Ordinary Differential Equations

Ollie Stubbs

January 22, 2021

1 Question 1

My program for the forward Euler method can be found in section 10.3.1. I calculated the required values, which show clearly the exponential oscillation in the case h=2.0, and I also added a column to show the value of $\frac{ln|\epsilon_n|}{x_n}$, since if $\epsilon_n \approx Ae^{\gamma x_n}$, $\frac{ln|\epsilon_n|}{x_n} \approx \gamma + \frac{ln(A)}{x_n}$, so should tend to γ as $n \to \infty$. As such from the first table we can estimate that for h = 2.0 γ is probably between 0.96 and 0.98.

The results for a variety of other values of h are also shown below. What we observe is that instability occurs for values of h larger than 0.5, 0.5 oscillates but does not grow, the magnitude of the error remains roughly constant, and for h less than 0.5 the error converges to zero. The interesting thing we observe is that the highest growth rate for the error appears to be for h=1.0, not 2.0 or 0.6, suggesting the growth rate has a maximum somewhere between 0.6 and 2.0, this is contrary to how one might expect smaller steps to make the approximation more accurate. I did not include a table for h=0.3 as the results did not illustrate anything of interest.

Table 1: Iterates with h = 2.0

N	x_N	Y_n	$y(x_N)$	ϵ_N	$\frac{ln\lvert\epsilon_n\rvert}{x_n}$
0	0	0	0	0	NA
1	2	8.000	0.036	7.964	1.037
2	4	-55.85	0.001	-55.84	1.006
3	6	391.0	0.000	391.0	0.995
4	8	-2736	0.000	-2736	0.989
5	10	19158	0.000	19158	0.986
6	12	-1.34×10^5	0.000	-1.34×10^5	0.984

Table 2: Iterates with h = 1.0

N	x_N	Y_n	$y(x_N)$	ϵ_N	$\frac{ln\lvert\epsilon_n\rvert}{x_n}$
0	0	0	0	0	NA
1	1	4.000	0.234	3.76	1.326
2	2	-11.459	0.036	-11.495	1.221
3	3	34.449	0.005	34.444	1.180
4	4	-103.34	0.001	-103.34	1.160
		⋮		⋮	
11	11	2.26×10^6	0.000	2.26×10^6	1.126
12	12	-6.78×10^6	0.000	-6.78×10^6	1.123

Table 3: Iterates with h = 0.6

N	x_N	Y_n	$y(x_N)$	ϵ_N	$\frac{ln\lvert\epsilon_n\rvert}{x_n}$
0	0	0	0	0	NA
1	0.6	2.400	0.421	1.979	1.138
2	1.2	-2.637	0.165	-2.802	0.859
3	1.8	3.910	0.053	3.857	0.750
4	2.4	-5.408	0.001	-5.424	0.705
		⋮		⋮	
19	11.4	843.1	0.000	843.1	0.591
20	12	-1180	0.000	-1180	0.589

Table 4: Iterates with h = 0.5

N	x_N	Y_n	$y(x_N)$	ϵ_N	$\frac{ln\lvert\epsilon_n\rvert}{x_n}$
0	0	0	0	0	NA
1	0.5	2.000	0.465	1.535	0.857
2	1.0	-1.264	0.234	-1.498	0.404
3	1.5	1.535	0.095	1.440	0.243
4	2.0	-1.435	0.036	-1.471	0.193
		⋮		⋮	
23	11.5	1.462	0.000	1.462	0.033
24	12	-1.462	0.000	-1.462	0.032

Table 5: Iterates with h = 0.4

N	x_N	Y_n	$y(x_N)$	ϵ_N	$\frac{ln\lvert\epsilon_n\rvert}{x_n}$
0	0	0	0	0	NA
1	0.4	1.600	0.495	1.105	0.250
2	0.8	-0.248	0.322	-0.563	-0.717
3	1.2	0.468	0.165	0.303	-0.996
4	1.6	-0.135	0.078	-0.214	-0.965
		⋮		⋮	
29	11.6	0.000	0.000	0.000	-1.241
30	12	0.000	0.000	0.000	-1.242

Table 6: Iterates with h = 0.2

N	x_N	Y_n	$y(x_N)$	ϵ_N	$\frac{ln\lvert\epsilon_n\rvert}{x_n}$
0	0	0	0	0	NA
1	0.2	0.800	0.442	0.358	0.250
2	0.4	0.696	0.495	0.201	-0.717
3	0.6	0.499	0.421	0.078	-0.996
4	0.8	0.341	0.322	0.018	-0.965
		⋮		⋮	
49	11.8	0.000	0.000	0.000	-2.123
50	12	0.000	0.000	0.000	-2.121

2 Question 2

Complementary solution C_n satisfies:

$$C_{n+1} = (1-4h)C_n \tag{1}$$

$$C_n = A(1-4h)^n \tag{2}$$

Particular solution P_n satisfies the whole equation, so guess that $P_n = Be^{-2hn}$, this yields:

$$Be^{-2h(n+1)} = B(1-4h)e^{-2hn} + 4he^{-2hn} \tag{3}$$

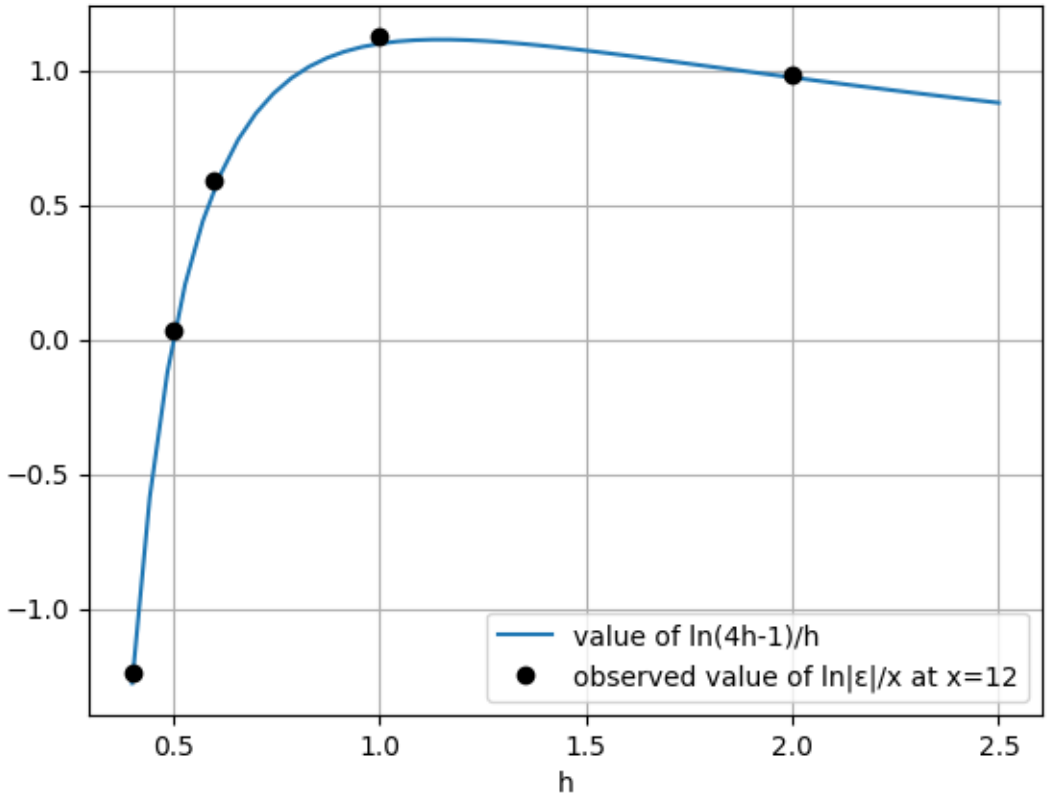
$$B = \frac{4h}{e^{-2h} + 4h - 1} \tag{4}$$

$e^{-2h} + 4h - 1$ is non-zero for $h > 0$ since it is zero at $h = 0$ and $\frac{d}{dh}e^{-2h} + 4h - 1 = -2e^{-2h} + 4 > 0$ for $h > 0$, so the expression must be greater than zero for $h > 0$.

