Project 1

Daniel Holm

1 Explicit adaptive Runge-Kutta methods

For an initial value problem $y' = f(t, y)$ the Runge-Kutta methods used in these tasks computes a handful of stage derivatives sampled at different points and then by using a linear combination of these we obtain a form of "average" derivative that is then used to advance the solution by taking a step of certain length along the above mentioned slope. An example is the classical 4th-order Runge-Kutta method (known as RK4) [1].

$$\begin{aligned} Y_1' &= f(t_n,\, y_n) \\ Y_2' &= f(t_n + \frac{h}{2},\, y_n + \frac{h}{2}Y_1') \\ Y_3' &= f(t_n + \frac{h}{2},\, y_n + \frac{h}{2}Y_2') \\ Y_4' &= f(t_n + h,\, y_n + hY_3') \end{aligned} \tag{1}$$

$$y_{n+1} = y_n + \frac{h}{6}(Y_1' + 2Y_2' + 2Y_3' + Y_4')$$

As we can see the four stage derivatives Y_1', Y_2', Y_3', Y_4' come together on the last line in order to create the new value where h denotes the length of the step.

In general, the method is defined by its coefficients a_{ij} that are used to compute the stage derivatives as well as the coefficients b_j that are used in the linear combination that updates the solution. These are often arranged in the Butcher tableau

c	A
y	b^T

For the classical RK4-method above (1), the Butcher tableau is

0	0	0	0	0
1/2	1/2	0	0	0
1/2	0	1/2	0	0
1	0	0	1	0
y	1/6	1/3	1/3	1/6

Table 1

1.1 Writing the RK4step method

The first task was to write a method that takes a step according to the equations above (1) and then simply outputs the updated value [2]. The method's signature looks like `RK4Step(f, uold, told, h)` where `f` corresponds to the function f in the actual differential equation. The parameters `uold` and `told` are the current value and time and `h` is the desired step size. The implementation consisted of simply doing the necessary math operations and returning the result. In order to test the method it was applied to solve the linear test equation $y' = ay$ with initial condition $y(0) = 1$ and a time interval between 0 and 10. By deciding the number of steps we can know exactly the size of the steps (since we also choose the time interval). By simply looping the step function and storing the returned values in an array the results can be plotted to see if it at least "looks" right. The values can also be directly compared to the exact solution to obtain errors.

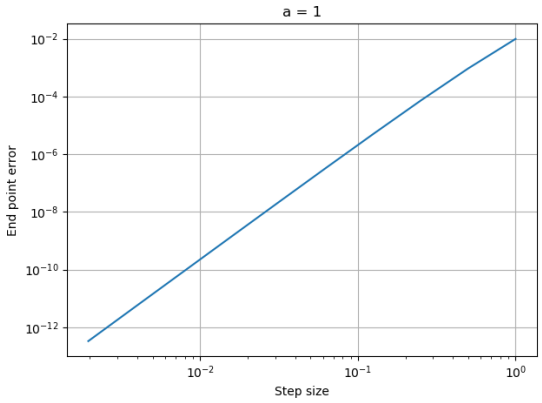

Figure 1: End point error as a function of step size with $a = 1$.

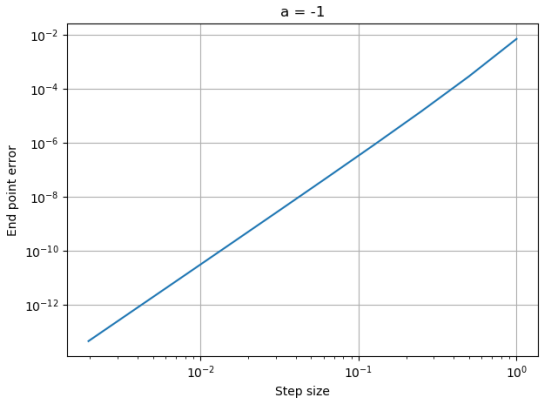

Figure 2: End point error as a function of step size with $a = -1$.

In both the plots above (figures 1 and 2) we see very straight lines with a logarithmic slope of about 4. We can therefore conclude that global error is indeed $O(h^4)$ and be sure that the implementation was correct.

