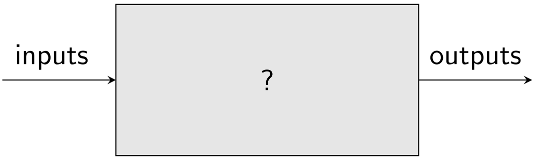
Simulation methods for differential equations

Rien Quirynen

July 28, 2015

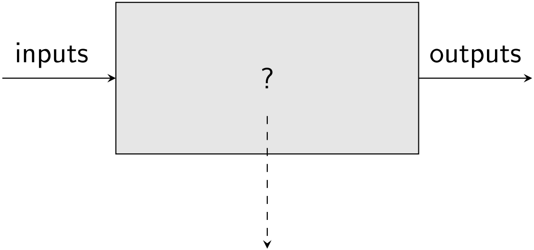
Introduction

The system of interest:



Introduction

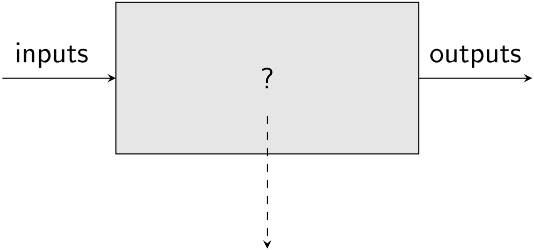
The system of interest:



dynamic model:

Introduction

The system of interest:

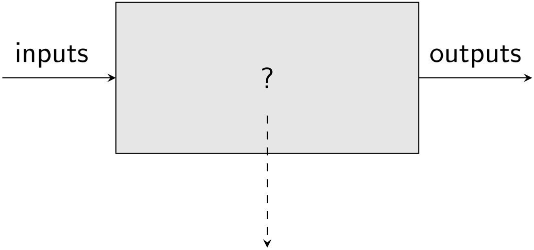


dynamic model:

deterministic set of differential equations (ODE/DAE/PDE)

Introduction

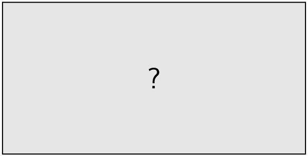
The system of interest:



dynamic model:
deterministic set of differential equations (**ODE**/**DAE**/PDE)

Introduction

The system of interest:



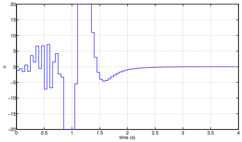
Introduction

The system of interest:

$$\boxed{\dot{x}(t) = f(t, x(t), \mathbf{u(t)})}$$

Introduction

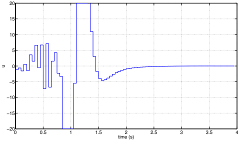
The system of interest:



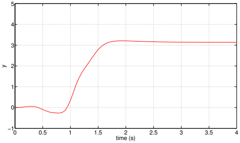
$$\dot{x}(t) = f(t, x(t), \mathbf{u(t)})$$

Introduction

The system of interest:



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



Introduction: IVP

THEOREM [Picard 1890, Lindelöf 1894]:

Initial value problem in ODE

$$\begin{aligned} \dot{x}(t) &= f(t, x(t), u(t), p), \quad t \in [t_0, t_{\mathrm{end}}], \\ x(t_0) &= x_0 \end{aligned}$$

- with given initial state x_0, parameters p, and controls $u(t)$,
- and Lipschitz continuous $f(t, x(t), u(t), p)$

Introduction: IVP

THEOREM [Picard 1890, Lindelöf 1894]:

Initial value problem in ODE

$$\begin{aligned} \dot{x}(t) &= f(t, x(t), u(t), p), \quad t \in [t_0, t_{\text{end}}], \\ x(t_0) &= x_0 \end{aligned}$$

- with given initial state x_0, parameters p, and controls $u(t)$,
- and Lipschitz continuous $f(t, x(t), u(t), p)$

has **unique** solution

$$x(t), \ t \in [t_0, t_{\text{end}}]$$

Introduction: numerical simulation

Aim of numerical simulation:

Compute $x(t)$, $t \in [t_0, t_{\mathrm{end}}]$ which approximately satisfies

$$\begin{aligned} \dot{x}(t) &= f(t, x(t), u(t), p), \quad t \in [t_0, t_{\mathrm{end}}], \\ x(t_0) &= x_0, \end{aligned}$$

and $z(t)$ in case of index-1 DAE

$$\begin{aligned} \dot{x}(t) &= f(t, x(t), z(t), u(t), p), \\ 0 &= g(t, x(t), z(t), u(t), p), \quad t \in [t_0, t_{\mathrm{end}}], \\ x(t_0) &= x_0 \end{aligned}$$

NOTE: interested in values at discrete times $t_i \in [t_0, t_{\mathrm{end}}]$, especially $t = t_{\mathrm{end}}$

Introduction: numerical simulation

Let us define the exact trajectory $x(t)$, $t \in [t_0, t_{\mathrm{end}}]$ and a set of discrete time steps $t_0, t_1, \dots$

Introduction: numerical simulation

Let us define the exact trajectory $x(t)$, $t \in [t_0, t_{\mathrm{end}}]$ and a set of discrete time steps $t_0, t_1, \ldots$

Local error:

$$e(t_i) = x(t_i) - x(t_i;\ t_{i-1}, x(t_{i-1}))$$

Introduction: numerical simulation

Let us define the exact trajectory $x(t)$, $t \in [t_0, t_{\mathrm{end}}]$ and a set of discrete time steps $t_0, t_1, \ldots$

Local error:

$$e(t_i) = x(t_i) - x(t_i;\ t_{i-1}, x(t_{i-1}))$$

Global error or "transported error":

$$E(t_i) = x(t_i) - x(t_i;\ t_0, x_0)$$

Introduction: important properties

Let us define the stepsize h such that $t_{i+1} = t_i + h$

Introduction: important properties

Let us define the stepsize h such that $t_{i+1} = t_i + h$

> **convergence**: A method is convergent when its values converge to the exact solution for $h \to 0$.

Introduction: important properties

Let us define the stepsize h such that $t_{i+1} = t_i + h$

convergence: A method is convergent when its values converge to the exact solution for $h \to 0$.

order: The method has order p if the local error

$$\lim_{h \to 0} e(t_i) = O(h^{p+1})$$

NOTE: consistency when $p > 0$ (necessary for convergence)

Introduction: important properties

Let us define the stepsize h such that $t_{i+1} = t_i + h$

convergence: A method is convergent when its values converge to the exact solution for $h \to 0$.

order: The method has order p if the local error

$$\lim_{h \to 0} e(t_i) = O(h^{p+1})$$

NOTE: consistency when $p > 0$ (necessary for convergence)

stability: 'damping' of errors, see stiffness

Introduction: important properties

Let us define the stepsize h such that $t_{i+1} = t_i + h$

convergence: A method is convergent when its values converge to the exact solution for $h \to 0$. $\to$ minimum!

order: The method has order p if the local error

$$\lim_{h \to 0} e(t_i) = O(h^{p+1})$$

NOTE: consistency when $p > 0$ (necessary for convergence)

stability: 'damping' of errors, see stiffness

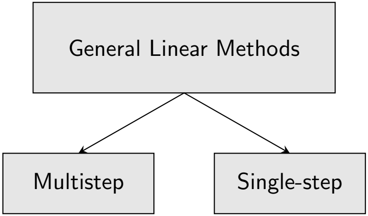
Overview

Classes of numerical methods:

