.0004946
9	1.9	1.1577236	1.1572284	0.0004952
10	2.0	1.1817275	1.1812322	0.0004952

5.4.7(b)

We approximate the solutions to the initial value problem in 5.4.3(b) using the Runge-Kutta midpoint method with a step size of $h = 0.2$ for $N = 10$ iterations. Using the code for the Python implementation of the midpoint method (midpoint_method.py), our output is the following table.

i	t_i	w_i	$y(t_i)$	$\lvert y(t_i) - w_i \rvert$
0	1.0	0.0000000	0.0000000	0.0000000
1	1.2	0.2198347	0.2212428	0.0014081
2	1.4	0.4861770	0.4896817	0.0035046
3	1.6	0.8061849	0.8127527	0.0065678
4	1.8	1.1884393	1.1994386	0.0109994
5	2.0	1.6438889	1.6612818	0.0173928
6	2.2	2.1868609	2.2135018	0.0266409
7	2.4	2.8364357	2.8765514	0.0401157
8	2.6	3.6184926	3.6784753	0.0599828
9	2.8	4.5688944	4.6586651	0.0897707
10	3.0	5.7386475	5.8741000	0.1354525

5.4.15(a)

For the initial value problem given in 5.4.3(a), we approximate its solutions using the Runge-Kutta Method of order 4 with a step size of $h = 0.1$, hence performing $N = 10$ iterations. Using the runge_kutta_order_4.py code, we obtain as output the table below.

i	t_i	w_i	$y(t_i)$	$\lvert y(t_i) - w_i \rvert$
0	1.0	1.0000000	1.0000000	0.0000000
1	1.1	1.0042815	1.0042817	0.0000002
2	1.2	1.0149520	1.0149523	0.0000003
3	1.3	1.0298133	1.0298137	0.0000003
4	1.4	1.0475336	1.0475339	0.0000004
5	1.5	1.0672620	1.0672624	0.0000004
6	1.6	1.0884323	1.0884327	0.0000004
7	1.7	1.1106547	1.1106551	0.0000004
8	1.8	1.1336532	1.1336536	0.0000004

9	1.9	1.1572281	1.1572284	0.0000004
10	2.0	1.1812319	1.1812322	0.0000004

5.4.15(b)

We now redo our approximations for the initial value problem in 5.4.3(a) using the Runge-Kutta method of order 4 with a step size of $h = 0.1$ (meaning that we perform $N = 10$ iterations). Then, using the runge_kutta_order_4.py code, we obtain the table below as our code output.

i	t_i	w_i	$y(t_i)$	$\lvert y(t_i) - w_i \rvert$
0	1.0	0.0000000	0.0000000	0.0000000
1	1.2	0.2212457	0.2212428	0.0000029
2	1.4	0.4896842	0.4896817	0.0000025
3	1.6	0.8127522	0.8127527	0.0000006
4	1.8	1.1994320	1.1994386	0.0000066
5	2.0	1.6612651	1.6612818	0.0000166
6	2.2	2.2134693	2.2135018	0.0000325
7	2.4	2.8764941	2.8765514	0.0000573
8	2.6	3.6783790	3.6784753	0.0000963
9	2.8	4.6585063	4.6586651	0.0001588
10	3.0	5.8738386	5.8741000	0.0002614

5.29

Consider the initial value problem

$$y' = -y + t + 1,\ 0 \le t \le 1,\ y(0) = 1.$$

We want to show that for any choice of step size h, Modifier Euler's Method and the Midpoint Method give the same approximations. Recall that Modified Euler's Method can be recursively defined as

$$\begin{aligned} w_0 &= \alpha, \\ w_{i+1} &= w_i + \frac{h}{2}\left(f(t_i, w_i) + f\big(t_{i+1}, w_i + hf(t_i, w_i)\big)\right), \end{aligned}$$

for $i = 0, 1, \ldots, N-1$ some initial condition $y(t_0) = \alpha$. For this IVP, we have that $w_0 = 1$ and the approximation at each step is given by