Then $Y_n = C_n + P_n = A(1-4h)^n + \frac{4h}{e^{-2h}+4h-1}e^{-2hn}$, have that $Y_0 = 0$, which implies $A = -\frac{4h}{e^{-2h}+4h-1}$.

We can see from this result that the oscillation occurs when $(1-4h) < -1$, ie when $h > 0.5$. In this case the magnitude of the error is approximately $|(1-4h)^n| = (4h-1)^n = e^{ln(4h-1)n}$. We have that $x_n = hn$ so can write this $e^{\frac{ln(4h-1)}{h}x_n}$ and so get an estimate that $\gamma = \frac{ln(4h-1)}{h}$. For h=2.0 $\gamma = \frac{ln(8-1)}{2} \approx 0.97$. In fact we can graph the value of gamma as h varies, which you can see in figure 1. The graph shows that $\gamma = \frac{ln(4h-1)}{h}$ fits well with the previously observed data.

Figure 1: Graph of $\frac{ln(4h-1)}{h}$ against h



If $h < 0.5$ the solution will converge to 0, and if $h = 0.5$ we would expect Y_n to converge to $\frac{2}{e^{-1}+2-1} \approx 1.46$. These results are consistent with the results of the program.

If we take the limit as n tends to infinity for fixed x, then have $h = \frac{x}{n}$ and

$$y(x) = \lim_{n\to\infty} \frac{4\frac{x}{n}}{e^{\frac{-2x}{n}} + 4\frac{x}{n} - 1}(e^{-2x} - (1 - \frac{4x}{n})^n) \tag{5}$$

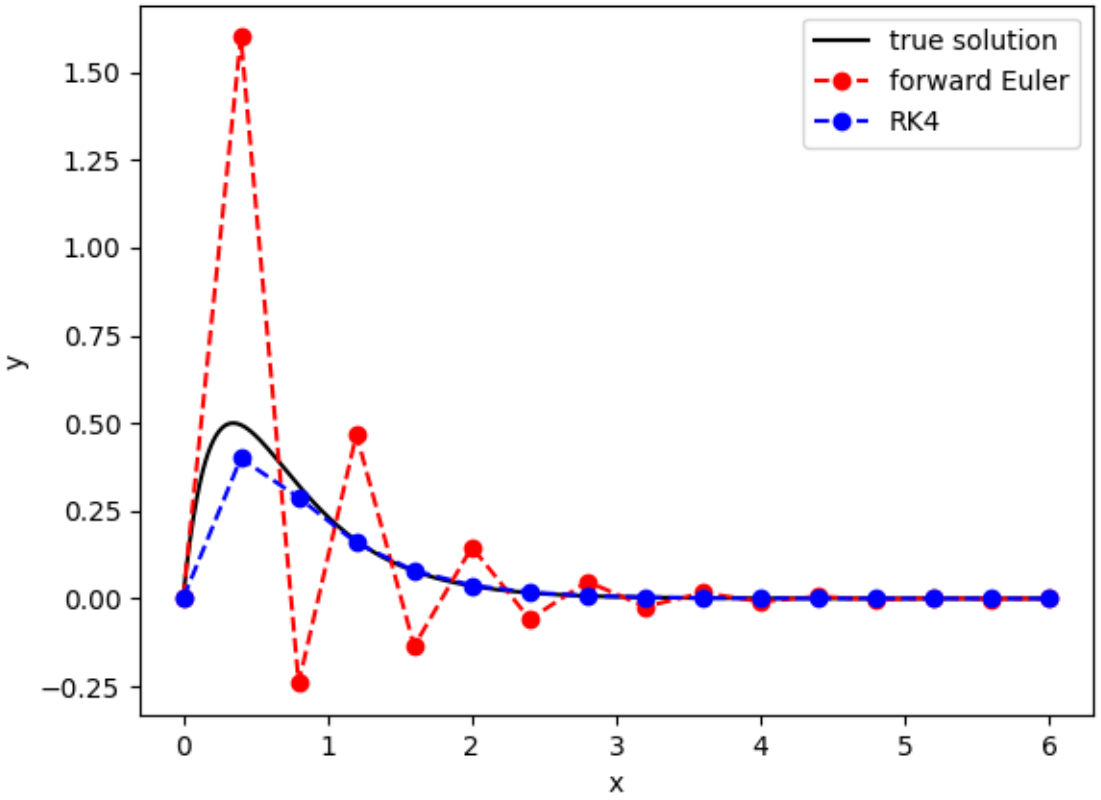
$$= \lim_{n\to\infty} \frac{4\frac{x}{n}}{1-2\frac{x}{n}+4\frac{x}{n}-1}(e^{-2x}-e^{-4x}) = 2e^{-2x}-2e^{-4x} \tag{6}$$

As desired.

3 Question 3

I ran the forward Euler algorithm, the RK4 algorithm and the exact solution function between 0 and 6 and plotted the results in figure 2

Figure 2: comparison of forward Euler method with RK4



4 Question 4

I ran both algorithms at each of the required values of h, the code can be found in section 10.4.9, and the results are in tables 7 and 8 respectively. Figure 3 shows a graph of the results.

From the graph we can observe that both methods behave erratically for the larger values of h, and that the RK4 methods behaves strangely once the value of the error gets below 10^{-14} or so. This can be explained by the fact that all these numbers are being represented as 64-bit floats, which struggle to be more precise than about 10^{-15}. However we notice that in between these two extremes the rate of change in the precision is very well behaved, forming a straight line in the log graph, meaning the error changes as some power of h. We would like to know what the gradient of the line is.

Both lines are very straight from 1.6×10^{-3} to 0.1, so will make an estimate for the gradient by taking the gradient between these two points. For the Euler method this is $\frac{ln(4.174\times10^{-3})-ln(5.59\times10^{-5})}{ln(0.1)-ln(1.6\times10^{-3})}$=1.037. For the RK4 method, this is $\frac{ln(5.512\times10^{-6})-ln(2.715\times10^{-13})}{ln(0.1)-ln(1.6\times10^{-3})} = 4.046$.