General Linear Methods

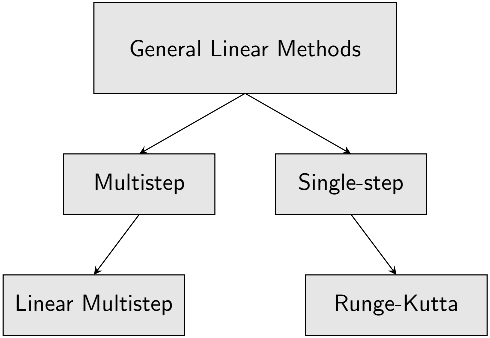
Overview

Classes of numerical methods:



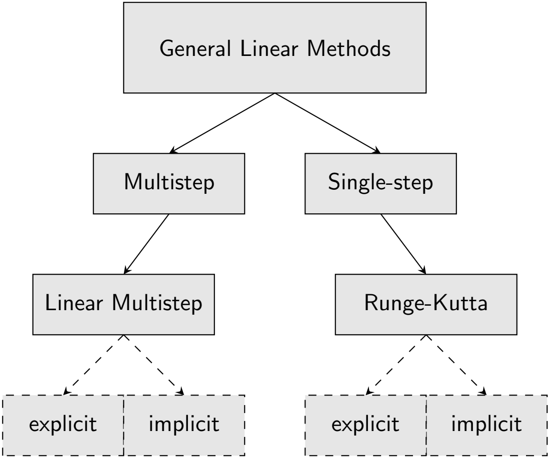
Overview

Classes of numerical methods:



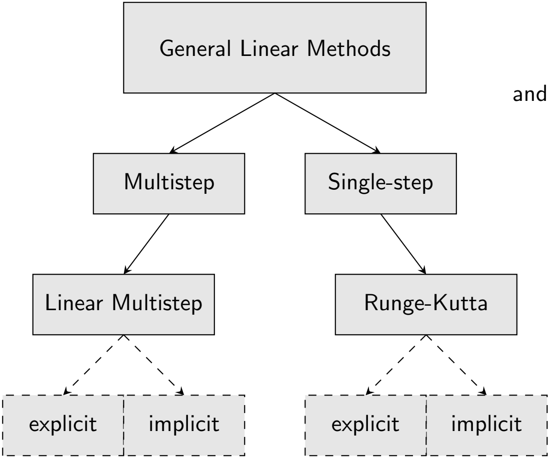
Overview

Classes of numerical methods:



Overview

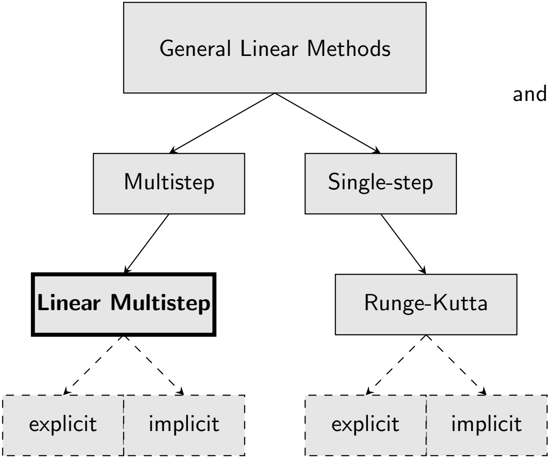
Classes of numerical methods:



and others ...

Overview

Classes of numerical methods:



and others ...

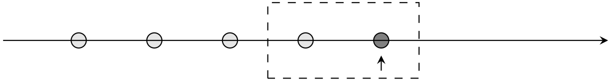
Multistep methods

Each method takes a step forward in time to find the next solution point, but this can be based either:

Multistep methods

Each method takes a step forward in time to find the next solution point, but this can be based either:

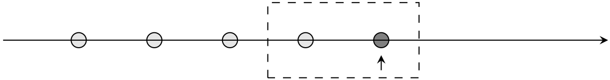
- on the previous point and its derivative, often with intermediate steps (see Runge-Kutta)



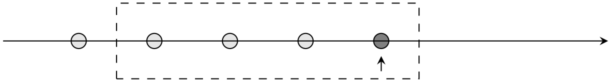
Multistep methods

Each method takes a step forward in time to find the next solution point, but this can be based either:

- on the previous point and its derivative, often with intermediate steps (see Runge-Kutta)



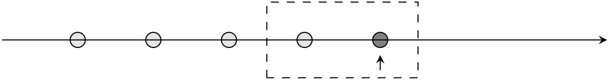
- on a certain amount of previous points and their derivatives



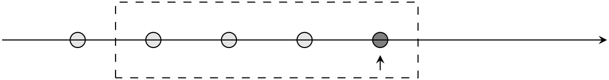
Multistep methods

Each method takes a step forward in time to find the next solution point, but this can be based either:

- on the previous point and its derivative, often with intermediate steps (see Runge-Kutta)



- on a certain amount of previous points and their derivatives



$\Rightarrow$ **good starting procedure needed!**

Linear multistep methods

Let us consider the simplified system $\dot{x}(t) = f(t, x(t))$.

A s-step LM method then uses x_i, $f_i = f(t_i, x_i)$ for $i = n - s, \ldots, n - 1$ to compute $x_n \approx x(t_n)$:

$$x_n + a_{s-1}x_{n-1} + \ldots + a_0 x_{n-s} = h\left(b_s f_n + b_{s-1} f_{n-1} + \ldots + b_0 f_{n-s}\right)$$

Linear multistep methods

Let us consider the simplified system $\dot{x}(t) = f(t, x(t))$.

A s-step LM method then uses x_i, $f_i = f(t_i, x_i)$ for $i = n-s, \ldots, n-1$ to compute $x_n \approx x(t_n)$:

$$\begin{aligned} \mathbf{x_n} + a_{s-1}x_{n-1} \; + \; \ldots \; + \; a_0 x_{n-s} \; = \\ h\left(\underline{b_s}\mathbf{f_n} \; + \; b_{s-1}f_{n-1} \; + \; \ldots \; + \; b_0 f_{n-s}\right) \end{aligned}$$

explicit $(b_s = 0)$ $\quad\leftrightarrow\quad$ implicit $(b_s \neq 0)$

Linear multistep methods

Let us consider the simplified system $\dot{x}(t) = f(t, x(t))$.

A s-step LM method then uses x_i, $f_i = f(t_i, x_i)$ for $i = n - s, \ldots, n - 1$ to compute $x_n \approx x(t_n)$:

$$\mathbf{x_n} + a_{s-1}x_{n-1} + \ldots + a_0 x_{n-s} = h\left(\underline{b_s}\mathbf{f_n} + b_{s-1}f_{n-1} + \ldots + b_0 f_{n-s}\right)$$

explicit ($b_s = 0$) $\leftrightarrow$ implicit ($b_s \neq 0$)

Three main families:

- Adams-Bashforth (explicit)
- Adams-Moulton (implicit)
- Backward differentiation formulas (BDF)

Intermezzo: stiffness

"... stiff equations are equations where certain implicit methods, in paricular BDF, perform better, usually tremendously better, than explicit ones."

- (Curtiss & Hirschfelder, 1952)

Intermezzo: stiffness

"... stiff equations are equations where certain implicit methods, in paricular BDF, perform better, usually tremendously better, than explicit ones."

- (Curtiss & Hirschfelder, 1952)

"... Around 1960, things became completely different and everyone became aware that the world was full of stiff problems."

