Initial value problem using vector notation:

$$y'(x) = f(x, y) \quad y(x_0) = a; \quad x \geq x_0$$

where $x \in \mathbb{R}$ and $y(x)$ is an N-dimensional function. The N-dimensional vector a is called the *Initial Condition*.

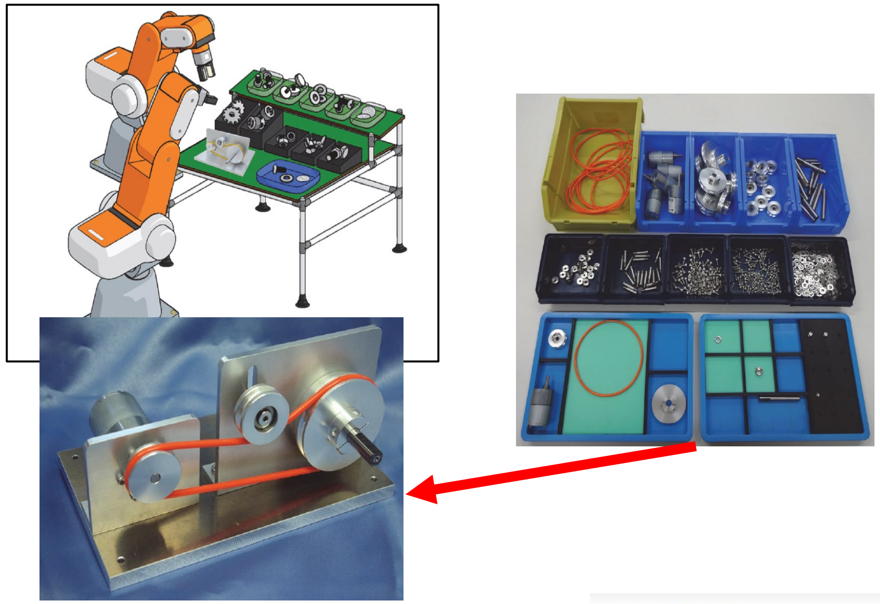
Numerical solution:

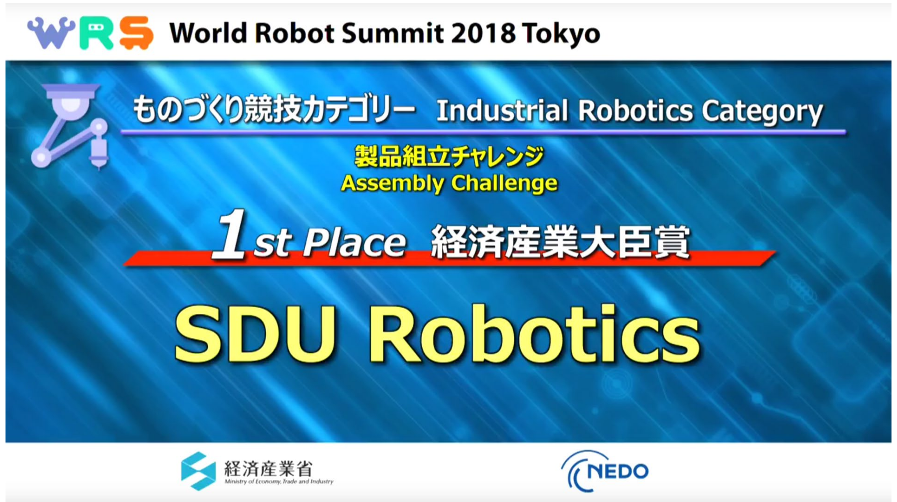
In numerical solutions to ordinary differential equations, we first define a stepsize h and

$$\begin{aligned} x_n &= x_0 + nh \quad & n = 0, 1, 2, \ldots \\ y_n &\simeq y(x_n) \quad & n = 0, 1, 2, \ldots \end{aligned}$$

where y_n is the numerical approximation to the true value $y(x_n)$.

Assembly challenge at the World Robot Summit in Tokyo, 2018
250 applicants from Europe, North America and Asia. Of these 16 were selected to participate including us.




World Robot Summit 2018 Tokyo

ものづくり競技カテゴリー Industrial Robotics Category

製品組立チャレンジ
Assembly Challenge

1st Place 経済産業大臣賞

SDU Robotics

経済産業省 Ministry of Economy, Trade and Industry

NEDO

One of the crucial technologies in our solution was ”**Programming by Demonstration**”.

Example from robotics:

Dynamic Movement Primitives for learning robot trajectories based on **Programming by Demonstration**.

