Lecture MATLAB Scip-continuation course, ODE

Differential equations

Heikki Apiola and juha Kuortti

February 15, 2018

Aalto University
juha.kuortti@aalto.fi, heikki.apiola@aalto.fi

Numerical methods of ODE's

Ordinary differential equation (ODE)

- A scalar (first order) differential equation is of the general form: $y' = f(t, y)$
- Solution: A differentiable function $t \to y(t)$, that satisfies: $y'(t) = f(t, y(t))$ on an interval $a < t < b$
- Initial value problem (IVP) Require: solution $y(t)$ satisfies the "initial condition" $y(t_0) = y_0$ for some $t_0 \in (a, b)$ and given initial value y_0. (Often t represents time and $t_0 = 0$.)

Differential equations,3 aspects

1. **Existence theorems, analytic solutions**
   - *Picard–Lindelöf(1870-1946)*
   - Various methods and tricks, most importantly for **linear equations**, Computer algebra (`CA`) helps and extends.
2. **Qualitative methods**: Make conclusions directly from the equations without solving them. *Direction fields, isoclines, critical points*. Global view.
3. **Numerical methods** Most ODE-systems of use in applications can't be solved analytically (or the solution – perhaps produced by `CA`– is too complicated for efficient computation). Numerical methods are increasingly important, especially with computers. **This is our main concern here**.

The interplay between all three aspects is most fruitful and necessary. Numerical methods alone are "blind", the 2 first give the necessary insight and help understand erros and limitations.

Differential equations, more

- The fundamental theorem of calculus gives:

$$y(t) = y(t_0) + \int_{t_0}^{t} y'(s)ds = \int_{t_0}^{t} f(s, y(s))ds.$$

  (Numerical) integration can't be used in general, since $y(s)$ is unknown, unless f depends only on t.
- Special cases:
  - f depends only on t $\Rightarrow$ Solution is the integral function of $f(t)$.
  - f depends only on y $\Rightarrow$ the equation is called autonomous. This simplifies the situation in ways to be discussed. Many of our examples, especially with systems will be *autonomous*.

Systems of ODE's, higher order equations

- Many models involve more than one unknown function, and/or higher order derivatives. They can be handeled by making y and f vector valued: $\vec{y}$ and $\vec{f}$.
- **Exampe:** Harmonic oscillator: $y'' = -y$.
  Denote:$y_1 = y, y_2 = y' = y_1'$, then we get the system:
  $$\begin{cases} y_1' = y_2 \\ y_2' = -y_1 \end{cases} \qquad \text{Thus, if } \vec{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix},$$
  we have the equation: $\vec{y'} = \vec{f}(t, \vec{y}) = \begin{bmatrix} y_2 \\ -y_1 \end{bmatrix}$.
- This transformation will become routine when we proceed.
- The user of ODE-solvers needs to be able to do it. The point here is that the methods for one equation translate almost verbatim to systems, just draw (or imagine) the vector-arrow.

Harmonic oscillator: code into MATLAB

The above equation will be coded into Matlab either as an m-file:

```
function yp=harmonicA(t,y)
% t is not used in this (autonomous) case.
% y is a column vector of 2 components.
yp=[y(2);-y(1)];
```

or as a function handle (or anynomous function):

```
harmonicB=@(t,y)[y(2);-y(1)]
```

- Note: The variable t has to be present even if it is not used in the function definition.
- The call of an ODE-solver has one of these two forms:
  (A) `ode23(@harmonicA,...)`
  (B) `ode23(harmonicB,...)`

Lecture task 1, solve harmonic oscillator equation

Let's go a little ahead of our agenda and solve the above system with MATLAB's basic solver ode23 (or `ode45`.)

- To use standard options and variable time step
  `» [T,Y]= ode23(@myODE,[0,10],y0)`
  Here `[0 10]` is the time span and `y0` is the initial value-columnvector at starting time 0.
- Solve the harmonic oscillator first with eg. y0=[1;0].
- `» ode23(F,[0 10],[...])` produces plots.
  Then, capture the output:

$y'' = -y$, output, visualization suggestions

```
[T,Y]=ode23(F,[0 10],[...]);
plot(T,Y),'-*')
legend('y_1(t)','y_2(t)');
grid on
figure  % Open new graphics window
plot(Y(:,1),Y(:,2),'-o')
title('Phase plane of y''''=-y')
axis square
grid on
```

- What are the sizes of `T` and `Y` and what are their contents?
- What does `steps=diff(T);` reveal, especially `[min(steps), max(steps)]` ?

