Catam 1.2 Ordinary Differential Equations

December 2022

1 Question 1

We used the program "AdamsBashforth.py", found on page 16, to compute results shown in tables starting on page 12. By consulting the results of the first such table, $h = 0.5$, we can calculate that the magnitude of the error tends to about 5.3723 for each increment to n.

Suppose that $E \propto e^{\gamma x}$

$$x = nh \implies E_n \propto e^{\gamma n/2}$$

$$\implies 5.37228 = e^{\gamma((n+1)-n)/2}$$

$$\implies \gamma = 2log(5.37228) \approx 3.3625$$

Looking through the rest we can clearly see that for $h > 0.125$ our error diverges, $h = 0.125$ the error remains at approximately the same magnitude $h < 0.125$, the error converges to 0. These mean in turn the value of the growth rate is positive, zero and negative over these values, and the computed output is unstable, metastable and stable.

2 Question 2

Finding the analytic solution we first get complimentary solutions of the form

$$Y_n = \alpha^n,\ \beta^n$$
$$\alpha = \frac{1 - 12h + \sqrt{1 - 8h + 144h^2}}{2},\ \beta = \frac{1 - 12h - \sqrt{1 - 8h + 144h^2}}{2}$$

Looking for a particular solution we search for one of the form ke^{-2hn}. Which yields

$$k = \frac{3h(3e^{-2h} - 1)}{e^{-4h} + (12h - 1)e^{-2h} - 4h}$$

Finally the final solution subject to background conditions should be

$$Y_n = -\frac{6h + k\beta - ke^{-2h}}{\beta - \alpha}\alpha^n + \frac{6h + k\alpha - ke^{-2h}}{\beta - \alpha}\beta^n + ke^{-2hn}$$

For instability to occur we need $|\alpha|$ or $|\beta| > 1$. As the instability is oscillatory we need $\alpha, \beta < -1$. Trivially $\beta < \alpha$ so we only consider β. This gives us after a bit of algebra $h = 1/8$ as the critical value which is precisely reflected in results from Question 1.

From this point the relevant term is $|\alpha^n| = e^{ln|\alpha|n} = e^{x(ln|\alpha|/h)}$ so has a growth rate of $ln|\alpha|/h$, which lines up precisely with expectations from Question 1.

Fixing $x = x_n = nh$ and taking $h \to 0$.

$$\alpha \to 1,\ \beta \to 0,\ k \to 1$$

We then can simplify to

$$e^{-2x} - e^{xln(\alpha)/h},\ \frac{ln(\alpha)}{h} \to -8,\ y(x) = e^{-2x} - e^{-8x}$$

If a more accurate method was used to compute Y_1, we note that α, β, k would all remain the same as they are independent of the initial conditions, all that will change are the coefficients of the α, β terms. Specifically

$$Y_n = -\frac{f(h) + k\beta - ke^{-2h}}{\beta - \alpha}\alpha^n + \frac{f(h) + k\alpha - ke^{-2h}}{\beta - \alpha}\beta^n + ke^{-2hn}$$

However, the coefficients are irrelevant in a discussion of growth rate, and the point of instability since it is entirely dependent on α. The only exception being if the coefficient is 0. But as we are attempting to approximate a function which is clearly non-zero at the point h, a better algorithm such as RK4 will not change the overall impact.

3 Question 3

We shall try each algorithm in turn providing a table of results and a plot $h = 0.08$, results are tabulated in the "Tables" section. The programs used are called "ForwardEuler.py", "AdamsBashforth.py" and "RungeKutta.py" found starting on page 15.

Figure 1: Forward Euler and Adams Bashforth

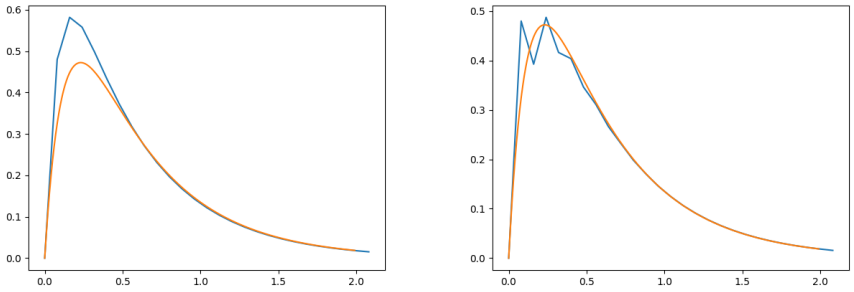
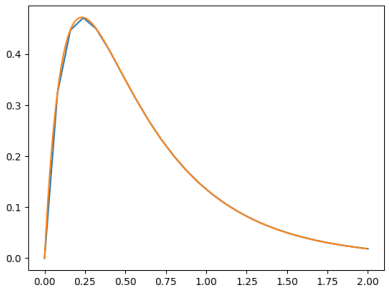


Figure 2: Runge Kutta



The blue plots are the values of the computation in each method, whereas the orange plot is the true solution.

4 Question 4

The algorithms "ErrorPlotEuler.py", "ErrorPlotAdams.py" and "ErrorPlotRunge.py" were used to generate these plots, they can be found starting on page 18.