1.2 Embedded pairs of RK methods

An example of a 3rd order RK method (RK3) is defined by the Butcher tableau

0	0	0	0
1/2	1/2	0	0
1	-1	2	0
z	1/6	2/3	1/6

Table 2

Some of the evaluations of the right hand side f are the same as in the RK4 above 1. Both methods can therefore be written simultaneously as

$$\begin{aligned} Y_1' &= f(t_n,\, y_n) \\ Y_2' &= f(t_n + \frac{h}{2},\, y_n + \frac{h}{2}Y_1') \\ Y_3' &= f(t_n + \frac{h}{2},\, y_n + \frac{h}{2}Y_2') \\ Z_3' &= f(t_n + h,\, y_n - hY_1' + 2Y_2') \\ Y_4' &= f(t_n + h,\, y_n + hY_3') \end{aligned} \tag{2}$$

$$\begin{aligned} z_{n+1} &= y_n + \frac{h}{6}(Y_1' + 4Y_2' + Y_3') \\ y_{n+1} &= y_n + \frac{h}{6}(Y_1' + 2Y_2' + 2Y_3' + Y_4'). \end{aligned}$$

We can obtain both a 3rd and a 4th order approximation by only performing one extra calculation Z_3'. This can then be used to estimate a local error difference $l_{n+1} = z_{n+1} - y_{n+1}$ [1].

When two methods use the same evaluations like this, they are called an embedded pair of RK methods. In this case, the pair is called RK34. Furthermore, in practice we will not even calculate z_{n+1}; instead one computes the error estimate directly from the stage derivatives in order to save an extra computation as well as sparing ourselves from potential round-off errors [2].

1.3 Writing the RK34Step method

Next task was to write a new method `RK34Step(f, uold, told, h)` that takes an RK4 step as normal but now also returns the error estimate by using the embedded RK3 [2]. Again the imlementation consisted of simply writing the math operations and returning the results.

1.4 Varying step size

Using the Euclidean norm $r_{n+1} = \|l_{n+1}\|_2$ the idea is to keep $r_{n+1} = \mathbf{TOL}$ according to some prescribed tolerance **TOL** along the function by varying the step size. This is how we make our solver adaptive. By borrowing some wisdom from control theory we will use a PI-controller in order to calculate the next step size using the current and previous local error estimates.

$$h_n = \left(\frac{\mathbf{TOL}}{r_n}\right)^{2/(3k)} \left(\frac{\mathbf{TOL}}{r_{n-1}}\right)^{-1/(3k)} \cdot h_{n+1} \tag{3}$$

Since RK4 has local error $O(h^5)$ and RK3 has $O(h^4)$ the difference between them is $O(h^4)$ hence we will use $k = 4$ for our adaptive RK34 solver later. Additionally, for the very first step we put $r_0 = \mathbf{TOL}$ and for the following steps recursion will operate as intended.

1.5 Writing the newStep method

Next task was to write a method `newStep(tol, err, errOld, hOld, k)` that, which given the tolerance, current and previous error estimate, the old step size and the order k, returned the new step size according to (3) above [2]. Implementing the method was very straight-forward.

1.6 Writing the adaptive solver

By combining `newStep` and `RK34Step` the task was to write an adaptive ODE solver `adaptiveRK34(f, y0, t0, tf, tol)` that returns the time points the method uses and the numerical values at those time points in two separate arrays. In order to start the integration an initial step size was selected according to the expression

$$h_0 = \frac{|t_f - t_0| \cdot \mathbf{TOL}^{1/4}}{100 \cdot (1 + \|f(t_0,\, u_0)\|)}. \tag{4}$$

The implementation consisted of looping steps until the next step would take you beyond the end point t_f. In the loop we take the step according to the calculated step size and update our current time point as well as storing the new values and time point in the arrays. The next step size is calculated and the local error estimates are updated for the next loop. The last step is taken manually in order to finish at exactly the end point t_f. Lastly the value and time arrays are returned.

In order to test the solver it was used to solve the matrix problem $y' = Ay$, $y_0 = (1, 1)^T$ and $t_0 = 0$, $t_f = 10$, for the matrix

$$A = \begin{pmatrix} -1 & 10 \\ 0 & -3 \end{pmatrix}.$$

The plots of the computed solutions and the exact solutions are plotted side by side. A tolerance of 10^{-8} was used.

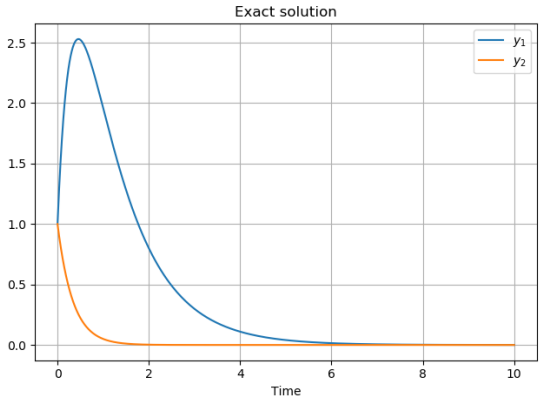

Figure 3: Exact solution using exponential matrix.