$$\begin{aligned} w_{i+1} &= w_i + \frac{h}{2}\left(f(t_i, w_i) + f\big(t_{i+1}, w_i + hf(t_i, w_i)\big)\right) \\ &= w_i + \frac{h}{2}\big(-w_i + t_i + 1 - w_i - hf(t_i, w_i) + t_{i+1} + 1\big) \\ &= w_i + \frac{h}{2}\left(-w_i + t_i + 1 - w_i - h(-w_i + t_i + 1) + t_{i+1} + 1\right) \\ &= w_i - \frac{h}{2} w_i + \frac{h}{2} t_i + \frac{h}{2} - \frac{h}{2} w_i + \frac{h^2}{2} w_i - \frac{h^2}{2} t_i - \frac{h^2}{2} + \frac{h}{2} t_{i+1} + \frac{h}{2} \\ &= w_i - \frac{h}{2} w_i + \frac{h}{2} t_i + \frac{h}{2} - \frac{h}{2} w_i + \frac{h^2}{2} w_i - \frac{h^2}{2} t_i - \frac{h^2}{2} + \frac{h}{2}(t_i + h) + \frac{h}{2} \\ &= w_i\left(1 - h + \frac{h^2}{2}\right) + t_i\left(h - \frac{h^2}{2}\right) + h \end{aligned}$$

for $i = 0, 1, \ldots, N-1$. Note that $t_{i+1} = t_i + h$. The Midpoint Method can also be defined recursively

$$\begin{aligned} w_0 &= \alpha, \\ w_{i+1} &= w_i + hf\left(t_i + \frac{h}{2}, w_i + \frac{h}{2} f(t_i, w_i)\right), \end{aligned}$$

for $i = 0, 1, \ldots, N-1$ and an initial condition $y(t_0) = \alpha$. For the IVP above, we have that $w_0 = 1$ and the approximation at each step is given by

$$\begin{aligned}
w_{i+1} &= w_i + hf\left(t_i + \frac{h}{2}, w_i + \frac{h}{2} f(t_i, w_i)\right) \\
&= w_i + h\left(-w_i - \frac{h}{2} f(t_i, w_i) + t_i + \frac{h}{2} + 1\right) \\
&= w_i + h\left(-w_i - \frac{h}{2}(-w_i + t_i + 1) + t_i + \frac{h}{2} + 1\right) \\
&= w_i - hw_i + \frac{h^2}{2} w_i - \frac{h^2}{2} t_i - \frac{h^2}{2} + ht_i + \frac{h^2}{2} + h \\
&= w_i\left(1 - h + \frac{h^2}{2}\right) + t_i\left(h - \frac{h^2}{2}\right) + h
\end{aligned}$$

for $i = 0, 1, \ldots, N-1$. This is exactly the same as Modified Euler's Method for the above problem. Hence, for any choice of h, both methods will give the same approximations for this initial value problem.

5.5.3(a)

Consider the initial value problem given by

$$y' = y/t - (y/t)^2,\ 1 \le t \le 2,\ y(1) = 1$$

along with the actual solution $y(t) = t/(1 + \ln t)$. We will approximate solutions to this initial value problem using the Runge-Kutta-Fehlberg Method. Using the Python code I wrote for this purpose, rkf.py, we obtain as output the table below.

i	t_i	w_i	h_i	$y(t_i)$
0	1.0000000	1.0000000	0.1330513	1.0000000
1	1.1101946	1.0051237	0.1101946	1.0051237
2	1.2191314	1.0175212	0.1089368	1.0175211
3	1.3572694	1.0396749	0.1381381	1.0396749
4	1.5290112	1.0732757	0.1717417	1.0732756
5	1.7470584	1.1213948	0.2180472	1.1213947
6	2.0286416	1.1881702	0.2815832	1.1881700
7	2.3994350	1.2795396	0.3707934	1.2795395
8	2.8985147	1.4041843	0.4990798	1.4041842
9	3.3985147	1.5285639	0.5000000	1.5285638
10	3.8985147	1.6514963	0.5000000	1.6514962
11	4.0000000	1.6762393	0.1014853	1.6762391