Figure 3: Forward Euler and Adams Bashforth

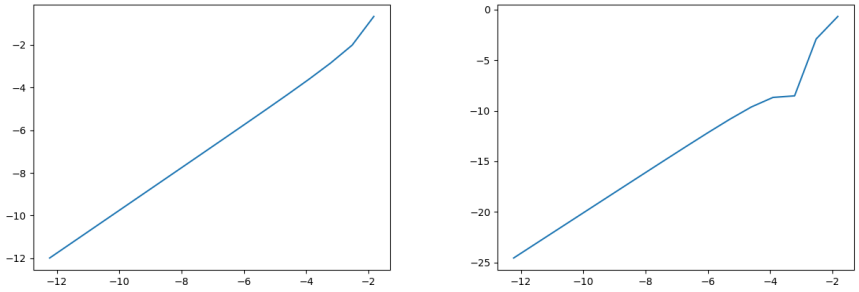
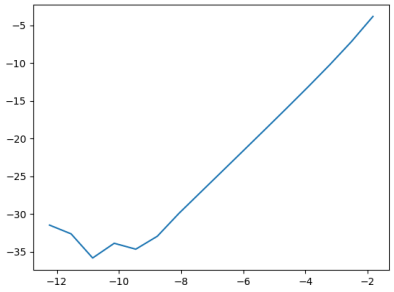


Figure 4: Runge Kutta



By a quick inspection of each graph we can see see that the gradients of each section with no significant deviations is about $1, 2$ and 4. This in turn tells us that $|E_n| \propto h, h^2$ and h^4. In Forward Euler, Adams Bashforth and Runge Kutta respectively.

This precisely lines up with our expectations of the algorithms, as we know that Forward Euler, Adams Bashforth and Runge Kutta are 1st, 2nd and 4th order accurate methods.

5 Question 5

$$\frac{d^2y}{dx^2} + p^2(1+x)^{-2}y = 0$$

Use substitution $e^z = 1 + x$

$$\frac{d^2y}{dz^2} - \frac{dy}{dz} + p^2 y = 0$$

Solving equation

$$y = Ae^{\lambda_1 z} + Be^{\lambda_2 z}, \lambda = \frac{1 \pm \sqrt{1-4p^2}}{2}$$

Returning to x

$$y(x) = A(1+x)^{\lambda_1} + B(1+x)^{\lambda_2}$$

Imposing first initial conditions

$$A = -B$$
$$A = \frac{1}{\lambda_1 - \lambda_2} = \frac{1}{\sqrt{1-4p^2}}$$

Thus final solution is

$$y(x) = \frac{1}{\sqrt{1-4p^2}}((1+x)^{\lambda_1} - (1+x)^{\lambda_2})$$

Returning to the problem but with boundary conditions $y(0) = y(1) = 0$. First lets rule out $p = 1/2$ as a solution.

$$y = (A + Bx)e^{x/2}$$

Boundary conditions give

$$A = 0, B = 0$$

Returning to general solution

$$y(x) = A(1+x)^{\lambda_1} + B(1+x)^{\lambda_2}$$

First boundary condition gives $A = -B$ and the second gives

$$A(2^{\lambda_1} - 2^{\lambda_2})$$

Real solutions of lambda are only valid if $\lambda_1 = \lambda_2$, but that case is already discussed. hence only complex solutions are valid. We then get

$$2^{i\sqrt{4p^2-1}\ /2} - 2^{-i\sqrt{4p^2-1}\ /2} = 0$$

$$ln(2)\frac{\sqrt{4p^2-1}}{2} = ln(2)\frac{-\sqrt{4p^2-1}}{2} + 2\pi n,\ n > 0$$

$$p_n = \sqrt{\frac{1}{4} + (\frac{\pi n}{ln(2)})^2}$$

Obviously the minimal solution is where $n = 1$.
Eigenfunctions are of the form as written above

$$y_n(x) = A((1+x)^{\lambda_1} - (1+x)^{\lambda_2})$$

$$y_n(x) = \frac{2(1+x)^{1/2}}{\sqrt{4p^2-1}} sin(\frac{ln(1+x)\sqrt{4p^2-1}}{2})$$

6 Question 6

A program called "SecondOrderRK4.py" has been written and can be found on page 22 to perform the task.

k	h	Y_n^k	E_n^k	E_n^{k-1}/E_n^k
0	0.1	0.13576482860337877	3.81503056832988e-05	N/A
1	0.05	0.13572944712824248	2.7688305470063845e-06	13.778
2	0.025	0.13572686124314226	1.8294544679164915e-07	15.135
3	0.0125	0.13572669000485416	1.1707158686924402e-08	15.627
4	0.00625	0.13572667903734273	7.396472601062243e-10	15.8284
5	0.003125	0.13572667834416235	4.646688589460268e-11	15.918
6	0.0015625	0.13572667830060856	2.9130864387383326e-12	15.951
7	0.00078125	0.13572667829787793	1.8246515409714448e-13	15.965
8	0.000390625	0.13572667829770108	5.606626274357040 5e-15	32.545
9	0.0001953125	0.13572667829769408	-1.3877787807814457e-15	-4.04
10	9.765625e-05	0.13572667829771873	2.325917236589703e-14	-0.060
11	4.8828125e-05	0.13572667829770346	7.993605777301127e-15	2.910
12	2.44140625e-05	0.13572667829760063	-9.4840801878604e-14	-0.084