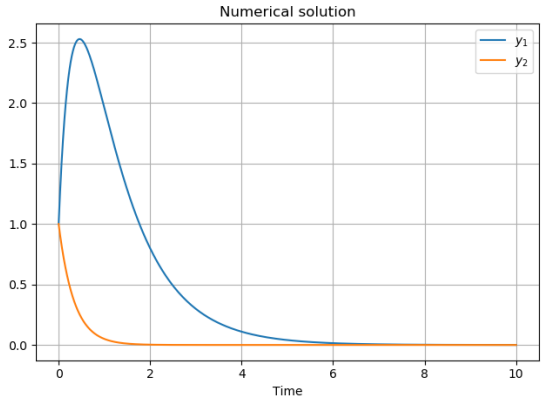

Figure 4: Numerical solution using the written solver

The plots above (figures 3 and 4) look more or less identical. A more rigorous check was not performed since this was mostly for quick verification that the written `adaptiveRK34` actually worked as intended.

2 The Lotka-Volterra equation

The Lotka-Volterra equation is a classical model in biological population dynamics. The equation models the the interaction between a predator species, y, and a prey species, x according to the differential equation system

$$\begin{aligned} \dot{x} &= ax - bxy \\ \dot{y} &= cxy - dy, \end{aligned} \tag{5}$$

where a, b, c, d are positive parameters. If there are no predators ($y = 0$), then the number of prey will grow exponentially with a rate of a. Similarly, if there are no prey ($x = 0$) the predators will die out exponentially with a rate of d since there is no food supply. The product term xy represents the probability that a predator and a prey will interact in their shared ecosystem. Since this encounter favors the predator the product term is negative in the first equation and positive in the second equation. Because the equation is separable we can divide the two equations and get

$$\frac{dx}{dy} = \frac{ax - bxy}{cxy - dy} = \frac{x(a - by)}{y(cy - d)}.$$

We can write this in differential terms,

$$\left(c - \frac{d}{x}\right) dx = \left(\frac{a}{y} - b\right) dy,$$

and by integrating we have $cx - d\ln x = a\ln y - by + K$ for some constant K. This means that the function

$$H(x,\, y) = cx + by - d\ln x - a\ln y \tag{6}$$

remains constant for all times along the solutions. This in turn means that the Lotka-Volterra equation has periodic solutions [1][2].

2.1 Applying the RK34 adaptive solver

In order to use the written `adaptiveRK34` method we first have to define a method `lotka(t, y)` that simply outputs the derivatives according to (5). This will be passed into the adaptive solver as the `f`-parameter. The parameters a, b, c, d were chosen as $(3, 9, 15, 15)$ and initial values were chosen to be close to the equilibrium point $\left(\frac{d}{c}, \frac{a}{b}\right)$ i.e. (1, 1). The system was to be simulated for at least 10 full periods with a tolerance of at least 10^{-6} although higher tolerances were tested as the code allowed for it without taking too much time.

2.1.1 Plots for different initial values

All of the simulations were run with a tolerance of 10^{-10} and took approximately 2 seconds.

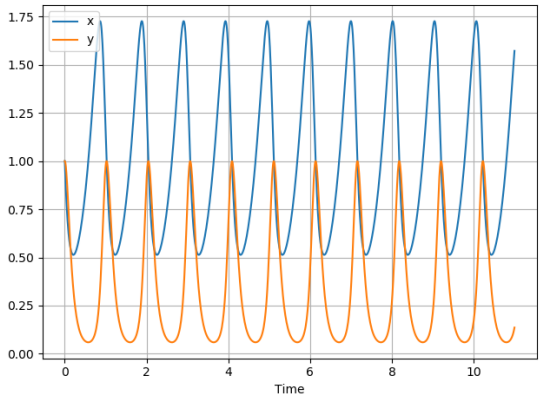

Figure 5: x and y as functions of time and the phase portrait for initial value $(1, 1)^T$

