Introduction to numerical simulation

Single step and multistep methods

Zdeněk Hurák
November 26, 2019

SIMULATION of a dynamic system is just another keyword for numerical solution of the underlying set of differential and/or algebraic equations. Unlike the modeling skills, which are easily understood as indispensable for an engineer, the simulation skills are often relegated to numerical mathematicians who develop the computational packages. These are then viewed as black boxes by engineers and used without any awareness of possible shortcomings. A prominent example is Simulink by The Mathworks. It is very common for engineering students to create the block diagrams modeling their system only to learn that the default setting of the simulation parameters does not yield satisfactory simulation results. A panic then arises leading to chaotic changes of many parameters of the solver. Even worse, the failure to detect problems with the numerical solver setting can yield simulation outcomes which are completely wrong and misleading. It becomes clear that at least basic understanding of the underlying techniques is a must for every engineer. Furthemore, it appears that certain directions in the area of numerical simulation were only started thanks to engineers who were able to identify the challenge, such as stepsize control and real-time simulation. Deeper understanding of simulation methods and their limits thus seems necessary for everyone who wants to pursue a career in control engineering. This lecture and the next are a modest contribution to starting your journey.

1 Discretization and Taylor series approximation

Most techniques for systems modeled by ordinary differential equations approach the problem by discretizing the time axis. The key tool for this is Taylor series approximation. Consider a system modeled by

$$\dot{x}(t) = f(x, u, t) \tag{1}$$

where $x(t)$ is a state vector, $u(t)$ is the input vector and consider that the state is known at some (initial) time t_0

$$x(t_0) = x_0. \tag{2}$$

The system state at time $t = t_0 + h$ can be obtained from a Taylor series approximation. Assuming a scalar model (for notational simplicity) the Taylor series is

$$x(t_0 + h) = x(t_0) + \left.\frac{\mathrm{d}x(t)}{\mathrm{d}t}\right|_{t_0} h + \left.\frac{\mathrm{d}^2 x(t)}{\mathrm{d}t^2}\right|_{t_0} \frac{h^2}{2!} + \ldots. \tag{3}$$

Plugging in the nonlinear state space model (1) yields

$$x(t_0 + h) = x(t_0) + \left. f(x, u, t)\right|_{t_0} h + \left.\frac{\mathrm{d}f(x, u, t)}{\mathrm{d}t}\right|_{t_0} \frac{h^2}{2!} + \ldots. \tag{4}$$

In the vector case when $\mathbf{x}(t) \in \mathbb{R}^n$, the expression is rewritten componentwise.

Apparently, the infinite series is not much useful for practical computations. Instead some finite truncation is used.

2 Approximation accuracy, errors

Obviously, the trunction introduces an error into the computation. It is not the only source of error but a very significant one. Rounding errors introduced by the use of a finite precision floating point arithmetics is another one.

2.1 Truncation errors and approximation order

Consider again the Taylor expansion, but now let's decide that we will only calculate the first, say, three terms and regard the rest as an error

$$x(t_0 + h) = x(t_0) + f(x(t_0))h + \frac{\mathrm{d}f(x(t_0))}{\mathrm{d}t}\frac{h^2}{2!} + \mathcal{O}(h^3). \tag{5}$$

This is an opportunity for us to introduce the popular *Big-Oh concept.*

A function $e(x)$ is said to be $\mathcal{O}(g(x))$ if and only if

$$\lim_{x \to 0} \frac{e(x)}{g(x)} \leq K > 0. \tag{6}$$

Often $g(x)$ is a polynomial and the lowest powers matter most for assymptotic considerations ($x \to 0$). We then say that a function $e(x)$ is $\mathcal{O}(x^n)$ or simply n-th order.

Now, what does all that mean? What is the use of such concept? Clearly if the parameter h (the size of the step in our case) approaches zero, the low-order terms dominate. The $\mathcal{O}(x^3)$ then does not reveal the exact size of the error, but it does reveal that decreasing the step to 1/10 would make the error thousand times smaller. Thus, we have an assymptotic behavior of the error, an *order of the error*.

Note that the same notion is used for studying assymptotic complexity of algorithms, where the integer variable is considered growing to infinity. It should be always clear from the context which concept is used.

The important issue is that by truncating to, say, the first three terms, that is, $n = 2$, not all the terms in Taylor expansion can be computed directly. Namely, the $\frac{\mathrm{d}f(x(t_0))}{\mathrm{d}t}$ is not available, it must be obtained somehow. There are two possible ways:

1. Compute the higher-order derivatives of $f(x)$ analytically. This only works if the differential equations is just one and relatively simple. This restriction is also valid when using some nifty computer algebra systems to compute the derivatives. This approach is not used often but it is good to be aware of this path. We will not develop it further in this course.

2. Estimate the higher-order derivatives of $f(t)$ (or $x(t)$) from the (previously computed) samples of x. The topic of approximate derivatives based on samples of a function usually earns separate chapters in books on numerical methods. Whichever class of algorithms is used, the order of the introduced error must be consistent with the order of the error enforced by the truncation of the original Taylor series. For example, with $n = 2$, the order of the truncation error is 3, hence the derivative of $f()$ should be approximated such that the corresponding term $\frac{\mathrm{d}f(x(t_0))}{\mathrm{d}t}\frac{h^2}{2!}$ still dominates, that is, it must of of order 2. Observing that it contains the quadratic h^2 part, the approximation to derivative must be of order 0. We will come back to this concept when talking about multistep methods. Note that in single step methods the higher order derivatives are evaluated indirectly through multiple function calls within a single integration step.

2.2 Rounding errors

Hopefully none of you needs to be persuaded that the computations carried on current computers using Matlab or similar SW packages are inexact. Ignoring this fact can lead to many frustrating situations. The other day one of the students claimed discovery of a bug in Matlab by noticing that $0.1+0.2 \neq 0.3$. I am not kidding.

```
>> (0.1+0.2)==0.3
ans =
     0
```

Note that the current standard for storing floating point numbers on computers is IEEE 754 an IEEE 854. These define what exactly is meant by **single** and **double** floating point precisions.

C-programmers would recognize the data type called **float** corresponding to IEEE 754 single precision. For quite some time Matlab did not even offer computations with single precision numbers. Currently their new data type is called **single**. In decimal expansion it is valid only down to 7 decimal places ($2^{-23} \approx 1 \times 10^{-7}$). Let us see the impact for the truncation process. Think of some example where $h = 0.001$, $|x| \approx |f| \approx |\dot{f}| \approx |\ddot{f}| = 1$.

$$\begin{aligned} |x(t_0+h)| &\approx |x(t_0)| + |f(x(t_0))h| + \left|\frac{\mathrm{d}f(x(t_0))}{\mathrm{d}t}\frac{h^2}{2!}\right| + \left|\frac{\mathrm{d}^2 f(x(t_0))}{\mathrm{d}t^2}\frac{h^3}{3!}\right| \\ &\approx 1.0 + 0.001 + 10^{-6} + 10^{-9}. \end{aligned} \tag{7}$$

Apparently, the fourth term contributes nothing to the outcome, why bothering to compute it then? The third term is just at the edge. Implementing a second-order method on a microcontroller with 8- or 16-bit arithmetics would be a nonsence (unless using software libraries for double arithmetics, which, however, make the system very heavy and slow).

