Lab module: Time-stepping Ordinary Differential Equations

Johan Jansson (jjan@kth.se)

February 2, 2017

0 Jupyter-FEniCS web PDE solver environment

The address of the web Jupyter-FEniCS cloud environment, described more in detail below, is provided via email with the ip of the cloud virtual machine and Jupyter login. To run a program in Jupyter-FEniCS, open the Python 2 notebook and select the Run command under the Cell menu.

You can find example Jupyter notebooks with implementations from the lab under the `src` directory in the zip archive of the lab module, which you can upload in the Jupyter interface.

NB!: The files in the web environment are **not** stored on disk, which means they will disappear if the system is rebooted. Do not forget to save your notebook regularly to your own computer, by using Download as > IPython Notebook (.ipynb) under File in your notebook.

You can also set up the environment at your own computer by using the command:

```
sudo docker run -t -i -p 80:8000 sputnikcem/fenics-jpy
```

and using the login name and passwords listed on the terminal window by accessing localhost from a web browser.

1 Introduction

In this lab session you will use the Jupyter and Numpy Python environment to formulate time-stepping methods for solving Ordinary Differential Equations (ODEs), and analyze the methods and investigate some of the concepts for ODE in the course.

The goal of this session is to:

1. Become familiar with time-stepping methods and a general software implementation for ODEs.
2. Learn the Forward Euler, Backward Euler and Trapezoid time-stepping methods, unified in the θ-method.
3. Understand the concepts *convergence*, *stability* and *energy conservation* of time-stepping methods.
4. Investigate particle systems modeled by systems of ODEs: Newton's second law - conservation of momentum - with elastic and gravity forces.

For reference material, please see the lecture notes from the DD1354 Models and Simulation course at KTH [2] and the book Computational Differential Equations [1].

2 Exercises

2.1 Time-stepping - constructing primitive functions

We would like to construct a primitive function $u(t)$ of its time-derivative given by the function $f(t)$. The problem can be formulated as follows:

Find the function $u(t)$ which solves the initial value problem:

$$\begin{cases} \dot{u}(t) = f(t) & \text{for } t \in [0, T] \\ u(0) = u_0 \end{cases} \tag{1}$$

We use a systematic time-stepping approach to construct the function $u(t)$ that we can then also apply to ODEs. The fundamental theorem of calculus gives the representation of the function:

$$u(\bar{t}) = \int_0^{\bar{t}} f(t)dt + u(0) \quad \text{for } \bar{t} \in [0, T] \tag{2}$$

Note: the solution $u(\bar{t})$ is a function which we can evaluate at any point in the time domain.

For a computable approximation, a realization of the *Riemann sum*, we introduce a partition of the time domain into subdomains with a chosen uniform refinement level n, and $I_i^n = (t_{i-1}^n, i_i^n]$ with length, or *step size* $k_n = 2^{-n}$ and $N = 2^n$ nodes $t_i^n = ik_n, i = 0, ..., N$. We approximate $u(t)$ by the continous piecewise linear function $U^n(t)$ defined by:

$$U^n(t_j^n) = \sum_{i=1}^{j} k_n f(t_j^n) \quad \text{for } j = 0, ..., N \tag{3}$$

Note: again, the solution $U^n(t)$ is a function, a continous piecewise linear function which we can evaluate at any point in the time domain, where the values in the nodes are given as above.

2.2 One time-step

We can step-by-step construct this sum by *time-stepping* like so:

$$U^n(t_i^n) = U^n(t_{i-1}^n) + k_n f(t_i^n) \quad \text{for } i = 0, ..., N \tag{4}$$

This time-stepping approach is what we will use to solve general ODEs, and now as a first step to construct primitive functions, solutions to (8).

We generalize the time-step formulation to allow evaluation of f in an arbitrary point in the subdomain I_i^n. We also give the solution function itself U as an argument to f to prepare for the more general ODE case. We thus define the *theta-method* as:

```
def step_prim(f, t0, u0, k, method): # One step for: u' = f(t), u(0) = u0, returns u(t + k), Theta method
  thetadict = {'Forward Euler': 0., 'Backward Euler': 1., 'Trapezoid': 0.5}; theta = thetadict[method]
  t = t0 + k;
  u = u0 + (1. - theta)*k*f(t0, u0) + theta*k*f(t, u)
  return u
```

Choosing theta as 0 gives the left end-point (aka. "Forward Euler"), 1 the right end-point (aka. "Backward Euler"), and 0.5 the mean value of the two endpoints (aka. "Trapezoid") of the subdomain.