Dynamic Movement Primitives are mathematically formalized as a *transformation system* (1), which controls the evolution of the trajectory, and a *canonical system* (2), which controls temporal evolution of the system:

Goal

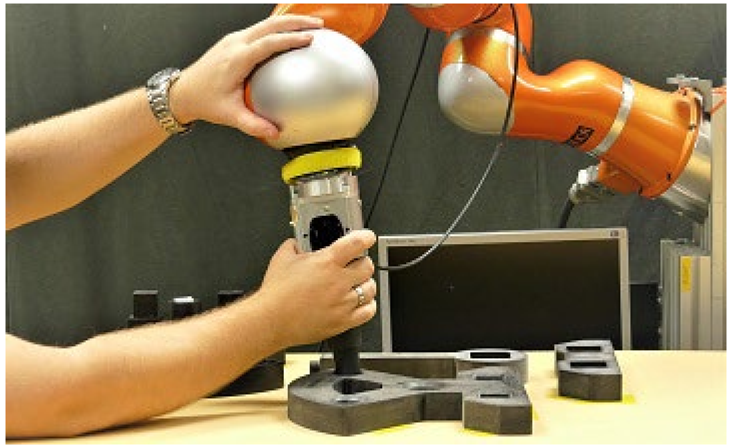
”Force term” defined by demonstrated trajectories

$$\tau^2 \ddot{y} = \alpha_y \left(\beta_y \left(g - y\right) - \tau \dot{y}\right) + f(x), \quad (1)$$

$$\tau \dot{x} = -\alpha_x x, \quad (2)$$

x is controlling the phase (where on the trajectory at time t).

A dot (over y and x) means derivative wrt. time



Programming by Demonstration

1st order Runge-Kutta (Euler):

$$y_{n+1} = y_n + hf(x_n, y_n) + O(h^2)$$

2nd order Runge-Kutta (Midpoint):

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf\left(x_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_1\right) \\ y_{n+1} &= y_n + k_2 + O(h^3) \end{aligned}$$

4th order Runge-Kutta:

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf(x_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_1) \\ k_3 &= hf(x_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_2) \\ k_4 &= hf(x_n + h, y_n + k_3) \\ y_{n+1} &= y_n + \tfrac{1}{6}k_1 + \tfrac{1}{3}k_2 + \tfrac{1}{3}k_3 + \tfrac{1}{6}k_4 + O(h^5) \end{aligned}$$

Trapezoidal method (implicit):

$$y_{n+1} = y_n + \frac{h}{2}\left(f(x_n, y_n) + f(x_{n+1}, y_{n+1})\right) + \mathcal{O}(h^3)$$

Leap-frog:

$$y_{n+1} = y_{n-1} + 2hf(x_n, y_n) + \mathcal{O}(h^3)$$

Example y’(x)=y(x)cos[x+y(x)]; y(0)=1

Midpoint (2nd order Runge-Kutta)
Richardson analysis at x=100

i	A(hi)	A(hi-1)-A(hi)	Rich-alp^k	A(hi)-A	Rich-fejl	Antal f-ber.
1.	0.0093614	*	*	-0.00101948	*	200.
2.	0.0114708	-0.00210941	*	0.00108992	*	600.
3.	0.0105727	0.000898101	-2.34874	0.00019182	0.000299367	1400.
4.	0.0104132	0.00015947	5.63178	0.0000323502	0.0000531567	3000.
5.	0.0103869	0.0000263589	6.04995	5.99125×10^{-6}	8.78631×10^{-6}	6200.
6.	0.0103821	4.73601×10^{-6}	5.56564	1.25524×10^{-6}	1.57867×10^{-6}	12 600.
7.	0.0103812	9.46299×10^{-7}	5.00477	3.08938×10^{-7}	3.15433×10^{-7}	25 400.
8.	0.010381	2.06623×10^{-7}	4.57984	1.02316×10^{-7}	6.88743×10^{-8}	51 000.

Trapezoidal
Richardson analysis at x=100

i	A(hi)	A(hi-1)-A(hi)	Rich-alp^k	A(hi)-A	Rich-fejl	Antal f-ber.
1.	0.0107355	*	*	0.000354652	*	716.
2.	0.0104665	0.000269017	*	0.0000856349	*	2088.
3.	0.0104023	0.0000641927	4.19077	0.0000214421	0.0000213976	4376.
4.	0.0103863	0.0000160509	3.99932	5.39122×10^{-6}	5.3503×10^{-6}	8436.
5.	0.0103823	4.01395×10^{-6}	3.99878	1.37727×10^{-6}	1.33798×10^{-6}	16 454.
6.	0.0103813	1.00358×10^{-6}	3.99964	3.73697×10^{-7}	3.34525×10^{-7}	32 448.
7.	0.010381	2.509×10^{-7}	3.99991	1.22797×10^{-7}	8.36333×10^{-8}	64 386.
8.	0.0103809	6.27253×10^{-8}	3.99998	6.00718×10^{-8}	2.09084×10^{-8}	127 898.

