Math 151B Homework 3

Nick Monozon

5.10.5

(a)

Consider the multistep method

$$w_{i+1} = -\frac{3}{2}w_i + 3w_{i-1} - \frac{1}{2}w_{i-2} + 3hf(t_i, w_i)$$

for $i = 2, \ldots, N-1$ and starting values w_0, w_1, w_2. We want to find the local truncation error for this method. Recall that the local truncation error is the amount by which the true solution fails to satisfy the difference equation

$$y(t_{i+1}) = -\frac{3}{2}y(t_i) + 3y(t_{i-1}) - \frac{1}{2}y(t_{i-2}) + 3hf\big(t_i, y(t_i)\big) + \tau_{i+1}(h).$$

We can rearrange to find that

$$\tau_{i+1}(h) = \frac{y(t_{i+1}) + \frac{3}{2}y(t_i) - 3y(t_{i-1}) + \frac{1}{2}y(t_{i-2})}{h} - 3f\big(t_i, y(t_i)\big).$$

Next, we want to find the third degree Taylor expansions of $y(t_{i+1})$, $y(t_{i-1})$, and $y(t_{i-2})$ about t_i. We first find that

$$y(t_{i+1}) = y(t_i) + hy'(t_i) + \frac{h^2}{2}y''(t_i) + \frac{h^3}{6}y'''(t_i) + \frac{h^4}{24}y^{(4)}(\xi_{i+1})$$

for some $\xi_{i+1} \in (t_i, t_{i+1})$. Next, we have that

$$y(t_{i-1}) = y(t_i) - hy'(t_i) + \frac{h^2}{2}y''(t_i) - \frac{h^3}{6}y'''(t_i) + \frac{h^4}{24}y^{(4)}(\xi_{i-1})$$

for some $\xi_{i-1} \in (t_{i-1}, t_i)$. Lastly, we have that

$$y(t_{i-2}) = y(t_i) - 2hy'(t_i) + 2h^2y''(t_i) - \frac{4}{3}h^3y'''(t_i) + \frac{2}{3}h^4y^{(4)}(\xi_{i-2})$$

for some $\xi_{i-2} \in (t_{i-2}, t_i)$. Using these Taylor expansions, we substitute them into the expression $\alpha = y(t_{i+1}) + \frac{3}{2}y(t_i) - 3y(t_{i-1}) + \frac{1}{2}y(t_{i-2})$ to obtain

$$\begin{aligned}
\alpha &= y(t_i) + hy'(t_i) + \frac{h^2}{2}y''(t_i) + \frac{h^3}{6}y'''(t_i) + \frac{h^4}{24}y^{(4)}(\xi_{i+1}) + \frac{3}{2}y(t_i) - 3\Big[y(t_i) - hy'(t_i) + \frac{h^2}{2}y''(t_i) - \frac{h^3}{6}y'''(t_i) + \frac{h^4}{24}y^{(4)}(\xi_{i-1})\Big] \\
&\quad + \frac{1}{2}\Big[y(t_i) - 2hy'(t_i) + 2h^2y''(t_i) - \frac{4}{3}h^3y'''(t_i) + \frac{2}{3}h^4y^{(4)}(\xi_{i-2})\Big] \\
&= \left(1 + \frac{3}{2} + \frac{1}{2} - 3\right)y(t_i) + (1 + 3 - 1)hy'(t_i) + \left(\frac{1}{2} - \frac{3}{2} + 1\right)h^2y''(t_i) + \left(\frac{1}{6} + \frac{1}{2} - \frac{2}{3}\right)h^3y'''(t_i) + \frac{h^4}{24}y^{(4)}(\xi_{i+1}) - \frac{h^4}{8}y^{(4)}(\xi_{i+1}) + \frac{h^4}{3}y^{(4)}(\xi_{i-2}) \\
&= 3hy'(t_i) + \left(\frac{1}{24} - \frac{1}{8} + \frac{1}{3}\right)h^4y^{(4)}(\xi_i) \\
&= 3hy'(t_i) + \frac{1}{4}h^4y^{(4)}(\xi_i)
\end{aligned}$$

for $\xi_i \in (t_{i-2}, t_{i+1})$. We can substitute this back into our expression for $\tau_{i+1}(h)$ to get that

$$\begin{aligned}
\tau_{i+1}(h) &= \frac{3hy'(t_i) + \frac{1}{4}h^4y^{(4)}(\xi_i)}{h} - 3f\big(t_i, y(t_i)\big) \\
&= 3y'(t_i) + \frac{1}{4}h^3y^{(4)}(\xi_i) - 3f\big(t_i, y(t_i)\big) \\
&= 3f\big(t_i, y(t_i)\big) + \frac{1}{4}h^3y^{(4)}(\xi_i) - 3f\big(t_i, y(t_i)\big) \\
&= \frac{1}{4}h^3y^{(4)}(\xi_i)
\end{aligned}$$

where we used the fact that $y'(t_i) = f\big(t_i, y(t_i)\big)$. Hence, the local truncation error for this method is given by

$$\tau_{i+1}(h) = \frac{1}{4}h^3 y^{(4)}(\xi_i)$$

for $\xi_i \in (t_{i-2}, t_{i+1})$. This concludes the exercise.