2.3 Stepping the whole time domain

Iterating over all subdomains, i.e. stepping through the entire time domain can be expressed as the following *solver*:

```
def solve_prim(f, I, u0, k, method): # Solve ODE u' = f(t, u), u(0) = u0, I = [t0, T], returns array of t and u
  tarr = linspace(I[0], I[1], (I[1] - I[0])/k + 1); uarr = zeros([size(u0), tarr.size]); uarr[:, 0] = u0
  t = I[0]; T = I[1]; u = u0; i = 1
  for t in tarr[1:]:
    u0 = u
    uarr[:, i] = step_prim(f, t - k, u0, k, method)
    i += 1
  return [tarr, uarr]
```

2.4 Numerical integration error and convergence order

A key question of a numerical method is to be able to estimate the *numerical error* and the *convergence order* of the error.

The numerical error, or *quadrature error* here is defined as:

$$Q_k = \left| \int_0^T f(t)dt - \sum_{i=1}^{j} k_n f(t_i^n) \right| \tag{5}$$

The numerical error, or *quadrature error* here is defined as:

$$Q_k \leq Ck^p \tag{6}$$

where C is a constant independent of the time step k.

We say that a numerical method converges of order p if the error can be estimated like above.

2.4.1 Questions

Try running the given time-stepping implementation to compute the primitive function $sin(t)$ of of $f(t) = cos(t)$:

$$\begin{cases} \dot{u}(t) = f(t) = cos(t) & \text{for } t \in [0, 6\pi] \\ u(0) = u_0 = 0 \end{cases} \tag{7}$$

given by the implementation:

```
def f_cos(t, u): # harmonic model ODE
  return array([cos(t)])
```

Answer the following questions:

- Try out the three timestepping methods for the $f(t) = cos(t)$ model problem. Compare against the exact solution $sin(t)$. Successively halve the timestep k as described above and experimentally investigate the convergence order of the three methods and tabulate or plot (in a log-log scale) the convergence order.
- Compute primitive functions of other given $f(t)$ functons. For example, the primitive function of $f(t) = x^x$ cannot be represented by elementary functions, what does it look like?

2.5 Solving ODEs by time-stepping - harmonic oscillator

We now generalize the time-stepping approach to general systems of ordinary differential equations (ODEs). We would now like to construct a function $u(t)$ which solves an ODE. The problem can be formulated as follows:

Find the function $u(t)$ which solves the general ODE:

$$\begin{cases} \dot{u}(t) = f(t, u(t)) & \text{for } t \in [0, T] \\ u(0) = u_0 \end{cases} \tag{8}$$

We can now simply generalize our time-stepping method to also be dependent on u: We can step-by-step construct this sum by *time-stepping* like so:

$$U^n(t_i^n) = U^n(t_{i-1}^n) + k_n f(t_i^n, U(t_i^n)) \quad \text{for } i = 0, ..., N \tag{9}$$

We now have to solve a non-linear algebraic equation for every time-step, we described how to do this via fixed-point iteration in the next section. Below we give the implementation of the generalized theta-method for ODEs:

```
def step(f, t0, u0, k, method): # One step for ODE: u' = f(t), u(0) = u0, returns u(t + k), Theta method
  thetadict = {'Forward Euler': 0., 'Backward Euler': 1., 'Trapezoid': 0.5}; theta = thetadict[method]
  M = 5; t = t0 + k; u = u0 # Starting guess
  for i in range(0, M): # M fixed-point iterations
    u = u0 + (1. - theta)*k*f(t0, u0) + theta*k*f(t, u)
  return u
```

now with the adjusted solver for all time-steps:

```
def solve(f, I, u0, k, method): # Solve ODE u' = f(t, u), u(0) = u0, I = [t0, T], returns array of t and u
  tarr = linspace(I[0], I[1], (I[1] - I[0])/k + 1); uarr = zeros([size(u0), tarr.size]); uarr[:, 0] = u0
  t = I[0]; T = I[1]; u = u0; i = 1
  for t in tarr[1:]:
    u0 = u
    uarr[:, i] = step(f, t - k, u0, k, method)
    i += 1
  return [tarr, uarr]
```