Example $y'(x)=y(x)\cos[x+y(x)]; y(0)=1$

Leapfrog
Richardson analysis at
x=10

i	A(hi)	A(hi-1)-A(hi)	Rich-alp^k	A(hi)-A	Rich-fejl	Antal f-ber.
1.	-5.56034	*	*	-5.62469	*	10.
2.	3.37249	-8.93284	*	3.30814	*	30.
3.	-4.46885	7.84134	-1.1392	-4.5332	2.61378	70.
4.	-0.706711	-3.76214	-2.08428	-0.77106	-1.25405	150.
5.	-0.0822316	-0.62448	6.02444	-0.146581	-0.20816	310.
6.	0.0290177	-0.111249	5.61334	-0.0353313	-0.0370831	630.
7.	0.0554915	-0.0264737	4.20226	-0.00885758	-0.00882457	1270.
8.	0.0621314	-0.00663992	3.98705	-0.00221766	-0.00221331	2550.

Leapfrog
Richardson analysis at
x=100

i	A(hi)	A(hi-1)-A(hi)	Rich-alp^k	A(hi)-A	Rich-fejl	Antal f-ber.
1.	1.63429×10^8	*	*	1.63429×10^8	*	100.
2.	10831.5	1.63419×10^8	*	10831.5	*	300.
3.	-15684.4	26515.9	6163.04	-15684.4	8838.63	700.
4.	25.8618	-15710.3	-1.6878	25.8514	-5236.76	1500.
5.	-27.5197	53.3815	-294.302	-27.5301	17.7938	3100.
6.	-22.1219	-5.39778	-9.88953	-22.1323	-1.79926	6300.
7.	-45.881	23.7591	-0.227188	-45.8914	7.91969	12700.
8.	3.47281	-49.3538	-0.481403	3.46243	-16.4513	25500.

1st order Runge-Kutta (Euler):

$$y_{n+1} = y_n + hf(x_n, y_n) + O(h^2)$$

2nd order Runge-Kutta (Midpoint):

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf\left(x_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_1\right) \\ y_{n+1} &= y_n + k_2 + O(h^3) \end{aligned}$$

4th order Runge-Kutta:

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf(x_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_1) \\ k_3 &= hf(x_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_2) \\ k_4 &= hf(x_n + h, y_n + k_3) \\ y_{n+1} &= y_n + \tfrac{1}{6}k_1 + \tfrac{1}{3}k_2 + \tfrac{1}{3}k_3 + \tfrac{1}{6}k_4 + O(h^5) \end{aligned}$$

Trapezoidal method (implicit):

$$y_{n+1} = y_n + \frac{h}{2}\left(f(x_n, y_n) + f(x_{n+1}, y_{n+1})\right) + \mathcal{O}(h^3)$$

Leap-frog:

$$\xcancel{y_{n+1} = y_{n-1} + 2hf(x_n, y_n) + \mathcal{O}(h^3)}$$

Order of a numerical method (global order):

In applications, we will not be interested in the error after one step, but the error at some fixed x. If we use a a number of subdivisions

$$x_n = x_0 + nh \quad n = 0, \ldots, M$$

so that $x = Mh$, we get a first approximation of the error at x as

$$\|y_M - y(x)\| \simeq M * \mathcal{O}(h^{k+1})$$

Since $M = \frac{x}{h}$, we expect to get

$$\|y_M - y(x)\| \simeq \mathcal{O}(h^k)$$

which is why k (and not $k+1$) is called the order of the method.

The term for one step $\mathcal{O}(h^{k+1})$ is called the "local order" of the method

The Leap-frog examples have shown that the order does not tell everything !!!