Table 1: $p = 4$

k	h	Y_n^k	E_n^k	E_n^{k-1}/E_n^k
0	0.1	-0.0856990642561456	0.0001446392846319866	N/A
1	0.05	-0.08583461818041246	9.085360365118644e-06	15.920
2	0.025	-0.08584314406303958	5.594777379991367e-07	16.239
3	0.0125	-0.08584366898613824	3.455463934431968e-08	16.191
4	0.00625	-0.08584370139635168	2.1444259007408206e-09	16.114
5	0.003125	-0.08584370340726369	1.3351389438476247e-10	16.061
6	0.0015625	-0.08584370353244843	8.329156808706273e-12	16.030
7	0.00078125	-0.08584370354025671	5.208750097907e-13	15.991
8	0.000390625	-0.08584370354074913	2.8449465006019636e-14	18.309
9	0.0001953125	-0.08584370354077916	-1.582067810090848e-15	-17.982
10	9.765625e-05	-0.08584370354075946	1.812439087700568e-14	-0.087
11	4.8828125e-05	-0.08584370354076325	1.4335754805472334e-14	1.2642
12	2.44140625e-05	-0.08584370354084657	-6.898648319264566e-14	-0.208

Table 2: $p = 5$

It can be seen that the global error is divided by 16 (until the numbers are so small that computational errors creep in) each time h is halved. This perfectly matches prior knowledge that RK4 is a fourth order convergent algorithm.

7 Question 7

The program to compute the eigenvalues of our equation is called "FalsePosition.py" and can be found on page 24.

First of all we know the true value of all eigenvalues (See Question 5) and it can be quickly justified that there is only one in the interval $(4, 5)$, that being the minimal eigenvalue.

Tabulated below are the results of the program in finding the root. Values of 2^{-10} and 10^{-7} may be chosen for h, ϵ respectively to ensure a result within $5 * 10^{-6}$ of the true value.

To explain this note we have two functions and two variables, ϕ, ϕ_c, and p, p_c. ϕ, and p are the pure mathematical objects being the true function evaluated at 1, and the exact eigenvalue. ϕ_c and p_c are the computed values that the program produces.

From our analysis in Question 6 we note that:

$$|\phi_c(p_c) - \phi(p_c)| = E_n < Ch^4 < h^3$$

This holds for small h, from the results in question 6 we can approximate that $C < 2$, however bounding with h^3 is a much safer bound as we cannot be sure what C is for general functions. The termination condition for the function is:

$$|\phi_c(p_c)| < \epsilon \implies |\phi(p_c)| < h^3 + \epsilon$$

Now consider a Taylor expansion of ϕ.

$$\phi(p_c) = 0 + \phi'(p)|p_c - p| + o(|p_c - p|)$$

$$\implies |p_c - p| \leq 2\frac{|\phi(p_c)|}{|\phi'(p)|}$$

For small $|p_c - p|$. Now with a little analysis of the derivative of ϕ near p we get:

$$|p_c - p| \leq (2h^3 + 2\epsilon)\frac{\pi\sqrt{2}}{ln(2)}$$

$$|p_c - p| \leq (2^{-29} + 2(10^{-7}))\frac{\pi\sqrt{2}}{ln(2)}$$

$$|p_c - p| \leq 5 * 10^{-6}$$

One can also note that for derivatives near the eigenvalues with magnitudes that are not large, the same result still holds.

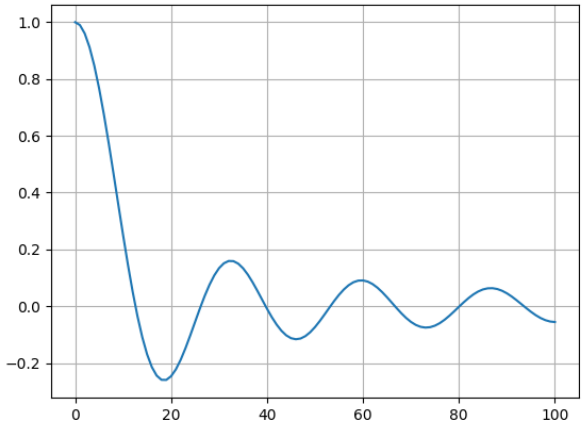
Lower Bound	Upper Bound	New Bound
4	5	4.612566883590025
4	4.612566883590025	4.565358345434106
4	4.565358345434106	4.560421032473599
4	4.560421032473599	4.559914073237854
4	4.559914073237854	4.559862119016884
4	4.559862119016884	4.559856795692291
4	4.559856795692291	4.559856250265735

Table 3: Finding Eigenvalue

8 Question 8

To begin this question first we must find approximations for each root. Using a program called "EightPlot.py", found on page 26, I graphed the value of $y(1)$ for each corresponding value of p. The value of $y(1)$ was computed using the RK4 algorithm.

Figure 5: Graph of $\phi(p)$ to p



Simply looking at the graph it is clear approximately where the first 5 roots lie, so we may assign intervals to search in. We also know the graph is accurate to within a very small deviation in the y-axis. So we can be sure that the ones we see are the minimal eigenvalues. Tabulated below are results for computations of the eigenvalues using a program called "FalsePosition8.py" found on page 28.

k	Interval	Eigenvalue
1	$[5, 20]$	12.576597865767088
2	$[20, 35]$	26.18277907407644
3	$[35, 45]$	39.71110441264036
4	$[45, 60]$	53.19890817262561
5	$[60, 77]$	66.66329793691702

Table 4: Eigenvalues

Finding the eigenvalues to an acceptable accuracy follows much the same method as before in question 7, we first note that at all points on the graph the magnitude of the derivative does not exceed 1, nor does it come close to 1. So with a choice of $h = 2^{-10}$, $\epsilon = 10^{-8}$ the error is comfortably below $5 * 10^{-6}$

A program called "NormalisedPlot.py" is used to plot these graphs, found on page 30.