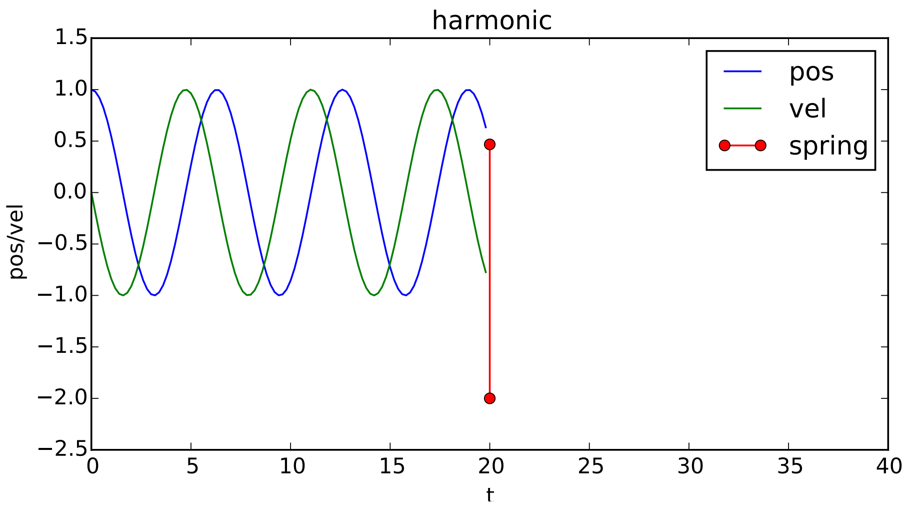
As a model problem we will use the *harmonic oscillator* - a linear mass-spring system with unit mass and length, by giving $f(t,u)$ as:

$$\begin{cases} u &= [u_0, u_1] \\ \dot{u_0}(t) &= u_1 \\ \dot{u_1}(t) &= -u_0 \quad \text{for } t \in [0,T] \\ u_0(0) &= 1 \\ u_1(0) &= 0 \end{cases} \tag{10}$$

with implementation:

```
def f_hrm(t, u): # harmonic model ODE
  return array([u[1], -u[0]])
```

with example solution:



2.5.1 Solving non-linear algebraic systems - fixed-point iteration

If the ODE is non-linear, i.e. the function $f(t,u)$ is non-linear in u, then the equation for one time-step is a non-linear algebraic equation, here we give the time-step equation for Backward Euler:

$$U(t_i) = U(t_{i-1}) + kf(t_i, U(t_i)) \tag{11}$$

where the value for $U(t_i)$ is the unknown.

A systematic method for solving non-linear algebraic equations is *fixed-point iteration*, where the equation is put in fixed-point form:

$$x = g(x) \tag{12}$$

and the iteration is performed as:

$$x^j = g(x^{j-1}) \tag{13}$$

where j is the iteration index.

The fixed-point iteration converges if the Lipschitz constant $L = |g'(x^j)| < 1$.

Our time-step equation is already in fixed-point form, and the Lipschitz constant is proportional to the time-step k, so for a small enough time-step, the iteration is guaranteed to converge.

2.5.2 Stability and energy conservation

The *stability* of a model means how the solution grows over time. We say that a model is stable if the solution, here measured as the maximum value over the time interval $|\cdot|_{L_\infty}$, can be bounded by a constant C:

$$|u(t)|_{L_\infty} \leq C \tag{14}$$

The stability of a numerical method is also of key importance. Again, if we can show that the numerical solution U is bounded by a constant, we say the method is stable:

$$|U(t)|_{L_\infty} \leq C \tag{15}$$

However, while stability is critical, a stable method may in some sense be too stable, damping out the solution unless the time-step is very small. The Backward Euler method can be described this way.

A more refined stability property is *energy conservation*, which is equivalent to showing that the total energy E of the model is exactly conserved over the whole time interval:

$$\dot{E} = 0 \tag{16}$$

and the same statement for the discrete energy E^k of the numerical solution:

$$\dot{E^k} = 0 \tag{17}$$

We now proceed to investigate the energy conservation of the harmonic model problem (10), and the Trapezoid method applied to it.

Energy conservation in the harmonic model problem We define position as $x = u_0$ and the velocity as $v = u_1$ in the model. The potential energy is then $P = x^2$, the kinetic energy $K = v^2$ and the total energy $E = P + K$.

We multiply equation 0 $\dot{x} = v$ by v and equation 1 $\dot{v} = -x$ by x, then add them together:

$$\dot{v}v + \dot{x}x = -xv + vx \tag{18}$$

$$\Rightarrow \tag{19}$$

$$\frac{d}{dt}(\frac{1}{2}v^2 + \frac{1}{2}x^2) = 0 \tag{20}$$

The total energy E is conserved!

Energy conservation for the Trapezoid method We write out the Trapezoid method for the full system, which is the theta-method with theta = 0.5. Again, we define $X = U_0$, $V = U_1$ and we arrange the timestep equation in the form:

$$X(t_i) - X(t_{i-1}) = k\frac{V(t_i) + V(t_i)}{2} \tag{21}$$

$$V(t_i) - V(t_{i-1}) = -k\frac{X(t_i) + X(t_{i-1})}{2} \tag{22}$$

We now multiply equation 0 by $\frac{V(t_i)+V(t_{i-1})}{2}$ and equation 1 by $\frac{X(t_i)+X(t_{i-1})}{2}$ and add the equations together to get:

$$(V(t_i) - V(t_{i-1}))\frac{V(t_i) + V(t_i)}{2} + (X(t_i) - X(t_{i-1}))\frac{X(t_i) + X(t_i)}{2} = \tag{23}$$