(b)

In part (a), we found that the local truncation error $\tau_{i+1}(h)$ for the multistep method is

$$\tau_{i+1}(h) = \frac{1}{4}h^3 y^{(4)}(\xi_i)$$

for $\xi_i \in (t_{i-2}, t_{i+1})$. The method is said to be consistent if the local truncation error $\tau_i(h) \to 0$ as $h \to 0$. We have that

$$\lim_{h\to 0} \tau_{i+1}(h) = \frac{1}{4}y^{(4)}(\xi_i) \lim_{h\to 0} h^3 = 0,$$

so this multistep method is consistent. Let's now investigate stability and convergence of the method. By Definition 5.22, the characteristic polynomial associated with a general multistep method

$$\begin{aligned} w_0 &= \alpha, w_1 = \alpha_1, \ldots, w_{m-1} = \alpha_{m-1} \\ w_{i+1} &= a_{m-1}w_i + a_{m-2}w_{i-1} + \cdots + a_0 w_{i+1-m} + hF(t_i, h, w_{i+1}, w_i, \ldots, w_{i+1-m}) \end{aligned}$$

is given by

$$P(\lambda) = \lambda^m - a_{m-1}\lambda^{m-1} - \cdots - a_1\lambda - a_0.$$

For this multistep method, we can see that $m = 3, a_2 = -3/2$, $a_1 = 3$, and $a_0 = -1/2$. The characteristic polynomial is therefore

$$P(\lambda) = \lambda^3 - a_2\lambda^2 - a_1\lambda - a_0 = \lambda^3 + \frac{3}{2}\lambda^2 - 3\lambda + \frac{1}{2}.$$

We can factor $P(\lambda)$ to get

$$P(\lambda) = (\lambda - 1)\left(\lambda^2 + \frac{5}{2}\lambda - \frac{1}{2}\lambda\right).$$

Our first root is clearly $\lambda_1 = 1$. We now want to solve for the roots of the quadratic in the second term. We have that

$$\lambda = \frac{-5 \pm \sqrt{5^2 - 4(2)(-1)}}{2(2)} = \frac{-5 \pm \sqrt{33}}{4} \Rightarrow \lambda_2 = (-5 - \sqrt{33})/4, \lambda_3 = (-5 + \sqrt{33})/4.$$

The method satisfies the root condition if $|\lambda_i| \leq 1$ for $i = 1, 2, 3$. We find that $|\lambda_2| = |(-5 - \sqrt{33})/4| \approx 2.686 > 1$, so the root condition is not satisfied for this method. By Definition 5.2.3, this implies that the multistep method is unstable. Thus, the method is not convergent. In summary, the method is consistent but neither stable nor convergent.

5.10.7

Consider the multistep method given by

$$w_{i+1} = -4w_i + 5w_{i-1} + 2h\left[f(t_i, w_i) + 2hf(t_{i-1}, w_{i-1})\right]$$

for $i = 1, 2, \ldots, N-1$ and starting values w_0, w_1. By Definition 5.22, the characteristic polynomial associated with a general multistep method

$$\begin{aligned} w_0 &= \alpha, w_1 = \alpha_1, \ldots, w_{m-1} = \alpha_{m-1} \\ w_{i+1} &= a_{m-1}w_i + a_{m-2}w_{i-1} + \cdots + a_0 w_{i+1-m} + hF(t_i, h, w_{i+1}, w_i, \ldots, w_{i+1-m}) \end{aligned}$$

is given by

$$P(\lambda) = \lambda^m - a_{m-1}\lambda^{m-1} - \cdots - a_1\lambda - a_0.$$

For the multistep method given above, we can see that $a_1 = -4$, $a_0 = 5$, and $m = 2$. Thus, the characteristic polynomial is

$$P(\lambda) = \lambda^2 - a_1\lambda - a_0 = \lambda^2 + 4\lambda - 5.$$

We now want to solve for the roots of $P(\lambda)$ to determine if the method satisfies the root condition. We find that

$$\lambda = \frac{-4 \pm \sqrt{4^2 - 4(1)(-5)}}{2(1)} = \frac{-4 \pm \sqrt{36}}{2} = \frac{-4 \pm 6}{2} \Rightarrow \lambda_1 = -5, \lambda_2 = 1.$$

This method satisfies the root condition if $|\lambda_i| \leq 1$ for each $i = 1, 2$. However, we have that $|\lambda_1| = 5 > 1$, so this method fails the root condition. Thus, by Definition 5.23, the multistep method is unstable.

