Project #1 in FMNN10 and NUMN12

Goals. The goal is to get started with MATLAB or Python as a tool for approximating solutions to differential equations, and in particular, to get acquainted with time-stepping methods for initial value ODEs, and standard techniques for analyzing and assessing such methods.

1. **The Euler methods.** Examine numerical stability and computational accuracy, and learn how to verify that your solver works as expected. Learn how to arrange computational experiments and how to plot and evaluate information that reveals method performance.

2. **Explicit Runge–Kutta methods and automatic step size selection.** You construct your own ODE solver, based on an explicit Runge–Kutta method with embedded error estimator. The error estimator is used to adjust the time step h along the integration, so that the error estimate is kept close to a prescribed accuracy tolerance `tol`.

3. **Stiff vs. nonstiff problems.** We focus on understanding the distinction between stiff and nonstiff problems, and why stiff problems require *implicit* methods with unbounded stability regions. We will study two nonlinear oscillatory systems, the nonstiff Lotka-Volterra population dynamics problem and the van der Pol equation, which may be stiff or nonstiff depending on problem parameters. You will work both with your own solver, and a professionally implemented stiff ODE solver.

Prepare well before you go to the classroom. Read the entire instruction and work out a plan for how to solve the problems. Get started as soon as possible. The teaching assistants can only advise you when you have a program to work with. You work individually or (preferably) with a fellow student on the programs and the project report.

General rules, and suggestions on how to write a good report

- Write your report in English (preferable) or Swedish. **The report should not exceed 10 pages.**
- Hand in your report in Canvas via the link on the course webpage.
- The submission deadline is non-negotiable. Plan your work accordingly.
- The report will be evaluated and commented on within a week.

- If there are one or two important things missing from the report, you will get the chance to complete these and resubmit your report before the exam. But if you have not included any information on your attempts to solve a task you will have to wait until the re-exam period. These projects are an intrinsic part of the course and you will learn much that is of use also on the exam, so please do not try to "get away with" the minimal necessary amount of work.

- There is no prescribed format for the report, except that it must describe your work in a scientific manner, providing the necessary theory, as well as describe how you proceed from observation to conclusion, including motivations wherever necessary. Make sure to discuss your observations and state your conclusions, even on minor questions raised in the assignment text. These are all educational objectives.

- Plots and graphs must be accurate, legible, illustrative, and have labels on both axes. If you use a loglog-plot showing convergence orders, turn the grid on, or plot a reference line with the nominal slope.

- A plot alone is not sufficient as a "result" – it has to be *interpreted*. A good rule is to accompany plots with three comments: *what* the plot depicts; what you can specifically *see* in the plot; and *what conclusions* you draw.

- Describe each problem you work on in your own words, write down definitions of methods and equations, as well as values of any constants used, etc. The report should be "self-contained."

- You are encouraged to discuss problems and technicalities with your fellow students, but you must develop your own programs, and write your own report (where "own" means your group of ≤ 2 students).

- On formatting: LaTeX is preferred, but as long as it's readable, .doc/.odt and such formats are also acceptable.

- It's better to include too much rather than too little of your codes. However, if you have the same type of plotting code repeated five times in a script, it can be omitted. You may include codes as an appendix, surpassing the ten page limit.

- At the end of the report, you must **explain your division of work** (who did what?), and **acknowledge instructors and fellow students who have contributed to your understanding and re-**

sults. You also need to include **literature references, or references to the lecture notes and book manuscript published on the course home page.**

1. Getting started – the explicit Euler method

You do not have to include the tasks in this first section in the project report, only the tasks in Section 2 – 4. But these tasks will help you structure your later work, and see how things work in a simpler case.

Consider the linear initial value problem

$$\dot{y} = Ay; \qquad y(t_0) = y_0, \tag{1}$$

where A is a square matrix, over the interval $t \in [t_0, t_\mathrm{f}]$. Write down the exact solution to the problem using the matrix exponential e^{tA}.

Task 1.1 Construct a function,

```
function unew = eulerstep(A, uold, h)
```

that takes a single explicit Euler step of size h, from the point `uold` to produce the next approximation, `unew`. This function will be extremely simple – the point is that when we program more advanced methods and solvers, we prefer to start from a function taking a single step. Note also that we mainly use this MATLAB-style notation in these instructions. Equivalent Python code will have different but similar notation. It might be more natural to use an object-oriented approach in Python; feel free to deviate from the suggested structure as long as you fulfil the objectives of the exercises. However, you must then even more clearly describe in the report what you have done and why.