- (G. Dahlquist, 1985)

Intermezzo: stiffness example

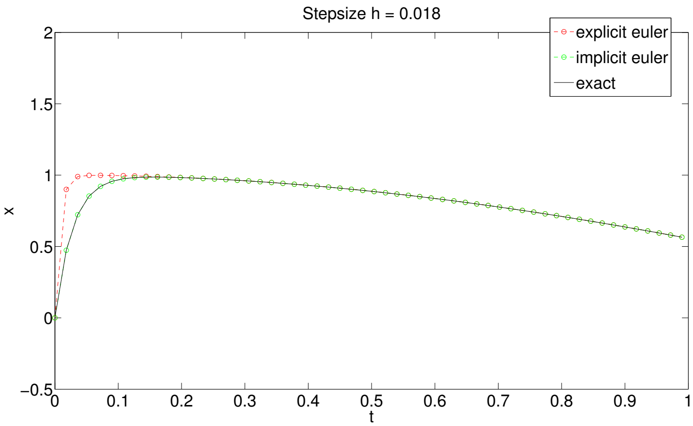
Let us consider the following simple one-dimensional system

$$\dot{x}(t) = -50(x(t) - cos(t))$$

Intermezzo: stiffness example

Let us consider the following simple one-dimensional system

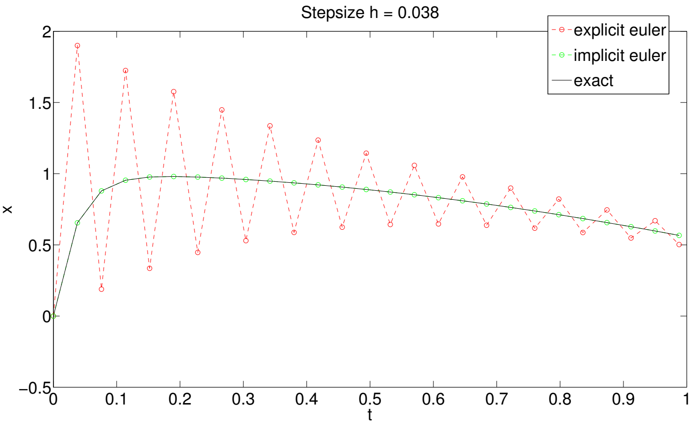
$$\dot{x}(t) = -50(x(t) - cos(t))$$



Intermezzo: stiffness example

Let us consider the following simple one-dimensional system

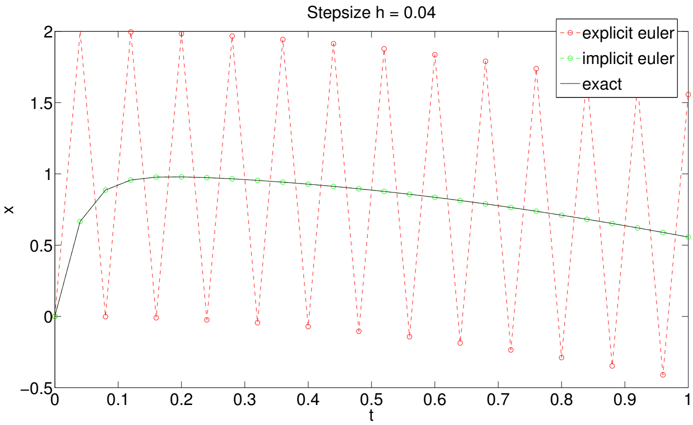
$$\dot{x}(t) = -50(x(t) - cos(t))$$



Intermezzo: stiffness example

Let us consider the following simple one-dimensional system

$$\dot{x}(t) = -50(x(t) - cos(t))$$



Intermezzo: stiffness

Stiffness depends largely on

Intermezzo: stiffness

Stiffness depends largely on

- the eigenvalues $\lambda(t)$ of the Jacobian $\frac{\partial f}{\partial x}$
- but also system dimension, smoothness of the solution, ...

Intermezzo: stiffness

Stiffness depends largely on

- the eigenvalues $\lambda(t)$ of the Jacobian $\frac{\partial f}{\partial x}$
- but also system dimension, smoothness of the solution, ...

$$\Downarrow$$

- various mathematical definitions exist
- new concepts needed:
  A-stability, I-stability, A(α)-stability, L-stability, ...

Intermezzo: stiffness

Stiffness depends largely on

- the eigenvalues $\lambda(t)$ of the Jacobian $\frac{\partial f}{\partial x}$
- but also system dimension, smoothness of the solution, ...

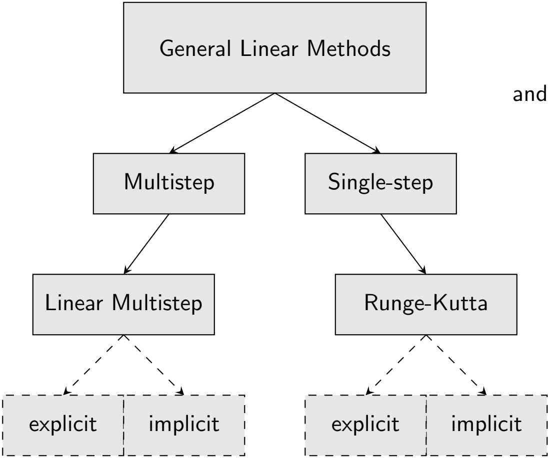
$$\Downarrow$$

- various mathematical definitions exist
- new concepts needed:
  A-stability, I-stability, A(α)-stability, L-stability, ...

Main message: stiff systems require (semi-)implicit methods!

Overview

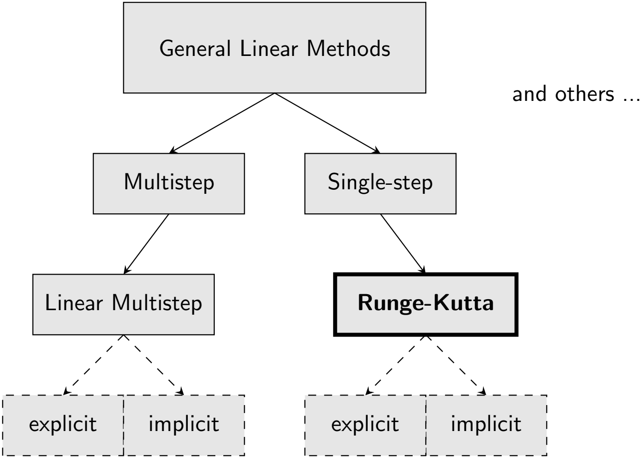
Classes of numerical methods:



and others ...

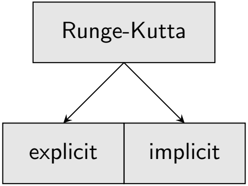
Overview

Classes of numerical methods:



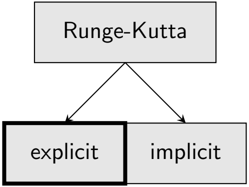
Overview

Runge-Kutta methods:



Overview

Runge-Kutta methods:



Explicit Runge-Kutta (ERK) methods

The simplest ERK method is explicit Euler

Explicit Runge-Kutta (ERK) methods

The simplest ERK method is explicit Euler

$$x_n = x_{n-1} + h\, f_{n-1}$$

which is consistent of order one (minimum).

Explicit Runge-Kutta (ERK) methods

The simplest ERK method is explicit Euler

$$x_n = x_{n-1} + h\, f_{n-1}$$

which is consistent of order one (minimum).