Stability of differential equations with respect to perturbations

$$y'(x) = f(y(x)); \quad y(x_0) = a$$

Consider now a small perturbation δ_0 on the initial condition

$$y(x_0) = a + \delta_0$$

The perturbation is e.g. due to finite real number representation (around 10^-17 for double prec.). Can also be initial measurement errors (much larger)

We then get a new solution $y(x) + \delta(x)$ where $\delta(x)$ is the resulting perturbation at x. We will now have a differential equation for $y + \delta$

$$(y'(x) + \delta'(x)) = f(y(x) + \delta(x)) \quad x \geq x_0$$

We may now Taylor expand the right hand side around $y(x)$:

$$\cancel{y'(x)} + \delta'(x) \simeq \cancel{f(y(x))} + J(y(x))\delta(x)$$

where $J(y)$ is the Jacobian to $f(y)$ (remember f contains N functions in the N dimensional vector y).

We then get a differential equation for $\delta(x)$

$$\delta'(x) \simeq J(y(x))\delta(x)$$

$$\delta'(x) \simeq J(y(x))\delta(x)$$

Assume now for analysis that $N = 1$. If now $J(y(x)) = k > 0$ for a while along the trajectory, say between $x = u$ and $x = u + X$, we get the solution

$$\delta(u + X) = e^{kX}\delta(u)$$

If now (more realistic) $J(y(x)) \geq k > 0$ for a while along the trajectory, say between $x = u$ and $x = u + X$, we get

$$|\delta(u + X)| \geq e^{kX}|\delta(u)|$$

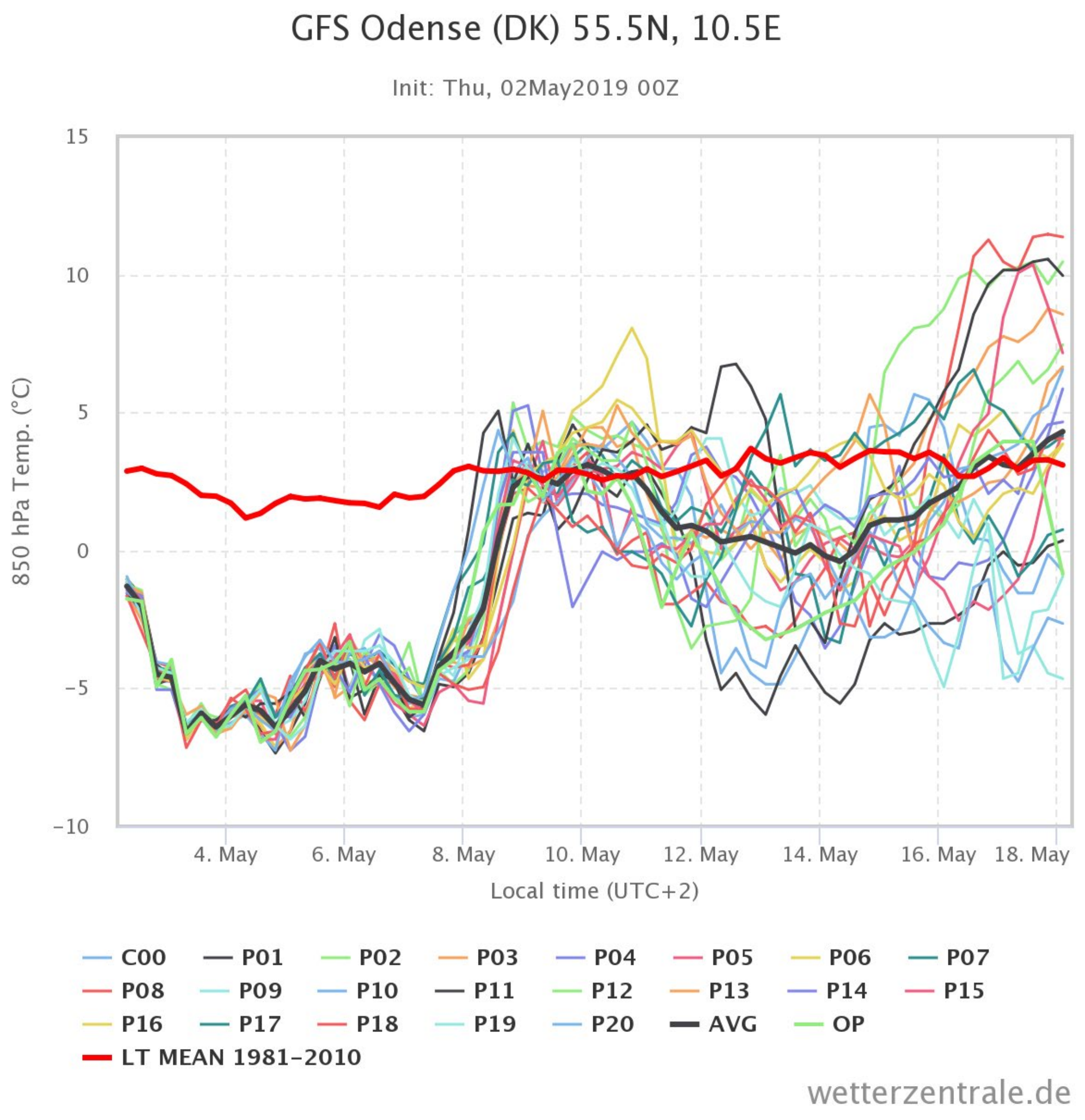
Hence, the perturbation leads to exponential separation. If this occurs for a sufficient big interval size X, it will be difficult or impossible to obtain any global accuracy after this interval.