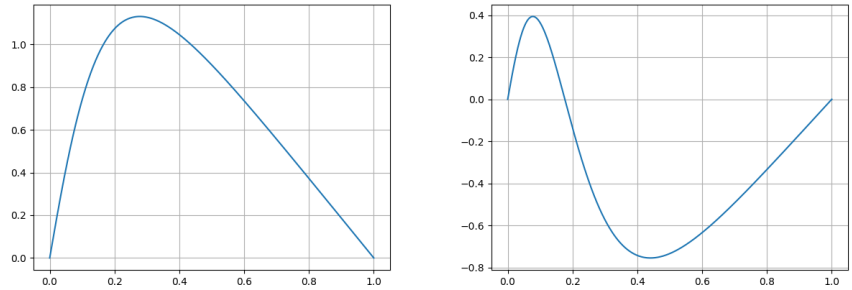


Figure 6: Eigenfunctions for $p = 12.57659786576708,\ 26.1827790740764$

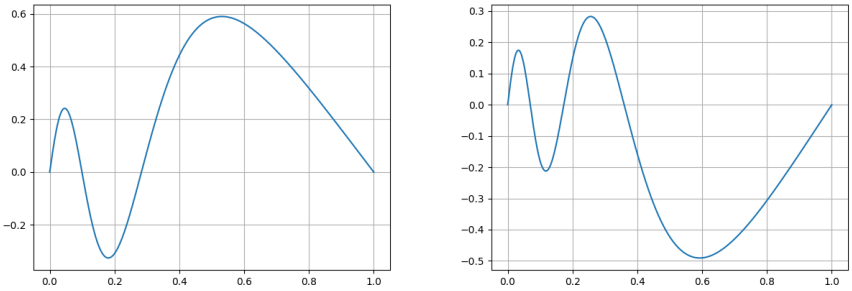


Figure 7: Eigenfunctions for $p = 39.7111044126403,\ 53.1989081726256$

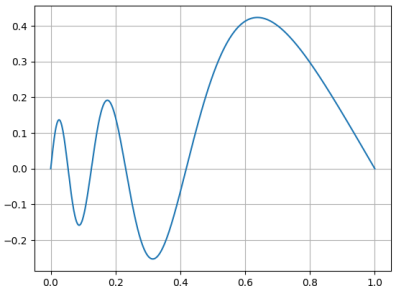


Figure 8: $p = 66.6632979369170$

9 Tables

n	x_n	$y_e(x_n)$	Y_n	E_n
0	0	0.0	0	0.0
1	0.5	0.34956380228270817	3.0	-2.650436197717292
2	1.0	0.13499982060871019	-14.84454251472851	14.97954233533722
3	1.5	0.04978092415551062	80.27990218645013	-80.23012126229462
4	2.0	0.018315526353559465	-431.06755707890716	431.0858726052607
5	2.5	0.006737944937931846	2315.9053295398835	-2315.8985915949456
6	3.0	0.002478752138915014	-12441.658914554071	12441.66139330621

Table 5: $h = 0.5$

n	x_n	$y_e(x_n)$	Y_n	E_n
0	0	0.0	0	0.0
1	0.375	0.4225794843731508	2.25	-1.8274205156268493
2	0.75	0.2206514079717635	-7.405762884499076	7.626414292470839
3	1.125	0.10527581475777767	29.516822014414075	-29.411546199656296
4	1.5	0.04978092415551062	-114.31282042446857	114.36260134862408
5	1.875	0.023517439953688612	444.4195617353706	-444.3960442954169
6	2.25	0.011108981308262565	-1726.9143347701497	1726.925443751458
7	2.625	0.0052475176409253425	6710.840549697807	-6710.835302180166
8	3.0	0.002478752138915014	-26078.308213344048	26078.310692096187

Table 6: $h = 0.375$

n	x_n	$y_e(x_n)$	Y_n	E_n
0	0	0.0	0	0.0
1	0.25	0.4711953764760207	1.5	-1.0288046235239792
2	0.5	0.34956380228270817	-2.3853060156465746	2.7348698179292827
3	0.75	0.2206514079717635	6.643442779144418	-6.4227913711726545
4	1.0	0.13499982060871019	-15.446058294480025	15.581058115088736
5	1.25	0.08203959869413631	37.67271613527552	-37.590676536581384
6	1.5	0.04978092415551062	-90.70830078055474	90.75808170471025
7	1.75	0.030196551893599405	219.1397748512448	-219.1095782993512
8	2.0	0.018315526353559465	-528.9572466716199	528.9755621979735
9	2.25	0.011108981308262565	1277.0728303444175	-1277.0617213631092
10	2.5	0.006737944937931846	-3083.0916488474104	3083.0983867923483
11	2.75	0.004086771159517259	7443.262956672583	-7443.258869901423
12	3.0	0.002478752138915014	-17969.61342041709	17969.61589916923

Table 7: $h = 0.25$

n	x_n	$y_e(x_n)$	Y_n	E_n
0	0	0.0	0	0.0
1	0.125	0.41092134189996254	0.75	-0.33907865810003746
2	0.25	0.4711953764760207	0.12615088095533045	0.3450444955206903
⋮	⋮	⋮	⋮	⋮
22	2.75	0.004086771159517259	-0.30804688139674685	0.31213365255626413
23	2.875	0.003182780693890872	0.31539269819780213	-0.31220991750391125
24	3.0	0.002478752138915014	-0.3096717006906251	0.3121504528295401