2.3 Local vs. global errors

The errors discussed so far describe how inacurate the computation of $x(t+h)$ is. How much it differs from a true value. The error is called local error. However, in the end it is very often the value at the end of the simulation interval that matters. For a “well-behaved” algorithm (to be defined in a while by introducing the concept of numerical stability), the global error arises as a composition of the individual local errors. Now we want to guess the order of the global error knowing the local one. The reasoning is roughly that for a unit simulation time interval, the number of steps of lenght h is $1/h$. Therefore, for a second-order algorithm, the global error is first-order.

2.4 Absolute vs. relative errors

This distinction is standard in all other disciplines. Shall we by happy with an error of the order 10^{-6}? Well, it depends. For a variable with the nominal value, say, 10^6, the error seems very small, but for a variable with the value 10^{-5} the error will have detrimental effect. The relative error then seems the right tool.

3 Basic first order methods

3.1 Forward Euler approximation

The last expression (4) gives suggestion for a very simple technique; just keep the first two terms on the right hand side and throw away the rest. We will study later how much error we introduce by this truncation of the infinite

Taylor series. The new algorithm giving an approximate solution relies on the fact that

$$x(t_0 + h) \approx x(t_0) + f(x(t_0))h. \tag{8}$$

This suggests an algorithm

$$x_{k+1} = x_k + f_k h, \tag{9}$$

where we use the new notation x_k as an approximation to $x(t_k)$ and $f_k = f(x_k, u_k, t_k)$ as an approximation to $f(x(t_k), u(t_k), t_k)$. Make sure you understand the difference between the two.

Note that in the linear case $\dot{x}(t) = ax(t)$ the truncated Taylor series turns into

$$\begin{aligned} x(t_0 + h) &\approx x(t_0) + ax(t_0)h \\ &\approx (1 + ah)x(t_0). \end{aligned} \tag{10}$$

(Do not let the linearity-in-x fool you. The derivatives in the Taylor approximation are done with respect to time.)

Hence the algorithm is

$$x_{k+1} = (1 + ah)x_k. \tag{11}$$

Note that extension to a vector case (x regarded as a vector variable) is straighforward. The only difference is that the Taylor expansion is now carried out in the vector sense (the scalar a turning into a matrix A).

This simple method is called *Forward Euler* integration method. The reason for calling it forward is obvious from Fig.1.

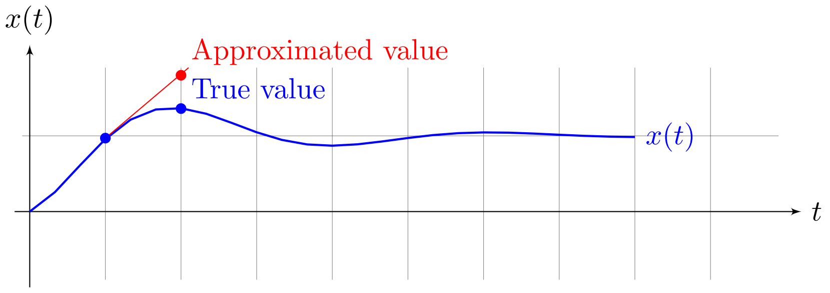

Figure 1: Numerical integration using Forward Euler method.

The increment in $x(t)$ is approximated by its linear part only (called differential). As an approximation of the derivative over the whole interval, the value of derivative at the beginning of the interval is taken.

Algorithm 1 Forward Euler integration (no input $u(t)$)

```
Require: $f(x(t),t)$,  $t_0$,  $t_f$,  $x(t_0)$,  $h$
Ensure: $\dot{x}(t) = f(x(t),t)$
  $t_k \leftarrow t_0$
  $x_k \leftarrow x(t_0)$
  while $t_k < t_f$ do
    $\dot{x}_k \leftarrow f(x_k, t_k)$
    $x_k \leftarrow x_k + h\dot{x}_k$
    $t_k \leftarrow t_k + h$
  end while
```

The algorithms is then very simple, see Alg.1.

This algorithm belongs to a class of algorithms denoted as *explicit* because the value of x at the next time instant can be calculated directly using the values at the previous time instants.

Example 3.1. *Consider the following scalar problem*

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \frac{x - 2tx^2}{1+t}, \quad x(0) = \frac{2}{5}. \tag{12}$$

The exact solution can be found analytically using Symbolic Toolbox for Matlab as

```
>> dsolve('Dx=(x-2*t*x^2)/(1+t)')
```

which returns

$$x(t) = \frac{t+1}{t^2 + C} \tag{13}$$

parameterized by the constant C in addition to the trivial (zero) solution.

Ploting the exact solutions for a few values of the constant C and superposing the numerical solution for a given initial value leads fo Fig.2.

A closer look at Fig.2 reveals that the the accurate (analytical) and the numerical solution do not match perfectly. This is a first evidence that the process of numerical solution brings about some innacuracies. This gets amplified as the sampling period h grows, see Fig.3.

In fact, the current simple algorithm is well known for its poor accuracy when longer integrations steps are used. Where does this poor performance come from? The key contribution to the error is the truncation of Taylor series; in fact, we only kept the first two terms. This suggests that the local truncation error is $\mathcal{O}(h^2)$ while the global error is $\mathcal{O}(h)$. Do the results of numerical computations correspond to this anticipation?

The differences between the computed values x_k at the end of the interval and the true value $x(t_f) = \frac{t_f+1}{t_f^2+C}$ where C was chosen consistently with the initial conditions ($C = 1/5$ for $x(0)$=5) are

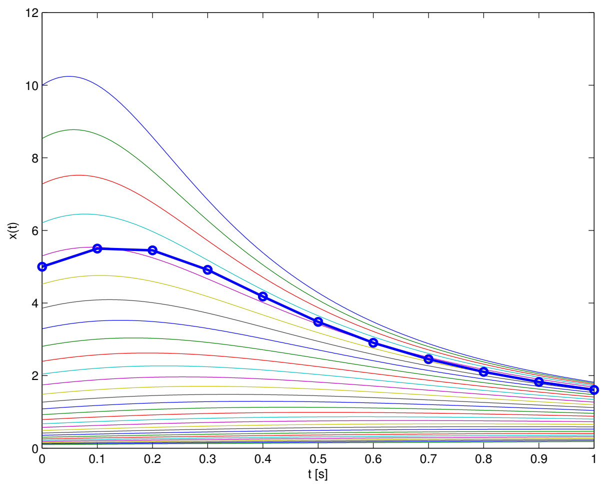

Figure 2: Analytical solutions to the example problem versus the numerical solution obtained by Forward Euler method for a given initial condition and $h = 0.1$.

h	n	$\lvert x_n - x(t_f)\rvert$
0.2	5	3.1155
0.1	10	0.3956
0.05	20	0.1191
0.025	40	0.0542
0.0125	80	0.0261

Clearly, halving the integration step h halves (approximately) the global error. Hence the global error is a linear function of the integration step and it is indeed $\mathcal{O}(h)$. This is not a very admirable property since for reducing the error by one order (in order to get one more valid digit in the result), the number of steps must be increased ten times.

Now, let's have a look at another variant of Euler method.

3.2 Backward Euler approximation

In the Forward Euler method, the increment in $x(t)$ was approximated by its linear part (linear in h, with the derivative of $x(t)$ at the begining of the discrete time interval playing the role of the constant of proportionality). How about using the value of the derivative at the end of the interval? Visually this is sketched at Fig.4.