Task 1.2 Using `eulerstep`, construct a function

```
function [approx,err] = eulerint(A, y0, t0, tf, N)
```

that integrates (1) using N equal steps on the interval $[t_0, t_\mathrm{f}]$. Make sure that you hit the endpoint, i.e., be careful not to take one step too few, or one too many. Store the endpoint numerical solution in the output variable `approx` and the endpoint error in the variable `err`.

You will need the matrix exponential e^{tA} to compute the exact solution and the error. A matrix exponential e^{A} can be computed in MATLAB by using the command `expm(A)`. Use the `help` command if you need to find out more, and don't confuse the function `expm` with `exp`. In Python, you

find the expm `function` in the `scipy.linalg` module. Use `?` to get more information.

Verify that your function works properly by testing it for a simple problem, e.g. the linear scalar test equation $\dot{y} = \lambda y$, with `t0=0` and `tf=1` and initial value `y0=1`. Choose a suitable value for λ. (Would a negative value be preferable or not?) Plot the solution as a function of t.

Task 1.3 We are now going to study how the *global error* at the endpoint depends on the step size h, or equivalently, on the total number of steps N taken to reach the end point. Write a function

```
function errVSh(A, y0, t0, tf)
```

that calls `eulerint` for various choices of N and plots the error as a function of $h = (t_\mathrm{f} - t_0)/N$ in a log–log diagram. Use the `loglog` plot command. (Why should a log–log diagram be used, i.e., how do we expect the error to depend on the step size? Check with the lecture notes. Also, why should one use an integer number of steps, N, and not vary h freely?)

To measure the error, we need a *norm*. (Why can't you use the `abs` function in MATLAB? Check what `abs` does when applied to a vector.) Use the function `norm` in MATLAB; this computes the Euclidean norm $\|r\|_2$ of a vector r. In Python, the `norm` function exists in (e.g.) the `scipy.linalg` module.

Start with the scalar case $A = \lambda$ and make several graphs for different choices of λ. Choose $N = 2^k$ for some suitable powers k. (Why is that smart, when you use a log–log plot?) Also, by using the MATLAB command `hold on` you can present *several graphs in the same diagram*. You can also use color graphs for easy identification. The `help` command in MATLAB is always very helpful and the key to getting further information. For example, you could try typing `help plot`. In Python, the module `matplotlib.pyplot` contains equivalent functions `plot` and `loglog`, and "hold on" is the default. For the report plots, remember to use the functions `xlabel`, `ylabel`, `title`, `legend` and `grid` (same names in MATLAB/Python).

After running your program, take a close look at your findings. How does the error behave as a function of h? What is the slope of the graph in the diagram? How can the slope be interpreted? Can you deduce that the method is convergent? How does the error graph behave when you change λ? Can you see if numerical instability occurs?

Task 1.4 Now we will investigate the error as a function of the time t. For this, modify your `eulerint` function such that it returns the whole vector

of approximations $y_0, y_1, \ldots, y_N$ and the corresponding errors $e_0, e_1, \ldots, e_N$. (You can then modify your `errVSh` function to just use the last element of this vector.) Plot the error vector vs. the time vector $t_0, t_1, \ldots, t_N$ in a lin-log diagram, with time on a linear scale. (Why should a lin-log diagram be used, i.e. how do we expect the error to behave as a function of time? Check with the lecture notes.) Run your first test for the scalar case and use both positive and negative values of λ. Does the error always grow with time? What happens if you consider instead the relative error $e_n/\|y(t_N)\|$?

Task 1.5 Repeat Task 1.3 and 1.4 using some matrix A of your on choice. You can choose any dimension you like for A, but make sure that A doesn't have too large eigenvalues (if necessary, check with `eig(A)` in MATLAB or `scipy.linalg.eig` in Python).

A simple test case could be

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

with $y_0 = (1 \quad 1)^{\mathrm{T}}$, and $t_0 = 0$ and $t_{\mathrm{f}} = 10$. Do you see any qualitative differences compared with what you saw in the scalar case? Try a few different matrices, of different sizes, and with different elements.