5.11.11

Recall that the Backward Euler one-step method is defined by

$$w_{i+1} = w_i + hf(t_{i+1}, w_{i+1})$$

for each $i = 0, \ldots, N-1$. Applying the Backward Euler method to the differential equation $y' = \lambda y$, we get that

$$\begin{aligned} w_{i+1} &= w_i + hf(t_{i+1}, w_{i+1}) \\ &= w_i + h\lambda w_{i+1} \end{aligned}$$

from which we can rerrange to get that

$$w_{i+1} - h\lambda w_{i+1} = (1 - h\lambda) w_{i+1} = w_i.$$

So then we have that

$$w_{i+1} = \frac{w_i}{1 - h\lambda} = \frac{1}{1 - h\lambda} w_i = Q(h\lambda) w_i.$$

Hence, we have that $Q(\lambda h) = 1/(1 - h\lambda)$, as desired.

5.9.1(a)

Consider the system of first-order differential equations

$$\begin{cases} u_1' = 3u_1 + 2u_2 - (2t^2 + 1)e^{2t} \\ u_2' = 4u_1 + u_2 + (t^2 + 2t - 4)e^{2t} \end{cases}$$

with $u_1(0) = 1$, $u_2(0) = 1$, $0 \leq t \leq 1$, and actual solutions $u_1(t) = \frac{1}{3}e^{5t} - \frac{1}{3}e^{-t} + e^{2t}$ and $u_2(t) = \frac{1}{3}e^{5t} + \frac{2}{3}e^{-t} + t^2 e^{2t}$. We want to approximation the solutions to these equations using the Runge-Kutta method for systems with a step size of $h = 0.2$, meaning we will perform $N = 5$ iterations. Using the MATLAB script runge_kutta_systems_2.m, we obtain as output the following table.

i	t_i	w_{1i}	u_{1i}	$\lvert w_{1i} - u_{1i} \rvert$	w_{2i}	u_{2i}	$\lvert w_{2i} - u_{2i} \rvert$
0	0.0	1.00000000	1.00000000	0.00000000	1.00000000	1.00000000	0.00000000
1	0.2	2.12036583	2.12500839	0.00464256	1.50699185	1.51158743	0.00459558
2	0.4	4.44122776	4.46511961	0.02389186	3.24224021	3.26598528	0.02374507
3	0.6	9.73913329	9.83235869	0.09322540	8.16341700	8.25629549	0.09287849
4	0.8	22.67655977	23.00263945	0.32607967	21.34352778	21.66887674	0.32534896
5	1.0	55.66118088	56.73748265	1.07630178	56.03050296	57.10536209	1.07485913

5.9.1(b)

Consider the system of first-order differential equations

$$\begin{cases} u_1' = -4u_1 - 2u_2 + \cos t + 4\sin t \\ u_2' = 3u_1 + u_2 - 3\sin t \end{cases}$$

with $u_1(0) = 0$, $u_2(0) = -1$, $0 \leq t \leq 2$, and actual solutions $u_1(t) = 2e^{-t} - 2e^{-2t} + \sin t$ and $u_2(t) = -3e^{-t} + 2e^{-2t}$. We want to approximation the solutions to these equations using the Runge-Kutta method for systems with a step size of $h = 0.1$,

so we require $N = 20$ iterations. Using the MATLAB script runge_kutta_systems_2.m, we obtain as output the following table.

