Aschaffenburg University of Applied Sciences
Faculty for Engineering
Prof. Dr. Michael Möckel

Email: Michael.Moeckel@th-ab.de
Telephone: +49 (0)6021 / 4206 - 507

Numerical Methods and Simulation
Module MT 03
Winter term 2022/23

https://moodle.th-ab.de/course/view.php?id=624

Problem Set 10

Numerical solvers for ODEs

Coursework

Exercise 1: Explicit Euler method as a solver for the IVP

An explicit ODE solver calculates an approximate numerical solution from a given initial point (x_0, y_0) by linear extrapolation w.r.t. the slope m.

$$\begin{aligned} x_{i+1} &= x_i + h \\ y_{i+1} &= y_i + m * h \end{aligned}$$

For a single step algorithm the value of the slope is calculated by evaluating the ODE at one point.

(1) The explicit Euler method is a simple example for such an algorithm. It constructs the slope directly as the derivative at the (already known) corner of the next interval.

$$m = \left.\frac{dy(x)}{dx}\right|_{x_i}$$

(2) Show that the Euler methods corresponds to a first order Taylor expansion of the solution around the point (x_i, y_i).

(3) Implement the explicit Euler method in Matlab yourself by writing your own ODE solver, following the conventional syntax for ODE solvers `[x,y] = MyEuler(odefun, span, y0, n)` with odefun being a function of two variables (x, y), span=$[a, b]$ being the interval for which the solution has to be calculated, $y0$ the initial value at $x = a$ (which can be a column vector for higher order ODEs) and n the number of steps.

(4) Run your solver `MyEuler` for simple examples, e.g. $dy/dx = x$ with the initial condition $y(0) = 0$ in the interval $[0, 3]$ or $dy/dx = \sin(x)$ with the initial condition $y(0) = -1$ in the interval $[0, 2\pi]$. Plot the results for different step sizes h and add the analytical solution to your plot.

(5) An obvious problem with the Euler method is that it estimates the solution over an time interval using only information from the beginning of the interval. Transform the following second order ODE to a system of first order ODEs and run the Euler method.

$$\frac{d^2x}{dt^2} + x = 0 \qquad x(0) = 1 \qquad \left.\frac{dx}{dt}\right|_{x=0} = 0$$

What does the ODE describe? Plot the two components of the solution w.r.t. each other. What result do you expect?

Some problems of single point can be avoided by using multipoint methods. Here we illustrate a multipoint improvement of the Euler method.

(6) *Modified Euler method.* Similar to the trapezoidal rule in numerical integration an improvement of the Euler method can be obtained by calculating the slope as the average of the derivatives at the beginning and the end of each interval

$$m = \frac{1}{2}\left(\left.\frac{dy(x)}{dx}\right|_{x_i} + \left.\frac{dy(x)}{dx}\right|_{x_i+h}\right)$$

Implement the modified Euler method.

Exercise 2: Solving ODEs in Matlab

Matlab has a large variety of solvers for differential equations. We will discuss a selection of solvers from the ode suite (there are other solvers, too!). They are labelled with a number and some additional letters, e.g. `ode23`, `ode45` or `ode23tb`. Each of them has advantages and disadvantages and is most useful for a certain class of ODEs. Most of them have a very similar Matlab syntax: `[X, Y] = solver(odefun, xspan, y0, options, parameters)`. Note that the ODE must be given in *normal form*, i.e. as a first order ODE. Then `odefun` represents the non-differential ('right hand') side of the ODE. `xspan` gives the range $[x_1, x_2]$ for which the ODE has to be solved, `y0` sets the right hand side of the initial condition $y(x_1) = y0$. `options` and `parameters` are optionally flags. Parameters are passed down to the function `odefun` and must fit the signature of this function. If parameters are set but no options chosen a placeholder `[]` must be inserted instead.
Note that ODEs are used a lot in physics and engineering for time dependent problems. Then the independent variable x is replaced by t and the dependent variable $y(x)$ is replaced by $x(t)$.
Solver. A good starting point is the general purpose solver `ode45` which uses a variable step Runge-Kutta algorithm. Its tolerances are set by default parameters which could be changed if needed.

(1) Solve the initial value problem $y'(x) = \sin(x)/(y-1)$ in the range $[0, 2\pi]$ with the initial condition $y(0) = 2$. Define an anonymous function `f @(x,y)` for the right hand side of the ODE and note that the independent variable must always come first. Then use f as the input function in `ode45` and plot the result.

(2) Consider a growth process given by the ODE $dx/dt = \exp(t)\cos(x)$ with the initial condition $x(0) = 0$ for $t \in [0, 3.5]$. Plot the solution.

Exercise 3: The Lorenz[1] system

Assume a, b, and r positive and consider the system of three coupled ODEs which represents a simplified model for the atmospheric turbulences in a thunderstorm. The derivative is w.r.t. time.

$$\begin{aligned} x' &= -ax + ay \\ y' &= rx - y - xz \\ z' &= -bz + xy \end{aligned}$$

Is this set of ODEs linear or nonlinear? Can they be re-written as a matrix equation?

(1) To represent this set of ODEs in a Matlab file (which can be used as an input for the `ode45` solver) one has to pass both the coordinates and the parameters to the file. It is advisable to create vectors `u = [x y z]` and `p = [a b r]` for this purpose (an alternative is to use *global parameters* with the command `global a b r`).