We can express our new algorithm as

$$x_{k+1} = x_k + f(x_{k+1}, t_{k+1})h. \tag{14}$$

The trouble with this method is that in order to calculate an approximation x_k to $x(t_k)$ at some given time t_k, we also need x_k in order to determine

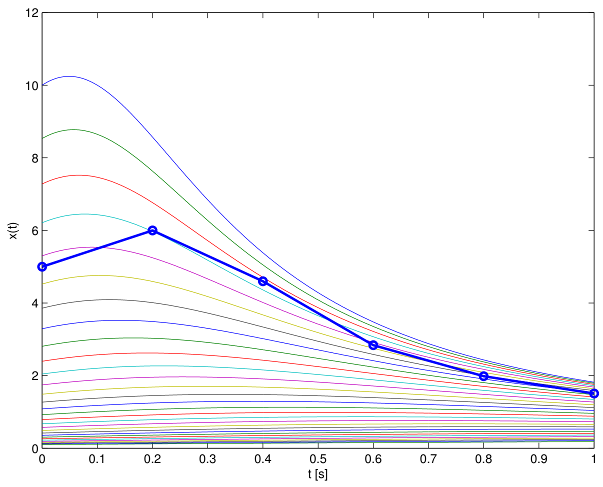

Figure 3: Analytical solutions to the example problem versus the numerical solution obtained by Forward Euler method for a given initial condition and $h = 0.2$.

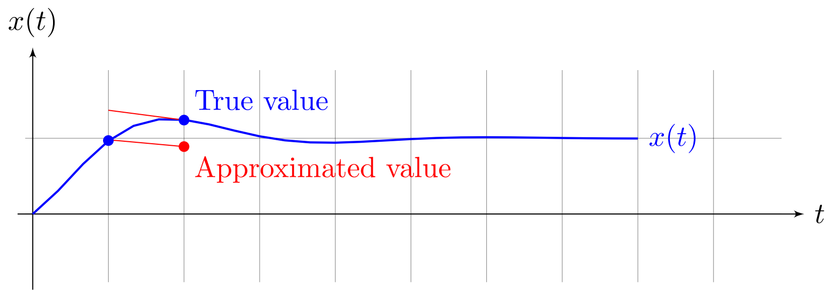

Figure 4: Numerical integration using Backward Euler method.

an approximation $f(x_k, t_k)$ to $f(x(t_k), t_k))$. This method represents a class of methods called *implicit methods*. When not treated properly, this could introduce so-called *algebraic loops*. What is the proper treatment? Usually some sort of solving (systems of) equations is inevitable. Consider a linear time invariant case to illustrate the point

$$\dot{x}(t) = Ax(t), \quad x(0) = x_0 \in \mathrm{R}^n. \tag{15}$$

For a regular sampling, the Backward Euler scheme leads to the following algorithm

$$x_{k+1} = x_k + hAx_{k+1}. \tag{16}$$

Bringing the x_{k+1} terms on the left side gives

$$(I - Ah)x_{k+1} = x_k, \tag{17}$$

which can be rewritten as

$$x_{k+1} = \underbrace{(I - Ah)^{-1}}_{F} x_k. \tag{18}$$

Apparently, in the vector case, there is no way to avoid the need to solve a set of linear equations (note that although we have stated a matrix inversion in the above formula, numerically it would be very inefficient to actually compute it. Instead, a set of linear equations 17 is solved with dedicated solvers (in Matlab use the \ operator).

A pseudocode for the Backward Euler algorithm for a linear system $\dot{x}(t) = Ax(t),\ x(t_0) = x_0$ is here

Algorithm 2 Backward Euler integration for a linear system (no input $u(t)$)

```
Require: $A, \quad t_0, \quad t_f, \quad x(t_0), \quad h$
Ensure: $\dot{x}(t) = Ax(t)$
  $t_k \leftarrow t_0$
  $x_k \leftarrow x(t_0)$
  while $t_k < t_f$ do
    solve $(I - Ah)x_{k+1} = x_k$ for the unknown $x_{k+1}$
    $t_k \leftarrow t_k + h$
    $x_k \leftarrow x_{x+1}$
  end while
```

In a nonlinear system, a nonlinear (typically algebraic) equation will have to solved instead of a linear one. There are a several approaches. Two of them will be discussed shortly, one of them based on fixed point iterations and the other one based on Newton iterations.

Before we delve into that discussion, it may be useful to point out that in the previous two methods we somehow arbitrarily approximated the slope (the derivative) of x by its derivative at the beginning or the end of the step. Both seem to do just an approximate job. But there is no reason why we could not use the derivative of x somewhere inside the interval. It is your task to propose an algorithm that would evaluate f at $t + h/2$.

Another method can be obtained by averaging the results of the two methods (FE and BE). Again, it is your task to write down such algorithm. Later you will also be asked to apply some analysis for the two new algorithms.

3.3 Predictor-corrector method

The celebrated principle of Fixed Point iterations (often regarded as one of the most fundamental results in analysis) deals with the nonlinear equation

$$x = g(x). \tag{19}$$

Existence and uniqueness of such solution is in a very general mathematical setting examined by Banach's *Fixed point theorem* (or *Fixed point principle* or *Contractive mapping theorem*). The message is that a unique solution to (19) exists if and only if $g(x)$ is *contractive* (look it up on your own). As a consequence of the contractivity of $g(x)$, it is guaranteed that if some x_0 is plugged into $g()$, the resulting $x_1 = g(x_0)$ comes a bit closer to the true solution x. Repeating this step, that is, substituting x_1 into $g()$ to get x_2 and then substituting $x_2 \ldots$, converges to the solution x.

Although beyond the scope of this course (most probably covered by some previous math course), we can only state as a fact that in our setting the function g is always contractive. It follows from the analysis of existence and uniqueness of the original ordinary differential equation (remember function f defining the differential equation needs to be Lipschitz for an equation to have a unique solution).

Why not applying the principle here? First we need to get some estimate of a solution in order to get close enough to the solution. This is *predicted* by the Forward Euler method. Then we can start the fixed point iterations in order to get a *correction* of the estimate. The iterations run as long as the difference between two successive approximations is above some predetermined treshold ε. Practically, just a few iterations suffice.

Let us examine what the algorithm does for a linear system. The sequence of matrices F in the discrete-time system $x(k+1) = Ax(k)$ is

$$\begin{aligned}
F^P &= I + Ah \\
F^{C_1} &= I + Ah + (Ah)^2 \\
F^{C_2} &= I + Ah + (Ah)^2 + (Ah)^3 \\
F^{C_3} &= I + Ah + (Ah)^2 + (Ah)^3 + (Ah)^4
\end{aligned} \tag{20}$$

For an infinite number of iterations this yields

$$F = I + Ah + (Ah)^2 + (Ah)^3 + (Ah)^4 + \ldots, \tag{21}$$

which can be shown to be equal to

$$F = (I - Ah)^{-1}. \tag{22}$$

Just multiply (21) from the left by Ah and subtract it from (21). Identity results.

Before we analyze the performance of this and the previous algorithm, let us consider one more modification.