For $N > 1$, we remember that if for an $N \times N$ matrix A, we have that

$$Az = \lambda z$$

where $\lambda \in \mathbb{C}$, we say that λ is an *Eigenvalue* for A with *corresponding Eigenvector* z.

If we now for simplicity assume that $J(y(x)) \equiv A$ is constant, the same problem of exponential separation will occur if A has an Eigenvalue with positive real part and there is a component from the associated Eigenvector in the perturbation.



The weather is an example where perturbations on initial conditions lead to large separations. Here you see a prediction of the atmospheric temperature at the 850HPa isobar over Odense from May 2nd 2019.

The main run is supplemented by 20 random perturbations on the initial condition.

The purpose of this is to predict the certainty of the temperature forecast (”the exponential factor on the perturbation”).

As you can see the temperature is fairly predictable for around a week (May, 9). After that, the trajectories quickly separates.

Exercise from last week:

$$\begin{aligned} y_1'(x) &= y_1(x)y_2(x) \\ y_2'(x) &= -y_1(x)^2 \end{aligned}$$

$$J(y(x)) = \begin{pmatrix} y_2(x) & y_1(x) \\ -2y_1(x) & 0 \end{pmatrix}$$

We then get Eigenvalues

$$\begin{aligned} \lambda_1 &= \frac{1}{2}\left(y_2(x) + \sqrt{y_2(x)^2 - 8y_1(x)^2}\right) \\ \lambda_2 &= \frac{1}{2}\left(y_2(x) - \sqrt{y_2(x)^2 - 8y_1(x)^2}\right) \end{aligned}$$

In the numerical solution, we quickly get $y_2(x) < 0$, in which case none of the Eigenvalues will have positive real values. The solutions therefore do not separate, and actually converge to the stable point $(y_1, y_2) = (0, -\sqrt{2})$ where the Eigenvalues are $\{0, -\sqrt{2}\}$.

Stability of the numerical integration method

Definition For the testproblem

$$y'(x) = \lambda y(x) \quad \boxed{\mathrm{Re}(\lambda) < 0;} \quad y(0) = 1; \quad \text{Solution: } y(x) = e^{\lambda x}$$

we first notice that $y(x) \to 0$ for $x \to \infty$. A numerical method using a stepsize h is said to be *stable* if and only if $y_n \to 0$ for $n \to \infty$. The condition is purely a property of $z = \lambda h$, and the subset of the complex plane with negative real values, where this property is satisfied is called the *stability region* for the method.