BUT: it is typically not a practical method... Why?

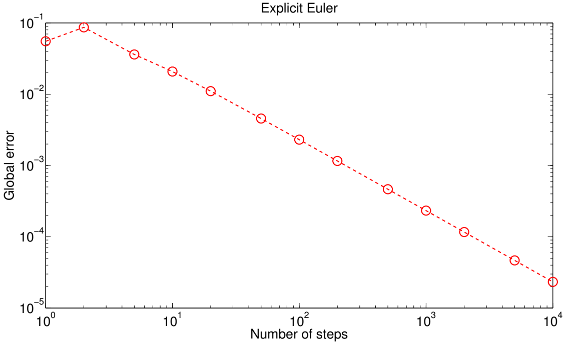
Explicit Runge-Kutta (ERK) methods

The simplest ERK method is explicit Euler

$$x_n = x_{n-1} + h\, f_{n-1}$$

which is consistent of order one (minimum).

BUT: it is typically not a practical method... Why?
Higher order methods need much fewer steps for same accuracy!



Explicit Runge-Kutta (ERK) methods

The most popular is the following 4^{th} order method

Explicit Runge-Kutta (ERK) methods

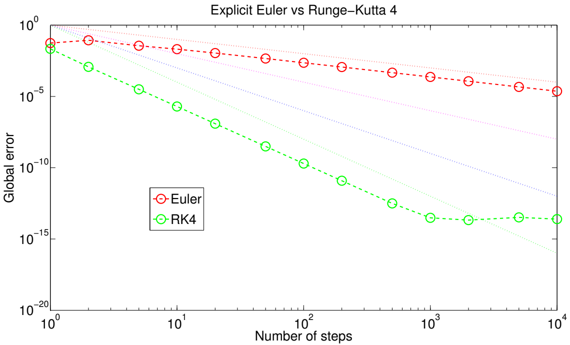
The most popular is the following 4^{th} order method

$$\begin{aligned}
k_1 &= f(t_{n-1}, x_{n-1}) \\
k_2 &= f(t_{n-1} + \frac{h}{2}, x_{n-1} + \frac{h}{2} k_1) \\
k_3 &= f(t_{n-1} + \frac{h}{2}, x_{n-1} + \frac{h}{2} k_2) \\
k_4 &= f(t_{n-1} + h, x_{n-1} + h\, k_3) \\
x_n &= x_{n-1} + \frac{h}{6} (k_1 + 2k_2 + 2k_3 + k_4)
\end{aligned}$$

Explicit Runge-Kutta (ERK) methods

The most popular is the following 4th order method

$$
\begin{aligned}
k_1 &= f(t_{n-1}, x_{n-1}) \\
k_2 &= f(t_{n-1} + \frac{h}{2}, x_{n-1} + \frac{h}{2} k_1) \\
k_3 &= f(t_{n-1} + \frac{h}{2}, x_{n-1} + \frac{h}{2} k_2) \\
k_4 &= f(t_{n-1} + h, x_{n-1} + h\, k_3) \\
x_n &= x_{n-1} + \frac{h}{6}\,(k_1 + 2k_2 + 2k_3 + k_4)
\end{aligned}
$$



Explicit Runge-Kutta (ERK) methods

The RK4 method

$$
\begin{aligned}
k_1 &= f(t_{n-1}, x_{n-1}) \\
k_2 &= f(t_{n-1} + \frac{h}{2}, x_{n-1} + \frac{h}{2} k_1) \\
k_3 &= f(t_{n-1} + \frac{h}{2}, x_{n-1} + \frac{h}{2} k_2) \\
k_4 &= f(t_{n-1} + h, x_{n-1} + h\, k_3) \\
x_n &= x_{n-1} + \frac{h}{6} (k_1 + 2k_2 + 2k_3 + k_4)
\end{aligned}
$$

Explicit Runge-Kutta (ERK) methods

So a general s-stage ERK method

$$
\begin{aligned}
k_1 &= f(t_{n-1}, x_{n-1}) \\
k_2 &= f(t_{n-1} + c_2\, h, x_{n-1} + a_{21}\, h\, k_1) \\
k_3 &= f(t_{n-1} + c_3\, h, x_{n-1} + a_{31}\, h\, k_1 + a_{32}\, h\, k_2) \\
&\vdots \\
k_s &= f(t_{n-1} + c_s\, h, x_{n-1} + a_{s1}\, h\, k_1 + a_{s2}\, h\, k_2 + \ldots + a_{s,s-1}\, h\, k_{s-1}) \\
x_n &= x_{n-1} + h \sum_{i=1}^{s} b_i\, k_i
\end{aligned}
$$

Explicit Runge-Kutta (ERK) methods

So a general s-stage ERK method

$$
\begin{aligned}
k_1 &= f(t_{n-1}, x_{n-1}) \\
k_2 &= f(t_{n-1} + c_2\,h, x_{n-1} + a_{21}\,h\,k_1) \\
k_3 &= f(t_{n-1} + c_3\,h, x_{n-1} + a_{31}\,h\,k_1 + a_{32}\,h\,k_2) \\
&\vdots \\
k_s &= f(t_{n-1} + c_s\,h, x_{n-1} + a_{s1}\,h\,k_1 + a_{s2}\,h\,k_2 + \ldots + a_{s,s-1}\,h\,k_{s-1}) \\
x_n &= x_{n-1} + h \sum_{i=1}^{s} b_i\,k_i
\end{aligned}
$$

$$
\begin{array}{c|cccc}
0 & & & & \\
c_2 & a_{21} & & & \\
c_3 & a_{31} & a_{32} & & \\
\vdots & \vdots & & \ddots & \\
c_s & a_{s1} & a_{s2} & \cdots & \\
\hline
 & b_1 & b_2 & \cdots & b_s
\end{array}
$$

Explicit Runge-Kutta (ERK) methods

So a general s-stage ERK method

$$
\begin{aligned}
k_1 &= f(t_{n-1}, x_{n-1}) \\
k_2 &= f(t_{n-1} + c_2\, h, x_{n-1} + a_{21}\, h\, k_1) \\
k_3 &= f(t_{n-1} + c_3\, h, x_{n-1} + a_{31}\, h\, k_1 + a_{32}\, h\, k_2) \\
&\vdots \\
k_s &= f(t_{n-1} + c_s\, h, x_{n-1} + a_{s1}\, h\, k_1 + a_{s2}\, h\, k_2 + \ldots + a_{s,s-1}\, h\, k_{s-1}) \\
x_n &= x_{n-1} + h \sum_{i=1}^{s} b_i\, k_i
\end{aligned}
$$

$$
\begin{array}{c|cccc}
0 & & & & \\
c_2 & a_{21} & & & \\
c_3 & a_{31} & a_{32} & & \\
\vdots & \vdots & & \ddots & \\
c_s & a_{s1} & a_{s2} & \cdots & \\
\hline
 & b_1 & b_2 & \cdots & b_s
\end{array}
$$

NOTE: each Runge-Kutta method is defined by its Butcher table!
other examples are e.g. the methods of Runge and Heun, ...