Algorithm 3 Predictor-corrector modification of Backward Euler integration

```
**Require:** $f(x(t)),\quad t_0,\quad t_f,\quad x_0,\quad h,\quad \varepsilon$
**Ensure:** $\dot{x}(t) = f(x(t)),\quad x(t_0) = x_0$
  $t_k \leftarrow t_0$
  $x_k \leftarrow x(t_0)$
  **while** $t < t_f$ **do**
    $\dot{x}_k \leftarrow f(x_k, t_k)$
    $x^P_{k+1} \leftarrow x_k + h\dot{x}_k$ {Predictor}
    $\dot{x}^P_{k+1} \leftarrow f(x^P_{k+1}, t_k + h)$
    $x^{C_1}_{k+1} \leftarrow x_k + h\dot{x}^P_{k+1}$ {1st corrector}
    $i \leftarrow 1$
    **while** $e > \varepsilon$ **do**
      $\dot{x}^{C_i}_{k+1} \leftarrow f(x^{C_i}_{k+1}, t_k + h)$
      $x^{C_i}_{k+1} \leftarrow x_k + h\dot{x}^{C_i}_{k+1}$ {$i^{\text{th}}$ corrector}
      $e \leftarrow x^{C_i} - x^{C_{i-1}}$
      $i \leftarrow i + 1$
    **end while**
    $x_k \leftarrow x^{C_i}_k$
    $t_k \leftarrow t_k + h$
  **end while**
```

3.4 Newton iterations in BE method

The predictor-corrector type of methods rely on *fixed point iterations* algorithm for solving the nonlinear algebraic equation. But other numerical techniques can be used. One of the most popular (because of its conceptual simplicity and fast convergence rate) is the Newton method. Let us recall that the method aims at solving

$$g(x) = 0, \tag{23}$$

that is, to find the root(s) of the function $g(x)$. The method is iterative as well and its i-th iteration is described by

$$x^{i+1} = x^i - \frac{g(x^i)}{\dot{g}(x^i)}. \tag{24}$$

The nonlinear algebraic equation that needs to be solved in every iteration of BE algorithm is

$$x_{k+1} = x_k + hf(x_{k+1}, t_{k+1}) \tag{25}$$

or, in the format ready for application of Newton method

$$x_k + hf(x_{k+1}, t_{k+1}) - x_{k+1} = 0. \tag{26}$$

The unknown variable is x_{k+1} and the Newton iteration is

$$x_{k+1}^{i+1} = x_{k+1}^{i} - \frac{x_k + hf(x_{k+1}^i, t_{k+1}) - x_{k+1}^i}{h \left.\frac{\partial f(x,t)}{\partial x}\right|_{x_{k+1}^i, t_{k+1}} - 1.0}. \tag{27}$$

This expression is easily reformulated for vector variables x. The only complication is that instead of a scalar derivative $\frac{\mathrm{d}f(x)}{\mathrm{d}x}$ we have to consider its matrix counterpart, so-called Jacobian matrix

$$J = \frac{\mathrm{d}f(x)}{\mathrm{d}x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \dots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \dots & \frac{\partial f_2}{\partial x_n} \\ \vdots & & & \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \dots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}. \tag{28}$$

$$x_{k+1}^{i+1} = x_{k+1}^{i} - \left[hJ_{k+1}^i - I\right]^{-1} \cdot \left[x_k + hf(x_{k+1}^i) - x_{k+1}^i\right] \tag{29}$$

The Jacobian matrix for a linear state-space model is just the matrix A; the vector version of the algorithms is then

$$\begin{aligned} x_{k+1}^{i+1} &= x_{k+1}^{i} - [Ah - I]^{-1} \cdot [x_k + (hA - I)x_{k+1}^i] \\ &= [I - Ah]^{-1} x_k \end{aligned} \tag{30}$$

And we see that in the linear case the behavior of BE is revoked.

4 Numerical stability

Numerical stability, loosely speaking, is a property of an algorithm, which guarantees that for slightly perturbed input data the computed solution will only by slighty inaccurate. One of the ways to approach numerical stability of algorithms for solving initial value problems is quite familiar to us, control engineers. The solution algorithm is viewed as a discrete-time approximation of the original continuos-time system. You may already know from an introductory course on automatic control that unstable discrete-time model can be obtained for a stable continuous-time model.

This direction will be explored a bit in this section. The fact is, that we only have analytical tests for stability of linear systems. Nonetheless the analysis can bring some insight. We will start with the Forward Euler method.

4.1 Domain of numerical stability of Forward Euler method

As we have already learnt, the Forward Euler integration process turns the original continuous-time system $\dot{x}(t) = Ax(t)$ into the approximate discrete-time model $x(k+1) = Fx(k)$, where $F = I + Ah$. Obviously the discretization

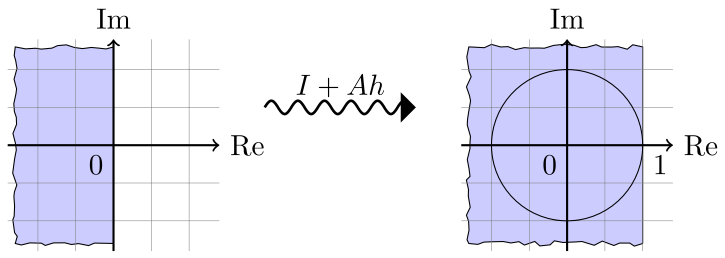

Figure 5: Mapping the left-half plane by the Forward Euler integration method.

maps the region of stability of the continous-time model (which is the open left half-plane) onto a shifted half-plane as in Fig.5.

Obviously some continous-time systems are unstable after discretization. More insight can be obtained if we scrutinize the subset of the complex plane that is mapped into the unit circle as is in Fig.6.

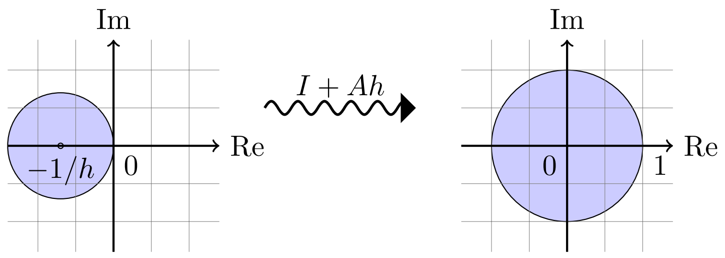

Figure 6: Preimage of the unit circle for Forward Euler integration method.

Note that it is more common to see the visualization of the domain of stability not in the λ-plane but in the "normalized" $h\lambda$-plane. But the message is the same. Whichever way, although derivation of the stability domain is straightforward in this case, in the more complicated cases to come it will not be easy, therefore numerical tools may come in handy. We can plot the *contours* of *spectral radius* $\rho(F)$, that is, the maximum (in absolute value) eigenvalue and the boundary of the stability domain is given by the contour for $\rho(F) = 1$. Matlab code is below

```
h = 1;  I = eye(2,2);

a = linspace(-4,2,100); % range of real values
b = linspace(0,5,100);     % range of imaginary values

rmax = zeros(length(a),length(b));

for ia = 1:length(a)
    for ib = 1:length(b)
        A = [a(ia) b(ib); -b(ib) a(ia)];
```

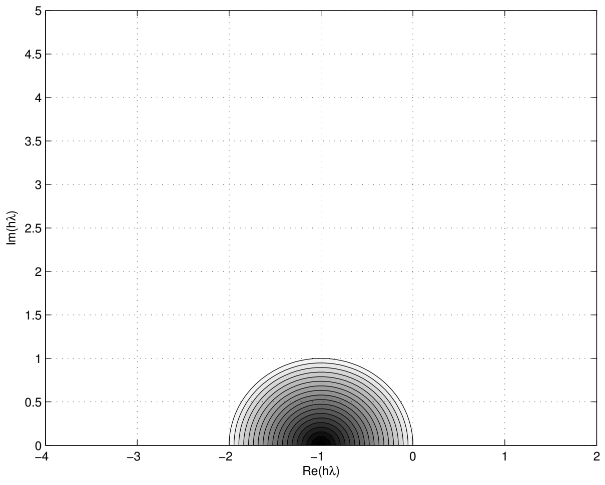

Figure 7: Stability domain once again. This time produced using contour plot for the spectral radius of F (for normalized $h = 1$, equivalently, the domain is plotted in $h\lambda$-plane.

```
          F = I + A*h;
          rmax(ia,ib) = max(max(abs(eig(F))));
     end
end

[A,B] = meshgrid(a,b);

colormap(gray)
contourf(A,B,rmax',linspace(0,1,20))
xlabel('Re(h\lambda)'), ylabel('Im(h\lambda)')
axis([a(1) a(end) b(1) b(end)]), grid on
```

which produces Fig. 7 equivalent to the analytically obtained 6. We will reuse this code later to plot stability domains for other methods. The only thing that needs to be replaced is the definition of the F matrix.