Table 8: $h = 0.125$

n	x_n	$y_e(x_n)$	Y_n	E_n
0	0	0.0	0	0.0
1	0.1	0.36940178896076026	0.6	-0.23059821103923983
2	0.2	0.4684235280409839	0.31685767777018364	0.15156585027080027
3	0.3	0.45809368280461393	0.5342972799546442	-0.07620359715003022
⋮	⋮	⋮	⋮	⋮
27	2.7	0.00451658052647292	0.004619994641380548	-0.00010341411490762814
28	2.8	0.003697863529499285	0.0036665308847209453	3.1332644778339896e-05
29	2.9	0.0030275546613586436	0.0030877947416588633	-6.024008030021963e-05
30	3	0.002478752138915007	0.002464493561449956	1.4258577465051103e-05

Table 9: $h = 0.1$

n	x_n	$y_e(x_n)$	Y_n	E_n
0	0	0.0	0	0.0
1	0.05	0.2345173720003202	0.30000000000000004	-0.06548262799967985
2	0.1	0.36940178896076026	0.37717683811618186	-0.007775049155421598
3	0.15	0.4396240087695158	0.44357396142617067	-0.003949952656654876
⋮	⋮	⋮	⋮	⋮
58	2.9	0.0030275546613586653	0.0030320468429054503	-4.492181546785021e-06
59	2.95	0.002739444762449994	0.00274350952445051 1	-4.064762000517176e-06
60	3	0.002478752138915027	0.002482430135000693	-3.67799608566588e-06

Table 10: $h = 0.05$

9.1 Question 3

n	x_n	$y_e(x_n)$	Y_n
0	0	0.0	0
1	0.08	0.3248513649231628	0.48
2	0.16	0.4481117366204968	0.5818290187037815
⋮	⋮	⋮	⋮
23	1.84	0.025222570086747276	0.02460059064527569
24	1.92	0.021493387924282795	0.0209632405532083
25	2	0.018315526353559448	0.01786369524479814

Table 11: Forward Euler

n	x_n	$y_e(x_n)$	Y_n
0	0	0.0	0
1	0.08	0.3248513649231628	0.48
2	0.16	0.4481117366204968	0.39274352805567214
⋮	⋮	⋮	⋮
23	1.84	0.025222570086747276	0.025319744035809732
24	1.92	0.021493387924282795	0.0215768227865756
25	2	0.018315526353559448	0.01838727001093234

Table 12: Adams Bashforth

n	x_n	$y_e(x_n)$	Y_n
0	0	0.0	0
1	0.08	0.3248513649231628	0.3241070783102275
2	0.16	0.4481117366204968	0.4473467299405445
⋮	⋮	⋮	⋮
23	1.84	0.025222570086747276	0.02522746937769847
24	1.92	0.021493387924282795	0.021497567222105336
25	2	0.018315526353559448	0.018319090144704935

Table 13: Runge Kutta

10 Code

10.1 ForwardEuler.py

```
# Forward Euler
import math
import matplotlib.pyplot as plt

h = 0.08
N = 27
y = [0 for i in range(N)]
x = [0 for i in range(N)]

y[0] = 0
x[0] = 0

def f(x, y):
    z = -8*y + 6 * ((math.e)**(-2*x))
    return z

def g(x):
    z = (math.e)**(-2*x) - (math.e)**(-8*x)
    return z

for i in range(0, N-1):
    print(str(i) + " & " + str(x[i]) + " & " + str(g(x[i])) + " & " + str(y[i]) + "\\\\")
    y[i+1] = y[i] + h*f(x[i], y[i])
    x[i+1] = x[i] + h


plt.plot(x, y)
w = [i*0.01 for i in range(200)]
z = [( g(i*0.01) ) for i in range(200)]
plt.plot(w, z)
plt.show()
```

10.2 AdamsBashforth.py

```
# Adams-Bashforth

import math
import matplotlib.pyplot as plt

h = 0.08
N = 27

y = [0 for i in range(N)]
x = [0 for i in range(N)]

y[0] = 0
x[0] = 0

def f(x, y):
    z = -8*y + 6 * ((math.e)**(-2*x))
    return z

def g(x):
    z = (math.e)**(-2*x) - (math.e)**(-8*x)
    return z

# Forward Euler for first one
print("0 & " + str(x[0]) + " & " + str(g(x[0])) + " & " + str(y[0]) + "\\\\")
y[1] = y[0] + h*f(x[0], y[0])
x[1] = x[0] + h

# A-B now
for i in range(1, N-1):
    print(str(i) + " & " + str(x[i]) + " & " + str(g(x[i])) + " & " + str(y[i]) + "\\\\")
    y[i+1] = y[i] + h*( 1.5 * f(x[i], y[i]) - 0.5 * f(x[i-1], y[i-1]) )
    x[i+1] = x[i] + h


plt.plot(x, y)
w = [i*0.01 for i in range(200)]
z = [( g(i*0.01) ) for i in range(200)]
plt.plot(w, z)
plt.show()
```