i	t_i	w_{1i}	u_{1i}	$\lvert w_{1i} - u_{1i}\rvert$	w_{2i}	u_{2i}	$\lvert w_{2i} - u_{2i}\rvert$
0	0.0	0.00000000	0.00000000	0.00000000	-1.00000000	-1.00000000	0.00000000
1	0.1	0.27204137	0.27204675	0.00000538	-1.07704549	-1.07705075	0.00000526
2	0.2	0.49548169	0.49549074	0.00000906	-1.11554333	-1.11555217	0.00000884
3	0.3	0.67952186	0.67953338	0.00001151	-1.12482019	-1.12483139	0.00001120
4	0.4	0.83138741	0.83140051	0.00001310	-1.11228951	-1.11230221	0.00001270
5	0.5	0.95671390	0.95672798	0.00001408	-1.08381950	-1.08383310	0.00001360
6	0.6	1.05986269	1.05987732	0.00001463	-1.04403240	-1.04404648	0.00001408
7	0.7	1.14417945	1.14419437	0.00001491	-0.99654769	-0.99656198	0.00001429
8	0.8	1.21220597	1.21222098	0.00001501	-0.94417953	-0.94419386	0.00001433
9	0.9	1.26585346	1.26586845	0.00001499	-0.88909695	-0.88911120	0.00001425
10	1.0	1.30654440	1.30655930	0.00001490	-0.83295364	-0.83296776	0.00001411
11	1.1	1.33532844	1.33534321	0.00001477	-0.77699300	-0.77700693	0.00001394
12	1.2	1.35297699	1.35299160	0.00001461	-0.72213299	-0.72214673	0.00001374
13	1.3	1.36006018	1.36007462	0.00001443	-0.66903470	-0.66904822	0.00001352
14	1.4	1.35700930	1.35702353	0.00001423	-0.61815747	-0.61817077	0.00001330
15	1.5	1.34416716	1.34418117	0.00001401	-0.56980329	-0.56981634	0.00001305
16	1.6	1.32182847	1.32184223	0.00001376	-0.52415236	-0.52416515	0.00001279
17	1.7	1.29027184	1.29028532	0.00001348	-0.48129154	-0.48130403	0.00001250
18	1.8	1.24978481	1.24979796	0.00001315	-0.44123705	-0.44124922	0.00001217
19	1.9	1.20068300	1.20069578	0.00001278	-0.40395251	-0.40396431	0.00001180
20	2.0	1.14332436	1.14333672	0.00001236	-0.36936318	-0.36937457	0.00001139

11.1.3(a)

Consider the boundary value problem

$$y'' = -3y' + 2y + 2x + 3,\ 0 \le x \le 1,\ y(0) = 2,\ y(1) = 1.$$

We want to apply the linear shooting method to approximate the solution to this BVP for a step size of $h = 0.1$. As such, we will perform $N = 10$ iterations. Using my Python implementation of the method (linear_shooting.py), we obtain the following table as our output.

i	x_i	u_{1i}	v_{1i}	w_i
0	0.0	2.00000000	0.00000000	2.00000000
1	0.1	2.03212917	0.08667083	1.40843171
2	0.2	2.11883042	0.15238213	1.02226375
3	0.3	2.24919314	0.20370236	0.78331775
4	0.4	2.41588955	0.24524882	0.65103904
5	0.5	2.61411953	0.28027320	0.59722789
6	0.6	2.84087251	0.31107168	0.60234998
7	0.7	3.09441166	0.33927230	0.65295283
8	0.8	3.37391374	0.36603627	0.73985699
9	0.9	3.67921813	0.39219903	0.85688993
10	1.0	4.01065236	0.41836911	1.00000000

11.1.3(b)

Consider the boundary value problem

$$y'' = -4x^{-1}y' - 2x^{-2}y + 2x^{-2}\ln x,\ 1 \le x \le 2,\ y(1) = -1/2,\ y(2) = \ln 2.$$

Applying the linear shooting method to approximate the solution to this BVP for a step size of $h = 0.05$, we perform $N = 20$ iterations. Using the Python code linear_shooting.py, we obtain as output the following table.

i	x_i	u_{1i}	v_{1i}	w_i
0	1.00	-0.50000000	0.00000000	-0.50000000
1	1.05	-0.49882877	0.04535093	-0.31742476
2	1.10	-0.49559872	0.08264377	-0.16502314
3	1.15	-0.49067261	0.11342052	-0.03698984
4	1.20	-0.48434486	0.13888778	0.07120709
5	1.25	-0.47685619	0.15999887	0.16314024
6	1.30	-0.46840470	0.17751367	0.24165106
7	1.35	-0.45915441	0.19204281	0.30901800
8	1.40	-0.44924180	0.20408060	0.36708182
9	1.45	-0.43878104	0.21402994	0.41734001
10	1.50	-0.42786800	0.22222131	0.46101857
11	1.55	-0.41658357	0.22892735	0.49912720
12	1.60	-0.40499618	0.23437421	0.53250207
13	1.65	-0.39316393	0.23875042	0.56183917
14	1.70	-0.38113629	0.24221386	0.58772060
15	1.75	-0.36895549	0.24489734	0.61063533
16	1.80	-0.35665764	0.24691301	0.63099588
17	1.85	-0.34427369	0.24835594	0.64915156
18	1.90	-0.33183020	0.24930700	0.66539929
19	1.95	-0.31935000	0.24983520	0.67999228
20	2.00	-0.30685272	0.24999960	0.69314718