These results suggest that in this region the error in the forward Euler method goes roughly like h, and the error in the Runge-Kutta method goes roughly like h^4, which fits with the theoretical accuracy of the methods: the Euler method has first order accuracy, and RK4 has fourth order accuracy.

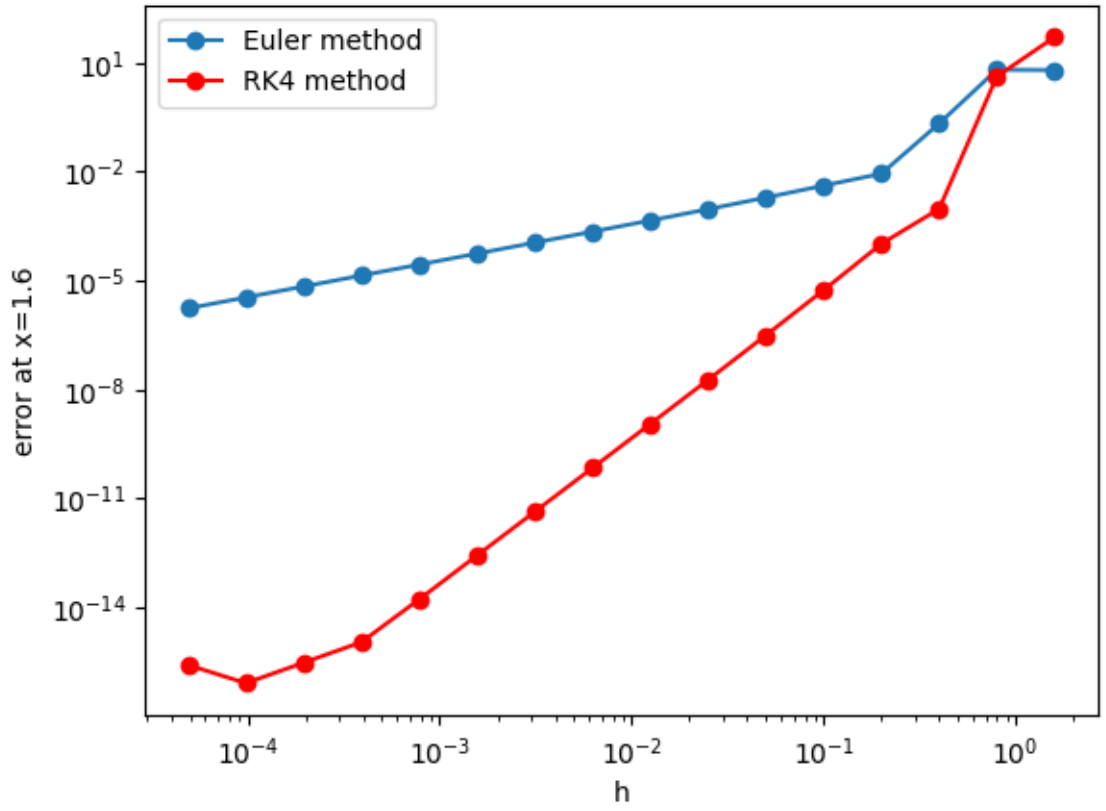
Table 7: Error at x=1.6 using forward Euler

h	$\lvert E_n \rvert$ at $x_n = 1.6$
4.8828×10^{-5}	1.7416×10^{-6}
9.7656×10^{-5}	3.4836×10^{-6}
1.9531×10^{-4}	6.9686×10^{-6}
3.9063×10^{-4}	1.3943×10^{-5}
⋮	⋮
0.1	4.1740×10^{-3}
0.2	8.8704×10^{-3}
0.4	2.1366×10^{-1}
0.8	6.4721
1.6	6.3218

Table 8: Error at x=1.6 using RK4

h	$\lvert E_n \rvert$ at $x_n = 1.6$
4.8828×10^{-5}	2.6368×10^{-16}
9.7656×10^{-5}	8.3267×10^{-17}
1.9531×10^{-4}	3.0531×10^{-16}
3.9063×10^{-4}	1.1241×10^{-15}
⋮	⋮
0.1	5.5117×10^{-6}
0.2	1.0423×10^{-4}
0.4	9.5114×10^{-4}
0.8	4.2435
1.6	51.339

Figure 3: comparison of accuracy of Euler and RK4 methods at x=1.6



5 Question 5

I ran the forward Euler method program with the required input, the start and end of the output is in table 9. The code for this is in section 10.4.10.

What we can observe from this is that the error grows exponentially with growth rate at x=10 roughly equal to 3.

If we integrate analytically as before using the same method we observe that the solution is:

$$Y_n = \frac{1 - h - e^{-h}}{1 + 4h - e^{-h}}(1 + 4h)^n + \frac{5h}{1 + 4h - e^{-h}}e^{-h} \tag{7}$$

We can see from this equation that the $(1+4h)^n$ term causes an exponential growth in the error. We would like to estimate what the growth rate would be in the case h=0.001, in this case, since n = $1000x_n$ that term becomes 1.004^n=$e^{ln(1.004)n}$=$e^{1000ln(1.004)x} \approx e^{4x}$

Table 9: Iterates with h=0.001

N	x_N	Y_n	$y(x_N)$	ϵ_N	$\frac{ln\lvert\epsilon_n\rvert}{x_n}$
0	0.000	1	1	0	NA
1	0.001	0.999	0.999	-4.998×10^{-7}	-1.45×10^5
2	0.002	0.998	0.998	-1.001×10^{-6}	-6.91×10^4
3	0.003	0.997	0.997	-1.504×10^{-6}	-4.47×10^4
4	0.004	0.996	0.996	-2.08×10^{-6}	-3.28×10^4
		$\vdots$		$\vdots$	
9999	9.999	-2.16×10^{13}	0.000	-2.16×10^{13}	3.071
10000	10.000	-2.17×10^{13}	0.000	-2.17×10^{13}	3.071

We may note that $\frac{1-0.001-e^{-0.001}}{1+0.004-e^{-0.001}} = -9.998 \times 10^{-5} \approx -10^{-4}$, so we would expect $\frac{ln\lvert\epsilon_n\rvert}{x} \approx \frac{ln(10^{-4})+ln(e^{4x})}{x} \approx \frac{-9.21}{x} + 4$, which in the case x=10 is 3.079, which is almost exactly the observed value in the table.

We now wish to know what the expected error at x=10 is as h tends to 0. We use the fact that n=$\frac{10}{h}$, then the error (as $h \to 0$) is approximately

$$\frac{1-h-e^{-h}}{1+4h-e^{-h}}(1+4h)^{\frac{10}{h}} \approx \frac{-h^2}{10h}e^{40} \approx -h \times 2.37 \times 10^{16} \tag{8}$$

This fits accurately with the observed error at x=10 in table 9, where h is 0.001. We also can see that while it is technically true that reducing the value of h reduces the size of the error, it only does so linearly, and the magnitude of the error is so large that in order to get any kind of accuracy at x=10 h would have to be so small (and therefore n so large) that it would take an unrealistic amount of computing time to find the solution.

RK4 would likely be much more effective at reducing the error since it has fourth order accuracy, so reducing the size of h will have a much bigger effect at reducing the error.