5.5.3(b)

We are given the initial value problem

$$y' = 1 + y/t + (y/t)^2,\ 1 \le t \le 3,\ y(1) = 0$$

and the corresponding actual solution $y(t) = t\tan(\ln t)$. We will again use the rkf.py Python code to approximate solutions to this IVP using the Runge-Kutta-Fehlberg Method. Doing this gives as output the table below.

i	t_i	w_i	h_i	$y(t_i)$
0	1.0000000	0.0000000	0.1450265	0.0000000
1	1.1450265	0.1560235	0.1980582	0.1560234
2	1.2822303	0.3254972	0.1372038	0.3254970
3	1.4225752	0.5232644	0.1403449	0.5232641

4	1.5482238	0.7234123	0.1256486	0.7234119
5	1.6655759	0.9320288	0.1173521	0.9320282
6	1.7773654	1.1521480	0.1117896	1.1521473
7	1.8847226	1.3851234	0.1073571	1.3851226
8	1.9881998	1.6316951	0.1034772	1.6316941
9	2.0880997	1.8923303	0.0998999	1.8923290
10	2.1846024	2.1673514	0.0965027	2.1673499
11	2.2778244	2.4569949	0.0932220	2.4569932
12	2.3678493	2.7614423	0.0900249	2.7614402
13	2.4547442	3.0808374	0.0868949	3.0808350
14	2.5385694	3.4152978	0.0838252	3.4152951
15	2.6193834	3.7649220	0.0808140	3.7649189
16	2.6972462	4.1297939	0.0778628	4.1297904
17	2.7722206	4.5099869	0.0749744	4.5099829
18	2.8443728	4.9055656	0.0721522	4.9055611
19	2.9137726	5.3165880	0.0693998	5.3165830
20	2.9804930	5.7431068	0.0667204	5.7431011
21	3.0000000	5.8741059	0.0195070	5.8741000

5.6.3(a)

Consider the initial value problem given by

$$y' = y/t - (y/t)^2,\ 1 \le t \le 2,\ y(1) = 1$$

along with the actual solution $y(t) = t/(1+\ln t)$. Using the ab3.py code to implement the Adams-Bashforth Three-Step Method with a step size of $h = 0.1$ (requiring $N = 10$ iterations) in Python, our output is given the table below.

i	t_i	w_i	$y(t_i)$	$\lvert y(t_i) - w_i\rvert$
0	1.0	1.0000000	1.0000000	0.0000000
1	1.1	1.0042815	1.0042817	0.0000002
2	1.2	1.0149520	1.0149523	0.0000003
3	1.3	1.0293579	1.0298137	0.0004558
4	1.4	1.0468730	1.0475339	0.0006609
5	1.5	1.0664788	1.0672624	0.0007836
6	1.6	1.0875837	1.0884327	0.0008490
7	1.7	1.1097691	1.1106551	0.0008859
8	1.8	1.1327465	1.1336536	0.0009070
9	1.9	1.1563092	1.1572284	0.0009193
10	2.0	1.1803057	1.1812322	0.0009265

Similarly using ab4.py to implement the Adams-Bashforth Four-Step Method with a step size of $h = 0.1$ for $N = 10$ iterations, we obtain the table below as output.

i	t_i	w_i	$y(t_i)$	$\lvert y(t_i) - w_i\rvert$
0	1.0	1.0000000	1.0000000	0.0000000
1	1.1	1.0042815	1.0042817	0.0000002
2	1.2	1.0149520	1.0149523	0.0000003
3	1.3	1.0298133	1.0298137	0.0000003
4	1.4	1.0477278	1.0475339	0.0001939
5	1.5	1.0675362	1.0672624	0.0002739
6	1.6	1.0887567	1.0884327	0.0003240
7	1.7	1.1109994	1.1106551	0.0003443
8	1.8	1.1340093	1.1336536	0.0003558

9	1.9	1.1575899	1.1572284	0.0003614
10	2.0	1.1815967	1.1812322	0.0003645

5.6.3(b)