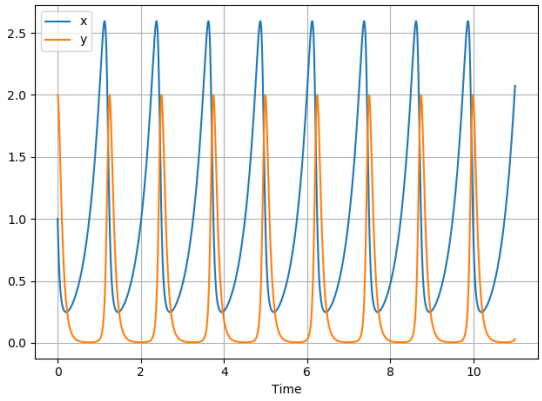

Figure 6: x and y as functions of time and the phase portrait for initial value $(1, 2)^T$

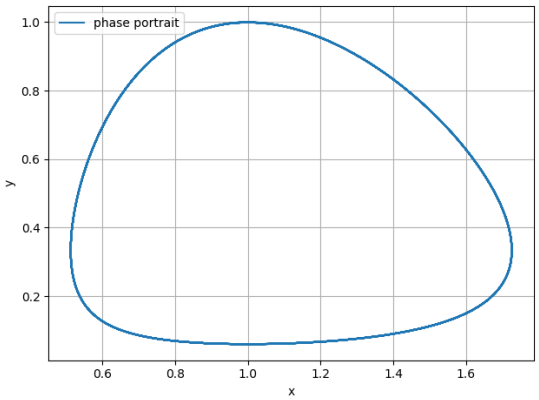

Figure 7: y as a function of x with initial value $(1,\ 1)^T$

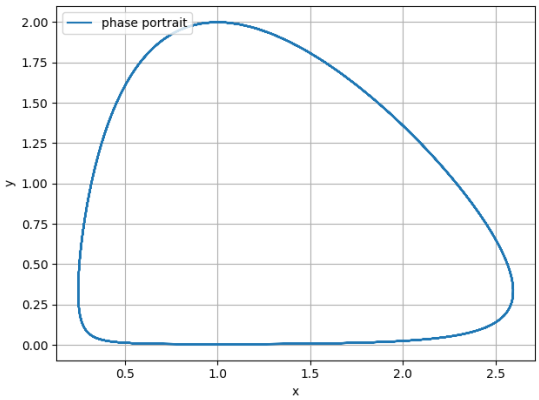

Figure 8: y as a function of x with initial value $(1,\ 2)^T$

In the plots above (figures 5, 6, 7 and 8) the solutions are indeed periodic as the phase portraits are closed orbits. The period time is slightly different for them because of the different initial values. For the left plot the period seems to be slightly larger than 1 while for the right plot is closer to 1.2. The solutions do, however, "look" quite similar.

2.2 Investigating error over time

By integrating over a very long time (100 - 1000 full periods depending on tolerance and patience) we can check if the numerically computed $H(x,\ y)$ according to (6) stays close to its initial value $H(x_0,\ y_0)$ over time. If the solver is good and the outputs are periodic as they should be this error should also not drift away over time. This was done by plotting the expression

$$\left| \frac{H(x,\ y)}{H(x_0,\ y_0)} - 1 \right| \tag{7}$$

as a function of time where we insert the computed x, y into the expression.

2.2.1 Error plot

The simulation was run with a tolerance of 10^{-6} over approximately 1000 full periods with initial value (1, 1) and took appromixately 30 seconds. The error over time according to (6 is shown in a linlog diagram below.

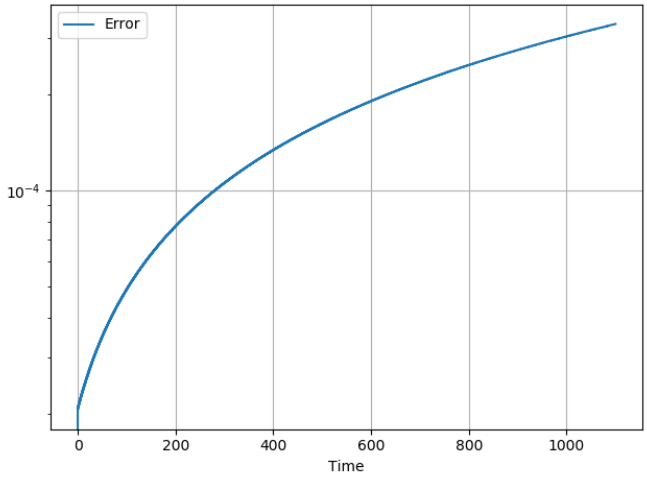

Figure 9: Error over time

We can see that the error does actually grow over time but seems to perhaps stabilize as the slope mellows out near the end.