10.3 RungeKutta.py

```
# Runge-Kutta

import math
import matplotlib.pyplot as plt

h = 0.08
N = 27

y = [0 for i in range(N)]
x = [0 for i in range(N)]


def f(x, y):
    z = -8*y + 6 * ((math.e)**(-2*x))
    return z

def g(x):
    z = (math.e)**(-2*x) - (math.e)**(-8*x)
    return z


for i in range(0, N-1):

    k1 = f(x[i], y[i])
    k2 = f(x[i] + 0.5 * h, y[i] + 0.5 * h * k1)
    k3 = f(x[i] + 0.5 * h, y[i] + 0.5 * h * k2)
    k4 = f(x[i] + h, y[i] + h * k3)
    y[i+1] = y[i] + h*( k1 + 2*k2 + 2*k3 + k4 ) / 6
    x[i+1] = x[i] + h


plt.plot(x, y)

plt.show()
```

10.4 ErrorPlotEuler.py

```
# Plot error
import math
import matplotlib.pyplot as plt

x = [0 for i in range(16)]
y = [0 for i in range(16)]

def GetError(h, n):
    N = n

    y = [0 for i in range(N+1)]
    x = [0 for i in range(N+1)]

    def f(x, y):
        z = -8*y + 6 * ((math.e)**(-2*x))
        return z

    def g(x):
        z = (math.e)**(-2*x) - (math.e)**(-8*x)
        return z

    for i in range(0, N):
        y[i+1] = y[i] + h*f(x[i], y[i])
        x[i+1] = x[i] + h

    return y[N] - g(0.16)


for k in range(16):
    n = 2**k
    h = 0.16/n
    En = GetError(h, n)
    x[k] = math.log(abs(h))
    y[k] = math.log(abs(En))

plt.plot(x,y)
plt.show()
```

10.5 ErrorPlotAdams.py

```
# Plot error
import math
import matplotlib.pyplot as plt

x = [0 for i in range(16)]
y = [0 for i in range(16)]

def GetError(h, n):
    N = n

    y = [0 for i in range(N+1)]
    x = [0 for i in range(N+1)]

    def f(x, y):
        z = -8*y + 6 * ((math.e)**(-2*x))
        return z

    def g(x):
        z = (math.e)**(-2*x) - (math.e)**(-8*x)
        return z

    y[1] = y[0] + h*f(x[0], y[0])
    x[1] = x[0] + h

    # A-B now
    for i in range(1, N):
        #print(str(i) + " & " + str(x[i]) + " & " + str(g(x[i])) + " & " + str(y[i]) + "\\\\
        y[i+1] = y[i] + h*( 1.5 * f(x[i], y[i]) - 0.5 * f(x[i-1], y[i-1]) )
        x[i+1] = x[i] + h

    return y[N] - g(0.16)


for k in range(16):
    n = 2**k
    h = 0.16/n
    En = GetError(h, n)
    print(En)
    x[k] = math.log(abs(h))
    y[k] = math.log(abs(En))

plt.plot(x,y)
plt.show()
```

10.6 ErrorPlotRunge.py

```
# Plot error
import math
import matplotlib.pyplot as plt

x = [0 for i in range(16)]
y = [0 for i in range(16)]

def GetError(h, n):
    N = n

    y = [0 for i in range(N+1)]
    x = [0 for i in range(N+1)]

    def f(x, y):
        z = -8*y + 6 * ((math.e)**(-2*x))
        return z

    def g(x):
        z = (math.e)**(-2*x) - (math.e)**(-8*x)
        return z

    for i in range(0, N):
        #print(str(i) + " & " + str(x[i]) + " & " + str(g(x[i])) + " & " + str(y[i]) + "\\\\

        k1 = f(x[i], y[i])
        k2 = f(x[i] + 0.5 * h, y[i] + 0.5 * h * k1)
        k3 = f(x[i] + 0.5 * h, y[i] + 0.5 * h * k2)
        k4 = f(x[i] + h, y[i] + h * k3)
        y[i+1] = y[i] + h*( k1 + 2*k2 + 2*k3 + k4 ) / 6
        x[i+1] = x[i] + h

    #print(x[N])
    return y[N] - g(0.16)


for k in range(16):
    n = 2**k
    h = 0.16/n
    En = GetError(h, n)
    #print(En)
    x[k] = math.log(abs(h))
    y[k] = math.log(abs(En))

    try:
```

```
        print( (y[k] - y[k-1]) / (x[k] - x[k-1]) )
    except:
        pass

plt.plot(x,y)
plt.xlim(-13, 0)
```