$$k\frac{V(t_i) + V(t_i)}{2}\frac{V(t_i) + V(t_{i-1})}{2} - k\frac{X(t_i) + X(t_{i-1})}{2}\frac{X(t_i) + X(t_{i-1})}{2} \tag{24}$$

$$\Rightarrow$$

$$\frac{V(t_i)^2 - V(t_{i-1})^2}{2} + \frac{X(t_i)^2 - X(t_{i-1})^2}{2} = 0 \tag{25}$$

$$\Rightarrow$$

$$X(t_{i-1})^2 + V(t_{i-1})^2 = X(t_i)^2 + V(t_i)^2 \tag{26}$$

The total energy of the numerical solution is conserved for every time step, i.e. for the entire time domain!

2.5.3 Questions

Try running the given time-stepping implementation to compute the trigonometric functions $sin(t)$ and $cos(t)$.

Answer the following questions:

- Try out the three timestepping methods for the harmonic oscillator model problem and compute and plot the total energy of the system. How does the energy behave over time? What can you say about the stability of the methods?
- Successively increase the timestep and verify the stability limit of the fixed-point iteration.

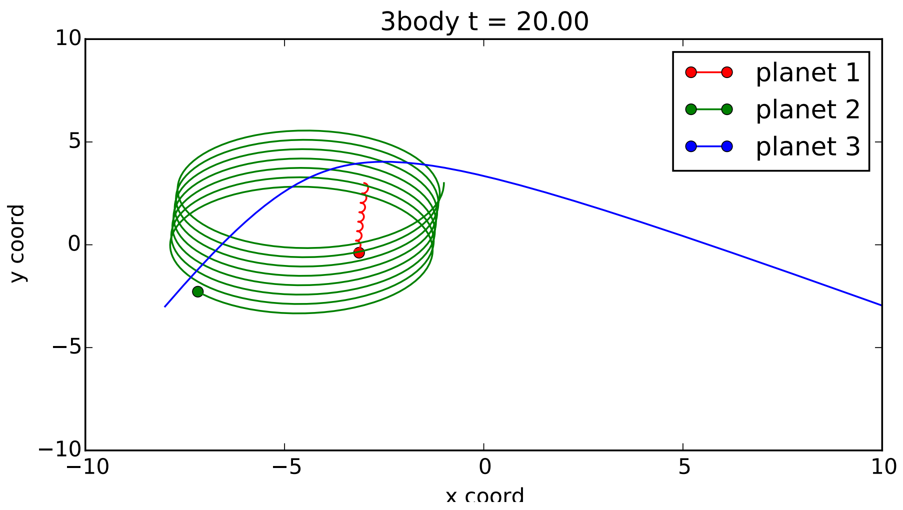
2.6 N-body systems

We now introduce a specific $f(t, u)$ modeling a system of N point-masses with mass m_i for point-mass i such as a planetary system or mass-spring system. The forces between point-masses i and j consist of an elastic force (Hooke's law) $K(r - L)e$ and a gravitational force $m_i m_j G e/r^2$, where K is the spring stiffness, L the rest length of the spring, r the distance between the point-masses, e a normalized direction vector between the point-masses and G the gravitational constant.

We give a Python formulation of the N-body model below, where the expressions are written out:

```python
def f_3body(t, u):
  fval[:] = 0.; m = [200, 4, 1]; K = 0.*100.0; L = 0.*2.0 # mass m, stiffness K, rest length L
  for i0 in range(0, M):
    for i1 in range(0, M):
      if(i0 != i1):
        r = norm(u[i1:2*M:M] - u[i0:2*M:M]); e = (u[i1:2*M:M] - u[i0:2*M:M])/r; # radius and direction
        fval[i0 + 2*M::M] += (K*(r - L)*e + m[i1]*m[i0]*e/r**2) / m[i0] # Elastic and gravity forces
  fval[:2*M] = u[2*M:];
  return fval
```

By setting the elastic stiffness to 0 we only have gravity and can model a planetary system, allowing us to investigate the behavior of the methods in a more complex and realistic system. An example solution is given below:



2.6.1 Questions

Try running the planetary simulator and try different initial conditions.

Answer the following questions:

- Try to model a planet with an orbiting moon and a comet entering and disrupting the orbit.
- Try out the three timestepping methods for the planetary system, and try a range of timestep sizes. What can you say about the performance and accuracy of the methods for the planetary system model?

References

[1] Kenneth Eriksson, Don Estep, Peter Hansbo, and Claes Johnson. *Computational Differential Equations*. Cambridge University Press New York, 1996.

[2] Johan Hoffman and Christopher Peters. Lecture notes for dd1354 models and simulation, 2016.