Project 0 in FMNN10 and NUMN20

Goals: The goal of this project 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.

Warm-up. This initial "warm-up" project is not mandatory, and you do not have to present your work on it in any way. However, we **strongly recommend** that you still do these exercises, because you will use exactly the same kind of code structures in the next project. It is much easier and less total work to figure out any complications in this simpler setting and then transfer that knowledge to the more complicated setting, than trying to do it all at once.

Start early. The idea is to start working with these exercises as soon as possible in the first week, identify things you have forgotten or are unsure about, and then get help with these on the first exercise. This year, one of the teaching assistants will also give a mini-lecture at this first exercise, which covers basic MATLAB syntax, useful code structures and how to handle typical error messages. You will get much more out of this if you have already made an attempt to solve the exercises.

Contents: Examine numerical stability and computational accuracy of the explicit and implicit Euler methods, 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.

The explicit Euler method

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 what you have done and why, when presenting your work (in the following projects).

Task 1.2 Using `eulerstep`, construct a function

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

that approximates the solution to (1) using N explicit Euler steps of equal size 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`. Also store and return the temporal grid in the variable `tgrid`. You will not explicitly need the grid inside this particular function, but you will in later similar functions. By defining it here where h is also defined, you don't need to set it up in your test script as well. This avoids code duplication and common errors related to that.

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. Note that we can only compute the error here because we know the exact solution to this simple test problem! In a real problem, we could only (at best) estimate it.

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 the `numpy.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. When presenting your work in the following projects, 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 grid $t_0, t_1, \ldots, t_N$ 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.

The implicit Euler method

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?

Bonus: the trapezoidal rule

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`?