3 The van der Pol equation

The van der Pol equation,

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= \mu \cdot (1 - y_1^2) \cdot y_2 - y_1 \end{aligned} \tag{8}$$

models an electric oscillator circuit. The solution is periodic with a period of at least 2μ. Depending on the choice of μ, the problem may be either stiff or non-stiff. The initial condition used for all simulations unless stated otherwise was $y(0) = (2 \quad 0)^T$ [1][2].

3.1 Applying the RK34 adaptive solver

First a method `vdp(t, y)` was implemented according to (8) that could be passed into the solver as the `f`-parameter. Again, we used the written `adaptiveRK34` method to solve, with $\mu = 100$ the system was simulated over a full period i.e. a time interval of $[0, 2\mu]$.

3.1.1 Some plots

The solution component y_2 was plotted over time as well as a function of y_1 with a few different initial values for the phase portrait.

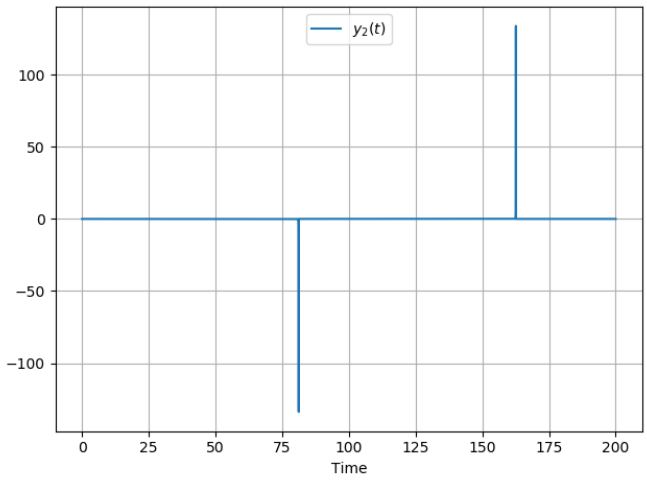

Figure 10: The solution component y_2 over time.

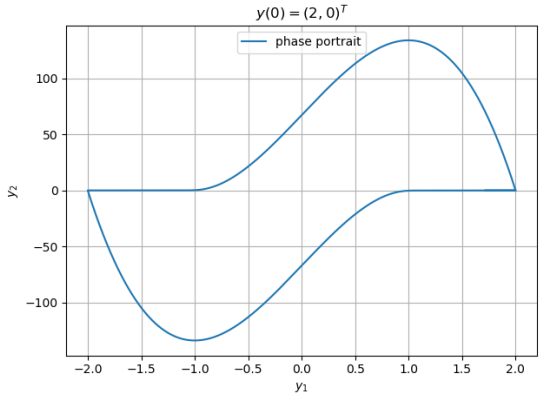

Figure 11: The phase portrait with initial value $(2, 0)^T$.

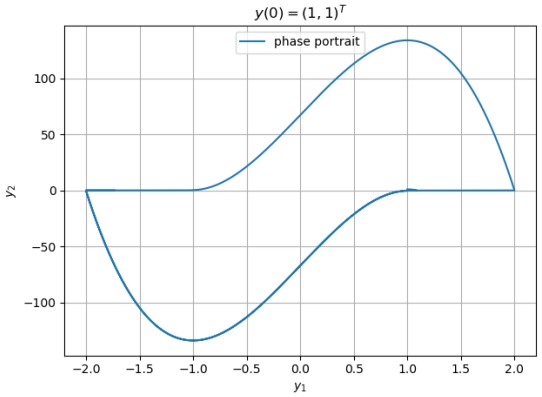

Figure 12: The phase portrait with initial value $(1, 1)^T$.

As we can see the phase portraits are identical save for the tiny transient in figure 12 located at $(y_1 = 1, y_2 = 0)$. The transient exists because of the deviation in initial value. The orbit should however tend to the same portrait regardless of initial value choice (within reason). This is known as a limit cycle.

3.2 Exploring stiffness

By using the explicit `adaptiveRK34` method we can explore how stiffness depends on the choice of μ. The specific choices were the "E6 series", $\mu = 10, 15, 22, 33, 47, 68, 100, 150, 220, 330, 470, 680, 100$ The system was simulated over time interval $[0, 0.07\mu]$ with a tolerance of 10^{-6} for all μ above and the number of steps the solver took for each μ was stored in an array.

3.2.1 Plots of N as a function of μ

The number of steps N as a function of μ was plotted in a loglog diagram.