11.2.3(a)

Consider the boundary-value problem

$$y'' = -e^{-2y},\ 1 \le x \le 2,\ y(1) = 0,\ y(2) = \ln 2$$

with actual solution $y(x) = \ln x$. We want to use the nonlinear shooting method with a tolerance of 10^{-4} for $N = 10$ iterations to approximate the solution to this problem. Using my Python implementation of the method, nonlinear_shooting_newton.m, we obtain as output the table below.

i	x_i	w_{1i}	$y(x_i)$	$\lvert w_{1i} - y(x_i)\rvert$	w_{2i}
0	1.0	0.00000000	0.00000000	0.00000000	1.00000166
1	1.1	0.09530982	0.09531018	0.00000036	0.90909178
2	1.2	0.18232094	0.18232156	0.00000061	0.83333370
3	1.3	0.26236347	0.26236426	0.00000080	0.76923079
4	1.4	0.33647129	0.33647224	0.00000095	0.71428547
5	1.5	0.40546403	0.40546511	0.00000108	0.66666622
6	1.6	0.47000243	0.47000363	0.00000120	0.62499939
7	1.7	0.53062693	0.53062825	0.00000132	0.58823454
8	1.8	0.58778522	0.58778666	0.00000144	0.55555468
9	1.9	0.64185232	0.64185389	0.00000156	0.52631480
10	2.0	0.69314549	0.69314718	0.00000169	0.49999890

11.2.3(b)

Consider the boundary-value problem given by

$$y'' = y' \cos x - y \ln y,\ 0 \le x \le \pi/2,\ y(0) = 1,\ y(\pi/2) = e$$

along with actual solution $y(x) = e^{\sin x}$. We will apply the nonlinear shooting method with a tolerance of 10^{-4} for $N = 10$ iterations to approximate the solution to this problem. Using the Python code nonlinear_shooting_newton.m, we obtain the table below as our output.

i	x_i	w_{1i}	$y(x_i)$	$\lvert w_{1i} - y(x_i)\rvert$	w_{2i}
0	0.00000000	1.00000000	1.00000000	0.00000000	1.00007499
1	0.15707963	1.16934787	1.16933413	0.00001374	1.15502446
2	0.31415927	1.36211436	1.36208552	0.00002884	1.29551712
3	0.47123890	1.57462734	1.57458304	0.00004431	1.40306671
4	0.62831853	1.80005614	1.79999746	0.00005869	1.45633156
5	0.78539816	2.02818535	2.02811498	0.00007037	1.43418949
6	0.94247780	2.24577743	2.24569937	0.00007807	1.32006974
7	1.09955743	2.43766321	2.43758190	0.00008131	1.10669908
8	1.25663706	2.58852370	2.58844295	0.00008075	0.79990902
9	1.41371669	2.68509837	2.68502044	0.00007793	0.42004153
10	1.57079633	2.71835646	2.71828183	0.00007463	-0.00001138

11.3.3(a)

Consider the boundary value problem from 11.1.3(a). We want to apply the linear finite-difference method to approximation the solution to this BVP with a step size of $h = 0.1$, meaning that we require $N = 10$ iterations. Using my Python implementation of the method (linear_finite_diff.py), we obtain as output the table below.

i	x_i	w_i
0	0.0	2.00000000
1	0.1	1.40535199
2	0.2	1.01809654
3	0.3	0.77913550
4	0.4	0.64736665
5	0.5	0.59427431
6	0.6	0.60014996
7	0.7	0.65145196
8	0.8	0.73896130
9	0.9	0.85649362
10	1.0	1.00000000

11.3.3(b)

Consider the boundary value problem

$$y'' = -4x^{-1}y' + 2x^{-2}y + 2x^{-2}\ln x,\ 1 \le x \le 2,\ y(1) = -1/2,\ y(2) = \ln 2.$$

Note that this is a slightly different problem than the one in 11.1.3(b). We want to apply the linear finite-difference method to approximation the solution to this BVP with a step size of $h = 0.05$, meaning that we we will perform $N = 20$ iterations. Using my Python implementation of the method (linear_finite_diff.py), we obtain the table below as our output.

i	x_i	w_i
0	1.00	-0.50000000
1	1.05	-0.31114686
2	1.10	-0.15662817
3	1.15	-0.02881690
4	1.20	0.07795820
5	1.25	0.16797186