Euler's method with stepsize h yields

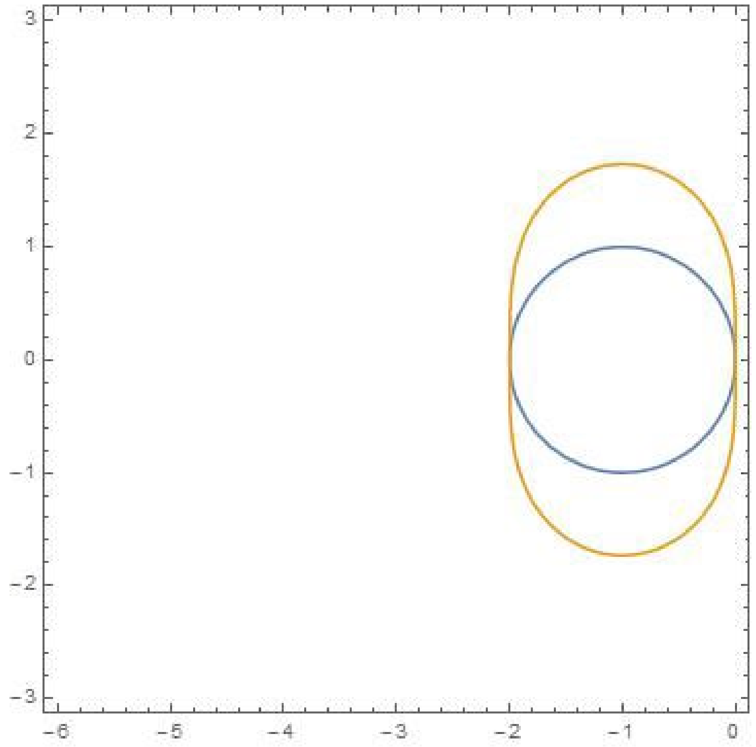
$$y_{n+1} = y_n + hf(x_n, y_n)$$

$$\begin{aligned} y_0 &= 1 \\ y_{n+1} &= y_n + \lambda h y_n = (1 + \lambda h) y_n \end{aligned}$$

Consider the second order Runge-Kutta method (midpoint method):

$$k_1 = hf(x_n, y_n)$$
$$k_2 = hf\left(x_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_1\right)$$
$$y_{n+1} = y_n + k_2 + O(h^3)$$

$$y_{n+1} = y_n + \lambda h y_{n+\frac{1}{2}}$$



Stability region for Euler (blue) and Midpoint (yellow)

Consider now the Trapezoidal method: $y_{n+1} = y_n + \frac{h}{2}(f(x_n, y_n) + f(x_{n+1}, y_{n+1}))$

$$y_{n+1} = y_n + \lambda h \frac{1}{2}(y_{n+1} + y_n)$$

which can be reordered to

$$(1 - \frac{\lambda h}{2})y_{n+1} = (1 + \frac{\lambda h}{2})y_n$$

Hence the stability region is the whole complex half plane with negative real values and hence the Trapezoidal method is numerical stable for all h-values. In particular if we have differential equations, where we know that $\text{Re}(\lambda) << 0$ (stiff differential equations), this method becomes relevant.

Consider now the Leap-frog method:

$$y_{n+1} = 2\lambda h y_n + y_{n-1}$$

An analysis, which is a bit more involved than for the other methods above, yields that the stability region is EMPTY !!! Hence, don't use this method except in special cases where you are not interested in global accuracy, but only in e.g. obtaining statistical behavior over long runs with differential equations where perturbations lead to fast separation. Actually it is used quite frequently for e.g. simulation dynamics of particles or molecules, albeit in a slightly modified form called Verlet integration.

A final remark:

In practise (for $N >> 1$), separation of trajectories and numerical stability cannot be studied via the Jacobian, so all the above is for theoretical study. Use Richardson Extrapolation to obtain global accuracy, and if a convergence cannot be observed, you will know that the global result cannot be trusted.

Stiff ODE's (NR sec. 17.5). Occurs in e.g.

- Chemical reaction kinetics (containing highly unstable components)
- Modeling materials with highly varying stiffness (e.g. composites)
- Control engineering

$u'(x)=998u(x)+1998v(x);\qquad u(0)=1$

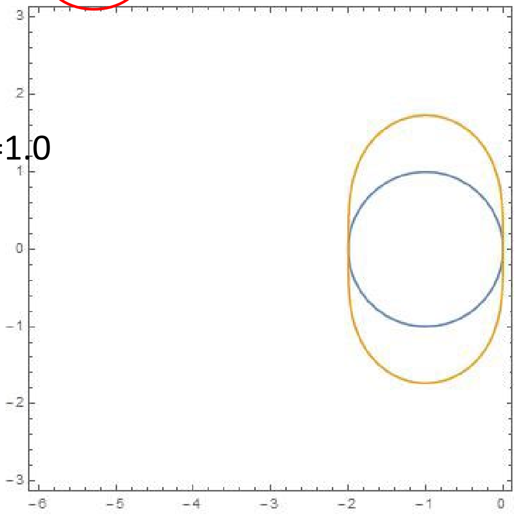
$v'(x)=-999u(x)-1999v(x);\qquad v(0)=0$

Midpoint method estimating A=u(1):