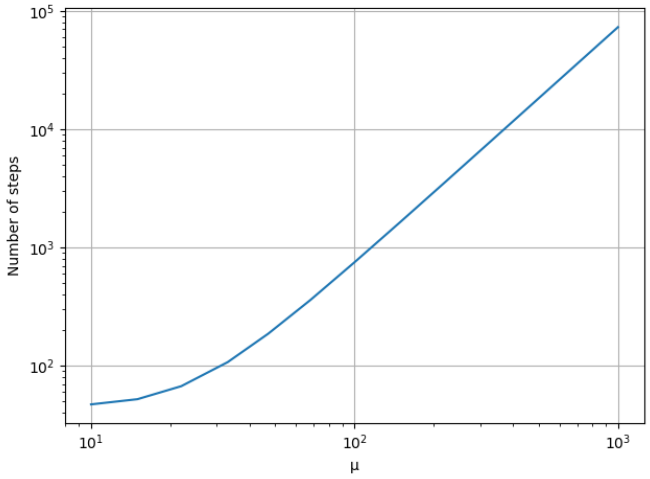

Figure 13

In the plot above (figure 13) we see a nearly straight line with a logarithmic slope of around 2. Since at $\mu = 10^2$ we have the number of steps slightly below 10^3 and at $\mu = 10^3$ the number of steps is about just as slightly below 10^5. This means that $N \sim C \cdot \mu^2$ and that the number of steps is proportional to the stiffness where the stiffness depends is simply μ^2.

3.3 Implicit vs. explicit solvers

In an implicit method the calculations are generally more expensive. This extra expense can, however, pay off if one can take significantly longer and therefore, fewer, steps compared to an explicit method. This happens in stiff problems. In explicit methods, the maximum stable step size is limited because the methods have bounded stability regions. A well designed implicit method can in contrast have an unbounded stability region. This permits us to take much larger steps without losing accuracy, making up for the extra computing. By using `scipy.integrate.BDF` in Python the operations described in 3.2 were repeated with this built-in ODE-solver [1][2].

3.3.1 Even more plots

Again, the number of steps N as a function of μ was plotted in a loglog diagram. The built in solver also tested for $\mu = 10000$ and $\mu = 15000$. These very high values would take extrem amounts of time with the written solver.

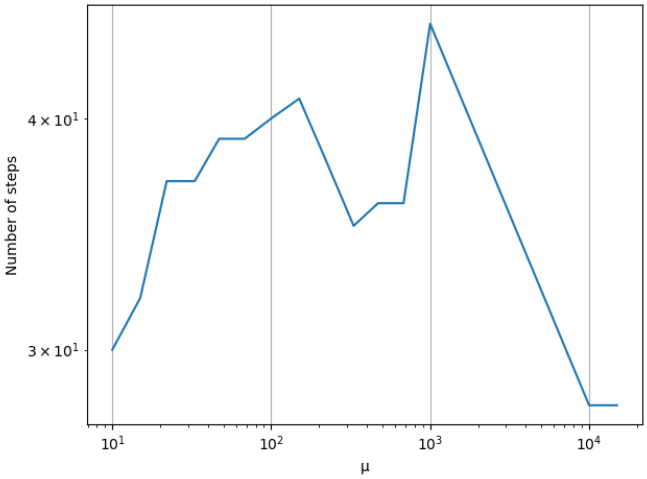

Figure 14

The number of steps that the built in solver requires is several orders of magnitude less than the nonstiff and explicit `adaptiveRK34`. The execution time for the built in solver was also significantly faster due to this. The graph also does not follow an obvious trend.

4 Conclusions

I have learned quite a lot during this project. It was very enjoyable to work with Python as I have not done that before and I am likely to convert to Python for these kinds of tasks instead of using MATLAB in the future. Additionally, learning these semi-basic ideas about numerical methods has been very interesting as they are quite clever, I think this course can get interesting. Being able to then actually apply these ideas with my own code is very exciting, like I am finally doing math for real.

I would like to thank a few of my fellow classmates and the good people of the internet for help with setting up the IDE, how to actually write Python code as well as help with LaTeX notation. For help with understanding the mathematical concepts I would thank the writers of the project instruction, namely my lecturer Tony Stillfjord and Gustaf Söderlind.