A scalar equation, direction fields and solution curves

Direction field, one scalar equation

Let's go back to one scalar equation to begin with

- At each point of the area of the $ty - plane$, where f is defined, the differential equation determines the direction of the tangent of the solution curve $y(t)$. (That's what the differential equation is all about.)
  At a grid of points (t_i, y_j) in the plane, draw a short line in the direction of the tangent $f(t_i, y_j)$ to get the **direction field**.
  MATLAB offers easy-to-use, efficient tools for drawing it.

Matlab tool for grid points: `meshgrid`

Recall **meshgrid**:

```
>> x=0:2;
>> y=3:6;
>> [X,Y]=meshgrid(x,y);
>> [X Y]     % X and Y side by side
                          y'     y'     y'
    x  0     1     2  |   3      3      3
    x  0     1     2  |   4      4      4
    x  0     1     2  |   5      5      5
    x  0     1     2  |   6      6      6
```

Thus `X` consists of `length(y) (=4)` x-rows,

`Y` consists of `length(x) (=3)` y'-columns,

Grid points (continued)

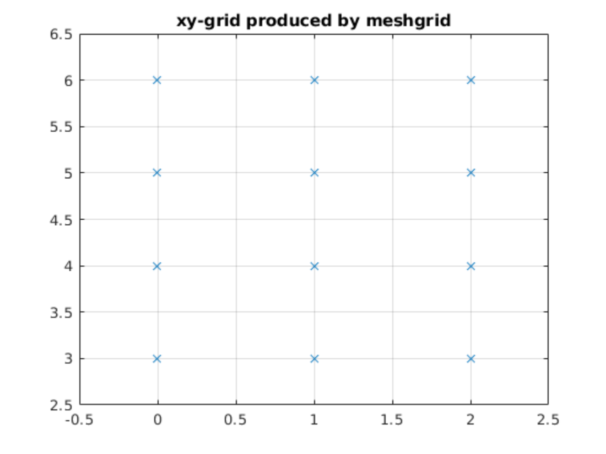
If you list `X` and `Y` in column order side by side, i.e.
`>> gridpoints=[X(:)Y(:)]` you will get a 3×4 rectangular grid of points, let's transpose the display to save space:

```
>> gridpoints'
   0    0    0    0    1    1    1    1    2    2    2    2
   3    4    5    6    3    4    5    6    3    4    5    6
```

This data just waits to be plotted:

```
>> plot(X(:),Y(:),'x')
>> axis([-.5 2.5 2.5 6.5])
>> grid on
>> title('xy-grid produced by meshgrid')
```

Grid points (continued)



More uses of meshgrid:

meshscript.m, meshscript.html (Published html).

Tools for direction field: `meshgrid` and quiver

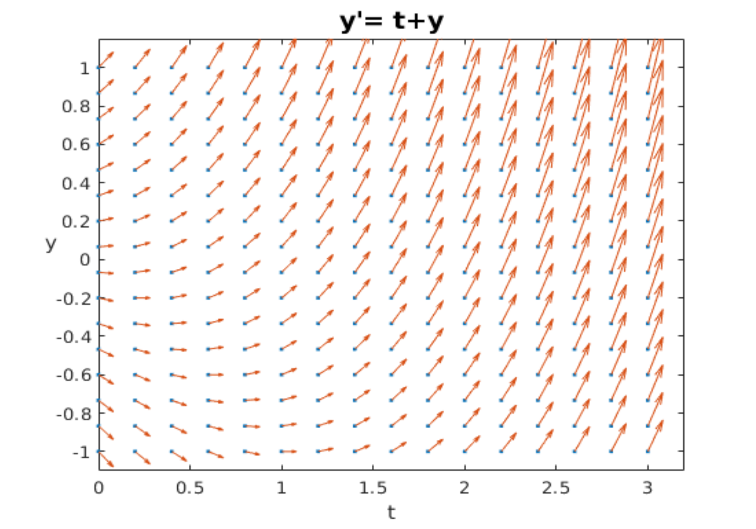
- The command `quiver(x,y,u,v,scale)` plots short arrows starting at the points $[x, y]$ in the direction of vectors with components $[u, v]$. `x,y,u,v` are matrices of the same size produced in most cases with the aid of `meshgrid`.
- This combination of *meshgrid* and *quiver* is good for all kinds of vector fields, like gradient field, etc.