10.7 SecondOrderRK4.py

```
# Runge-Kutta on second order

import math
import matplotlib.pyplot as plt
import numpy


def ye(x):
    a = (4 * p**2 - 1)**0.5
    b = (2/a) * ((1+x)**0.5) * math.sin(numpy.log(1+x)*a/2)
    return b

def ComputeYn(h, N):
    global alpha
    global p
    alpha = 2
    p = 5

    x = [0 for i in range(N+1)]
    y = [0 for i in range(N+1)]
    z = [0 for i in range(N+1)]

    z[0] = 1


    def f(x, y, z):
        return z

    def g(x, y, z):
        return -(p**2) * ((1+x)**(-alpha)) * y



    for i in range(0, N):

        x[i+1] = x[i] + h

        k1 = f(x[i], y[i], z[i])
        t1 = g(x[i], y[i], z[i])

        k2 = f(x[i] + 0.5 * h, y[i] + 0.5 * h * k1, z[i] + 0.5 * h * t1)
        t2 = g(x[i] + 0.5 * h, y[i] + 0.5 * h * k1, z[i] + 0.5 * h * t1)

        k3 = f(x[i] + 0.5 * h, y[i] + 0.5 * h * k2, z[i] + 0.5 * h * t2)
```

```
        t3 = g(x[i] + 0.5 * h, y[i] + 0.5 * h * k2, z[i] + 0.5 * h * t2)

        k4 = f(x[i] + h, y[i] + h * k3, z[i] + h * t3)
        t4 = g(x[i] + h, y[i] + h * k3, z[i] + h * t3)

        y[i+1] = y[i] + h*( k1 + 2*k2 + 2*k3 + k4 ) / 6
        z[i+1] = z[i] + h*( t1 + 2*t2 + 2*t3 + t4 ) / 6


    #plt.plot(x, y)

    return y[N]


# Actual program
x = [0 for i in range(13)]
y = [0 for i in range(13)]
E = [0 for i in range(13)]



for k in range(13):
    h = 0.1 / (2**k)
    N = 10 * (2**k)

    x[k] = h
    y[k] = ComputeYn(h,N)
    E[k] = y[k] - ye(1)

    if (k != 0):
        print(str(k) + " & " + str(x[k]) + " & " + str(y[k]) + " & " + str(E[k]) + " & " + s
    else:
        print(str(k) + " & " + str(x[k]) + " & " + str(y[k]) + " & " + str(E[k]) + " & \\\\'


#plt.plot(x, y)
plt.plot(x, E)

plt.show()
```

10.8 FalsePosition.py

```
# False position method
import math
import matplotlib.pyplot as plt
import numpy

def y(p, x):
    a = (4 * p**2 - 1)**0.5
    b = (2/a) * ((1+x)**0.5) * math.sin(numpy.log(1+x)*a/2)
    return b

# Compute value of y_p(1) with associated step size.
def phi(p):
    h = (2**(-10))
    N = (2**10)
    #print("THONK")
    return ComputeYn(h, N, p, 2)


# Compute the value of Y_N with step size h with p and alpha given with RK4.
def ComputeYn(h, N, p, alpha):
    x = [0 for i in range(N+1)]
    y = [0 for i in range(N+1)]
    z = [0 for i in range(N+1)]

    z[0] = 1


    def f(x, y, z):
        return z

    def g(x, y, z):
        return -(p**2) * ((1+x)**(-alpha)) * y

    for i in range(0, N):

        x[i+1] = x[i] + h

        k1 = f(x[i], y[i], z[i])
        t1 = g(x[i], y[i], z[i])

        k2 = f(x[i] + 0.5 * h, y[i] + 0.5 * h * k1, z[i] + 0.5 * h * t1)
        t2 = g(x[i] + 0.5 * h, y[i] + 0.5 * h * k1, z[i] + 0.5 * h * t1)

        k3 = f(x[i] + 0.5 * h, y[i] + 0.5 * h * k2, z[i] + 0.5 * h * t2)
```

```
        t3 = g(x[i] + 0.5 * h, y[i] + 0.5 * h * k2, z[i] + 0.5 * h * t2)

        k4 = f(x[i] + h, y[i] + h * k3, z[i] + h * t3)
        t4 = g(x[i] + h, y[i] + h * k3, z[i] + h * t3)

        y[i+1] = y[i] + h*( k1 + 2*k2 + 2*k3 + k4 ) / 6
        z[i+1] = z[i] + h*( t1 + 2*t2 + 2*t3 + t4 ) / 6

    return y[N]



def FindRoot (p1, p2, phi1, phi2, eps):
    ps = ( phi2 * p1 - phi1 * p2 )/( phi2 - phi1 )
    print(str(p1) + " & " + str(p2) + " & " + str(ps) + "\\\\")

    phis = phi(ps)
    if (abs(phis) < eps): return ps
    if (phis * phi1 > 0): return FindRoot(ps, p2, phis, phi2, eps)
    return FindRoot(p1, ps, phi1, phis, eps)



eps = 1 * 10**(-7)
p2 = 5
p1 = 4
p = FindRoot(p1, p2, phi(p1), phi(p2), eps)
true = (1/4 + (math.pi/numpy.log(2))**2)**0.5
print(p, phi(p), y(p, 1), true, p-true)
```

10.9 EightPlot.py

```
# False position method
import math
import matplotlib.pyplot as plt
import numpy

global alpha
alpha = 10


# Compute value of y_p(1) with associated step size.
def phi(p):
    h = 0.1 / (2**10)
    N = 10 * (2**10)
    return ComputeYn(h, N, p, alpha)


# Compute the value of Y_N with step size h with p and alpha given with RK4.
def ComputeYn(h, N, p, alpha):
    x = [0 for i in range(N+1)]
    y = [0 for i in range(N+1)]
    z = [0 for i in range(N+1)]

    z[0] = 1


    def f(x, y, z):
        return z

    def g(x, y, z):
        return -(p**2) * ((1+x)**(-alpha)) * y

    for i in range(0, N):

        x[i+1] = x[i] + h

        k1 = f(x[i], y[i], z[i])
        t1 = g(x[i], y[i], z[i])

        k2 = f(x[i] + 0.5 * h, y[i] + 0.5 * h * k1, z[i] + 0.5 * h * t1)
        t2 = g(x[i] + 0.5 * h, y[i] + 0.5 * h * k1, z[i] + 0.5 * h * t1)

        k3 = f(x[i] + 0.5 * h, y[i] + 0.5 * h * k2, z[i] + 0.5 * h * t2)
        t3 = g(x[i] + 0.5 * h, y[i] + 0.5 * h * k2, z[i] + 0.5 * h * t2)
```

```
        k4 = f(x[i] + h, y[i] + h * k3, z[i] + h * t3)
        t4 = g(x[i] + h, y[i] + h * k3, z[i] + h * t3)

        y[i+1] = y[i] + h*( k1 + 2*k2 + 2*k3 + k4 ) / 6
        z[i+1] = z[i] + h*( t1 + 2*t2 + 2*t3 + t4 ) / 6

    return y[N]



x = [ i for i in range(101)]
y = [ phi(i) for i in range(101)]

plt.plot(x,y)
plt.grid()
plt.show()
```