Intermezzo: Step size control

Typically:

no constant step size but suitable error control

Intermezzo: Step size control

Typically:

no constant step size but suitable error control

based on a local error estimate:

$$e_i \approx \|x(t_i) - x(t_i;\, t_{i-1}, x(t_{i-1}))\|$$

Intermezzo: Step size control

Example:

Euler: $x_n = x_{n-1} + h\, f_{n-1}$

Intermezzo: Step size control

Example:

Euler: $x_n = x_{n-1} + h\, f_{n-1}$

Let us create a reference solution using 2 steps with $h/2$:

$$
\begin{aligned}
x_{n-1/2} &= x_{n-1} + \frac{h}{2}\, f_{n-1} \\
\tilde{x}_n &= x_{n-1/2} + \frac{h}{2}\, f_{n-1/2}
\end{aligned}
$$

Intermezzo: Step size control

Example:

Euler: $x_n = x_{n-1} + h\, f_{n-1}$

Let us create a reference solution using 2 steps with $h/2$:

$$x_{n-1/2} = x_{n-1} + \frac{h}{2}\, f_{n-1}$$
$$\tilde{x}_n = x_{n-1/2} + \frac{h}{2}\, f_{n-1/2}$$

$e_n = \tilde{x}_n - x_n \quad \Rightarrow \quad$ accept/reject

and update the step size: $h_n = 0.9\, h_{n-1} \sqrt[p+1]{\frac{TOL}{E}}$

Intermezzo: Step size control

Example:

Euler: $x_n = x_{n-1} + h\, f_{n-1}$

Let us create a reference solution using 2 steps with $h/2$:

$$x_{n-1/2} = x_{n-1} + \frac{h}{2}\, f_{n-1}$$
$$\tilde{x}_n = x_{n-1/2} + \frac{h}{2}\, f_{n-1/2}$$

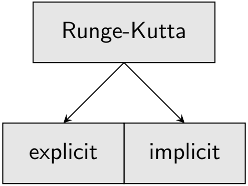
$e_n = \tilde{x}_n - x_n \quad \Rightarrow \quad$ accept/reject

and update the step size: $h_n = 0.9\, h_{n-1} \sqrt[p+1]{\frac{TOL}{E}}$

Embedded methods: Fehlberg (e.g. RKF45), Dormand-Prince, ...

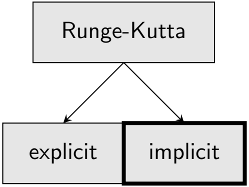
Overview

Runge-Kutta methods:



Overview

Runge-Kutta methods:



Implicit Runge-Kutta (IRK) methods

IRK as the natural generalization from ERK methods:

$$
\begin{array}{c|cccc}
0 & & & & \\
c_2 & a_{21} & & & \\
c_3 & a_{31} & a_{32} & & \\
\vdots & \vdots & & \ddots & \\
c_s & a_{s1} & a_{s2} & \cdots & \\
\hline
 & b_1 & b_2 & \cdots & b_s
\end{array}
$$

Implicit Runge-Kutta (IRK) methods

IRK as the natural generalization from ERK methods:

$$
\begin{array}{c|cccc}
0 & & & & \\
c_2 & a_{21} & & & \\
c_3 & a_{31} & a_{32} & & \\
\vdots & \vdots & & \ddots & \\
c_s & a_{s1} & a_{s2} & \cdots & \\
\hline
 & b_1 & b_2 & \cdots & b_s
\end{array}
\quad \Rightarrow \quad
\begin{array}{c|ccc}
c_1 & a_{11} & \cdots & a_{1s} \\
c_2 & a_{21} & \cdots & a_{2s} \\
\vdots & \vdots & & \vdots \\
c_s & a_{s1} & \cdots & a_{ss} \\
\hline
 & b_1 & \cdots & b_s
\end{array}
$$

Implicit Runge-Kutta (IRK) methods

IRK as the natural generalization from ERK methods:

$$
\begin{array}{c|cccc}
0 & & & & \\
c_2 & a_{21} & & & \\
c_3 & a_{31} & a_{32} & & \\
\vdots & \vdots & & \ddots & \\
c_s & a_{s1} & a_{s2} & \cdots & \\
\hline
 & b_1 & b_2 & \cdots & b_s
\end{array}
\quad \Rightarrow \quad
\begin{array}{c|ccc}
c_1 & a_{11} & \cdots & a_{1s} \\
c_2 & a_{21} & \cdots & a_{2s} \\
\vdots & \vdots & & \vdots \\
c_s & a_{s1} & \cdots & a_{ss} \\
\hline
 & b_1 & \cdots & b_s
\end{array}
$$

e.g.

$x_n = x_{n-1} + h\, f_{n-1}$

Implicit Runge-Kutta (IRK) methods

IRK as the natural generalization from ERK methods:

$$
\begin{array}{c|cccc}
0 & & & & \\
c_2 & a_{21} & & & \\
c_3 & a_{31} & a_{32} & & \\
\vdots & \vdots & & \ddots & \\
c_s & a_{s1} & a_{s2} & \cdots & \\
\hline
 & b_1 & b_2 & \cdots & b_s
\end{array}
\quad \Rightarrow \quad
\begin{array}{c|ccc}
c_1 & a_{11} & \cdots & a_{1s} \\
c_2 & a_{21} & \cdots & a_{2s} \\
\vdots & \vdots & & \vdots \\
c_s & a_{s1} & \cdots & a_{ss} \\
\hline
 & b_1 & \cdots & b_s
\end{array}
$$

e.g.

$x_n = x_{n-1} + h\, f_{n-1}$

$x_n = x_{n-1} + h\, f_n$

Implicit Runge-Kutta (IRK) methods

IRK as the natural generalization from ERK methods:

$$
\begin{aligned}
k_1 &= f\left(t_{n-1} + c_1\, h, x_{n-1} + h \sum_{j=1}^{s} a_{1j}\, k_j\right) \\
&\vdots \\
k_s &= f\left(t_{n-1} + c_s\, h, x_{n-1} + h \sum_{j=1}^{s} a_{sj}\, k_j\right) \\
x_n &= x_{n-1} + h \sum_{i=1}^{s} b_i\, k_i
\end{aligned}
$$

$$
\begin{array}{c|ccc}
c_1 & a_{11} & \cdots & a_{1s} \\
c_2 & a_{21} & \cdots & a_{2s} \\
\vdots & \vdots & & \vdots \\
c_s & a_{s1} & \cdots & a_{ss} \\
\hline
 & b_1 & \cdots & b_s
\end{array}
$$

Implicit Runge-Kutta (IRK) methods

IRK as the natural generalization from ERK methods:

$$
\begin{aligned}
k_1 &= f\left(t_{n-1} + c_1\, h, x_{n-1} + h \sum_{j=1}^{s} a_{1j}\, k_j\right) \\
&\vdots \\
k_s &= f\left(t_{n-1} + c_s\, h, x_{n-1} + h \sum_{j=1}^{s} a_{sj}\, k_j\right) \\
x_n &= x_{n-1} + h \sum_{i=1}^{s} b_i\, k_i
\end{aligned}
$$

$$
\begin{array}{c|ccc}
c_1 & a_{11} & \cdots & a_{1s} \\
c_2 & a_{21} & \cdots & a_{2s} \\
\vdots & \vdots & & \vdots \\
c_s & a_{s1} & \cdots & a_{ss} \\
\hline
 & b_1 & \cdots & b_s
\end{array}
$$

pro: nice properties (order, stability)