Example of direction field

Let's look at the differential equation $y' = t + y$ The derivative: $y' = f(t, y) = t + y$ gives the slope of the tangent at the point (t, y). Thus the direction of the tangent vector (t, y) is given by $\vec{v} = (1, f(t, y))$.

```
close all
n=16;
tpoints=linspace(0,3,n); ypoints=linspace(-1,1,n);
[t,y]=meshgrid(tpoints,ypoints); % Recall 3d-graphics.
plot(t(:),y(:),'.') % Grid points:
f=@(t,y) t+y
vt=ones(size(y));
vy=f(t,y); % The derivative of solution curve
hold on; quiver(t,y,vt,vy,1.5);
```

```
xlabel('t');ylabel('y','Rotation',0)
xlim([0 3.2]);ylim([-1.1 1.15]); % Tune axis limits
```



Lecture task 2: Direction field and solution curves

Let's continue using MATLAB's ODE-solvers before Euler-introduction:

- Load ODEdirfield.m into your MATLAB-editor.
- Run the code one block of code at a time (`CTR-ENTER`), choosing different initial conditions, and also possibly changing your equation.
- Here you have a piece of code, you can use in principle for any first order ODE later in this course and in your life.

Back to basics, Euler, who else!

One scalar equation

Recall: Initial value problem (IVP):

$$y' = f(t,y), \quad y(t_0) = y_0$$

Let's draw a "direction field arrow" at the initial point (t_0, y_0).
The slope is $y'(t_0) = f(t_0, y_0)$
Let

$$t_1 = t_0 + h, y_1 = y_0 + hy'(t_0) = y_0 + h\,f(t_0, y_0).$$

Thus y_1 is the $y-$ value of the line tangent to the solution curve at t_0 evaluated at t_1.
For h small one can assume the error to be small as well.

Euler's method

Repeating the above step leads to the iteration:
Given initial point (t_0, y_0), compute: [1]

$$y_{k+1} = y_k + h_k f(t_k, y_k), k = 0, \ldots, n$$

Example $y' = t + y, \quad y(0) = 0.$
In this case we know the exact solution. $y(t) = e^t - t - 1.$

Let's demonstrate the use of Matlab's symbolic toolbox.

[1] h_k indicates variable time steps.

Some uses of the symbolic toolbox

```
>> help dsolve   % Symbolic ODE-solver
>> syms y(t)
>> dsolve(diff(y(t),t)==t+y(t))  % General solution:
ans = C1*exp(t) - t - 1
>> dsolve(diff(y(t),t)==t+y(t),y(0)==0) % Initial ...
    value given.
ans = exp(t) - t - 1
```

Little practice: Check the result with these commands:

```
>> syms t
>> y=exp(t) - t -1
>> diff(y,t) == y+t     % Diff equ satisfied ?
>> subs(y,t,0)          % Initial condition ? (help subs)
```

Eulerexample 1

Load the file Eulerexample1.m into MATLAB. It uses the same diff. equation $y' = t + y$.
Study and experiment, one block at a time.