Task 1.6 With the functions you have built, it should be very simple to also construct `ieulerstep`, `ieulerint` and `ierrVSh` for analyzing the *implicit Euler method*. See lecture notes for method formulas. You will need to solve a linear equation system: which? Note that because it is linear, you do not need Newton's method. Copy the files, modify them, and repeat tasks 1.3–5. Then try the matrix

$$A = \begin{pmatrix} -1 & 100 \\ 0 & -30 \end{pmatrix}$$

with $y_0 = (1 \quad 1)^{\mathrm{T}}$, and $t_0 = 0$ and $t_{\mathrm{f}} = 10$. Apply both the explicit and implicit Euler methods. For the plot of the error as a function of time t, use $N = 100$ and $N = 1000$. Do both methods have the same stability characteristics? What are the major differences in your observations, and can you give a full explanation?

Task 1.7 If time permits, construct `TRstep`, `TRint`, `TRerrVSh` for analyzing the *Trapezoidal Rule*. See the lecture notes for the method formula.

Note that you should be able to use the same code structure that you already have, and that this task may take no more than five to ten minutes if you just make copy of your previous files and replace the function call to

the solver by a call to `TRstep` etc. If you have an object-oriented Python code, you might get away with even fewer changes.

In particular, repeat Task 1.3 and compare to the Euler methods. Verify that the Trapezoidal Rule is a *second-order* convergent method. Make an error plot, solving the same linear differential equation using both the implicit Euler method and the Trapezoidal Rule, and compare the accuracy. How much more accuracy do you obtain using `TRstep`?

2. Project tasks. Explicit adaptive Runge–Kutta methods

Theory. An explicit Runge–Kutta method for the initial value problem $y' = f(t, y)$ is a method of the form exemplified by the classical 4th-order Runge–Kutta method (also known as RK4)

$$\begin{aligned} Y_1' &= f(t_n, y_n) \\ Y_2' &= f(t_n + h/2, y_n + hY_1'/2) \\ Y_3' &= f(t_n + h/2, y_n + hY_2'/2) \\ Y_4' &= f(t_n + h, y_n + hY_3') \\ y_{n+1} &= y_n + \frac{h}{6}(Y_1' + 2Y_2' + 2Y_3' + Y_4'). \end{aligned}$$

A single step of the method can then be described as follows. The method "samples" the right-hand side $f(t, y)$ at four different points to compute the four **stage derivatives** Y_1', Y_2', Y_3' and Y_4'. Then it forms a linear combination of these derivatives to obtain the "average derivative" to advance the solution from y_n to y_{n+1}.

A general Runge–Kutta method is defined by its coefficients a_{ij} for evaluating the Y_i', and the coefficients b_j for forming the linear combination of these derivatives to update the solution. The method can be written

$$hY_i' = hf(t_n + c_i h, y_n + \sum_{j=1}^{s} a_{ij} hY_j'), \qquad i = 1, \ldots, s,$$

$$y_{n+1} = y_n + \sum_{j=1}^{s} b_j hY_j'.$$

We see that the method is represented by two *coefficient vectors*, c and b, and the *coefficient matrix* A, usually arranged in the **Butcher tableau**

$$\begin{array}{c|c} c & A \\ \hline y & b^{\mathrm{T}} \end{array}$$

For the classical RK4 method, 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

All methods used in practice have $c_i = \sum_j a_{ij}$, so it is sufficient to know the matrix A and the vector b. Check that this condition holds for RK4.

Task 2.1 Write a MATLAB function

```
unew = RK4step(f, told, uold, h)
```

that takes a single step with the classical RK4 method. Here `f` is the function defining the differential equation. Then test it by solving a simple problem of your choice. For example, you could use the linear test equation $y' = \lambda y$, and **verify that the global error is $\mathrm{O}(h^4)$ by plotting the error in a log–log diagram**, like you did for the Euler methods. This check ensures that your implementation is correct. Note that you now need to define the differential equation (`f`) in a separate MATLAB m-file.