6 Question 6

If p = 0 equation (13a) becomes $\frac{d^2y}{dx^2} = 0$ which has solution $y = Ax + B$. If p=0, the particular solution must satisfy $B = 0$ and $A = 1$ so get solution $y = x$.

If p non-zero then boundary conditions force:

$$A\sin(p-\phi) = 0 \;\;\&\;\; A\sin(p-\phi) - Ap\cos(p-phi) = -Ap\cos(p-\phi) = 1 \tag{9}$$

Which gives particular solution $\phi = p + n\pi$ for $n \in \mathbb{Z}$, and $A = \frac{(-1)^{n+1}}{p}$, which by periodicity and the fact sin is odd gives the particular solution for $p \neq 0$:

$$y = -\frac{1+x}{p}\sin(p(1+x)^{-1} - p) \tag{10}$$

On the other hand the (13b) conditions enforce the following equations for $p \neq 0$:

$$A\sin(p-\phi) = 0 \;\;\&\;\; 2A\sin(\frac{p}{2} - \phi) = 0 \tag{11}$$

This is true for any A, but does require that $p - \phi = n\pi \;\;\&\;\; \frac{p}{2} - \phi = m\pi \implies p = 2k\pi$ for some $k \in \mathbb{Z}$.

If $p = 0$, have $y = Ax + B$, with requirement $B = 0 \;\;\&\;\; A + B = 0 \implies y = 0$, so smallest non-negative eigenvalue is 2π and in general eigenfunctions (using periodicity of sin to remove ϕ) are $y_k = A(1+x)\sin(2k\pi(1+x)^{-1})$ with eigenvalue $2k\pi$.

7 Question 7

I modified my program for the Runge-Kutta algorithm to work with coupled ODEs, and also to allow p to be passed as an argument to the program, the code can be found in section 10.3.3. The results for p = 6 and p=7 are in tables 10 and 11 respectively, and the results are shown in figure 4. The graph shows the linear relationship we might expect from the Runge-Kutte method, behaving strangely once the error became smaller than the machine accuracy. We might then ask what the gradient of the linear portion of the graph is. Taking estimates from $h = 3.9 \times 10^{-4}$ to $h = 0.05$ we get an estimate for the gradient of 3.97 for p=6, and 4.01 for p=7, which is consistent with the theoretical fourth-order accuracy of the Runge-Kutta method.

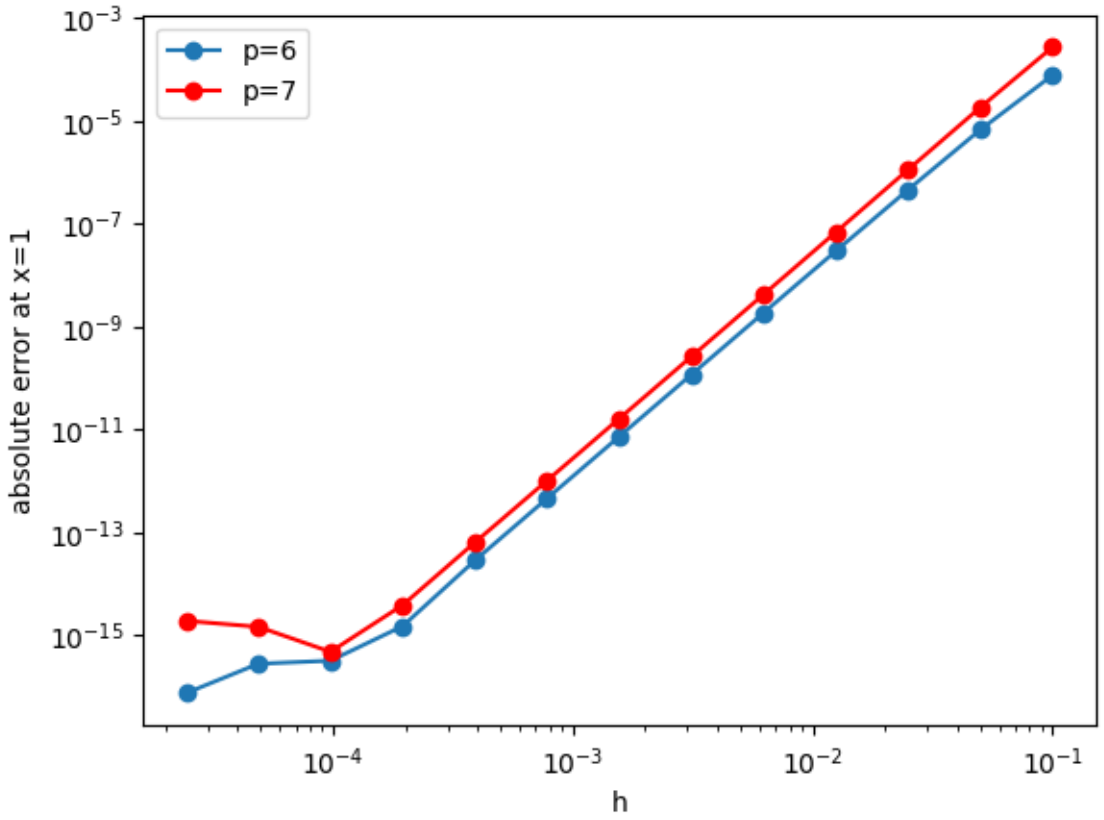
Table 10: Evaluation at x=1 for p=6

k	h	Y_n	$Y_n - y(1)$
0	0.1	0.047118	7.8172×10^{-5}
1	0.05	0.47047	6.8166×10^{-6}
2	0.025	0.47040	4.6574×10^{-7}
3	0.0125	0.47040	2.9972×10^{-8}
4	0.00625	0.47040	1.8941×10^{-9}
5	0.003125	0.47040	1.1893×10^{-10}
6	0.0015625	0.47040	7.4491×10^{-12}
7	7.8125×10^{-4}	0.47040	4.6565×10^{-13}
8	3.9063×10^{-4}	0.47040	2.8977×10^{-14}
9	1.9531×10^{-4}	0.47040	1.5335×10^{-15}
10	9.7656×10^{-5}	0.47040	-3.2613×10^{-16}
11	4.8828×10^{-5}	0.47040	2.8449×10^{-16}
12	2.4414×10^{-5}	0.47040	-7.6328×10^{-17}

Table 11: Evaluation at x=1 for p=7

k	h	Y_n	$Y_n - y(1)$
0	0.1	-0.99950	2.7401×10^{-4}
1	0.05	-0.10021	1.8526×10^{-5}
2	0.025	-0.10022	1.1413×10^{-6}
3	0.0125	-0.10022	6.9846×10^{-8}
4	0.00625	-0.10022	4.3039×10^{-9}
5	0.003125	-0.10022	2.6684×10^{-10}
6	0.0015625	-0.10022	1.6606×10^{-11}
7	7.8125×10^{-4}	-0.10022	1.0352×10^{-12}
8	3.9063×10^{-4}	-0.10022	6.4893×10^{-14}
9	1.9531×10^{-4}	-0.10022	3.9413×10^{-15}
10	9.7656×10^{-5}	-0.10022	4.8572×10^{-16}
11	4.8828×10^{-5}	-0.10022	1.4849×10^{-15}
12	2.4414×10^{-5}	-0.10022	1.9429×10^{-15}