Besides the boundary of the stability domain, these graphs also show the shape of the spectral radius inside the stability domain. The darkest spots represent the minimum. We will find a use for this information later when discussing simulations of stiff systems.

Example 4.1. *Consider a scalar LTI system*

$$\dot{x}(t) = ax(t), \quad x(0) = 1, \tag{31}$$

and pick the values of a as -0.1, -1.0, -2.0, -3.0. The response to initial values is plotted in Fig.8. using the ***impulse()*** *command in Matlab, which computes the "accurate" solution via the exponential. On top of the analytical solutions are visualized the numerical solutions using FE and* $h = 1$.

Apparently, for $a = -0.1$ *the numerical solution agrees fairly well with the analytical solution. For a bit faster system* ($a = -1$)*, the response computed*

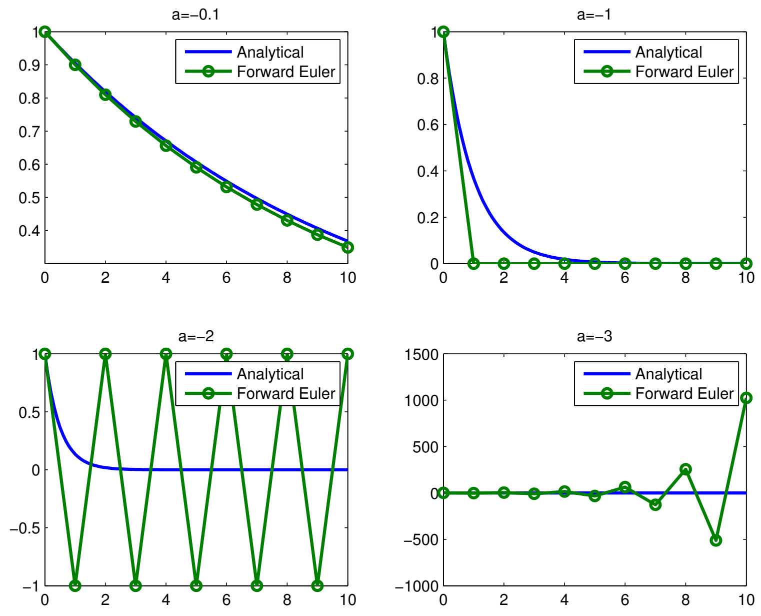

Figure 8: Numerical experiment with Forward Euler approximation for $\dot{x}(t) = ax(t)$.

numerically using FE resembles the accurate solution only faintly. Moving the pole of the original continuous-time system further to the left, the FE solution becomes oscilate even though the original system is nicely stable. Making the system even faster with $a = -3$ (still stable, in fact, the most stable of all the systems considered), the FE computed solution is unstable.

Demonstrated by the numerical example, we are now aware of poor performance of FE method for stable but fast systems. But it should not go unnoticed that the method is not able to simulate lightly damped systems (poles near the imaginary axis), no matter how fast sampled.

4.2 Domain of numerical stability of Backward Euler method

Determining the domain of numerical stability for BE method is a little bit more tedious. In fact, it is one of a few instance where the analytical determination is feasible, for most algorithms we will use some computational algorithm based on gridding. But let us go for the job of determining the relationship between the two complex planes. The mapping considered here is $(I - Ah)^{-1}$. I can be viewed as a composition of three mappings: first the complex number is multiplied by a real negative scalar $-h$. Then it is shifted by 1. Finally, it is inverted. Examining the border of continous-time stability, the first two steps map it to a vertical line crossing the real axis at 1. What remains to interpret geometrically is the inversion of this line in the complex plane. As Fig.9 reveals, the line is mapped onto a circle centered at 1/2 and with a radius of 1/2. And the whole left half plane is mapped onto this closed disc.

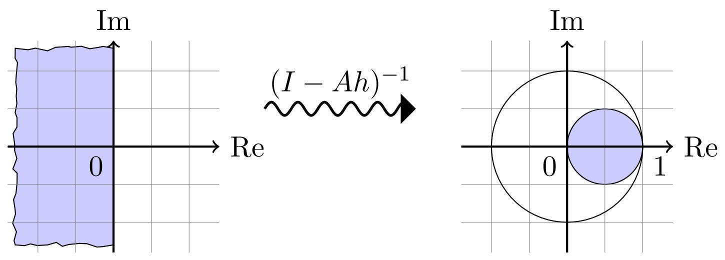

Figure 9: Mapping the left-half plane by the Backward Euler integration method.

A sketch of the proof serves as a nice illustration of how convenient complex numbers are when it comes to planar geometry. Keep considering the scalar (first-order) system $\dot{x}(t) = ax(t)$ for notational simplicity (in the higher order case the scalar a turns into a matrix A which necessitates considering eigenvalues). The vertical line shifted to 1 in the complex plane can be described as a complex number

$$1 + ja\omega, \quad \forall \omega \in \mathbb{R}. \tag{32}$$

We need to compute its inverse and for that purpose it is more convenient to have the complex number in the other format

$$1 + ja\omega = \sqrt{1 + a^2\omega^2} e^{-j\underbrace{\arctan a\omega}_{\alpha}}. \tag{33}$$

Its inverse can be easily found to be

$$\frac{1}{1 + ja\omega} = \frac{1}{\sqrt{1 + a^2\omega^2}} e^{j\arctan a\omega}. \tag{34}$$

With the hindsight of Fig.9, shift this set by 1/2 to the left. A centered circle with a diameter 1/2 should result. In the viewpoint of complex numbers, the resulting set shall be characterized by the constraint $|z|/ \leq 1/2$. Let us verify this. For that purpose, we need to bring the expression (34) into the Im-Re format, in which easy subtraction of 1/2 can be carried out.

$$\frac{1}{\sqrt{1 + a^2\omega^2}} e^{j\arctan a\omega} = \frac{1}{\sqrt{1 + a^2\omega^2}} \cos\alpha + j\frac{1}{\sqrt{1 + a^2\omega^2}} \sin\alpha - 1/2 + 1/2. \tag{35}$$

Realizing that

$$\cos\alpha = \frac{1}{\sqrt{1+a^2\omega^2}}, \qquad \sin\alpha = \frac{a\omega}{\sqrt{1+a^2\omega^2}} \tag{36}$$

makes the rest quite straighforward. Just check that the absolute value of the first two components in the complex number (34) is after substiting the previous expressions equal identically to 1/2. Hence, the full set is a circle wirth a radius 1/2 and centered at 1/2.

Fig.9 reveals that BE method always maps stable continuous-time systems into stable discrete-time systems, hence the resulting simulation algorithms are stable. So far so good. But to gain more insight, let us see what is actually mapped to the unit circle. Fig.10 gives an answer. Proving the figure is an easy task for you now but you can also use the provided Matlab code to obtain the stability domain numerically. Do it.