Consider the initial value problem given by

$$y' = y/t - (y/t)^2,\ 1 \le t \le 2,\ y(1) = 1$$

along with the actual solution $y(t) = t/(1+\ln t)$. Using the Adams-Bashforth Three-Step Method with a step size of $h = 0.1$ (for $N = 10$ iterations), we use the Python code ab3.py to obtain the below output.

i	t_i	w_i	$y(t_i)$	$\lvert y(t_i) - w_i \rvert$
0	1.0	0.0000000	0.0000000	0.0000000
1	1.2	0.2212457	0.2212428	0.0000029
2	1.4	0.4896842	0.4896817	0.0000025
3	1.6	0.8124317	0.8127527	0.0003210
4	1.8	1.1982110	1.1994386	0.0012277
5	2.0	1.6584313	1.6612818	0.0028504
6	2.2	2.2079987	2.2135018	0.0055031
7	2.4	2.8667672	2.8765514	0.0097842
8	2.6	3.6617484	3.6784753	0.0167269
9	2.8	4.6305275	4.6586651	0.0281375
10	3.0	5.8268008	5.8741000	0.0472992

Additionally, we implement the Adams-Bashforth Four-Step Method with a step size of $h = 0.1$ (also for $N = 10$ iterations) in Python (ab4.py), obtaining the following table as our output.

i	t_i	w_i	$y(t_i)$	$\lvert y(t_i) - w_i \rvert$
0	1.0	0.0000000	0.0000000	0.0000000
1	1.2	0.2212457	0.2212428	0.0000029
2	1.4	0.4896842	0.4896817	0.0000025
3	1.6	0.8127522	0.8127527	0.0000006
4	1.8	1.1990422	1.1994386	0.0003964
5	2.0	1.6603060	1.6612818	0.0009758
6	2.2	2.2117448	2.2135018	0.0017570
7	2.4	2.8735320	2.8765514	0.0030194
8	2.6	3.6733266	3.6784753	0.0051487
9	2.8	4.6498937	4.6586651	0.0087714
10	3.0	5.8589944	5.8741000	0.0151056

5.10.1

Proof. Suppose that the hypotheses of Theorem 5.20 are satisfied. Hence, assume that $\phi(t, w, h)$ is continuous and satisfies a Lipschitz condition in w on

$$D = \left\{(t, w, h) : a \le t \le b, -\infty < w < \infty, 0 \le h \le h_0\right\}$$

with Lipschitz constant L. Now, suppose that sequences $\{u_i\}_{i=1}^{N}$ and $\{v_i\}_{i=1}^{N}$ satisfy the difference equation

$$w_{i+1} = w_i + h\phi(t_i, w_i, h).$$

From this assumption, we have that

$$u_{i+1} = u_i + h\phi(t_i, u_i, h) \quad \text{and} \quad v_{i+1} = v_i + h\phi(t_i, v_i, h).$$

It follows that

$$u_{i+1} - v_{i+1} = u_i - v_i + h\left[\phi(t_i, u_i, h) - \phi(t_i, v_i, h)\right]$$

from which we also further obtain

$$|u_{i+1} - v_{i+1}| = \left|u_i - v_i + h\left[\phi(t_i, u_i, h) - \phi(t_i, v_i, h)\right]\right|.$$

By the Triangle Inequality,

$$\begin{aligned} |u_{i+1} - v_{i+1}| &\leq |u_i - v_i| + h\left|\phi(t_i, u_i, h) - \phi(t_i, v_i, h)\right| \\ &\leq |u_i - v_i| + hL|u_i - v_i| \\ &= (1 + hL)|u_i - v_i|. \end{aligned}$$

At this point, we observe that

$$\begin{aligned} |u_{i+1} - v_{i+1}| &\leq (1 + hL)|u_i - v_i| \\ &\leq (1 + hL)^2|u_{i-1} - v_{i-1}| \\ &\leq (1 + hL)^3|u_{i-2} - v_{i-2}| \\ &\vdots \\ &\leq (1 + hL)^{i+1}|u_0 - v_0|. \end{aligned}$$

