Training Unit 5: ODE numerical solver

1 INTRODUCTION TO TRAINING UNIT

After covering the theory of numerical solvers in the lecture, it's time to program your own solver and gain practical experience with solvers in general. Since numerical solvers are widely used in physics, engineering, and various other fields to model complex systems, understanding their behaviour, strengths, and limitations is essential.
As discussed, there are many types of numerical solvers available. Some of the best-known include:

- Euler's Method
- Runge-Kutta Family of Solvers
- Trapezoidal Method
- Heun's Method

In this training unit, you'll start by implementing a basic numerical solver. To verify its accuracy, you'll compare the results with an analytical solution and analyse the deviation between your numerical approximation and the exact solution.Next, you'll implement a second numerical solver, as in most real-world scenarios, an analytical solution is typically unavailable. Finally, you'll develop a variable step size control based on the deviation between the two numerical solvers.

2 PREPARATION

- For this unit you will need your Matlab and mathematical skills

3 REPORT

- The report for this training unit is based on your **Matlab live script**, that you will start from the first task.
- Make sure to structure your work in the live script using the available editor tools (structure your code in section, add headers to your sections, table of contents, use bold or italic formats, bullet points, etc.)
- As you will consistently add new features to your live script, make sure to add new sections to each task, to provide a well-documented script. This will require you to copy-paste some of your code throughout the document. Use the subtask number and header also for your script as section header for a more consistent documentation
- Try to make use of UI-controls throughout the script, to generate an interactive script, especially for the last training unit task
- Be sure to check the correct function of your scripts before uploading it. If your script **does not work**, you **will not receive any points** for this training unit.
- Make sure, that the script shows **your work** on this training unit task. Using solutions from other colleagues will result in zero points for you and your colleague.
- **Caution (!)** be sure to check the **due date** that is set for uploading the files on Moodle!

4 LABORATORY WORK

4.1 Introduction and setting-up your live script

- For this training unit you will work in a Matlab live script, which serves as your *main coding file* as well as your *main result presentation file* and your *documentation file*.
- For getting ready to use live scripts and if you have never used on before, make sure to read through this short tutorial (LINK)
- Try using the control functionality within MATLAB Live Scripts to modify parameters easily and intuitively. This will allow you to adjust values interactively, observe the effects on your results in real-time, and streamline experimentation with different parameter settings
- Explore the Live Script functionality by experimenting with different features and reviewing some of the community-created Live Scripts, which are available free of charge. These examples can give you a better sense of the possibilities with Live Scripts (LINK). (Hint: If you're logged into your MathWorks Online account, you can view and interact with these Live Scripts directly in your browser using MATLAB Online

4.2 Derive the analytical solution to the ODE under investigation

- To test the numerical solver, we first need a differential equation that has both an analytical and a numerical solution. This allows us to compare the results and verify that the solver has been implemented correctly.
- For a first try, let's take following differential equation:

$$EQ1: f(t,y) = \frac{dy}{dt} = \frac{-y}{\tau}$$

- Recall the steps for solving first-order differential equations and try to derive the analytical solution from EQ1. For documentation, use MATLAB's built-in *Equation Editor* located in the Live Script toolbar under *Insert* > *Equation*, and outline the main steps to reach the analytical result, which is:

$$y(t) = y_0 \cdot e^{-\frac{t}{\tau}}$$

4.3 Implement the analytical solution to your live script

- To be able to plot the analytical solution, the initial condition is required. For this example, we take:

$$y_0 = 1$$

- Try to implement the **analytical solution** in you Matlab live script and calculate the solution with following parameters:

$$t_0 = 0$$
$$t_{final} = 10$$
$$h = 1$$
$$\tau = 2$$

- Plot the result of the calculation in your live script. Does it resemble the **e-function** as expected?

4.4 Implement the numerical solution to your live script

- With the analytical solution established, use the numerical solver formula provided below to implement the solver
- To begin with a straightforward solver, implement the Forward Euler method as discussed in the lecture
- The forward Euler method is represented by following formula:

$$y_{k+1} = y_k + h \cdot f(t_k, y_k)$$

- After implementing the solver, run a test with the differential equation from the subtask before
- Plot the result of the solver by using the same parameters as for the analytical solution
- Does the numerical solution resemble the analytical one?
- Plot the results in one plot and remember to use proper axis naming and curve legends
- Now let's improve the readability of the live script by:
  - Exporting the analytical and numerical solution to external Matlab functions
  - What parameters should be given to the function, what should be returned
  - Review the data type, that you used to store the result of the function calls. Could there be better suited data types to store the values y_k and the time stamps? (Try to use a Matlab table as a data storage)

4.5 Investigate the error between analytical solution and numerical solution

- Add a new function to your live script, that takes the two calculated solutions of the differential equation and calculates the following error values:
  - Absolute error

$$E_k = x_{true\ value} - x_{estimation}$$

  - Relative error

$$e_k = \frac{E_k}{x_{true\ value}} = \frac{x_{true\ value} - x_{estimation}}{x_{true\ value}} = 1 - \frac{x_{estimation}}{x_{true\ value}} \cdot 100\ [\%]$$