References

[1] Gustaf Söderlind, *Numerical Methods for Differential Equations - An Introduction to Scientific Computing*, October 27, 2019

[2] Tony Stillfjord, Gustaf Söderlind, *Project #1 in FMNN10 and NUMN12*, 2019

A functions.py

A Python file named *functions* contained the basic methods (such as the step methods) for all the tasks. Separate files contained code for the more specific tasks (such as plot commands).

```
import numpy as np
from scipy.linalg import expm, norm

def RK4Step(f, uold, told, h):
    y1 = f(told, uold)
    y2 = f(told + h / 2, uold + h / 2 * y1)
    y3 = f(told + h / 2, uold + h / 2 * y2)
    y4 = f(told + h, uold + h * y3)

    return uold + h / 6 * (y1 + 2 * y2 + 2 * y3 + y4)


def RK34Step(f, uOld, tOld, h):
    y1 = f(tOld, uOld)
    y2 = f(tOld + h / 2, uOld + h / 2 * y1)
    y3 = f(tOld + h / 2, uOld + h / 2 * y2)
    y4 = f(tOld + h, uOld + h * y3)

    z3 = f(tOld + h, uOld - h * y1 + 2 * h * y2)

    return uOld + h / 6 * (y1 + 2 * y2 + 2 * y3 + y4), \
           h / 6 * (2 * y2 + z3 - 2 * y3 - y4)


def newStep(tol, err, errOld, hOld, k):
    r1 = norm(err)
    r0 = norm(errOld)

    return (tol / r1) ** (2 / (3 * k)) * (tol / r0) ** (-1 / (3 * k)) * hOld


def adaptiveRK34(f, y0, t0, tf, tol):
    t = np.array([t0])  # array of time points, times will be appended in loop
    y = y0  # array of values corresponding to time points in array above, values
    # will be appended in loop

    if np.size(y0) == 1:
```

```
        axis = None
    else:
        axis = 1

    tOld = t0  # initial time, updates every loop
    yOld = y0  # initial value, updates every loop
    errOld = tol  # initial error set to tol, updates every loop

    hOld = ((tf - t0) * tol ** .25) / (100 * (1 + norm(f(t0, y0))))
    # initial step size, updates every loop

    while (tOld + hOld) < tf:  # if the desired step size stays within
        # time boundary tf
        t = np.append(t, [tOld + hOld])
        # appends the current time after the step to time array
        yOld, err = RK34Step(f, yOld, tOld, hOld)
        # calculates value and error after step and updates
        y = np.append(y, yOld, axis)  # appends the new value to value array

        tOld += hOld  # updates the current time to time after the step
        hOld = newStep(tol, err, errOld, hOld, 4)
        # calculates and updates step size for next step
        errOld = err  # updates the error at current time for next loop

    hLast = tf - tOld  # length of last step is simply the remaining distance
    t = np.append(t, tf)  # we know the last time value is tf
    yOld, err = RK34Step(f, yOld, tOld, hLast)  # calculates the value at time tf
    y = np.append(y, yOld, axis)  # appends the last value

    return t, y  # returns time and value array
```

B Code for task 2

B.1 task2.1.py

```
import numpy as np
import functions as func
from scipy.linalg import norm
import matplotlib.pyplot as plotter

a = 1 # or a = -1, ran it with different a to produce two plots
```

```
def eq(t, y):
    return a * y


y0 = 1
t0 = 0
tf = 1


def RK4Int(f, y0, t0, tf, N):
    h = (tf - t0) / N
    uold = y0
    told = t0
    approx = np.array([y0])
    err = np.array([0])

    for i in range(1, N + 1):
        unew = func.RK4Step(f, uold, told, h)
        approx = np.append(approx, [unew])
        exact = y0 * np.exp(a * h * i)
        err = np.append(err, norm(exact - unew))
        uold = unew
        told += h

    return approx, err


def errRK4(f, y0, t0, tf, k=10):
    N = np.arange(k)
    N = np.power(2, N)

    h = (tf - t0) / N

    errors = np.zeros(k)

    for i in range(k):
        approx, err = RK4Int(f, y0, t0, tf, N[i])
        errors[i] = err[-1]

    plotter.loglog(h, errors)
    plotter.grid()
```

```
errRK4(eq, y0, t0, tf)
plotter.xlabel('Step size')
plotter.ylabel('End point error')
plotter.title('a = 1')
plotter.savefig('a = 1')
plotter.show()
```