Since $h > 0$ and $L > 0$, we have that $(1 + hL)^{i+1} > 0$ for all $i \geq 0$. The chain of inequalities established above implies that

$$(1 + hL)|u_i - v_i| \leq (1 + hL)^{i+1}|u_0 - v_0|.$$

Dividing each side by $1 + hL$ gives

$$|u_i - v_i| \leq (1 + hL)^i|u_0 - v_0|.$$

Thus, if we let $K = (1 + hL)^i > 0$, we have that

$$|u_i - v_i| \leq K|u_0 - v_0|$$

for each $i = 1, 2, \ldots, N$. This gives the desired result and we're done. □

modified_euler.py

This code provides my implementation of Modified Euler's Method in Python. As provided, this is the solution for 5.4.3(b).

```python
# Imports
import numpy as np
import pandas as pd
import math

# For more decimal places
pd.set_option("display.precision", 7)

## Arguments

# Function
f = lambda t, y: 1 + y/t + (y/t)**2
# Left endpoint
t_0 = 1
# Right endpoint
t_1 = 3
# Step size
h = 0.2
# Initial condition
alpha = 0

N = int((t_1-t_0)/h)
t = np.arange(t_0, t_1+h, h)
```

```
w = np.zeros(len(t))
w[0] = alpha

# Approximations
for i in range(0, len(t) - 1):
  w[i+1] = w[i] + h/2*(f(t[i], w[i]) +  f(t[i+1], w[i] + h*f(t[i], w[i])))
# Output
df = pd.DataFrame({('t_i'): t, 'w_i': w})
print(df)

## Error determination

# Exact solution
y = lambda t: t*np.tan(np.log(t))
yt = np.zeros(len(t))

# Looping to determine actual values
for i in range(0, len(t)):
  yt[i] = y(t[i])
# Output
df2 = pd.DataFrame({('t_i'): t, 'w_i': w,
                    'y(t_i)': yt, '|y(t_i) - w_i|': abs(w-yt)})
print(df2)
```

midpoint_method.py

The code below gives my implementation of the Midpoint Method in Python. This particular instance provides the output for 5.4.7(b).

```
# Imports
import numpy as np
import pandas as pd
import math

# For more decimal places
pd.set_option("display.precision", 7)

## Arguments

# Function
f = lambda t, y: 1 + y/t + (y/t)**2
# Left endpoint
t_0 = 1
# Right endpoint
t_1 = 3
# Step size
h = 0.2
# Initial condition
alpha = 0

N = int((t_1-t_0)/h)
t = np.arange(t_0, t_1+h, h)
w = np.zeros(len(t))
w[0] = alpha

# Approximations
for i in range(0, len(t) - 1):
  w[i+1] = w[i] + h*f(t[i] + h/2, w[i] + h/2*f(t[i], w[i]))
# Output
df = pd.DataFrame({('t_i'): t, 'w_i': w})
print(df)

## Error determination

# Exact solution
y = lambda t: t*np.tan(np.log(t))
yt = np.zeros(len(t))

# Looping to determine actual values
```

```python
for i in range(0, len(t)):
  yt[i] = y(t[i])
# Output
df2 = pd.DataFrame({('t_i'): t, 'w_i': w,
                    'y(t_i)': yt, '|y(t_i) - w_i|': abs(w-yt)})
print(df2)
```