Implicit Runge-Kutta (IRK) methods

IRK as the natural generalization from ERK methods:

$$\mathbf{k_1} = f\left(t_{n-1} + c_1\, h, x_{n-1} + h \sum_{j=1}^{s} a_{1j}\, \mathbf{k_j}\right)$$

$$\vdots$$

$$\mathbf{k_s} = f\left(t_{n-1} + c_s\, h, x_{n-1} + h \sum_{j=1}^{s} a_{sj}\, \mathbf{k_j}\right)$$

$$x_n = x_{n-1} + h \sum_{i=1}^{s} b_i\, k_i$$

$$\begin{array}{c|ccc}
c_1 & a_{11} & \cdots & a_{1s} \\
c_2 & a_{21} & \cdots & a_{2s} \\
\vdots & \vdots & & \vdots \\
c_s & a_{s1} & \cdots & a_{ss} \\
\hline
 & b_1 & \cdots & b_s
\end{array}$$

pro: nice properties (order, stability)

con: large nonlinear system

Implicit Runge-Kutta (IRK) methods

IRK as the natural generalization from ERK methods:

$$
\begin{aligned}
\mathbf{k_1} &= f\left(t_{n-1} + c_1\,h,\, x_{n-1} + h\sum_{j=1}^{s} a_{1j}\,\mathbf{k_j}\right) \\
&\vdots \\
\mathbf{k_s} &= f\left(t_{n-1} + c_s\,h,\, x_{n-1} + h\sum_{j=1}^{s} a_{sj}\,\mathbf{k_j}\right) \\
x_n &= x_{n-1} + h\sum_{i=1}^{s} b_i\,k_i
\end{aligned}
$$

$$
\begin{array}{c|ccc}
c_1 & a_{11} & \cdots & a_{1s} \\
c_2 & a_{21} & \cdots & a_{2s} \\
\vdots & \vdots & & \vdots \\
c_s & a_{s1} & \cdots & a_{ss} \\
\hline
 & b_1 & \cdots & b_s
\end{array}
$$

pro: nice properties (order, stability)

con: large nonlinear system $\Rightarrow$ Newton's method

Implicit Runge-Kutta (IRK) methods

Explicit ODE system:

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

$$\begin{aligned}
k_1 &= f\left(t_{n-1} + c_1\, h, x_{n-1} + h \sum_{j=1}^{s} a_{1j}\, k_j\right) \\
&\vdots \\
k_s &= f\left(t_{n-1} + c_s\, h, x_{n-1} + h \sum_{j=1}^{s} a_{sj}\, k_j\right) \\
x_n &= x_{n-1} + h \sum_{i=1}^{s} b_i\, k_i
\end{aligned}$$

Implicit Runge-Kutta (IRK) methods

Explicit ODE system:

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

$$k_1 = f\left(t_{n-1} + c_1\, h, x_{n-1} + h \sum_{j=1}^{s} a_{1j}\, k_j\right)$$

$$\vdots$$

$$k_s = f\left(t_{n-1} + c_s\, h, x_{n-1} + h \sum_{j=1}^{s} a_{sj}\, k_j\right)$$

$$x_n = x_{n-1} + h \sum_{i=1}^{s} b_i\, k_i$$

Implicit ODE/DAE (index 1):

$$0 = f(t, \dot{x}(t), x(t), z(t))$$

$$0 = f\left(t_{n-1} + c_1\, h, k_1, x_{n-1} + h \sum_{j=1}^{s} a_{1j}\, k_j, Z_1\right)$$

$$\vdots$$

$$0 = f\left(t_{n-1} + c_s\, h, k_s, x_{n-1} + h \sum_{j=1}^{s} a_{sj}\, k_j, Z_s\right)$$

$$x_n = x_{n-1} + h \sum_{i=1}^{s} b_i\, k_i$$

Implicit Runge-Kutta (IRK) methods

Explicit ODE system:

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

$$\mathbf{k_1} = f\left(t_{n-1} + c_1\, h, x_{n-1} + h \sum_{j=1}^{s} a_{1j}\, \mathbf{k_j}\right)$$

$$\vdots$$

$$\mathbf{k_s} = f\left(t_{n-1} + c_s\, h, x_{n-1} + h \sum_{j=1}^{s} a_{sj}\, \mathbf{k_j}\right)$$

$$x_n = x_{n-1} + h \sum_{i=1}^{s} b_i\, k_i$$

Implicit ODE/DAE (index 1):

$$0 = f(t, \dot{x}(t), x(t), z(t))$$

$$0 = f\left(t_{n-1} + c_1\, h, \mathbf{k_1}, x_{n-1} + h \sum_{j=1}^{s} a_{1j}\, \mathbf{k_j}, \mathbf{Z_1}\right)$$

$$\vdots$$

$$0 = f\left(t_{n-1} + c_s\, h, \mathbf{k_s}, x_{n-1} + h \sum_{j=1}^{s} a_{sj}\, \mathbf{k_j}, \mathbf{Z_s}\right)$$

$$x_n = x_{n-1} + h \sum_{i=1}^{s} b_i\, k_i$$

Collocation methods

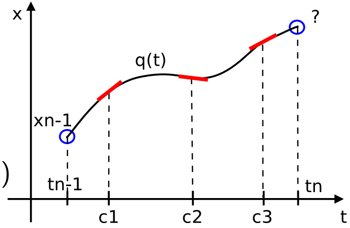
Important family of IRK methods:

- distinct c_i's (nonconfluent)
- polynomial $q(t)$ of degree s

Collocation methods

Important family of IRK methods:

- distinct c_i's (nonconfluent)
- polynomial $q(t)$ of degree s



$$
\begin{aligned}
q(t_{n-1}) &= x_{n-1} \\
\dot{q}(t_{n-1} + c_1 h) &= f(t_{n-1} + c_1 h, q(t_{n-1} + c_1 h)) \\
&\vdots \\
\dot{q}(t_{n-1} + c_s h) &= f(t_{n-1} + c_s h, q(t_{n-1} + c_s h))
\end{aligned}
$$

continuous approximation

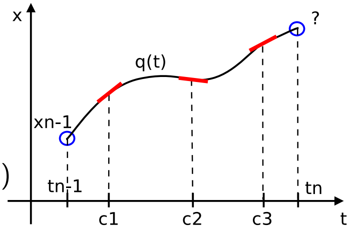
$\Rightarrow \quad x_n = q(t_{n-1} + h)$

Collocation methods

Important family of IRK methods:

- distinct c_i's (nonconfluent)
- polynomial $q(t)$ of degree s

$$
\begin{aligned}
q(t_{n-1}) &= x_{n-1} \\
\dot{q}(t_{n-1} + c_1 h) &= f(t_{n-1} + c_1 h, q(t_{n-1} + c_1 h)) \\
&\vdots \\
\dot{q}(t_{n-1} + c_s h) &= f(t_{n-1} + c_s h, q(t_{n-1} + c_s h))
\end{aligned}
$$



continuous approximation

$\Rightarrow \quad x_n = q(t_{n-1} + h)$

NOTE: this is very popular in direct optimal control!

Collocation methods

How to implement a collocation method?

$$
\begin{aligned}
q(t_{n-1}) &= x_{n-1} \\
\dot{q}(t_{n-1} + c_1 h) &= f(t_{n-1} + c_1 h, q(t_{n-1} + c_1 h)) \\
&\vdots \\
\dot{q}(t_{n-1} + c_s h) &= f(t_{n-1} + c_s h, q(t_{n-1} + c_s h))
\end{aligned}
$$

Collocation methods

How to implement a collocation method?

$$
\begin{aligned}
q(t_{n-1}) &= x_{n-1} \\
\dot{q}(t_{n-1} + c_1 h) &= f(t_{n-1} + c_1 h, q(t_{n-1} + c_1 h)) \\
&\vdots \\
\dot{q}(t_{n-1} + c_s h) &= f(t_{n-1} + c_s h, q(t_{n-1} + c_s h))
\end{aligned}
$$

This is nothing else than ...

$$
\begin{aligned}
k_1 &= f(t_{n-1} + c_1\, h, x_{n-1} + h \sum_{j=1}^{s} a_{1j}\, k_j) \\
&\vdots \\
k_s &= f(t_{n-1} + c_s\, h, x_{n-1} + h \sum_{j=1}^{s} a_{sj}\, k_j) \\
x_n &= x_{n-1} + h \sum_{i=1}^{s} b_i\, k_i
\end{aligned}
$$

where the Butcher table is defined by the collocation nodes c_i.