6	1.30	0.24448670
7	1.35	0.31002173
8	1.40	0.36654278
9	1.45	0.41559928
10	1.50	0.45842388
11	1.55	0.49600574
12	1.60	0.52914512
13	1.65	0.55849447
14	1.70	0.58458955
15	1.75	0.60787335
16	1.80	0.62871452
17	1.85	0.64742171
18	1.90	0.66425489
19	1.95	0.67943423
20	2.00	0.69314718

runge_kutta_systems_2.m

```
%% Input information
a = 0;          % left endpoint
b = 1;          % right endpoint
h = 0.2;        % stepsize
N = (b-a)/h;    % number of subintervals
alpha1 = 1;     % initial condition 1
alpha2 = 1;     % initial condition 2

f1 = @(t,u1,u2) 3*u1 + 2*u2 - (2*t^2 + 1)*exp(2*t);
f2 = @(t,u1,u2) 4*u1 + u2 + (t^2 + 2*t - 4)*exp(2*t);

% Exact solutions
u1 = @(t) 1/3*exp(5*t) - 1/3*exp(-t) + exp(2*t);
u2 = @(t) 1/3*exp(5*t) + 2/3*exp(-t) + t^2*exp(2*t);

%% Performing the method

% Starting values
t = a;
w1 = alpha1;
w2 = alpha2;

% Output header and starting iteration
fprintf('t_i \t w_1i \t\t u_1i \t\t |w_1i - u_1i| \t w_2i \t\t u_2i \t\t |w_2i - u_2i| \n')
fprintf('%.1f \t %.8f \t %.8f \t %.8f \t %.8f \t %.8f \n',t,w1,u1(t), ...
    abs(w1-u1(t)),w2,u2(t), abs(w2-u2(t)))

for i=1:N

    k(1,1) = h * f1(t, w1, w2);
    k(1,2) = h * f2(t, w1, w2);

    k(2,1) = h * f1(t + h/2, w1 + k(1,1)/2, w2 + k(1,2)/2);
    k(2,2) = h * f2(t + h/2, w1 + k(1,1)/2, w2 + k(1,2)/2);

    k(3,1) = h * f1(t + h/2, w1 + k(2,1)/2, w2 + k(2,2)/2);
    k(3,2) = h * f2(t + h/2, w1 + k(2,1)/2, w2 + k(2,2)/2);

    k(4,1) = h * f1(t + h, w1 + k(3,1), w2 + k(3,2));
    k(4,2) = h * f2(t + h, w1 + k(3,1), w2 + k(3,2));

    w1 = w1 + (k(1,1) + 2*k(2,1) + 2*k(3,1) + k(4,1))/6;
    w2 = w2 + (k(1,2) + 2*k(2,2) + 2*k(3,2) + k(4,2))/6;

    t = a + i*h;

    fprintf('%.1f \t %.8f \t %.8f \t %.8f \t %.8f \t %.8f \t %.8f \n',t,w1,u1(t), ...
```

```
    abs(w1-u1(t)),w2,u2(t), abs(w2-u2(t)))
end
```

linear_shooting.py

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

# For more decimal places
pd.set_option("display.precision", 8)

# Function
p = lambda x: -3
q = lambda x: 2
r = lambda x: 2*x + 3
# Left endpoint
a = 0
# Right endpoint
b = 1
# Step size
h = 0.1

# Left endpoint condition
alpha = 2
# Right endpoint condition
beta = 1

N = int((b-a)/h)
xs = np.arange(a, b+h, h)
w1 = np.zeros(len(xs))
u1 = np.zeros(len(xs))
u2 = np.zeros(len(xs))
v1 = np.zeros(len(xs))
v2 = np.zeros(len(xs))
w1[0] = alpha

# Initial conditions for IVPs
u1[0] = alpha
u2[0] = 0
v1[0] = 0
v2[0] = 1

# Approximating equations for system of ODEs
for i in range(0, len(xs) - 1):

  x = a + i*h

  k11 = h*u2[i]
  k12 = h*(p(x)*u2[i] + q(x)*u1[i] + r(x))
  k21 = h*(u2[i] + 1/2*k12)
  k22 = h*(p(x+h/2)*(u2[i] + 1/2*k12) + q(x+h/2)*(u1[i] + 1/2*k11) + r(x+h/2))
  k31 = h*(u2[i] + 1/2*k22)
  k32 = h*(p(x+h/2)*(u2[i]+1/2*k22) + q(x+h/2)*(u1[i] + 1/2*k21) + r(x+h/2))
  k41 = h*(u2[i] + k32)
  k42 = h*(p(x+h)*(u2[i] + k32) + q(x+h)*(u1[i] + k31) + r(x+h))
  u1[i+1] = u1[i] + 1/6*(k11 + 2*k21 + 2*k31 + k41)
  u2[i+1] = u2[i] + 1/6*(k12 + 2*k22 + 2*k32 + k42)