A striking conclusion is that even some unstable continous-time systems can appear as stable when simulated using BE algorithm. No matter how short the discretization period is. This is a general feature of implicit methods and one should be aware of it. The fact is that in the past not much attention was paid to unstable systems in numerical mathematics, and yet in engineering we need to simulate these too.

On the other hand, the method handles stable and fast systems much better than FE method. These properties are illustrated by the numerical example.

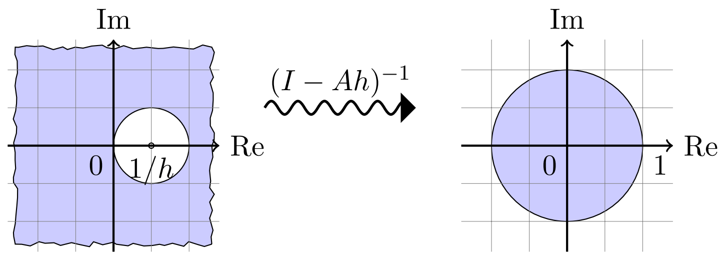

Figure 10: Preimage of the unit circle for Backward Euler integration method.

Example 4.2. *Consider again the system $\dot{x}(t) = ax(t)$, $x(0) = 1$. Use the fixed discretization period $h = 1$ and the finite simulation time $t_f = 10\,s$. Perform BE simulation for a few as.*

The results are in Fig.11.

4.3 Domain of numerical stability of Backward Euler method with predictor-corrector

The Backward Euler method with predictor-corrector iterations seems to be equivalent to the pure BE method if the number of iterations is infinite. However, there is a pitfall hidden here. We subtracted two series, that is, 21 and

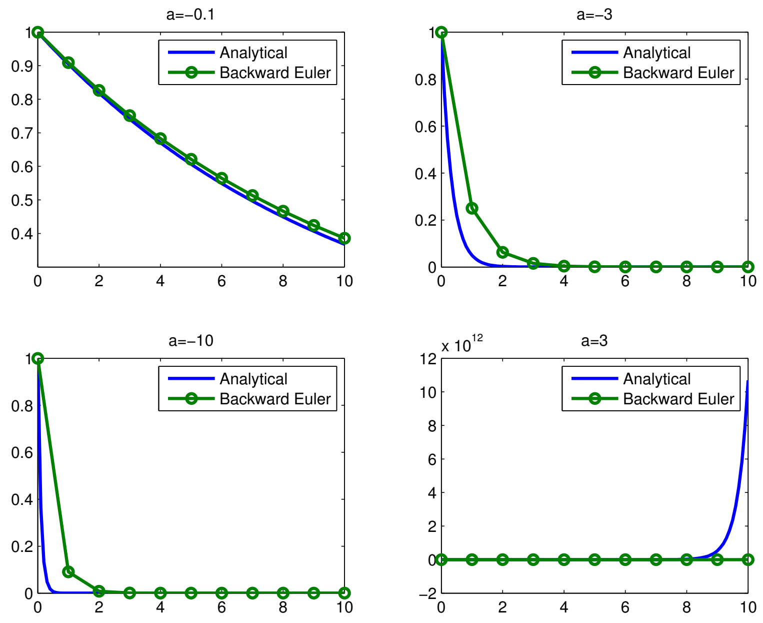

Figure 11: Numerical experiment with Backward Euler approximation for $\dot{x}(t) = ax(t)$.

scaled 21. In order for this operation to be valid, we must consider its region of convergence, that is, all the eigenvalues of Ah must be inside a unit circle. Intersection with the numerical stability domain of BE algorithm yields the half moon domain as in Fig.12. Clearly, the method works well for a very constrained set of systems.

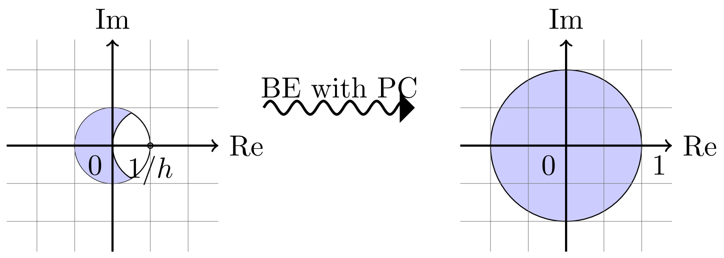

Figure 12: Preimage of the unit circle for Backward Euler integration with predictor-corrector iterations.

4.4 Domain of numerical stability of Backward Euler method with Newton iterations

Having already analyzed the behavior of BE method with Newton iterations for a linear system in (30), we can immeadiately conclude that Newton iterations do not alter the domain of stability of BE method.

5 Single-step methods

5.1 Heun's method

Recall the predictor-corrector scheme introduced previously

$$\begin{aligned} \dot{x}_k &= f(x_k, t_k) \\ x^P_{k+1} &= x_k + h\dot{x}_k \\ \dot{x}^P_{k+1} &= f(x^P_{k+1}, t_k + h) \\ x^C_{k+1} &= x_k + h\dot{x}^P_{k+1}. \end{aligned} \tag{37}$$

Substituting all the terms into just one we obtain

$$x_{k+1} = x_k + hf(x_k + hf_k, t_k + h). \tag{38}$$

Now expand $f()$ in Taylor series and truncate it after the linear term

$$f(x_k + hf_k, t_k + h) \approx f(x_k, t_k) + \left.\frac{\partial f(x,t)}{\partial x}\right|_{x_k,t_k} hf_k + \left.\frac{\partial f(x,t)}{\partial t}\right|_{x_k,t_k} h. \tag{39}$$

Plugging this into the (38) we obtain

$$x_{k+1} = x_k + hf_k + h^2 \underbrace{\left(\left.\frac{\partial f(x,t)}{\partial x}\right|_{x_k,t_k} f_k + \left.\frac{\partial f(x,t)}{\partial t}\right|_{x_k,t_k} \right)}_{\dot{f}(x_k,t_k)}. \tag{40}$$

Now compare this with Taylor expansion of $x(t_k + h)$ truncated after the quadratic term

$$x(t_k + h) \approx x(t_k) + hf(x_k, t_k) + \frac{1}{2}h^2 \dot{f}(x(t_k), t_k). \tag{41}$$

Clearly these two resemble each other very well, up to the factor 1/2 with the quadratic term. Hence, the PC based algorithm can be modified so that it gives an answer that agrees with the first three terms of Taylor expansion of the true solution.

$$\begin{aligned} x_{k+1}^{PC} &= x_k + hf_k + h^2\dot{f}(x_k,t_k) \\ x_{k+1} &= \frac{1}{2}\left(x_{k+1}^{PC} + x_{k+1}^{FE}\right). \end{aligned} \tag{42}$$

In other words

$$\begin{aligned} \dot{x}_k &= f(x_k,t_k) \\ x_{k+1}^{P} &= x_k + h\dot{x}_k \\ \dot{x}_{k+1}^{P} &= f(x_{k+1}^{P},t_{k+1}) \\ x_{k+1}^{C} &= x_k + \frac{1}{2}h\left(\dot{x}_{k+1}^{P} + \dot{x}_k\right). \end{aligned} \tag{43}$$

Derivation of this algorithm is a nice demonstration of how one large family of methods is derived: you come up with some algorithmic scheme, for instance predictor-corrector but with a limited number of corrector stages, which certainly introduces some error, and you then try to modify the result so that it agrees with the first $n+1$ terms of the Taylor expansion. In the case of Heun's method $n = 2$ and the local error is of order 3.