Figure 4: Variation of error with value of h in RK4 method

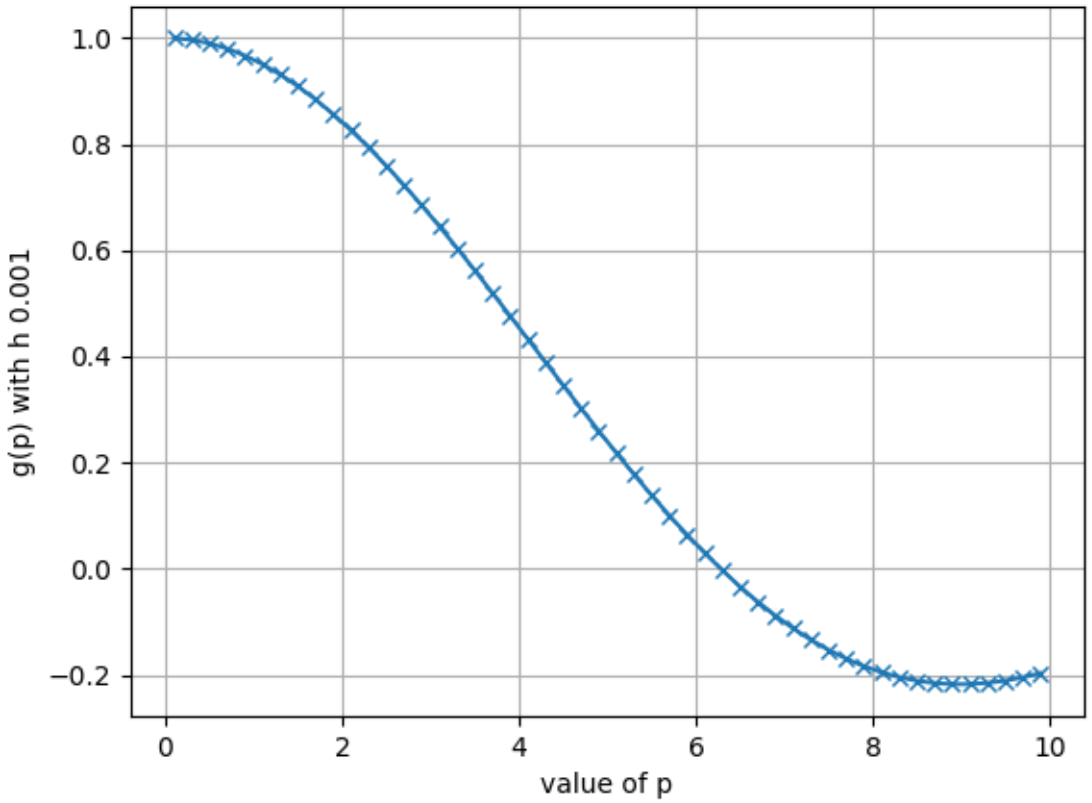


8 Question 8

First we note that if a non-zero solution y to 13a-13b exists for a given p then, since $\frac{dy}{dx}(0) = 0$ will yield a zero solution we have $\frac{dy}{dx} = \beta$. We can then see that $\frac{1}{\beta}y$ is also a solution and satisfies the initial conditions we are testing. Therefore the values of p for which solutions to 15 are zero at 1 are exactly the eigenvalues of 13a-13b.

First let f(p) be the analytic value of y(1) for parameter p, where y(x) is the solution to equations 13a) and 15, and let g(p) be the Runge-Kutta approximation at x=1 with parameter p. We would firstly like to know how small f(p) needs to be to guarantee that p is satisfactorily close to the root, and then how small h and ϵ need to be to guarantee f is that small. Next I graphed the values of RK4 with h=0.001 for a variety of values of p before and after 6 in figure 5.

Figure 5: variation of g(p) with p



Even if we are unsure about the precise accuracy of RK4, we can be sure from this graph that any root we find between 6 and 7 is in fact the smallest non-negative root.

Next we would like a lower bound on the magnitude of $\frac{df}{dp}$ between 6 and 7. With the knowledge that for $h = \frac{0.1}{2^4}$ RK4 was accurate to well within 10^{-8} at both 6 and 7 we can assume it will be that accurate between them. So for $x \in [6,7]$ $|\frac{g(x)-g(x+\delta)}{\delta}| < |\frac{f(x)-f(x+\delta)}{\delta}| + \frac{2\times10^{-8}}{\delta}$. So taking $\delta = 0.001$ we get an accuracy of 2×10^{-5}, I then calculated the value of $\frac{g(0.001n+6)-g(0.001(n+1)+6)}{0.001}$ for $0 \leq n \leq 999$ using the code is in section 10.4.17, the maximum value was -0.11949 and the minimum was -0.17282, so we can very safely assume $\frac{df}{dp} < -0.1$, and that therefore if $p*$ is the root of f, $f(p*+\epsilon) \approx \epsilon\frac{df}{dp} < -0.1\epsilon \implies |10f(p*+\epsilon)| > |\epsilon|$, so if we find p st $|f(p)| < 5\times10^{-7}$, then $|p-p*| < 5\times10^{-6}$ as desired.

So let us use $\epsilon = 4\times10^{-7}$ and $h = \frac{0.1}{2^4}$. From before we have with this value of h, the error in the RK4 method is less than 10^{-8} so for $|g(p)| < \epsilon$, $|f(p)| < |g(p)| + 10^{-8} < 4\times10^{-7} + 10^{-8} < 5\times10^{-7}$ with some room for error. You can find the code I used for the false position method [here], and the results in table 12.