Theory. In a similar manner, a 3rd order RK method called 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

Here we see that some of the evaluations of the right-hand side f are the same as for the classical RK4 method. In fact, we can write both methods *simultaneously* as

$$\begin{aligned}
Y_1' &= f(t_n, y_n) \\
Y_2' &= f(t_n + h/2, y_n + hY_1'/2) \\
Y_3' &= f(t_n + h/2, y_n + hY_2'/2) \\
Z_3' &= f(t_n + h, y_n - hY_1' + 2hY_2') \\
Y_4' &= f(t_n + h, y_n + hY_3') \\
y_{n+1} &= y_n + \frac{h}{6}\,(Y_1' + 2Y_2' + 2Y_3' + Y_4') \\
z_{n+1} &= y_n + \frac{h}{6}\,(Y_1' + 4Y_2' + Z_3').
\end{aligned}$$

With *five* evaluations of the right-hand side f instead of four (the extra evaluation being Z_3'), we can obtain *both* a 3rd order approximation z_{n+1} and a 4th order approximation y_{n+1} from the same starting point y_n. This can be used to *estimate a local error by the difference* $\ell_{n+1} := z_{n+1} - y_{n+1}$.

When two methods use the same function evaluations, we say that we have an **embedded pair** of RK methods. The embedded pair above is called RK34. In practice, one doesn't compute z_{n+1}. Instead, one computes the error estimate directly from

$$\ell_{n+1} := \frac{h}{6}\,(2Y_2' + Z_3' - 2Y_3' - Y_4').$$

This is not only to save one unnecessary computation, but because subtracting two very similar values is sensitive to round-off errors and can lead to an inaccurate estimate of the error.

Task 2.2 Starting from your `RK4step`, write a MATLAB function

```
[unew, err] = RK34step(f, told, uold, h)
```

that takes a single step with the classical RK4 method and puts the result in `unew`, and uses the embedded RK3 to compute *a local error estimate* in `err` as described above.

Theory. Let us use the Euclidean norm throughout and assume that the local error $r_{n+1} := \|\ell_{n+1}\|_2$ is of the form

$$r_{n+1} = \varphi_n h_n^k$$

if the step size h_n was used. The coefficient φ_n is called the **principal error function** and depends on the method as well as on the problem.

Our goal is now to keep $r_n = \text{TOL}$ for a prescribed accuracy tolerance TOL. The idea is to vary the step size so that **the magnitude of the local error remains constant along the solution**. In order to do this, let us assume that φ_n varies slowly (treat it as if it were "constant").

Now suppose that r_n was a bit off, i.e., there is a deviation between r_n and the desired value TOL. How would we change the step size in order to eliminate this deviation? If φ is constant we seek a step size h_n such that

$$\begin{aligned} r_n &= \varphi h_{n-1}^k \\ \text{TOL} &= \varphi h_n^k, \end{aligned}$$

where the second equation says that the step size has been changed to h_n so that the error becomes equal to TOL in magnitude. As we know the old

step size h_{n-1}, as well as the estimated error r_n and the tolerance TOL, we can solve for the next step size h_n. Thus, eliminating φ from the equations above by dividing the equations, we find

$$h_n = \left(\frac{\text{TOL}}{r_n}\right)^{1/k} \cdot h_{n-1}.$$

This is the simplest recursion for controlling the step size and is the desired algorithm for making the RK34 method **adaptive**. The power k is the order of the error estimator. Because RK4 has local error $\mathrm{O}(h^5)$ and RK3 has local error $\mathrm{O}(h^4)$, the difference between the two methods is $\mathrm{O}(h^4)$. This means that you should use the value $k = 4$ for your adaptive RK34 method.

This simple step size controller is, however, not particularly good. Turning to control theory for better alternatives, you will use a proportional–integral controller (PI controller) with your RK34 code,

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

which is more robust. Here we use the current error estimate, as well as the previous error estimate. On the very first step, however, no "previous" error estimate is available. For $n = 1$ we then put $r_0 = \text{TOL}$, after which the recursion will start operating as intended.

Task 2.3 Write a function

```
hnew = newstep(tol, err, errold, hold, k)
```

which, given the tolerance `tol`, a local error estimate `err` and a previous error estimate `errold`, the old step size `hold`, and the order `k` of the error estimator, computes the new step size `hnew` using (2).