runge_kutta_order_4.py

The chunk below provides my Python code for the implementation of the Runge-Kutta Method of order 4. In particular, the code below gives the solution for 5.4.15(b).

```python
import numpy as np
import pandas as pd
import math

# For more decimal places
pd.set_option("display.precision", 7)

## Arguments
# Function
f = lambda t, y: 1 + y/t + (y/t)**2
# Left endpoint
t_0 = 1
# Right endpoint
t_1 = 3
# Step size
h = 0.2
# Initial condition
alpha = 0

N = int((t_1-t_0)/h)
t = np.arange(t_0, t_1+h, h)
w = np.zeros(len(t))
w[0] = alpha

# Approximations
for i in range(0, len(t) - 1):
  k1 = h*f(t[i], w[i])
  k2 = h*f(t[i] + h/2, w[i] + 1/2*k1)
  k3 = h*f(t[i] + h/2, w[i] + 1/2*k2)
  k4 = h*f(t[i+1], w[i] + k3)
  w[i+1] = w[i] + 1/6*(k1 + 2*k2 + 2*k3 + k4)
# Output
df = pd.DataFrame({('t_i'): t, 'w_i': w})
print(df)

## Error determination

# Exact solution
y = lambda t: t*np.tan(np.log(t))
yt = np.zeros(len(t))

# Looping to determine actual values
for i in range(0, len(t)):
  yt[i] = y(t[i])
# Output
df2 = pd.DataFrame({('t_i'): t, 'w_i': w,
                    'y(t_i)': yt, '|y(t_i) - w_i|': abs(w-yt)})
print(df2)
```

rkf.py

The code below implements the Runge-Kutta-Fehlberg Method in Python using the pseudocode outlined in Algorithm 5.3 of the textbook. The chunk below in particular provides approximations for 5.5.3(a).

```python
# Imports
import numpy as np
```

```python
import pandas as pd
import math

# For more decimal places
pd.set_option("display.precision", 7)

## Argument specification

# Function
f = lambda t, y: y/t - (y/t)**2
# Left endpoint
a = 1
# Right endpoint
b = 4
# Maximum step size
hmax = 0.5
# Minimum step size
hmin = 0.05
# Tolerance
TOL = pow(10, -6)
# Initial condition
alpha = 1

# Maximum number of steps (unknown a priori)
N_max = int((b-a)/hmin) + 1

# Initialize arrays
t = np.zeros(N_max)
w = np.zeros(N_max)
h_out = np.zeros(1)

# Initial conditions
t[0] = a
w[0] = alpha
h_out[0] = hmax
h = hmax
i = 0
FLAG = 1

# Implementing the RKF algorithm for approximations
while (FLAG == 1):
  K1 = h*f(t[i],w[i])
  K2 = h*f(t[i] + 1/4*h, w[i] + 1/4*K1)
  K3 = h*f(t[i] + 3/8*h, w[i] + 3/32*K1 + 9/32*K2)
  K4 = h*f(t[i] + 12/13*h, w[i] + 1932/2197*K1 - 7200/2197*K2 + 7296/2197*K3)
  K5 = h*f(t[i] + h, w[i] + 439/216*K1 - 8*K2 + 3680/513*K3 - 845/4104*K4)
  K6 = h*f(t[i] + 1/2*h, w[i] - 8/27*K1 + 2*K2 - 3544/2565*K3 + 1859/4104*K4 - 11/40*K5)

  h_out = np.append(h_out, h)
  R = (1/h)*abs(1/360*K1 - 128/4275*K3 - 2197/75240*K4 + 1/50*K5 + 2/55*K6)
  delta = 0.84*pow(TOL/R, 1/4)

  if R <= TOL:
    t[i+1] = t[i] + h
    w[i+1] = w[i] + 25/216*K1 + 1408/2565*K3 + 2197/4104*K4 - 1/5*K5
    i += 1

  delta = 0.84*pow(TOL/R, 1/4)

  if delta <= 0.1:
    h = 0.1*h
  elif delta >= 4:
    h = 4*h
  else:
    h = delta*h

  if h > hmax:
    h = hmax

  if t[i] >= b:
```

```
    FLAG = 0
  elif (t[i] + h) > b:
    h = b - t[i]
  elif h < hmin:
    FLAG = 0

# Posterior update to approximation arrays
t = t[:i+1]
w = w[:i+1]
h_out = h_out[2:]

# Actual solution
y = lambda t: t/(1+np.log(t))
yt = np.zeros(len(t))

# Looping to determine actual values
for idx, val in enumerate(t):
  yt[idx] = y(val)

# Output
df = pd.DataFrame({('t_i'): t, 'w_i': w, 'h_i': h_out, 'y(t_i)': yt})
print(df)
```