Table 12: Iterates with h=0.00625 and $\epsilon = 4\times10^{-7}$

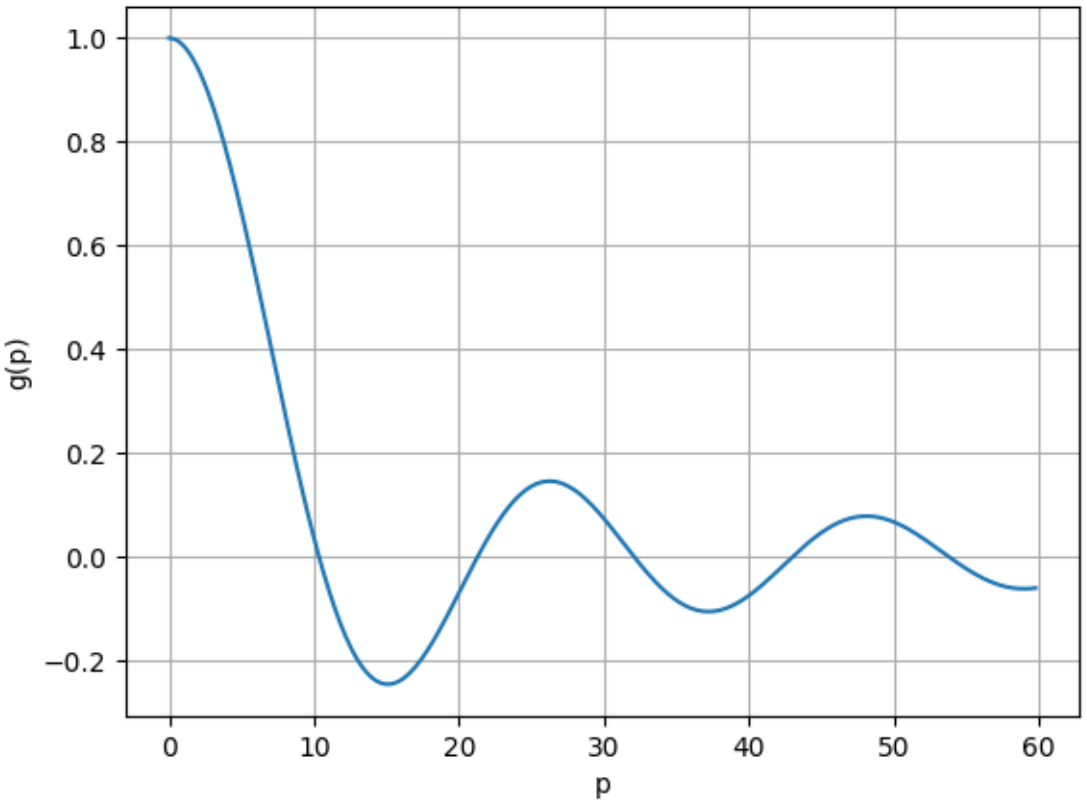
N	p	$\lvert p-2\pi\rvert$
1	6.3194	3.6241×10^{-2}
2	6.2847	1.5318×10^{-3}
3	6.2832	6.4181×10^{-5}
4	6.2832	2.7026×10^{-6}
5	6.2832	1.2834×10^{-7}

9 Question 9

As we worked out before, the question is essentially to find the first five roots of f(p), the solution at x=1 to equation 13a-15 with $\alpha = 8$. Firstly I roughly the graphed the RK4 estimate to understand the nature of the

roots, which you can see in figure 6.

Figure 6: Estimated value value of f at x=1 for various p



From this it was possible to work out that g does not have any double roots for the first five eigenvalues, so it was possible to look for locations of roots by calculating spaced out values of g(p) and looking for where they swapped signs, and then use the false position method from earlier. Also from the graph it was possible to infer that a spacing of 1 would be adequate to identify roots. The next question is what value of h, and what error, is necessary to ensure the root is found to sufficient accuracy.

Without knowing the true solution it is difficult to know exactly how inaccurate the estimate of g is for a given h, however the limits of my computer mean any value smaller than 0.0001 cannot be run in a reasonable amount of time. The evidence from $\alpha = 4$ suggest this will almost certainly exceed the limits of floating point arithmetic in accuracy, but it still takes too long to be used for the initial 'searching for sign change' period, so I used two values of h, 0.01 for the 'searching for sign change' phase, and 0.0001 for the false position method search.

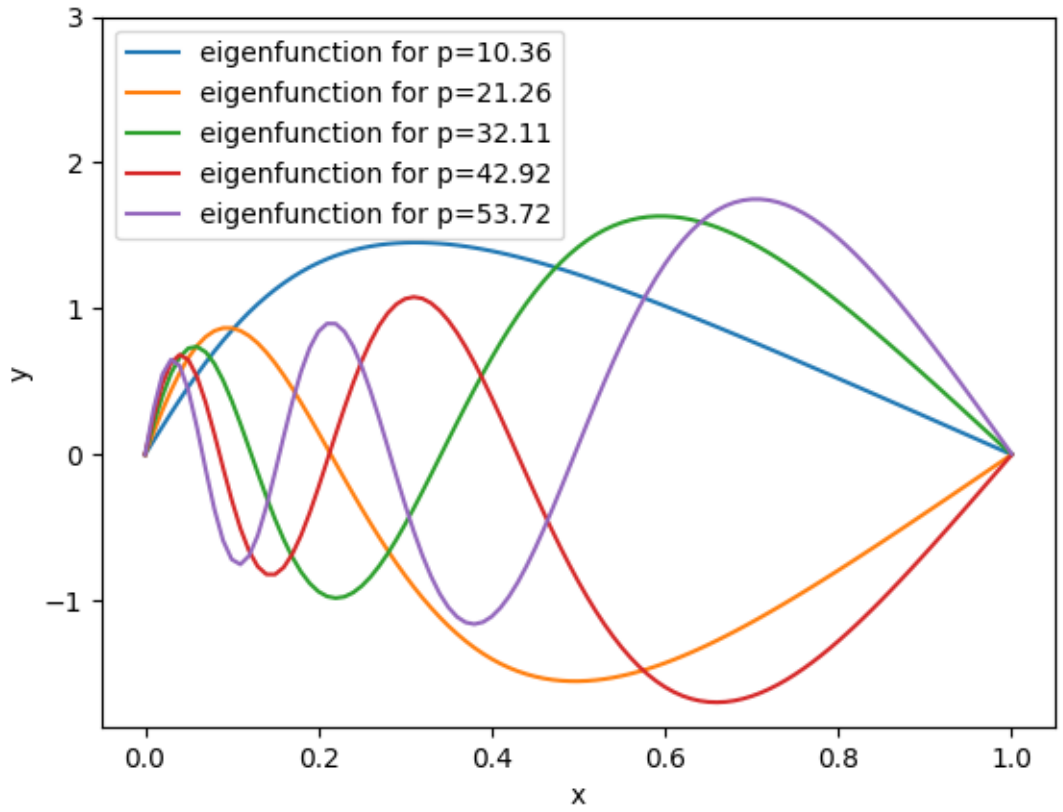
For the error in the false position search I used an estimate of the gradient by working out the gradient between the endpoints (that were found to have alternate sign), and then using the fact from Question 8 that $f(p * +\epsilon) \approx \epsilon \frac{dg}{dp}$, I also accounted for the potential slight inaccuracy of the gradient by dividing the result by ten, so the final formula for the error was $0.1 \times \frac{dg}{dp} \times 5 \times 10^{-6}$.

Running the program revealed that the first five eigenvalues were 10.35892127, 21.26392963, 32.10538534, 42.91933634, 53.71911122. Finding the RK4 solution to equation 13a-15 with p each of these values, and normalising the solution, gave the eigenfunctions, plotted in figure 7.

All eigenfunctions are roughly wave-like, growing larger as x gets larger. The first eigenfunction has no distinct roots between 0 and 1, the second has one distinct root, the third two etc. This makes sense as in general distinct eigenfunctions have distinct numbers of roots, since they are orthogonal.

Physically, the waves reflect the fact that this is an oscillating string, and the change in shape the mass distribution. This is because a small movement near x=0, where the string is heavy, takes the same energy as a large movement near x=1. The increased number of waves for larger p is explained exactly by equation 13a - the larger p is, the greater the vertical acceleration of the string, and therefore the greater the frequency of the vibrations.