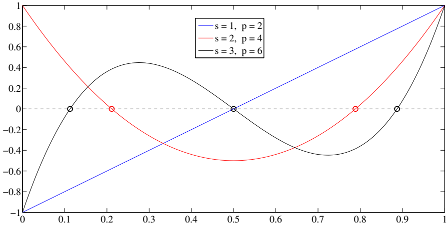
Collocation methods

Example: The Gauss methods

Collocation methods

Example: The Gauss methods

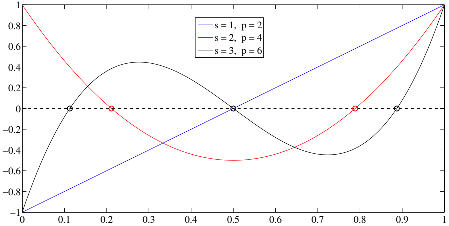
- roots of Legendre polynomials
- A-stable
- optimal order ($p = 2s$)



Collocation methods

Example: The Gauss methods

- roots of Legendre polynomials
- A-stable
- optimal order ($p = 2s$)

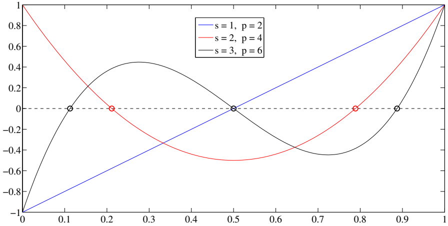


$$
\begin{aligned}
c_1 &= \frac{1}{2}, & s &= 1, \quad p = 2,\\
c_1 = \frac{1}{2} - \frac{\sqrt{3}}{6},\ c_2 &= \frac{1}{2} + \frac{\sqrt{3}}{6}, & s &= 2, \quad p = 4,\\
c_1 = \frac{1}{2} - \frac{\sqrt{15}}{10},\ c_2 = \frac{1}{2},\ c_3 &= \frac{1}{2} + \frac{\sqrt{15}}{10}, & s &= 3, \quad p = 6.
\end{aligned}
$$

Collocation methods

Example: The Gauss methods

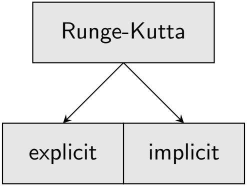
- roots of Legendre polynomials
- A-stable
- optimal order ($p = 2s$)

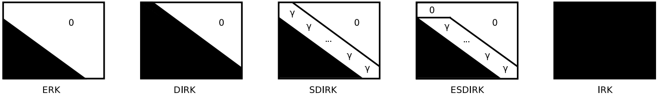


At least as popular:
Radau IIA methods ($p = 2s - 1$, stiffly accurate, L-stable)

Overview

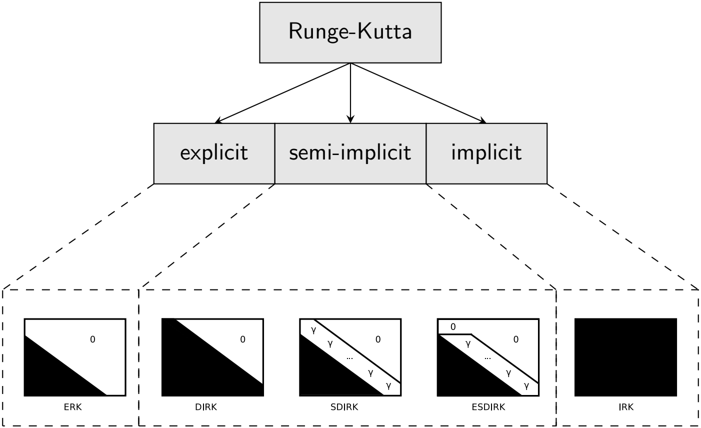
Runge-Kutta methods:





Overview

Runge-Kutta methods:



Semi-implicit Runge-Kutta methods

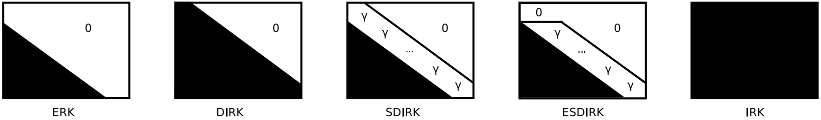
The matrix A is not strictly lower triangular . . .

Semi-implicit Runge-Kutta methods

The matrix A is not strictly lower triangular ...

but there is a specific structure!

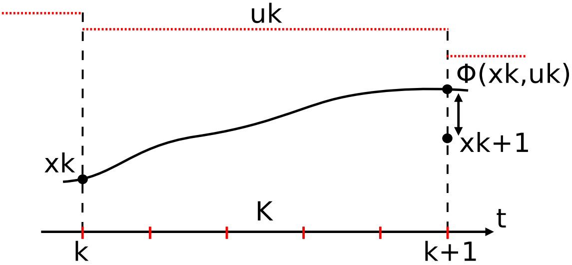
- Diagonal IRK (DIRK)
- Singly DIRK (SDIRK)
- Explicit SDIRK (ESDIRK)



Intermezzo: sensitivity propagation

Task of the integrator in nonlinear optimal control

- $x_{k+1} = \Phi_k(x_k, u_k)$
- nonlinear equality constraint



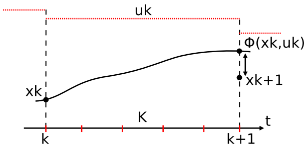
Intermezzo: sensitivity propagation

Task of the integrator in nonlinear optimal control

- $x_{k+1} = \Phi_k(x_k, u_k)$
- nonlinear equality constraint

$$\downarrow$$

- linearization at $\bar{w}_k = (\bar{x}_k, \bar{u}_k)$



$$0 = \boldsymbol{\Phi_k(\bar{w}_k)} - x_{k+1} + \frac{\partial \boldsymbol{\Phi_k}}{\partial \mathbf{w}}\boldsymbol{(\bar{w}_k)}(w_k - \bar{w}_k)$$

- integration and sensitivity generation is typically a major computational step

Intermezzo: sensitivity propagation

"integrate-then-differentiate"

- derivatives of result
- Internal Numerical Differentiation (IND)
- direct IFT approach

Intermezzo: sensitivity propagation

"integrate-then-differentiate"

- derivatives of result
- Internal Numerical Differentiation (IND)
- direct IFT approach

"differentiate-then-integrate"

- sensitivity equations
- extends IVP (forward)
- or new IVP (reverse)

$\Rightarrow$ They are different

Intermezzo: sensitivity propagation

"integrate-then-differentiate"

- derivatives of result
- Internal Numerical Differentiation (IND)
- direct IFT approach

$$\dot{x} = f(x)$$

"differentiate-then-integrate"

- sensitivity equations
- extends IVP (forward)
- or new IVP (reverse)

$\Rightarrow$ They are different ... or not?