ab3.py

The code snippet below is my Python implementation of the Adams-Bashforth Three-Step Method. This instance provides approximations for the first part of 5.6.3(a) in the textbook.

```
# Imports
import numpy as np
import pandas as pd
import math

# For more decimal places
pd.set_option("display.precision", 7)

## Argument specification

# Function
f = lambda t, y: y/t - (y/t)**2
# Left endpoint
a = 1
# Right endpoint
b = 2
# Step size
h = 0.1
# First initial condition
alpha = 1

N = int((b-a/h))
t = np.arange(a, b+h, h)
w = np.zeros(len(t))
w[0] = alpha

# For order 4 Runge-Kutta approximations
rk = np.zeros(3)
rk[0] = alpha

# Runge-Kutta order 4 approximations for alpha1 and alpha2
for i in range(0, 2):
  k1 = h*f(t[i], rk[i])
  k2 = h*f(t[i] + h/2, rk[i] + 1/2*k1)
  k3 = h*f(t[i] + h/2, rk[i] + 1/2*k2)
  k4 = h*f(t[i+1], rk[i] + k3)
  rk[i+1] = rk[i] + 1/6*(k1 + 2*k2 + 2*k3 + k4)

w[0] = alpha
w[1] = rk[1]
w[2] = rk[2]
```

```
# Adam-Bashforth 4-step approximations
for i in range(2, len(t) - 1):
  w[i+1] = w[i] + h/12*(23*f(t[i], w[i]) - 16*f(t[i-1], w[i-1]) + 5*f(t[i-2], w[i-2]))
# Output
df = pd.DataFrame({('t_i'): t, 'w_i': w})
print(df)

# Exact solution
y = lambda t: t/(1+np.log(t))
yt = np.zeros(len(t))

# Looping to determine actual values
for i in range(0, len(t)):
  yt[i] = y(t[i])
# Output
df2 = pd.DataFrame({('t_i'): t, 'w_i': w, 'y(t_i)': yt, '|y(t_i) - w_i|': abs(yt - w)})
print(df2)
```

ab4.py

The code snippet below is my Python implementation of the Adams-Bashforth Four-Step Method. This instance provides approximations for the second part of 5.6.3(a) in the textbook.

```
# Imports
import numpy as np
import pandas as pd
import math

# For more decimal places
pd.set_option("display.precision", 7)

## Argument specification
# Function
f = lambda t, y: y/t - (y/t)**2
# Left endpoint
a = 1
# Right endpoint
b = 2
# Step size
h = 0.1
# First initial condition
alpha = 1

N = int((b-a/h))
t = np.arange(a, b+h, h)
w = np.zeros(len(t))
w[0] = alpha

# For order 4 Runge-Kutta approximations
rk = np.zeros(4)
rk[0] = alpha

# Runge-Kutta order 4 approximations for alpha1, alpha2, alpha3
for i in range(0, 3):
  k1 = h*f(t[i], rk[i])
  k2 = h*f(t[i] + h/2, rk[i] + 1/2*k1)
  k3 = h*f(t[i] + h/2, rk[i] + 1/2*k2)
  k4 = h*f(t[i+1], rk[i] + k3)
  rk[i+1] = rk[i] + 1/6*(k1 + 2*k2 + 2*k3 + k4)

w[0] = alpha
w[1] = rk[1]
w[2] = rk[2]
w[3] = rk[3]

# Adam-Bashforth 4-step approximations
for i in range(3, len(t) - 1):
```

```
  w[i+1] = w[i] + h/24*(55*f(t[i], w[i]) - 59*f(t[i-1], w[i-1]) + 37*f(t[i-2], w[i-2]) - 9*f(t[i
    -3], w[i-3]))
# Output
df = pd.DataFrame({('t_i'): t, 'w_i': w})
print(df)

# Exact solution
y = lambda t: t/(1+np.log(t))
yt = np.zeros(len(t))

# Looping to determine actual values
for i in range(0, len(t)):
  yt[i] = y(t[i])
# Output
df2 = pd.DataFrame({('t_i'): t, 'w_i': w, 'y(t_i)': yt, 'y(t_i) - w_i|': abs(yt - w)})
print(df2)
```