B.2 task2.py

```
import numpy as np
import matplotlib.pyplot as plotter
import functions as f
from scipy.linalg import expm, norm

A = np.array([[-1, 10], [0, - 3]])


def eq(t, y):
    return np.dot(A, y)


y0 = np.array([[1], [1]])
t0 = 0
tf = 10
tol = 1e-8

t, y = f.adaptiveRK34(eq, y0, t0, tf, tol)

exact = y0
for i in t[1:]:
    exact = np.append(exact, np.dot(expm(A * i), y0), axis=1)

# Comment out different parts of the code for different plots

plotter.plot(t, y[0], label=r'$y_1$')
plotter.plot(t, y[1], label=r'$y_2$')
# plotter.plot(t, exact[0], label=r'$y_1$')
# plotter.plot(t, exact[1], label=r'$y_2$')
plotter.xlabel('Time')
plotter.title('Numerical solution')  # or 'Exact solution'
plotter.legend()
```

```
plotter.grid()
plotter.show()
```

C Code for task 3

C.1 task3.py

```
import numpy as np
import matplotlib.pyplot as plotter
from scipy.linalg import norm
import functions as f
import time

a, b, c, d = 3, 9, 15, 15

u0 = np.array([[1], [1]])  # different initial values between plots

t0 = 0
tf = 11  # or 11 * 100 for error plot

tol = 1e-10  # or 1e-6 for error plot


def lotka(t, u):
    x = a * u[0] - b * u[0] * u[1]
    y = c * u[0] * u[1] - d * u[1]
    return np.array([x, y])


def H(u):
    return c * u.item(0) + b * u.item(1) - d * np.log(u.item(0))\
           - a * np.log(u.item(1))


H0 = H(u0)


def eq(h):
    return norm(h / H0 - 1)


hArr = np.array([eq(H0)])
```

```
ex = time.time()

t, u = f.adaptiveRK34(lotka, u0, t0, tf, tol)

for i in u[:, 1:].T:
    temp = H(i)
    temp = eq(temp)
    hArr = np.append(hArr, [temp])

ex = time.time() - ex
print(ex)

# I comment out different parts to produce the different plots

plotter.plot(t, u[0], label='x')
plotter.plot(t, u[1], label='y')
# plotter.plot(u[0], u[1], label='phase portrait')
# plotter.semilogy(t, hArr, label='Error')
plotter.grid()
plotter.xlabel('Time')  # or 'x' for phase portrait
# plotter.ylabel('y') # for phase portrait
plotter.legend(loc='upper left')
plotter.show()
```

D Code for task 4

D.1 task4.1.py

```
import functions as f
import numpy as np
import matplotlib.pyplot as plotter

u = 100


def vdp(t, y):
    y1 = y[1]
    y2 = u * (1 - y[0] ** 2) * y[1] - y[0]
    return np.array([y1, y2])
```

```
y0 = np.array([[1], [2]])
t0 = 0
tf = 2 * u

tol = 1e-10

t, y = f.adaptiveRK34(vdp, y0, t0, tf, tol)

# Comment out different parts of the code for different plots

# plotter.plot(t, y[1, :], label=r'$y_2(t)$')
plotter.plot(y[0, :], y[1, :], label='phase portrait')
plotter.grid()
# plotter.xlabel('Time')
plotter.xlabel(r'$y_1$')
plotter.ylabel(r'$y_2$')
plotter.legend(loc='upper center')
plotter.title(r'$y(0) = (1, 2)^T$')
plotter.show()
```

D.2 task4.py

```
import numpy as np
import matplotlib.pyplot as plotter
from scipy.integrate import BDF
import functions as f
import time

u = np.array([10, 15, 22, 33, 47, 68, 100, 150, 220, 330, 470, 680, 1000,
              10000, 15000])
steps1 = np.zeros([13])
steps2 = np.zeros(15)

y0 = np.array([[2], [0]])
yi = np.array([2, 0])
tol = 1e-6

ex = time.time()

for i in range(15):  # or range(13) when using the written solver
    def vdp(t, y):
        y1 = y[1]
```

```
        y2 = u[i] * (1 - y[0] ** 2) * y[1] - y[0]
        return np.array([y1, y2])


    # t1, y1 = f.adaptiveRK34(vdp, y0, 0, .07 * u[i], tol)
    bdf = BDF(vdp, 0, yi, .07 * u[i])  # built-in solver
    k = 0
    while bdf.t < .07 * u[i]:
        bdf.step()
        k += 1
    # steps1[i] = t1.size
    steps2[i] = k

ex = time.time() - ex
print(ex)

# Comment out different parts of the code for different plots,
# written solver vs. built-in solver

# plotter.loglog(u[:13], steps1)
plotter.loglog(u, steps2)
plotter.grid()
plotter.xlabel('\u03BC')
plotter.ylabel('Number of steps')
plotter.show()
```