5.2 Runge-Kutta methods

Heun's method presented in the previous section can be generalized and the generalization leads to a class of methods denoted as Runge-Kutta methods (named after two mathematicians: Runge and Kutta).

$$\begin{aligned} \dot{x}_k &= f(x_k,t_k) \\ x_{k+1}^{P} &= x_k + h\beta_{11}\dot{x}_k \\ \dot{x}_{k+1}^{P} &= f(x_{k+1}^{P},t_k+\alpha h) \\ x_{k+1}^{C} &= x_k + h\left(\beta_{22}\dot{x}_{k+1}^{P} + \beta_{21}\dot{x}_k\right). \end{aligned} \tag{44}$$

Plugging the equations into each other and expanding into Taylor series yields

$$x_{k+1}^{C} = x_k + h(\beta_{21}+\beta_{22})f_k + \frac{h^2}{2}\left(2\beta_{11}\beta_{22}\left.\frac{\partial f(x,t)}{\partial x}\right|_{x_t,t_k} f_k + 2\alpha_1\beta_{22}\left.\frac{\partial f(x,t)}{\partial t}\right|_{x_k,t_k}\right). \tag{45}$$

Comparing with the Taylor expansion for $x(k+1)$ we learn that the following three equations need to be satisfied

$$\begin{aligned}\beta_{21} + \beta_{22} &= 1 \\ 2\alpha_1\beta_{22} &= 1 \\ 2\beta_{11}\beta_{22} &= 1\end{aligned} \tag{46}$$

This set of equations parameterizes the family of methods. Heun's method appears to be just one particular instance

$$\alpha_1 = 1, \quad \beta_{11} = 1, \quad \beta_{21} = 0.5, \quad \beta_{22} = 0.5. \tag{47}$$

It is customary in the ODE literature to use a special format for writing down these coefficients, so-called *Butcher tableau*, which for the Heun's method is of the form

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline & 1/2 & 1/2 \end{array}$$

The first row describes evaluation of the function at time t_k. The second row corresponds to predictor and the bottom row corresponds to corrector.

Different values give a different algorithm. Another popular algorithm is *explicit midpoint rule* with Butcher tableau

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1/2 & 1/2 & 0 \\ \hline & 0 & 1 \end{array}$$

The corresponding equations implementing this scheme are

$$\begin{aligned}\dot{x}_k &= f(x_k, t_k) \\ x^P_{k+\frac{1}{2}} &= x_k + \frac{h}{2}\dot{x}_k \\ \dot{x}^P_{k+\frac{1}{2}} &= f(x^P_{k+\frac{1}{2}}, t_{k+\frac{1}{2}}) \\ x^C_{k+1} &= x_k + h\dot{x}^P_{k+\frac{1}{2}}\end{aligned}$$

Both algorithms belong to the family of Runge-Kutta of second order. However, the whole idea can be extended to higher orders by including more predictor or corrector stages. Among all possibilities, the fourth-order and four-stage Runge Kutta method is notoriously known and used. Its α and β parameters, written in a matrix-vector way are

$$\alpha = \begin{bmatrix} 1/2 \\ 1/2 \\ 1 \\ 1 \end{bmatrix}, \quad \beta = \begin{bmatrix} 1/2 & 0 & 0 & 0 \\ 0 & 1/2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1/6 & 1/3 & 1/3 & 1/6 \end{bmatrix} \tag{48}$$

Equivalently, the Butcher tableau is

$$
\begin{array}{c|cccc}
0 & 0 & 0 & 0 & 0 \\
1/2 & 1/2 & 0 & 0 & 0 \\
1/2 & 0 & 1/2 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 \\
\hline
 & 1/6 & 1/3 & 1/3 & 1/6
\end{array}
$$

It should be an easy task for you to rewrite it into equations:

$$
\begin{aligned}
\dot{x}_k &= f(x_k, t_k) \\
x^{P_1} &= x_k + \frac{h}{2}\dot{x}_k \\
\dot{x}^{P_1} &= f(x^{P_1}, t_{k+\frac{1}{2}}) \\
x^{P_2} &= x_k + \frac{h}{2}\dot{x}^{P_1} \\
\dot{x}^{P_2} &= f(x^{P_2}, t_{k+\frac{1}{2}}) \\
x^{P_3} &= x_k + h\dot{x}^{P_2} \\
\dot{x}^{P_3} &= f(x^{P_3}, t_{k+1}) \\
x_{k+1} &= x_k + \frac{h}{6}\left(\dot{x}_k + 2\dot{x}^{P_1} + 2\dot{x}^{P_2} + \dot{x}^{P_3}\right).
\end{aligned}
\tag{49}
$$

One point is particularly worth noting: the function $f()$ is evaluated a couple of times inside the integration step. It is not automatically guaranteed that the times of function evaluations are nondecreasing or even strictly growing. In the above method the times are certainly not strictly increasing. The function is evaluated first at t_k, then twice at $t_k + 1/2h$ and finally at $t_k + h$.

5.3 Stability domains of RK algorithms

Here we should explore how the stability domain looks like. Especially in comparison with the simple algorithms like FE or BE. Let's start with Heun's method. We have seen that its F matrix is

$$
F = I + Ah + A^2h^2. \tag{50}
$$

Its stability domain is visualized in Fig.13.

Obviously, the stability domain is extended in comparisson to FE method, "prolonged" vertically. This is certainly plausible because a large portion of the left-half plane is mapped onto the unit circle. In the figure we not only visualize the boundary of the stability domain, but by plotting contours for a few values, we have also the information about the value of the underlying function inside the stability domain. This will become useful soon.

Finding the F matrix is somewhat tedious but straightforward. Just substitute the equations in (49) into each other and you obtain

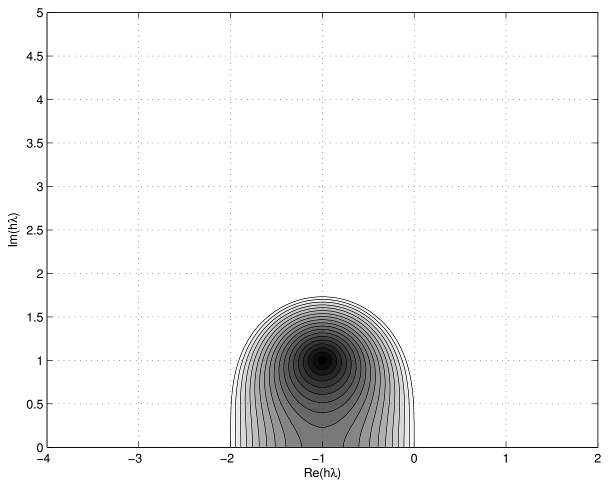

Figure 13: Stability domain for Heun's method (just the upper half-plane).

$$F = I + Ah + \frac{1}{2}A^2h^2 + \frac{1}{6}A^3h^3 + \frac{1}{24}A^4h^4. \tag{51}$$

The stability domain is at Fig. 14.

5.4 Stiff systems, A-stability, L-stability

Here we will examine if a nice stability domain is all we need to guarantee faithful solution of stiff systems. From the discussion of A-stability above, we can develop an impression that in a situation when the stability domain includes the whole left half-plane, the problems with stability are gone. Bad luck. Things are not that easy for a particular type of systems called *stiff systems*. Loosely speaking, stiff systems exhibit both slow and fast dynamics. The fast yet stable dynamics is what causes a problem. One can view the leftmost poles of the system as being the most stable poles because they are at greatest distance from the imaginary axis, right? Nope, recall the fundamental concept of Riemann sphere in complex analysis. According to it, there is just one infinity, whichever directions in the complex plane you are approaching it. That is, if the first order system $\dot{x}(t) = -\lambda x(t),\ \lambda > 0$ is gettig faster and faster, more and more stable, ultimately as $\lambda \to \infty$ it becomes "unstable".