10.10 FalsePosition8.py

```
# False position method
import math
import matplotlib.pyplot as plt
import numpy

# Compute value of y_p(1) with associated step size.
def phi(p):
    h = (2**(-16))
    N = (2**16)
    return ComputeYn(h, N, p, 10)


# Compute the value of Y_N with step size h with p and alpha given with RK4.
def ComputeYn(h, N, p, alpha):
    x = [0 for i in range(N+1)]
    y = [0 for i in range(N+1)]
    z = [0 for i in range(N+1)]

    z[0] = 1


    def f(x, y, z):
        return z

    def g(x, y, z):
        return -(p**2) * ((1+x)**(-alpha)) * y

    for i in range(0, N):

        x[i+1] = x[i] + h

        k1 = f(x[i], y[i], z[i])
        t1 = g(x[i], y[i], z[i])

        k2 = f(x[i] + 0.5 * h, y[i] + 0.5 * h * k1, z[i] + 0.5 * h * t1)
        t2 = g(x[i] + 0.5 * h, y[i] + 0.5 * h * k1, z[i] + 0.5 * h * t1)

        k3 = f(x[i] + 0.5 * h, y[i] + 0.5 * h * k2, z[i] + 0.5 * h * t2)
        t3 = g(x[i] + 0.5 * h, y[i] + 0.5 * h * k2, z[i] + 0.5 * h * t2)

        k4 = f(x[i] + h, y[i] + h * k3, z[i] + h * t3)
        t4 = g(x[i] + h, y[i] + h * k3, z[i] + h * t3)

        y[i+1] = y[i] + h*( k1 + 2*k2 + 2*k3 + k4 ) / 6
```

```
        z[i+1] = z[i] + h*( t1 + 2*t2 + 2*t3 + t4 ) / 6

    return y[N]



def FindRoot (p1, p2, phi1, phi2, eps):
    ps = ( phi2 * p1 - phi1 * p2 )/( phi2 - phi1 )
    phis = phi(ps)
    #print(p1, p2, phi1, phi2, ps, phis)
    if (abs(phis) < eps): return ps

    if (phis * phi1 > 0): return FindRoot(ps, p2, phis, phi2, eps)
    return FindRoot(p1, ps, phi1, phis, eps)



eps = 1 * 10**(-8)
p2 = [20, 35, 45, 60, 77]
p1 = [5, 20, 35, 45, 60]
for i in range(5):
    p = FindRoot(p1[i], p2[i], phi(p1[i]), phi(p2[i]), eps)
    print(p, phi(p))
```

10.11 NormalisedPlot.py

```
# False position method
import math
import matplotlib.pyplot as plt
import numpy


# Compute the value of Y_N with step size h with p and alpha given with RK4.
def ComputeYn(h, N, p, alpha):
    x = [0 for i in range(N+1)]
    y = [0 for i in range(N+1)]
    z = [0 for i in range(N+1)]

    z[0] = 1

    def f(x, y, z):
        return z

    def g(x, y, z):
        return -(p**2) * ((1+x)**(-alpha)) * y

    for i in range(0, N):

        x[i+1] = x[i] + h

        k1 = f(x[i], y[i], z[i])
        t1 = g(x[i], y[i], z[i])

        k2 = f(x[i] + 0.5 * h, y[i] + 0.5 * h * k1, z[i] + 0.5 * h * t1)
        t2 = g(x[i] + 0.5 * h, y[i] + 0.5 * h * k1, z[i] + 0.5 * h * t1)

        k3 = f(x[i] + 0.5 * h, y[i] + 0.5 * h * k2, z[i] + 0.5 * h * t2)
        t3 = g(x[i] + 0.5 * h, y[i] + 0.5 * h * k2, z[i] + 0.5 * h * t2)

        k4 = f(x[i] + h, y[i] + h * k3, z[i] + h * t3)
        t4 = g(x[i] + h, y[i] + h * k3, z[i] + h * t3)

        y[i+1] = y[i] + h*( k1 + 2*k2 + 2*k3 + k4 ) / 6
        z[i+1] = z[i] + h*( t1 + 2*t2 + 2*t3 + t4 ) / 6


    return x, y


def Normalise(h, N, p, alpha, y):
```

```
    # integrate and divide by integral
    integral = 0

    for i in range(1, N):
        integral += h * ( (1 + i*h)**(-10) * (p * y[i])**2 )

    integral += h * ((p*y[0])**2) / 2
    integral += h * (2**(-10) * (p*y[N])**2) / 2

    for i in range(0, N+1):
        y[i] /= integral

    return y


p = 66.6632979369170

h = (2**(-10))
N = (2**10)
x, y = ComputeYn(h, N, p, 10)
z = Normalise(h, N, p, 10, y)

plt.plot(x,z)
plt.grid()
plt.show()
```