Task 2.4 Combining `RK34` and `newstep`, write an adaptive ODE solver

```
[t,y] = adaptiveRK34(f, t0, tf, y0, tol)
```

which solves $y' = f(t, y); \quad y(t_0) = y_0$ on the interval $[t_0, t_\mathrm{f}]$, while keeping the error estimate equal to `tol` using the step size control algorithm you implemented above. In the vector `t` you store the time points the method uses, and in `y` you store the corresponding numerical approximation, as a row vector for each value of `t`. See below how you need to arrange the function `f`. Make sure that in its last step, the method *exactly hits the end point* `tf`. Thus, in the last step your solver `adaptiveRK34` has to "override" the value `hnew` supplied by `newstep`.

In order to start the integration, you also need to pick an initial step size. We suggest using the formula

$$h_0 = \frac{|t_\mathrm{f} - t_0| \cdot \mathrm{TOL}^{1/4}}{100 \cdot (1 + \|f(y_0)\|)}.$$

3. Project tasks. A nonstiff problem

Theory. A classical model in biological population dynamics is the **Lotka–Volterra equation**,

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

where a, b, c, d are positive parameters. The equation models the interaction between a predator species, y (foxes), and a prey, x (rabbits). If no foxes are present ($y = 0$) the rabbits multiply and grow exponentially. On the other hand, if there are no rabbits ($x = 0$), the foxes have no food supply and die at a rate determined by d. The product term, xy, represents the probability that a fox encounters a rabbit within their shared ecosystem. This encounter benefits the fox, which eats the rabbit, so the product term is positive in the second (fox) equation, and negative in the first (rabbit) equation.

The Lotka–Volterra equation is *separable*. By dividing the two equations, we get

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

Written in terms of differentials, we have

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

and by integration we obtain $cx - d\log x = a\log y - by + K$. Hence the function

$$H(x, y) = cx + by - d\log x - a\log y$$

remains constant at all times, i.e., $H(x, y)$ is **invariant** along solutions. This means (non-trivially, proof omitted) that the Lotka–Volterra equation has **periodic solutions**.

Task 3.1 Choose the parameters a, b, c, d as $(3, \quad 9, \quad 15, \quad 15)$ and pick some suitable positive initial values, preferably not too far from the equilibrium point $(d/c, a/b)$, e.g. $(1, 1)$. Use your own adaptive RK34 solver to

solve the problem. Run with a tolerance of (say) 10^{-6} or 10^{-8}. (You can use tighter tolerances still, if your code is good enough.) Simulate the system for at least 10 full periods.

Are the solutions periodic as claimed, and how long, approximately, is the period? Plot x and y as functions of time, and plot y as a function of x, i.e., the *phase portrait*. It is often easier to check periodicity there, because the latter plot is a closed orbit if the solution is periodic. You may have to experiment a little with initial conditions and how long time you integrate in order to get a nice plot.

Investigate what happens if you change the initial conditions. Do you get the same periodic solution and does the period remain the same?

Integrate over a very long time (100 – 1,000 full periods, depending on choice of tolerance) to check whether the numerically computed $H(x, y)$ stays near its initial value $H(x(0), y(0))$ or drifts away. This can be done by plotting

$$|H(x, y)/H(x(0), y(0)) - 1|$$

as a function of time, where you insert the computed values of x and y into the expression. Choose a suitable type of plot. Should it be loglog or linlog?

Hints. In order to solve the Lotka–Volterra equation, you need to write a function that computes the right-hand side of the ODE in the following format,

```
function dudt = lotka(t,u)
a = ... ;
b = ... ;
c = ... ;
d = ... ;
dudt = [ a*u(1)-b*u(1)*u(2); c*u(1)*u(2)-d*u(2) ];
```

Note that the argument t is necessary, although it will not be used by your solver in this case. The reason is that this format is used by MATLAB's and Python's built-in solvers, whether or not the right-hand side depends explicitly on t. By following this format you can easily switch to using those built-in solvers, in case your own solver is not efficient enough.

4. Project tasks. Nonstiff and stiff problems

Theory. In the beginning of this instruction set, you used both the *explicit Euler method*,

$$y_{n+1} = y_n + hf(t_n, y_n)$$

and the *implicit Euler method*,

$$y_{n+1} = y_n + hf(t_{n+1}, y_{n+1}).$$

In an **implicit** method, every step is (far) more expensive due to the necessary (nonlinear) equation solving. This extra expense can pay off, however, **if one can take much longer steps** with the implicit method.