The methods that can cope with this not only have their stability domain large enough (ideally covering the whole left half-plane) but also have to satisfy an additional restriction

$$\lim_{\lambda h \to \infty} \rho(F(\lambda h)) = 0. \tag{52}$$

A-stable method that satisfies the above restriction is called $L - stable$. Apparently, the BE method satisfies this as seen in Fig. 15. Recall that the

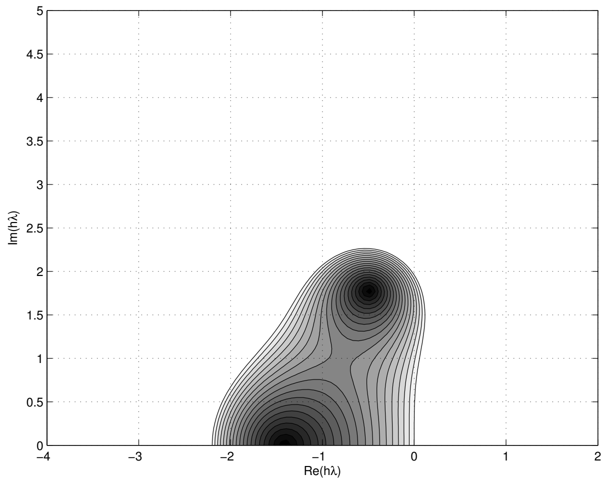

Figure 14: Stability domain for RK4 algorithm.

stability domain is obtained as $\rho(F(\lambda h)) \leq 1$.

For the second time in this lecture this confirms the fact that implicit methods do a good job when simulating stiff systems. In fact, it is only the implicit methods that are usable for stiff systems.

5.5 Stepsize and order control

We have learnt that the step size is what matters from the viewpoint of stability and accuracy. The selection of the integration step h now seems crucial. But why stick to a fixed value of h? Actually it seems a good idea to adjust the integration step online, based on the error. Is the error larger than what could be accepted? Then decrease the integration step. Nice application of basic feedback control principles. However, it is not clear where to get an error. After all, we do not know the true value. The solution is suprisingly simple: integrate over one integration step with two different algorithms and take the difference of their outcomes as the estimate of the error.

True, theoretically it can happen that both algorithms undershoot, in which case their difference will give a false indicator of a small error. But this rarely happens! This principle inded works fairly well.

On the other hand, the computational load doubles, which is a pitty. To releave this extra computational load, the two algorithms are not completely different, in fact, one can algorithm can be obtained just by adding one more stage to the RK algorithm. If the coefficients of the algorithm are chosen well, the new (more accurate) algorithm reuses much of the function evaluations done by the lower order algotithm. The default algorithm in Matlab—ode45—is doing exactly that, implementing RK4 and RK5.

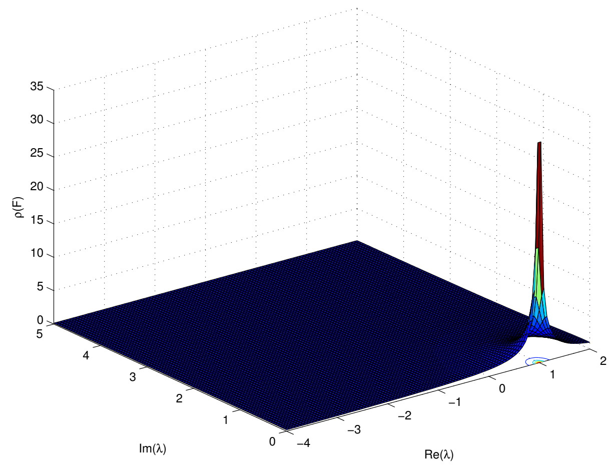

Figure 15: Spectral radius for matrix F in $x(k+1) = x(k)$ for BE method.

6 Multiple step methods

Unlike the single-step methods, which need to evaluate the function $f()$ several times during the step (in so-called microsteps), in a class of methods denoted as multistep, each step evaluates $f()$ just once. The trick is that unlike in single step methods, the information from the previous steps is not completely discarded. Quite the oposite, it is well used in the upcoming steps.

[...]

7 Literature

The topic of numerical solution of ordinary differential equation has been extensively described in the literature. Among the huge number of textbooks we have chosen [2] as the key text for preparation of this lecture and the next. The reason is that this monograph seems to be best tailored not only to the engieering language but also to the engineering needs. For instance, the interpretation of discretization in time domain as finding a discrete-time approximation to the original continuos-time system is emphasized. This then suggests using system-theoretic tools for investigation of stability of these methods (systems). Some advanced topics are also included in the book, which do not appear elsewhere, such as real-time simulation. Hardly useful in off-line mathematics but of use in some engineering applications where the simulation outcomes are in real-time compared to the true system outputs for the purpose of fault diagnosis. Well, yet another reason for preference of this book is the fact that it is avaible to students of Czech Technical University in Prague through the institutional subscription, see the web for more information.

Nonetheless, it appears that any serious study benefits from obtaining an alternative (or perhaps more classical) viewpoint. Among the classics, [1] is also available electronically for CTU students but it may be perhaps to terse. Other often cited introductory books are [6] and [3]. The [4] seems also suitable for undergraduate students, although I am not acquainted with it.

Some practical aspects are nicely exposed in a book written by one of the founders of The Mathworks company (the producer of Matlab) [5]. The book is also freely available at `http://www.mathworks.com/moler/chapters.html`. The chapter 7 deals with numerical solutions of ODEs. It is just 53 pages. Go ahead and read it.

One particular topic, namely, the topic of stiff ordinary differential equations is also analyzed nicely in a blog by Cleve Moler at `http://www.mathworks.com/company/newsletters/news_notes/clevescorner/may03_cleve.html`.

Another interesting blog is run by Guy Rouleau and Seth Popinchalk, application engineers at The Mathworks. Their article “THE Most Useful Command for Debugging Variable Step Solver Performance” is worth reading. You can access it at `http://blogs.mathworks.com/seth/2012/06/04/the-most-useful-command-for-debugging-variable-step-solver-performance/`.

Needless to say, dozens of nice and freely downloadable lecture notes are around on the internet, just search for the keywords ”numerical solution of ordinary differential equation” in case you still had not enough:-)

References

[1] John C. Butcher. *Numerical Methods for Ordinary Differential Equations.* John Wiley & Sons, Ltd, Chichester, UK, March 2008.

[2] François E. Cellier and Ernesto Kofman. *Continuous System Simulation.* Springer, softcover reprint of hardcover 1st ed. 2006 edition, October 2010.

[3] David F. Griffiths and Desmond J. Higham. *Numerical Methods for Ordinary Differential Equations: Initial Value Problems.* Springer, 1st edition, December 2010.

[4] Arieh Iserles. *A First Course in the Numerical Analysis of Differential Equations.* Cambridge University Press, 2nd edition, December 2008.

[5] Cleve B. Moler. *Numerical Computing with MATLAB, Revised Reprint.* Society for Industrial and Applied Mathematics, rev rep edition, July 2008.

[6] Lawrence F. Shampine. *Numerical solution of ordinary differential equations.* Springer, 1st edition, August 1994.