i	A(hi)	A(hi-1)-A(hi)	Rich-alp^k	n	Rich-error	#f comp.
1.	-499 000.	*	*	1.	*	2.
2.	-1.55005×10^{10}	1.55×10^{10}	*	2.	*	6.
3.	-9.2364×10^{17}	9.2364×10^{17}	1.67814×10^{-8}	4.	3.0788×10^{17}	14.
4.	-1.22105×10^{31}	1.22105×10^{31}	7.56432×10^{-14}	8.	4.07016×10^{30}	30.
5.	-2.68758×10^{52}	2.68758×10^{52}	4.54331×10^{-22}	16.	8.95859×10^{51}	62.
6.	-1.40813×10^{85}	1.40813×10^{85}	1.90861×10^{-33}	32.	4.69377×10^{84}	126.
7.	-9.90828×10^{129}	9.90828×10^{129}	1.42117×10^{-45}	64.	3.30276×10^{129}	254.
8.	-9.54329×10^{175}	9.54329×10^{175}	1.03825×10^{-46}	128.	3.1811×10^{175}	510.
9.	-4.01048×10^{172}	-9.53928×10^{175}	-1.00042	256.	-3.17976×10^{175}	1022.
10.	0.735759	-4.01048×10^{172}	2378.59	512.	-1.33683×10^{172}	2046.
11.	0.735759	3.51399×10^{-7}	-1.14129×10^{179}	1024.	1.17133×10^{-7}	4094.
12.	0.735759	8.77843×10^{-8}	4.00299	2048.	2.92614×10^{-8}	8190.
13.	0.735759	2.19367×10^{-8}	4.00171	4096.	7.31223×10^{-9}	16 382.
14.	0.735759	5.483×10^{-9}	4.00085	8192.	1.82767×10^{-9}	32 766.
15.	0.735759	1.37061×10^{-9}	4.00041	16 384.	4.5687×10^{-10}	65 534.
16.	0.735759	3.42623×10^{-10}	4.00035	32 768.	1.14208×10^{-10}	131 070.