This happens in **stiff problems.** Because all explicit methods have bounded stability regions, the maximum stable step size is limited (see lecture notes). A well designed implicit method, however, can have an *unbounded stability region*, permitting much larger steps without losing accuracy. This makes up for the extra work incurred by equation solving. The explicit and implicit Euler methods are the prototypical examples of this.

The **van der Pol equation**,

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

models an electric oscillator circuit. The solution is periodic, with a period of approximately 2μ. The system may be stiff or nonstiff depending on the parameter μ. Use the initial condition $y(0) = \begin{bmatrix} 2 & 0 \end{bmatrix}^{\mathrm{T}}$.

Task 4.1 Solve the van der Pol equation for $\mu = 100$ on the interval $[0, 2\mu]$ using your own explicit, adaptive RK34 code. Plot the solution component y_2 as function of time. Further, plot y_2 as a function of y_1 (the phase portrait). In the latter plot, try modifying the initial values, and check that the solution always tends to the same oscillation. Unlike the Lotka–Volterra equation, where you get different orbits, the van der Pol equation only has a single orbit, known as a **limit cycle**.

Task 4.2 We are now going to explore stiffness, and how it depends on μ. In all computations, use the initial condition $y(0) = \begin{bmatrix} 2 & 0 \end{bmatrix}^{\mathrm{T}}$, and for every given μ, solve the problem on the time interval $[0, 0.7\mu]$, still using your own adaptive solver.

Solve the problem for the "E6 series" of values of μ, i.e., solve the problem for $\mu = 10, 15, 22, 33, 47, 68, 100, 150, 220, 330, 470, 680, 1000$. You may have to cut this series short if your code takes exceedingly long time to solve for say $\mu = 1000$. If the computation suddenly takes a suspiciously short time, check the results: "values" such as 10^{87} or NaN indicate that the adaptivity broke down and a much too large step size was used. Then we are outside the stability region, and both the RK approximations blow up in unpredictable

ways. This can happen due to a too large initial step size, and a too stiff problem. Such results are not proper approximations.

Collect data by recording how many steps your solver needs to complete the integration in each case. Plot the total number of steps N as a function of μ in a loglog diagram. Use a suitable tolerance for all computations, at least $\text{TOL} = 10^{-6}$. Can you conclude that $N \sim C \cdot \mu^q$? What is the power q? The increase in the number of steps needed is proportional to the stiffness. How does stiffness depend on μ?

Task 4.3 Read the documentation in MATLAB (using `help` and other sources) on how to use the stiff solver `ode15s` to solve differential equations. In Python, use the function `scipy.integrate.solve_IVP` and specify the parameter `method='BDF'`. This is roughly equivalent to `ode15s`. Repeat the experiments from Tasks 4.2 using one of these solvers, using the same data as before. What happens? Which code performs better, and why? When you plot the number of steps as a function of μ, how does the graph differ from the one you got with your own nonstiff RK34 code? Can you run the built-in solver for $\mu = 10,000$ and higher? (Don't even think of doing that with your own *explicit* code.) Does it take longer wall-clock time to solve a problem with μ large?

Please note that the Matlab/Python solvers do not outperform your adaptive RK34 method because they are "better implemented", it is because they use implicit methods. We could do that as well, and the performance would be similar. But setting up an *implicit* Runge-Kutta method requires a bit more work, and so does getting a proper error estimate via an embedded scheme. If time permits, you can try applying your non-adaptive implicit Euler (or trapezoidal rule) method. It will likely work for any μ, but due to the lack of adaptiveness it will fail to pick up the parts where the solution changes very quickly, unless the step size is very small.

Final remarks. The project report needs to address the tasks raised in Sections 2 – 4 of this handout. You don't need to report on tasks in Section 1, as they are only intended to get started, building elementary solvers and learning the techniques needed for the project tasks.

In your report, you should also comment on what you have learned in this computer project. Do not forget to acknowledge help and inspiration that you received, and include literature references as well as references to the lecture notes published on the web wherever needed. (See the instructions.)

Please feel free to give positive as well as negative feedback for future use in the course.