Figure 7: Estimation of first five eigenfunctions with α = -8



10 Code listing

10.1 Overview

I did this project using Python 3.8.6 64 bit, and the package numpy version 1.19.3, which adds many MATLAB functions and structures to Python. The first part of my code imported a varierty of python packages that I would use in the code.

```
import numpy as np
import math
import matplotlib.pyplot as plt
import time
```

10.2 Functions on x,y

10.2.1 Function 5a

```
def func5a(x,y):
    output = -4*y+4*np.exp(-2*x)
    return output
```

10.2.2 Function 6

```
def func5aTruth(x):
    return (-2*np.exp(-4*x)+2*np.exp(-2*x))
```

10.2.3 Function 8a

```
def func8a(x,y):
    output = 4*y-5*np.exp(-x)
```

10.2.4 Function 8c

```
def func8aTruth(x):
    return np.exp(-x)
```

10.2.5 Function 16

```
def func16(x,y,p):
    output=np.zeros(2)
    output[0]=y[1]
    output[1] = -p**2*(1+x)**(-8)*y[0]

    return output
```

10.2.6 Function 16 analytic solution

```
def func16Truth(x,p):
    return -(1/p)*(1+x)*np.sin(p*(1+x)**(-1)-p)
```

10.3 Method Functions

These functions performed the various algorithms throughout the project. All values in tables were obtained by callind the function with the given initial conditions.

10.3.1 Forward Euler

```
def forwardEuler(startx,starty,steps,interval,fxy,truth = None, log = False):
    #initialise array with correct start values
    output = np.zeros((2,steps+1))
    output[0][0]=startx
    output[1][0]=starty
    for i in range(1,steps+1):
        #use the forward euler method iteratively
        output[0][i]=startx+i*interval
        output[1][i]=output[1][i-1]+interval*fxy(output[0][i-1],output[1][i-1])
    if truth == None:
        pass
    else:
        #This section adds an array of true values and errors to the output
        correctOutput=np.zeros((2,steps+1))
        correctOutput[0][0]=startx
        for i in range(1,steps+1):
            correctOutput[0][i]=startx+i*interval
        correctOutput[1]=truth(correctOutput[0])
        vec = output[1]-correctOutput[1]

        matrix = np.vstack((correctOutput[1],vec))

        output =np.vstack((output,matrix))
        if log:
            #adds the log of the errors to the output
            logged = np.zeros((1,steps+1))
            for i in range(steps+1):
```

```
            if vec[i]==0:
                logged[0,i]=13

            elif i ==0:
                pass
            else:
                logged[0,i]=np.log(abs(vec[i]))/(i*interval)
        output = np.vstack((output,logged))


    return output
```

Values in table for Q1 came from calling this function with the g

10.3.2 Fourth order Runge-Kutta

```
def rungeKutta(startx,starty,steps,interval,fxy):
    #initialise correctly sized array with correct initial values
    output = np.zeros((2,steps+1))
    output[0][0]=startx
    output[1][0]=starty
    for i in range(1,steps+1):
        #perform Runge-Kutta iteration
        output[0][i]=startx+i*interval
        xn=output[0][i-1]
        yn = output[1][i-1]
        k1 = interval*fxy(xn,yn)
        k2 = interval*fxy(xn+(1/2)*interval,yn+(1/2)*k1)
        k3 = interval*fxy(xn+(1/2)*interval,yn+(1/2)*k2)
        k4 = interval*fxy(xn+interval,yn+k3)
        output[1][i]=yn+(1/6)*(k1+2*k2+2*k3+k4)
    return output
```

10.3.3 2-dimensional Runge-Kutta

```
def rungeKuttaMulti(startx,starty,steps,interval,dimension,fxy,p,truth = None):
    #almost the same as Runge-Kutta with slight modifications to include
    #a dimension argument
    xoutput = np.zeros((steps +1))
    youtput = np.zeros((steps+1,dimension))
    xoutput[0]=startx
    youtput[0]=starty
    for i in range(1,steps+1):
        xoutput[i]=startx+i*interval
        xn=xoutput[i-1]
        yn = youtput[i-1]
        k1 = interval*fxy(xn,yn,p)
        k2 = interval*fxy(xn+(1/2)*interval,yn+(1/2)*k1,p)
        k3 = interval*fxy(xn+(1/2)*interval,yn+(1/2)*k2,p)
        k4 = interval*fxy(xn+interval,yn+k3,p)
        youtput[i]=yn+(1/6)*(k1+2*k2+2*k3+k4)
    if truth == None:
        return [xoutput,youtput]
    else:
        #this section is copied from the forwardEuler function
        #It adds the correct values and errors to the output
        correctOutput=np.zeros((2,steps+1))
        correctOutput[0][0]=startx
        for i in range(1,steps+1):
            correctOutput[0][i]=startx+i*interval
        correctOutput[1]=truth(correctOutput[0],p)
```

```
        vec = youtput[:,0]-correctOutput[1]

        matrix = np.vstack((correctOutput[1],vec))

        return [xoutput,youtput,matrix]
    return [xoutput,youtput]
```

For use in question 8 and 9 I also added a function to perform an RK4 estimate with initial conditions (15) with differing values of h and p.

```
def RK4estimate(p,n):
    return rungeKuttaMulti(0,np.array([0,1]),n,(1/n),2,func16,p)[1][n][0]
```

10.3.4 False Position Method

```
def falsePosition(left,right,func,error,n):
    #starts by evaluating each boundary
    lefteval = func(left,n)
    righteval = func(right,n)
    iterates = []
    for i in range(100):
        #If more than 100 iterations have taken place stop
        #then check if either boundary is within the given error
        if abs(lefteval)<error:

            iterates.append(left)
            return(iterates)
        elif abs(righteval)<error:

            iterates.append(right)
            return(iterates)
        else:
            #calculate the 'middle' value p*
            middle = (righteval*left - lefteval*right)/(righteval-lefteval)
            iterates.append([middle,abs(middle-2*np.pi)])
            middleeval = func(middle,n)
            if middleeval*lefteval<0:
                #if sign difference is with left side set right boundary
                #to the middle value
                right = middle
                righteval = middleeval
            else:
                #otherwise set left boundary to middle value
                left = middle
                lefteval = middleeval
```

10.3.5 Eigenvalue Finding Algorithm

```
def eigenvalueFinder(func,error,number,spacing,roughn,precisen):
    #timer to ensure program does not run for too long
    start_time = time.time()
    finished = False
    #initialise array for values where sign changes
    lefts = np.zeros((number,2))
    #set initial left boundary to start point
    left = spacing
    lefteval = func(left,roughn)
    count = 0
    while not finished:
        #calculate right boundary and evaluate function there
        right = left+spacing
```

```
        righteval = func(right,roughn)
        if lefteval*righteval<=0:
            #check whether there is sign change
            #if yes, add left value to array
            grad = abs((righteval-lefteval)/spacing)
            lefts[count,0]=left
            lefts[count,1]=grad
            count+=1
            if count >=number:
                #check whether we have reached five
                #sign changes
                finished = True
            else:

                pass
        else:
            #otherwise keep looking
            pass
        #then make new left boundary previous
        #right boundary
        left = right
        lefteval = righteval

        if time.time() - start_time > 30:
            finished = True
        else:
            pass
    #now we have sign changes, perform false position search
    #with boundary at the values with the opposite sign
    results = np.zeros((5))
    for i in range(5):
        results[i] = falsePosition(lefts[i,0],lefts[i,0]+spacing,func,
        error*(0.1*lefts[i,1]),precisen)[-1]
    return results
```