Writing Euler's method as a function

In the above script one could define `f=@(t,y)t+y` and write a generic code using $f(t, y)$ in the script. Better still: Write a function **myEuler**: (Type `>>which euler` to see why you should avoid the name euler.)

```
function [T,Y]=myEuler(f,Tspan,y0,n)
% Euler's method for solving a single IVP
%  - Function call:
% [T,Y]=myEuler(f,Tspan,y0,n)
%  - Input arguments:
% f     -- function handle defining the diff. equ.
% Tspan -- vector [a b].
% y0    -- Initial value at the point a.
% n     -- Nr. of subintervals.
```

Euler code continued

```
%  - Output arguments:
% T      -- ``Time-vector''
% Y      --  Vector of Euler-solutions at T-points.
% Example:  y'=t+y, y(0)=1
%           f=@(t,y)t+y;
%           [T,Y]=myEuler(f,[0 4],1,6);
%            plot(T,Y,'*--');grid on
% Code starts here:
a=Tspan(1);b=Tspan(2);
h=(b-a)/n;
% Complete the code
....
```

Lecture task

- Download the file: myEulerTemplate.m
  or just copy/paste the above code into your Matlab editor.
- Rename into ``myEuler.m'' (Make sure, the function name is `myEuler` as well.) Complete the code. When done, type: `>>help myEuler` and run the help-example. Then try some other examples.

In addition to getting to know Euler's method and its coding in Matlab, you will get an understanding of how the ODE-functions in MATLAB are built and used.

A few words about error analysis

- Standard tool in numerical analysis:
  The Taylor expansion of the (unknown) solution function $y(t)$.
  $$y(t+h) = y(t) + h\,y'(t) + O(h^2) = y(t) + h\,f(t, y(t)) + O(h^2).$$
- *Taylor's theorem* gives the formula $\frac{y''(\xi)}{2}h^2$ for the *local truncation error* $=$ *error made at one step*, which is of the form $O(h^2)$ (proportional to h^2 for small h).
- Taking n steps, the global error is of the order nh^2, where n is proportional to $\frac{1}{h}$, thus the global error is of the order $O(h)$.
  (This reasoning is valid for s.k. *stable* equations, see later.)
- Typical error behavior: **The (global) error is approximately halved when the stepsize is halved**.
  Euler's method, though inefficient, is the easy-to-understand starting point of all numerical methods of ODE's

Lecture exe 3 Eulerloop.m

Load the file Eulerloop.m
into Matlab and run one block at a time, let's discuss it ...

Better numerical methods, MATLAB's ODE-suite

Midpoint Euler

Euler's method is of the form:

$$t_{i+1} = t_i + h, y_{i+1} = y_i + m\,h,$$

where m=slope. For Euler, m is the slope $f(t_i, y_i)$, at the start of the step t_i, that is "follow your nose". For fancier methods, you first "sniff ahead". *Midpoint Euler* uses the slope m at the midpoint of the segment of an Euler step, that is:

$$m = f(t_i + \frac{h}{2}, y_i + \frac{h}{2} f(t_i, y_i)).$$

Runge-Kutta

- The 4^{th} order *Runge-Kutta* is the most commonly used method of that order, and converges considerably more rapidly than *Euler*.
- It uses a slope that is a weighted average of 4 "intermediate" slopes:
  $m_1 = f(t_i, y_i)$
  $m_2 = f(t_i + \frac{h}{2} m_1, y_i + \frac{h}{2} m_1)$
  $m_3 = f(t_i + \frac{h}{2}, y_i + \frac{h}{2} m_2)$
  $m_4 = f(t_i + h, y_i + h\, m_3)$
  $m_{RK} = \frac{1}{6}(m_1 + 2m_2 + 2m_3 + m_4)$
- MATLAB-implementation is straihgtforward, let's look at it more closely in connection with systems. (Here's the link already: rk4V.m)

Systems of ODE's, equations of order > 1

Systems of ODE's

```
function [T,Y]=eulerV(Fsys,Tspan,ya,n)
% ...
ya=ya(:)'    % Make row vector
a=Tspan(1);b=Tspan(2);
h=(b-a)/n;
N=length(ya);
Y=zeros(n+1,N);  % j^{th} col:  Y(1,j), Y(2,j), ..., ...
    Y(N,j)
T=a:h:b;
Y(1,:)=ya;        % First row
for i=1:n
   Y(i+1,:)=Y(i,:)+h*(Fsys(T(i),Y(i,:))');
 end;
```

EulerV example

Predator-pray (rabbits and foxes)

$$\begin{cases} \frac{dr}{dt} = 2r - \alpha\, r\, f\,, r(0) = r_0 \\ \frac{df}{dt} = -f + \alpha\, r\, f\,, f(0) = f_0 \end{cases}$$