Eigenvalues of the Jacobian of the right hand side are $\{-1, -1000\}$

h=1.0



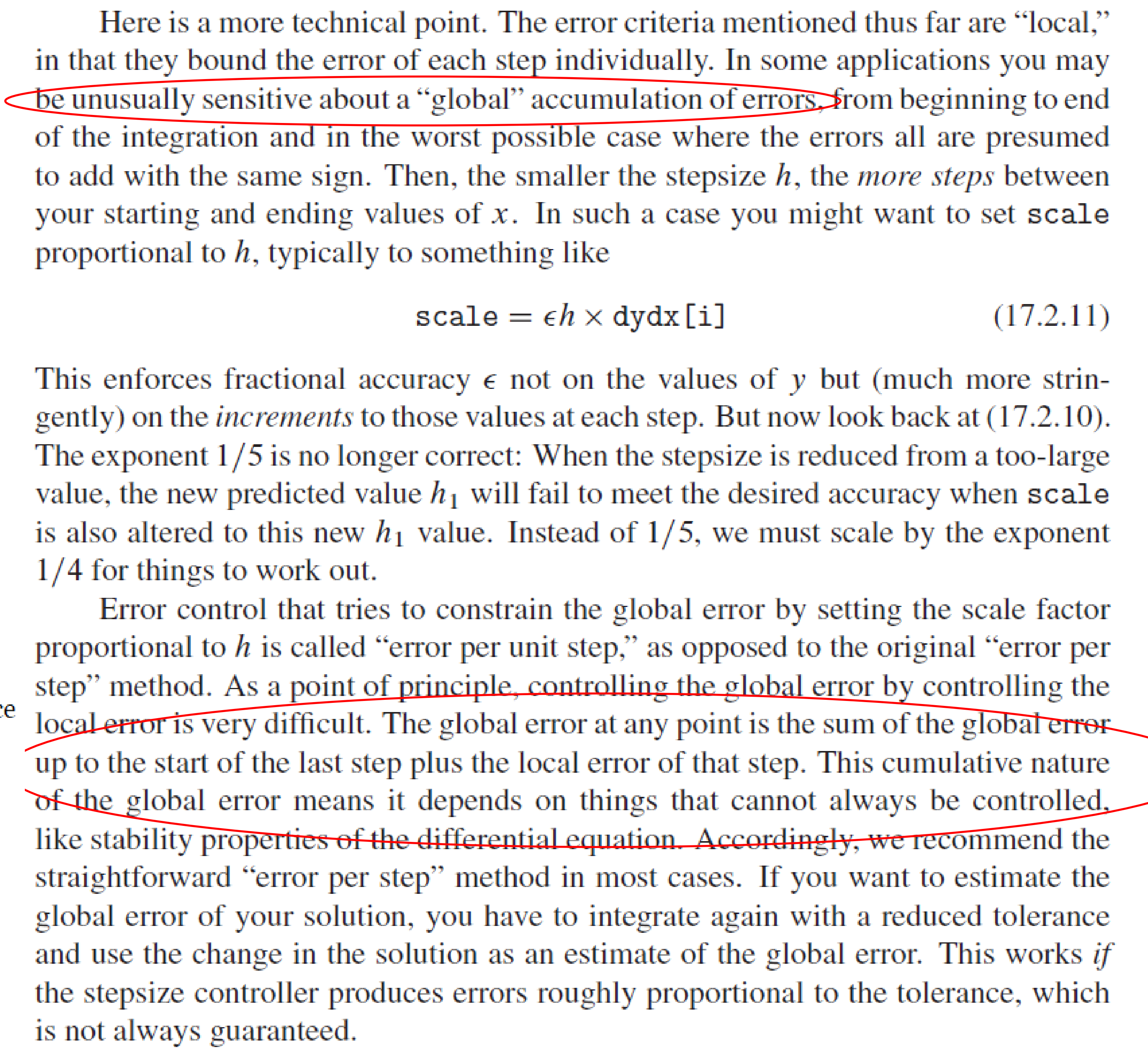
h=1.0/256, hence (-1000)h<-2

h=1.0/512, hence (-1000)h>-2

Meaning inside stab. region

Adaptive Stepsize Methods – a warning

- Numerical Recipes section 17.2
- No chance to derive proven global accuracy
- The text to the right is taken from NR p. 914
- Implicit adaptive stepsize methods (Rosenbrock, NR section 17.5.1) helps with numerical stability, but still no chance to check the error with Richardson.

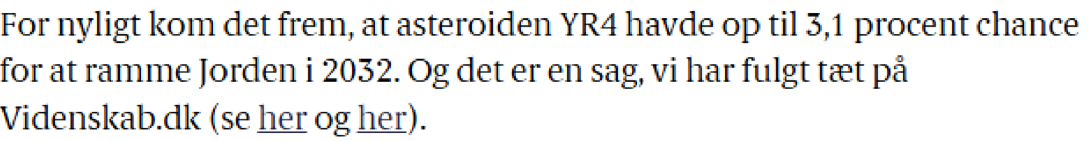


Here is a more technical point. The error criteria mentioned thus far are "local," in that they bound the error of each step individually. In some applications you may be unusually sensitive about a "global" accumulation of errors, from beginning to end of the integration and in the worst possible case where the errors all are presumed to add with the same sign. Then, the smaller the stepsize h, the *more steps* between your starting and ending values of x. In such a case you might want to set `scale` proportional to h, typically to something like

$$\texttt{scale} = \epsilon h \times \texttt{dydx[i]} \qquad (17.2.11)$$

This enforces fractional accuracy ϵ not on the values of y but (much more stringently) on the *increments* to those values at each step. But now look back at (17.2.10). The exponent $1/5$ is no longer correct: When the stepsize is reduced from a too-large value, the new predicted value h_1 will fail to meet the desired accuracy when `scale` is also altered to this new h_1 value. Instead of $1/5$, we must scale by the exponent $1/4$ for things to work out.

Error control that tries to constrain the global error by setting the scale factor proportional to h is called "error per unit step," as opposed to the original "error per step" method. As a point of principle, controlling the global error by controlling the local error is very difficult. The global error at any point is the sum of the global error up to the start of the last step plus the local error of that step. This cumulative nature of the global error means it depends on things that cannot always be controlled, like stability properties of the differential equation. Accordingly, we recommend the straightforward "error per step" method in most cases. If you want to estimate the global error of your solution, you have to integrate again with a reduced tolerance and use the change in the solution as an estimate of the global error. This works *if* the stepsize controller produces errors roughly proportional to the tolerance, which is not always guaranteed.



For nyligt kom det frem, at asteroiden YR4 havde op til 3,1 procent chance for at ramme Jorden i 2032. Og det er en sag, vi har fulgt tæt på Videnskab.dk (se her og her).

Exercise for next week

Consider the set of differential equations

$u'(x)= Cos[-1+x+u(x)+3v(x)]$ $\qquad$ $u(0)=1$

$v'(x)=-u(x)^2+2Sin[v(x)]$ $\qquad$ $v(0)=0$

a) Perform one step with the Midpoint method (2nd order Runge-Kutta) with stepsize h. Clearly illustrate each step.

We denote the solution from a) with u_1,v_1.

b) How is the error $||u(h)-u_1,v(h)-v_1||$ expected to depend on the stepsize h (use O() notation).

c) We now want to estimate u(x) at x=1. Implement the Midpoint method and Richardson extrapolation to achieve a proven accuracy on u(1) of 10^{-6}. Present the output in a table form similar to the ones used in class.

d) Determine analytically the Jacobian used for the Trapezoidal method for the above problem.