  - Average total error

$$e_{mean} = \frac{1}{n}\sum_{i=1}^{n} e_i$$

- Plot the error results in a clear and meaningful way. Try adding the local error to the existing plots using MATLAB's `hold on` function.
- Use a "Subplot" or "tiledlayout" for the relative error to visualize both figures in one plot (upper plot showing the analytical, numerical solution and the absolute error, lower plot showing the relative error)
- Connect both **x-axis** (upper and lower plot) together, using the "linkaxes" function
- Now both figures should be synchronised, test it by panning one subplot in the x-direction. The other subplot should be synchronized as well regarding the x-axis. This makes further investigations easier

4.6 Try out different step sizes and investigate the behaviour of the solver

- Change the step size h and investigate its effects on the errors.
  - If the time t_{final} is too short / long, change this parameter as well
  - Does it get instable at specific step sizes?
  - How does the mean error e_{mean} change over the step size control h (can you create a plot that shows the dependency $e(h)$)
- What can you examine further? → Make sure to document it in your script
- As you will require several plots, to undermine your investigations, add more figures to your script and describe each figure with your investigation results. (e.g. add 3 plots for 3 step sizes and add documentation to each plot, or examine Matlab's plotmatrix function for that purpose)

4.7 Add a second solver to your live script

- As discussed in the lecture unit, you already know that, normally the exact solution of the differential equation is unknown (if it would be known, a numerical solver would be superfluous)
- Therefore, the way to obtain an accurate result normally, is by applying a second algorithm and comparing both results against each other. This shall be your next task.
- We shall take the Mid-Point (MP) method as our second solver, which is a particular case of Runge-Kutta second order method:

$$k_1 = f(t_k, y_k)$$

$$k_2 = f\left(t_k + \frac{h}{2}, y_k + k_1 \cdot \frac{h}{2}\right)$$

$$y_{k+1} = y_k + h \cdot k_2$$

- Implement the numerical solver as you already did with the forward Euler method and test it again with the differential equation EQ1

4.8 Investigate your second solver

- As before, investigate your new solver by adding a new graph to your plots
- How does it behave in comparison to Euler's forward method
- How does the error change with step size control
- Make sure to document it in you live script adding several figures to your live script

4.9 Add a step size control to your numerical calculation

- The last step in this training unit is the automatic step size control, as it is performed within Simulink automatically by using the variable-step solver (e.g. ODE45)
- Therefore, we need to specify the maximum tolerable error which shall be $r = 0.05$
- Within your main loop, you need to calculate the error **ε** between the Mid-point and Euler method at every iteration
- If the local error **ε** is smaller than the specified tolerance, the next iteration can be calculated
- If the local error **ε** is greater than the specified tolerance, the step size needs to be changed, and a new iteration shall be started
- The error and new step size calculation shall be implemented according to following formulas:

$$h_{new} = \frac{h_{old} \cdot r}{\varepsilon}$$

$$\varepsilon = \left| y_{k_{MP}} - y_{k_{Euler}} \right|$$

- Test your implementation with different tolerance values and check if the algorithm works fine
- If something does not work as intended (e.g. Matlab runs into an **endless loop**) use **Strg+C** to end current simulation run
- Add representative plots that show the variation of the step size
- Analyse the influence of r on h . What can you see?

4.10 Additional task

- Now it would be a good time to add some of Matlabs Live Scripts **controls**. Try to make this live script as simple and user-friendly as possible.
- Ideal would be a small App-like UI, where only the control fields are visible (Check out Example "Estimating Sunrise and Sunset" under this LINK, to get some ideas of what it could look like)
- Note (!): Do not delete the previous figures and implementations, as these are part of your documentation. Add a new section, that includes all your previous work into one App-like UI design
- Parameters like h, r, t_{final}, should be changeable by using sliders or similar UI functions.

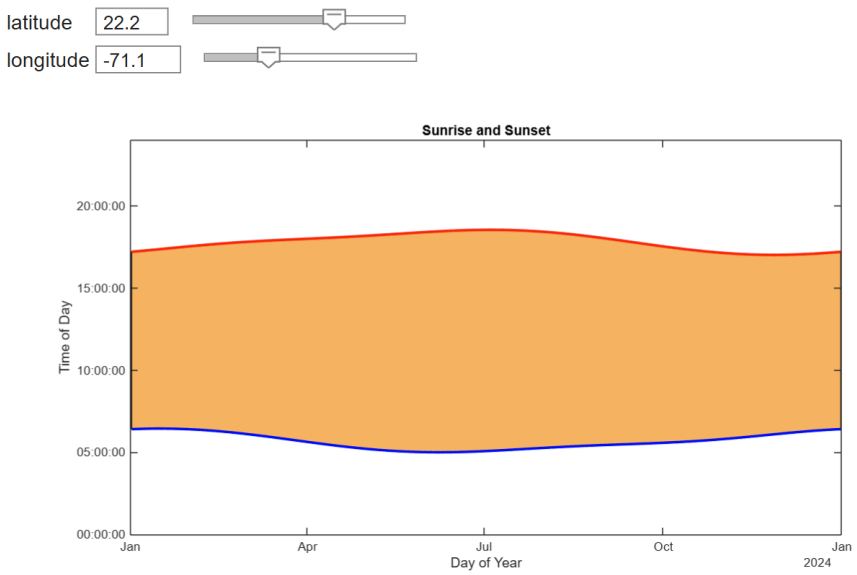

Figure 1: Mathworks Live Script "Estimating Sunrise and Sunset" [1]