Denote: $y_1 = r, y_2 = f$

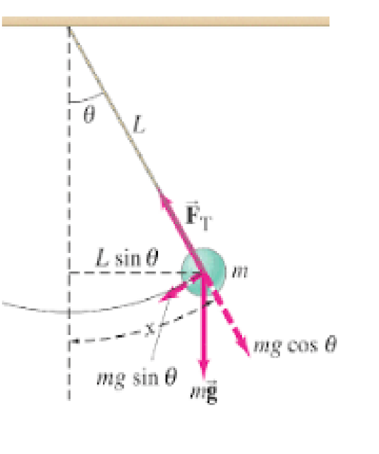
```
function ydot = rabfox(t,y)
alpha=0.01
ydot = [2*y(1)-alpha*y(1)*y(2)
        -y(2)+alpha*y(1)*y(2)];
```

Parameter α taken as a local variable, function handle “direct” definition doesn't work. Later we will see more elegant ways.

Lecture task, eulerV

- Load the file eulerV.m
  Write the above "rabfox"-code into a file `rabfox.m`.
- Write a script `runrabfox.m`. Edit some parameters: Take `r0=300, f0=150`.
- Run and Plot $r(t)$ and $f(t)$ on the time-axis and phase-plane in separate figures. Use `legend` in the time-picture and title in both.
- Experiment with about `tf=8` and especially N, starting at `N=20`. Increase to something like $N = 200$ and more. How small step is needed to see (in figures) (almost) periodicity.
- Publish your script.

Pendulum-example



The equation of motion can be written as a differential equation for $\Theta(t)$.

Arc length $s(t) = L\Theta(t) \Rightarrow$ acceleration: $s''(t) = L\Theta''(t)$
The equation of motion: $mL\Theta''(t) = -mg\sin(\Theta(t))$, or

$$\Theta''(t) = -\frac{g}{L}\sin(\Theta(t))$$

Denoting $y_1 = \Theta, y_2 = \Theta' = y_1'$ leads to the system:

$$\begin{cases} y_1' = y_2 \\ y_2' = -\frac{g}{L}\sin(y_1) \end{cases}$$

Take $g/L = 1$ and write the equation in vector form:

$$\vec{y}' = \begin{bmatrix} y_1' \\ y_2' \end{bmatrix} = \vec{f}(t, \vec{y}) = \begin{bmatrix} y_2 \\ -\sin(y_1) \end{bmatrix}$$

Note: I wrote $\vec{f}(t, \vec{y})$ although in this case the function $\vec{f}$ doesn't depend on t ("autonomous system").

To solve numerically with MATLAB:

1. Write code for the function $\vec{f}$, call it `myPendulum`. Either define a function handle:
   `myPendulum=@(t,y)[y(2);-sin(y(1))` or edit an m-file:
   ```
   function dy=myPendulum(t,y)...
   ```
2. Call the solver: `[T,Y] = ode45(myPendulum,Tspan,y0);`
   Note: In the m-file case you must include the `@`-sign, i.e. `[T,Y] = ode45(@myPendulum,Tspan,y0);` to tell the solver (`ode45`) that the argument is a function handle. (Rule of memory: There must be one `@`-sign here or there.)
3. Results: `T` is a column vector of time points used.
   `Y` is a 2-column matrix: col_j : y_j-values, $j = 1, 2$
   In this case: Col_1: Θ-values, Col_2: Θ'-values.

Results, continued

- In other words: The i^{th} row of `Y` approximates the solution $(y_1(t), y_2(t))$ at $t =$`T(i)`.
- Visualization:
  `plot(T,Y)` plots the solutions $y_1(t)$ and $y_2(t)$ on the given `Tspan` (Remember: Since `Y` is a matrix (with 2 columns), this command plots both columns against the `T`-column. (Same as `plot(T,Y(:,1),T,Y(:,2))`)
- In case of an autonomous system (like the pendulum) it is often more instructive to look at the phase plane, i.e. "velocity vs. position", i.e. the curve $(y_1(t), y_2(t)), t \in [a, b]$. The ode45 output matrix `Y` gives the required data right away: Just type: `plot(Y(:,1),Y(:,2));` NICE!