(2) Write a Matlab function `[uprime] = LorenzModel.m(t, u, p)` which implements the system of ODEs and returns the vector of the derivatives as a column vector.

(3) Solve the Lorenz model for the initial conditions `u0 = [1 2 3]` and the parameters `p = [10 8/3 28]` by calling the solver `[t, u] = ode45(@LorenzModel, [0,7], u0, [] , p)` and plot the result.

(4) Inspect the plot and discuss the *transient* behaviour of the system (for small times) and the *steady state* behaviour at long times.

(5) We want to study the long-term behaviour and investigate the dependence of the system of ODEs on the choice of the initial conditions. Use randomized initial conditions, solve the model for an enlarged time span of `[0, 100]` and plot only values after the transient regime for $t > 10$. This can be done by selecting a subset of data points v by `v = u(find(t>10), :)`. Plot the coordinates (u_1, u_2, u_3) as points of a 3D plot using `plot3(v(:,1), v(:, 2), v(:,3))`. Put all commands in a new m-file `LorenzRun.m`. This could contain the following lines:

```
a = 10; b = 8/3; r = 28; p = [a b r];
u0 = 100*(rand(3,1) - 0.5);
[t, u] = ode45(@LorenzModel, [0,7], u0, [] , p)
v = u(find(t>10), :)
plot3(v(:,1), v(:, 2), v(:,3))
```

The plot (a kind of butterfly) shows the so-called *Lorenz attractor*. Run it several times, i.e. with different random initial conditions. It should always result in a similar plot. This is a particular feature caused by the nonlinearity of the Lorenz system.

[1] E. N Lorenz, meteorologist and mathematician, published 1963

Exercise 4: Comparing different solvers for initial value problems

An initial value problem consists of an ODE and an initial condition. Consider the following simple sinusoidal example with an arbitrary frequency f.

$$\frac{dy}{dx} = \sin(fx) \quad , \quad y(0) = 2$$

Firstly, we consider this ODE at a large frequency $f = 1000$.

(1) Solve the initial value problem by hand for an arbitrary frequency and plot the exact solution for $f = 1000$. How many points would you choose for a meaningful discretization of the interval $[0, 3]$?

(2) Use two different solvers to calculate a numerical solution with Matlab for $x \in [0, 3]$ and for $f = 1000$: (i) `ode21` which is a standard solver for non-stiff problems and and (ii) `ode21s` which is a standard solver for stiff problems. Plot the result.

(3) Error handling. Obviously, the result of `ode21` solver is insufficient. However, you can improve the result by increasing the numerical accuracy. You can do so by specifying the numerical resolution (tolerances) of the solvers. First type `odeset` in the command window and study the output! What are the default values for the realtive and absolute tolerance? Then set your own levels of accuracy. For this create an `options` variable by `options=odeset('RelTol',10e-5,'AbsTol',10e-8)` and add the variable `options` to the solvers parameters.

(4) Try other solvers with default tolerances, e.g. `ode45` (Runge-Kutta standard solver) or `ode23t` (based on trapezoidal rules).

(5) Include `tic` and `toc` commands around the ode solver and use a meaningful `disp()` command outside of the solver to identify the various runtimes of the solvers. Compare!

(6) Repeat the same analysis for smaller frequencies (f = 100, f = 30) and plot the results in a new figure.

Exercise 5: Delayed differential equations (DDE)

.
A delayed differential equation is a differential equation in which the derivative of the unknown function $y(t)$ at time t depends on the unknown function at an earlier time (e.g. $y(t-2)$), i.e. there is a delayed feedback. Consider the following example for $t \in [0, 300]$:

$$\frac{dy}{dt} = \frac{2y(t-2)}{1+y(t-2)^5} - y(t) \qquad , \qquad y(t) = 0.5 \quad \text{for} \quad t < 0 \tag{1}$$

(1) Implement the right hand side of the DDE as a Matlab function `function dydt = dderhs(t,y,z)`. Introduce a new variable $z = (t-2)$.

(2) Solve the DDE with the special solver `dde23(ddefun,lags,history,tspan,options)` which receives the delay (lag) and the history of the function before the initial time as the first arguments `res=dde23(@dderhs,2,0.5,[0,300])` and returns a struct `res` with various components. Plot the two components of the result w.r.t. each other `plot(res.x, res.y)`.

Overview: Main numerical methods for solving ODEs

Solving ODEs numerically is done with incremental algorithms: They start at the point of the initial value and, step by step, construct the solution at later points. There are different types of algorithms:

(1) *Single step* algorithms calculate the next function point from the values of the function and the derivative at the current point only, i.e. $y(x_{i+1}) = A_{SS}(y(x_i), y'(x_i))$.

(2) *Multistep* algorithms calculate the next function point from the values of the function and the derivative at the current and at earlier points, i.e. $y(x_{i+1}) = A_{MS}(y(x_i), y'(x_i), y(x_i - 1), y'(x_i - 1)\ldots)$.

(3) *Explicit* methods (algorithms) do not depend on $y(x_{i+1})$ (explicit formula for y_{i+1}).

(4) *Implicit* methods (algorithms) *do* depend on $y(x_{i+1})$, i.e. $y(x_{i+1}) = A^{\text{imp}}(y(x_i), y'(x_i), \ldots, y(x_{i+1}))$. Such algorithms are normally nonlinear and require numerical techniques based on root finding.