10.4 Output used in Answer

10.4.1 Table 1

```
print(forwardEuler(0,0,6,2,func5a,truth = func5aTruth,log=True)
```

10.4.2 Table 2

```
print(forwardEuler(0,0,12,1,func5a,truth = func5aTruth,log=True)
```

10.4.3 Table 3

```
print(forwardEuler(0,0,20,0.6,func5a,truth = func5aTruth,log=True)
```

10.4.4 Table 4

```
print(forwardEuler(0,0,24,0.5,func5a,truth = func5aTruth,log=True)
```

10.4.5 Table 5

```
print(forwardEuler(0,0,30,0.4,func5a,truth = func5aTruth,log=True)
```

10.4.6 Table 6

```
print(forwardEuler(0,0,60,0.2,func5a,truth = func5aTruth,log=True)
```

10.4.7 Figure 1

```
xaxis =np.linspace(0.4,2.5)
yaxis = np.divide(np.log(4*xaxis-1),xaxis)
plt.plot(xaxis,yaxis,label = 'value of ln(4h-1)/h')
plt.plot([0.4,0.5,0.6,1,2],[-1.242,0.032,0.589,1.123,0.984],'o',color =
'black',label='observed value of ln|\u03B5|/x at x=12')
plt.xlabel('h')
plt.grid()
plt.legend(loc='lower right')
plt.show()
```

10.4.8 Figure 2

```
results1 = forwardEuler(0,0,15,0.4,func5a)
results2 = rungeKutta(0,0,15,0.4,func5a)
xaxis = np.linspace(0,6,1000)
yaxis = func5aTruth(xaxis)
plt.plot(xaxis,yaxis,color = 'black', label = 'true solution')
plt.plot(results1[0],results1[1],'--',color = 'red',
marker = 'o',label = 'forward Euler')
plt.plot(results2[0],results2[1],'--',color = 'blue',
marker = 'o', label = 'RK4')
plt.legend()
plt.xlabel('x')
plt.ylabel('y')
print(results2)
plt.show()
```

10.4.9 Tables 7 and 8, Figure 3

```
xaxis = np.zeros(16)
euleryaxis=np.zeros(16)
rkyaxis = np.zeros(16)
correct = func5aTruth(1.6)
for i in range(16):
    h=1.6/(2**(15-i))
    xaxis[i]=h
    euleryaxis[i]=abs(forwardEuler(0,0,2**(15-i),h,func5a)[1,-1]-correct)
    rkyaxis[i]=abs(rungeKutta(0,0,2**(15-i),h,func5a)[1,-1]-correct)
print(xaxis)
print(euleryaxis)
print(rkyaxis)
plt.loglog(xaxis,euleryaxis,marker='o', label = 'Euler method')
plt.loglog(xaxis,rkyaxis,marker='o',color = 'red', label = 'RK4 method')
plt.xlabel('h')
plt.ylabel('error at x=1.6')
plt.legend(loc = 'upper left')
plt.show()
```

10.4.10 Table 9

```
print(forwardEuler(0,1,10000,0.001,func8a,truth = func8aTruth,log=True)[:,:5])
```

10.4.11 Tables 10 and 11, Figure 4

```
for i in range(13):
    h = 0.1/(2**i)
    steps = 10*(2**i)
    values = rungeKuttaMulti(0,np.array([0,1]),steps,h,2,
    func16,6,truth = func16Truth)
    print(h,values[1][steps][0],values[2][1][steps])
for i in range(13):
    h = 0.1/(2**i)
    steps = 10*(2**i)
    values = rungeKuttaMulti(0,np.array([0,1]),steps,h,2,
    func16,7,truth = func16Truth)
    print(h,values[1][steps][0],values[2][1][steps])
xaxis = []
yaxis1 = []
yaxis2 = []
for i in range(13):
    h = 0.1/(2**i)
    steps = 10*(2**i)
    values6 = rungeKuttaMulti(0,np.array([0,1]),steps,h,2,
    func16,6,truth = func16Truth)
    values7 = rungeKuttaMulti(0,np.array([0,1]),steps,h,2,
    func16,7,truth = func16Truth)
    xaxis.append(h)
    yaxis1.append(abs(values6[2][1][steps]))
    yaxis2.append(abs(values7[2][1][steps]))
plt.loglog(xaxis,yaxis1,marker='o',label = 'p=6')
plt.loglog(xaxis,yaxis2,marker='o',color='red',label = 'p=7')
plt.xlabel('h')
plt.ylabel('absolute error at x=1')
plt.legend(loc = 'upper left')
plt.show()
```

10.4.12 Figure 5

```
xaxis = []
yaxis = []
for i in range(50):
    p = 0.1+i/5
    xaxis.append(p)
    y = rungeKuttaMulti(0,np.array([0,1]),1000,0.001,2,
    func16,p)[1][1000][0]
    yaxis.append(y)
plt.plot(xaxis,yaxis,marker = 'x')
plt.xlabel('value of p')
plt.ylabel('g(p) with h 0.001')
plt.grid()
plt.show()
```

10.4.13 Table 12

```
print(falsePosition(6,7,RK4estimate,4*10**(-7),160))
```

10.4.14 Figure 6

```
xaxis = []
yaxis = []
for i in range(300):
```

```
    xaxis.append(i*0.2)
    yaxis.append(RK4estimate(i*0.2,40))
plt.plot(xaxis,yaxis)
plt.xlabel('p')
plt.ylabel('g(p)')
plt.grid()
plt.show()
```

10.4.15 First five eigenvalues

```
eigenvalues = eigenvalueFinder(RK4estimate,5*10**(-6),5,1,100,10000)
print(eigenvalues)
```

10.4.16 Five normalised eigenfunctions

```
eigenvalues = eigenvalueFinder(RK4estimate,5*10**(-6),5,1,100,10000)
print(eigenvalues)
print('found values, doing graphs now')
xaxis = np.zeros((101))
for i in range(101):
    x=i*0.01
    xaxis[i]=x
for i in range(5):
   yaxis=rungeKuttaMulti(0,np.array([0,1]),100,
   0.01,2,func16,eigenvalues[i])[1][:,0]
   squared = np.square(yaxis)
   integral = np.trapz(squared,x=xaxis)
   root = np.sqrt(integral)
   yaxis = yaxis*(1/root)
   plt.plot(xaxis,yaxis,label = 'eigenfunction for p='+"{:.2f}".format(eigenvalues[i]))
   print('done ',i)
plt.xlabel('x')
plt.ylabel('y')
plt.legend(loc = 'upper left')
plt.ylim(top = 3)
plt.show()
```

10.4.17 Maximum and Minimum values of g

```
yvalues = []
y1 = rungeKuttaMulti(0,np.array([0,1]),160,0.00625,2,
func16,6)[1][160][0]
for i in range(1000):
    p = 6+0.001*i
    y2 = rungeKuttaMulti(0,np.array([0,1]),160,0.00625,2,
    func16,p+0.001)[1][160][0]
    yvalues.append((y2-y1)*1000)
    y1=y2
print(max(yvalues),min(yvalues))
```