  kp11 = h*v2[i]
  kp12 = h*(p(x)*v2[i] + q(x)*v1[i])
  kp21 = h*(v2[i] + 1/2*kp12)
  kp22 = h*(p(x+h/2)*(v2[i] + 1/2*kp12) + q(x+h/2)*(v1[i] + 1/2*kp11))
  kp31 = h*(v2[i] + 1/2*kp22)
  kp32 = h*(p(x+h/2)*(v2[i] + 1/2*kp22) + q(x+h/2)*(v1[i] + 1/2*kp21))
  kp41 = h*(v2[i] + kp32)
  kp42 = h*(p(x+h)*(v2[i] + kp32) + q(x+h)*(v1[i] + kp31))
  v1[i+1] = v1[i] + 1/6*(kp11 + 2*kp21 + 2*kp31 + kp41)
```

```
  v2[i+1] = v2[i] + 1/6*(kp12 + 2*kp22 + 2*kp32 + kp42)

# Initializing arrays
w1 = np.zeros(len(xs))
w2 = np.zeros(len(xs))
W1 = np.zeros(len(xs))
W2 = np.zeros(len(xs))

# Conditions
w1[0] = alpha
w2[0] = (beta - u1[-1])/v1[-1]

# Approximations
for i in range(0, len(xs)):
  W1[i] = u1[i] + w2[0]*v1[i]
  # For first derivative (not outputted here)
  W2[i] = u2[i] + w2[0]*v2[i]

# Output
df = pd.DataFrame({('x_i'): xs, 'u_1i': u1, 'v_1i': v1, 'w_i': W1})
print(df)
```

nonlinear_shooting_newton.m

```
%% Input Information
a = 0;          % left endpoint
b = pi/2;       % right endpoint
alpha = 1;      % boundary condition at left endpoint
beta = exp(1);  % boundary condition at right endpoint
N = 10;         % number of subintervals
tol = 1e-4;     % tolerance
M = 10;         % maximum number of iterations

f = @(x,y,y_prime) y_prime*cos(x) - y*log(y);
partialf_partialy = @(x,y,y_prime) -log(y) - 1;
partialf_partialy_prime = @(x,y,y_prime) cos(x);

% Exact solution
y = @(x) exp(sin(x));

%% Performing the method

h = (b-a)/N;
j = 1;
TK = (beta - alpha)/(b-a);

fprintf('x_i \t\t w_1i \t\t y(x_i) \t |w_1i-y(x_i)| \t w_2i\n')

while(j <= M)
    w(1,1) = alpha;
    w(2,1) = TK;
    u1 = 0;
    u2 = 1;

    for i=2:N+1
        x = a + (i-2)*h;

        k(1,1) = h * w(2,i-1);
        k(1,2) = h * f( x, w(1,i-1), w(2,i-1) );

        k(2,1) = h * ( w(2,i-1) + k(1,2)/2 );
        k(2,2) = h * f( x + h/2, w(1,i-1) + k(1,1)/2, w(2,i-1) + k(1,2)/2 );

        k(3,1) = h * ( w(2,i-1) + k(2,2)/2 );
        k(3,2) = h * f( x + h/2, w(1,i-1) + k(2,1)/2, w(2,i-1) + k(2,2)/2 );

        k(4,1) = h * ( w(2,i-1) + k(3,2) );
        k(4,2) = h * f( x + h, w(1,i-1) + k(3,1), w(2,i-1) + k(3,2) );

```

```
        w(1,i) = w(1,i-1) + ( k(1,1) + 2*k(2,1) + 2*k(3,1) + k(4,1) )/6;
        w(2,i) = w(2,i-1) + ( k(1,2) + 2*k(2,2) + 2*k(3,2) + k(4,2) )/6;

        k_prime(1,1) = h * u2;
        k_prime(1,2) = h * ( partialf_partialy( x, w(1,i-1), w(2,i-1) ) * u1 ...
            + partialf_partialy_prime( x, w(1,i-1), w(2,i-1) )*u2 );

        k_prime(2,1) = h * ( u2 + k_prime(1,2)/2 );
        k_prime(2,2) = h * ( partialf_partialy( x + h/2, w(1,i-1), w(2,i-1) ) * ( u1 + k_prime
    (1,1)/2 ) ...
            + partialf_partialy_prime( x + h/2, w(1,i-1), w(2,i-1) ) * ( u2 + k_prime(1,2)/2 ) )
    ;

        k_prime(3,1) = h * ( u2 + k_prime(2,2)/2 );
        k_prime(3,2) = h * ( partialf_partialy( x + h/2, w(1,i-1), w(2,i-1) ) * ( u1 + k_prime
    (2,1)/2 ) ...
            + partialf_partialy_prime( x + h/2, w(1,i-1), w(2,i-1) ) * ( u2 + k_prime(2,2)/2 ) )
    ;

        k_prime(4,1) = h * ( u2 + k_prime(3,2) );
        k_prime(4,2) = h * ( partialf_partialy( x + h, w(1,i-1), w(2,i-1) ) * ( u1 + k_prime
    (3,1) ) ...
            + partialf_partialy_prime( x + h, w(1,i-1), w(2,i-1) ) * ( u2 + k_prime(3,2) ) );

        u1 = u1 + ( k_prime(1,1) + 2*k_prime(2,1) + 2*k_prime(3,1) + k_prime(4,1) )/6;
        u2 = u2 + ( k_prime(1,2) + 2*k_prime(2,2) + 2*k_prime(3,2) + k_prime(4,2) )/6;
    end

    if(abs(w(1,N+1) - beta) <= tol)
        for i = 1:N+1
            x = a + (i-1) * h;
            fprintf('%.8f \t %.8f \t %.8f \t %.8f \t %.8f \n',x,w(1,i),y(x),abs(w(1,i)-y(x)),w
    (2,i))
        end

        fprintf('Convergence in %d iterations t = %.7f\n',j,TK)
        break;
    end

    TK = TK - ( w(1,N+1) - beta )/u1;

    j = j+1;
end
```

linear_finite_diff.py

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

# For more decimal places
pd.set_option("display.precision", 8)
# Functions
p = lambda x: -4/x
q = lambda x: 2/x**2
r = lambda x: (-2/x**2)*np.log(x)
# Left endpoint
a = 1
# Right endpoint
b = 2
# Left endpoint value
alpha = -1/2
# Right endpoint value
beta = np.log(2)
# Step size
h = 0.05
# Determination of N
N = int((b-a)/h) - 1
```

```
24
25 # Initializing arrays (used in approximation computation)
26 arr_a = np.zeros(N+2)
27 arr_b = np.zeros(N+2)
28 arr_c = np.zeros(N+2)
29 arr_d = np.zeros(N+2)
30 arr_l = np.zeros(N+2)
31 arr_u = np.zeros(N+2)
32 arr_z = np.zeros(N+2)
33
34 # Mesh points
35 x = np.arange(a, b+h, h)
36
37 # Step 1
38 h = (b-a)/(N+1)
39 arr_a[1] = 2 + h**2*q(x[1])
40 arr_b[1] = -1 + (h/2)*p(x[1])
41 arr_d[1] = -h**2*r(x[1]) + (1+(h/2)*p(x[1]))*alpha
42
43 # Step 2
44 for i in range(2, N):
45   arr_a[i] = 2 + h**2*q(x[i])
46   arr_b[i] = -1 + (h/2)*p(x[i])
47   arr_c[i] = -1 - (h/2)*p(x[i])
48   arr_d[i] = -h**2*r(x[i])
49
50 # Step 3
51 arr_a[-2] = 2 + h**2*q(x[-2])
52 arr_c[-2] = -1 - (h/2)*p(x[-2])
53 arr_d[-2] = -h**2*r(x[-2]) + (1-(h/2)*p(x[-2]))*beta
54
55 # Step 4
56 arr_l[1] = arr_a[1]
57 arr_u[1] = arr_b[1]/arr_a[1]
58 arr_z[1] = arr_d[1]/arr_l[1]
59
60 # Step 5
61 for i in range(2, N):
62   arr_l[i] = arr_a[i] - arr_c[i]*arr_u[i-1]
63   arr_u[i] = arr_b[i]/arr_l[i]
64   arr_z[i] = (arr_d[i] - arr_c[i]*arr_z[i-1])/arr_l[i]
65
66 # Step 6
67 arr_l[-2] = arr_a[-2] - arr_c[-2]*arr_u[-3]
68 arr_z[-2] = (arr_d[-2] - arr_c[-2]*arr_z[-3])/arr_l[-2]
69
70 # Step 7
71 w = np.zeros(N+2)
72 w[0] = alpha
73 w[-1] = beta
74 w[-2] = arr_z[-2]
75
76 # Step 8
77 for i in range(N-1, 0, -1):
78   w[i] = arr_z[i] - arr_u[i]*w[i+1]
79
80 # Output (step 9)
81 df = pd.DataFrame({'x_i': x, 'w_i': w})
82 print(df)
```