Intermezzo: sensitivity propagation

"integrate-then-differentiate"

- derivatives of result
- Internal Numerical Differentiation (IND)
- direct IFT approach

$$\dot{x} = f(x)$$

$$\Downarrow \text{ integrate}$$

$$x_{k+1} = x_k + h\, f(x_k)$$

"differentiate-then-integrate"

- sensitivity equations
- extends IVP (forward)
- or new IVP (reverse)

$$\begin{pmatrix} \dot{x} \\ \dot{S} \end{pmatrix} = F(X) = \begin{pmatrix} f(x) \\ \frac{\partial f}{\partial x} S \end{pmatrix}$$

$\Rightarrow$ They are different ... or not?

Intermezzo: sensitivity propagation

"integrate-then-differentiate"

- derivatives of result
- Internal Numerical Differentiation (IND)
- direct IFT approach

$$\dot{x} = f(x)$$

$$\Downarrow \text{ integrate}$$

$$x_{k+1} = x_k + h\, f(x_k)$$

$$S_{k+1} = S_k + h\, \frac{\partial f(x_k)}{\partial x} S_k$$

"differentiate-then-integrate"

- sensitivity equations
- extends IVP (forward)
- or new IVP (reverse)

$$\begin{pmatrix} \dot{x} \\ \dot{S} \end{pmatrix} = F(X) = \begin{pmatrix} f(x) \\ \frac{\partial f}{\partial x} S \end{pmatrix}$$

$$\Downarrow \text{ integrate}$$

$$X_{k+1} = X_k + h\, F(X_k)$$

$\Rightarrow$ They are different ... or not?

Intermezzo: sensitivity propagation

Variational Differential Equations
"differentiate-then-integrate"

Solve additional matrix differential equation

$$\dot{x} = f(x) \qquad\qquad x(0) = x_0\,,\; x(t_N) = x_N$$

$$\dot{S} = \frac{\partial f}{\partial x} S \qquad\qquad S(0) = d\,,\;\; S(t_N) = \frac{\partial x_N}{\partial x_0} d$$

Intermezzo: sensitivity propagation

Variational Differential Equations

"differentiate-then-integrate"

Solve additional matrix differential equation

$$
\begin{aligned}
\dot{x} &= f(x) & \qquad x(0) &= x_0\,,\ x(t_N) = x_N \\
\dot{S} &= \frac{\partial f}{\partial x} S & \qquad S(0) &= d\,,\ \ S(t_N) = \frac{\partial x_N}{\partial x_0} d
\end{aligned}
$$

Very accurate at reasonable costs, but:

- Have to get expressions for $\frac{\partial f}{\partial x}(\cdot)$.
- Computed sensitivity is not 100 % identical with derivative of (discretized) integrator result $\Phi(\cdot)$.
- What about implicit integration schemes?

Intermezzo: sensitivity propagation

External Numerical Differentiation (END)
"integrate-then-differentiate"

Finite differences: perturb x_0 and call integrator several times

$$\frac{x(t_N; x_0 + \epsilon\, e_i) - x(t_N; x_0)}{\epsilon}$$

Intermezzo: sensitivity propagation

External Numerical Differentiation (END)
"integrate-then-differentiate"

Finite differences: perturb x_0 and call integrator several times

$$\frac{x(t_N; x_0 + \epsilon\, e_i) - x(t_N; x_0)}{\epsilon}$$

Very easy to implement, but several problems:

- Relatively expensive with overhead of error control.
- How to choose perturbation stepsize? Rule of thumb: $\epsilon = \sqrt{\mathrm{TOL}}$ where TOL is integrator tolerance.
- Loss of half the digits of accuracy: if integrator accuracy has value of $\mathrm{TOL} = 10^{-4}$, derivative has only two valid digits!

Intermezzo: sensitivity propagation

External Numerical Differentiation (END)
"integrate-then-differentiate"

Finite differences: perturb x_0 and call integrator several times

$$\frac{x(t_N; x_0 + \epsilon\, e_i) - x(t_N; x_0)}{\epsilon}$$

Very easy to implement, but several problems:

- Relatively expensive with overhead of error control.
- How to choose perturbation stepsize? Rule of thumb: $\epsilon = \sqrt{\mathrm{TOL}}$ where TOL is integrator tolerance.
- Loss of half the digits of accuracy: if integrator accuracy has value of $\mathrm{TOL} = 10^{-4}$, derivative has only two valid digits!
- Due to adaptivity, each call might have different discretization grids: output $x(t_N; x_0)$ is not differentiable!

Intermezzo: sensitivity propagation

Internal Numerical Differentiation (IND)
"integrate-then-differentiate"

Like END, but evaluate simultaneously all perturbed trajectories x_i with frozen discretization grid.

Up to round-off and linearization errors identical with derivative of numerical solution $\Phi(\cdot)$, but:

- How to choose perturbation stepsize?

Intermezzo: sensitivity propagation

Internal Numerical Differentiation (IND)
"integrate-then-differentiate"

Like END, but evaluate simultaneously all perturbed trajectories x_i with frozen discretization grid.

Up to round-off and linearization errors identical with derivative of numerical solution $\Phi(\cdot)$, but:

- How to choose perturbation stepsize? Rule of thumb: $\epsilon = \sqrt{\mathrm{PREC}}$ where PREC is **machine precision**.

Intermezzo: sensitivity propagation

Internal Numerical Differentiation (IND)
"integrate-then-differentiate"

Like END, but evaluate simultaneously all perturbed trajectories x_i with frozen discretization grid.

Up to round-off and linearization errors identical with derivative of numerical solution $\Phi(\cdot)$, but:

- How to choose perturbation stepsize? Rule of thumb: $\epsilon = \sqrt{\mathrm{PREC}}$ where PREC is **machine precision**.

Note: adaptivity of nominal trajectory only, reuse of matrix factorization in implicit methods, so not only more accurate, but also cheaper than END!

Intermezzo: sensitivity propagation

Algorithmic Differentiation (AD)
"integrate-then-differentiate"

Use Algorithmic Differentiation (AD) to differentiate each step of the integration scheme.

Intermezzo: sensitivity propagation

Algorithmic Differentiation (AD)
"integrate-then-differentiate"

Use Algorithmic Differentiation (AD) to differentiate each step of the integration scheme. Illustration: AD for Euler

$$\dot{x} = f(x)$$

Intermezzo: sensitivity propagation

Algorithmic Differentiation (AD)
"integrate-then-differentiate"

Use Algorithmic Differentiation (AD) to differentiate each step of the integration scheme. Illustration: AD for Euler

$$\dot{x} = f(x)$$
$$\Downarrow \text{ integrate}$$
$$x_{k+1} = x_k + h\, f(x_k)$$

Intermezzo: sensitivity propagation

Algorithmic Differentiation (AD)
"integrate-then-differentiate"

Use Algorithmic Differentiation (AD) to differentiate each step of the integration scheme. Illustration: AD for Euler

$$\dot{x} = f(x)$$

$$\Downarrow \text{ integrate}$$

$$x_{k+1} = x_k + h\, f(x_k)$$

$$S_{k+1} = S_k + h\,\frac{\partial f(x_k)}{\partial x} S_k$$

Up to machine precision 100 % identical with derivative of numerical solution $\Phi(\cdot)$, but:

Intermezzo: sensitivity propagation

Algorithmic Differentiation (AD)

"integrate-then-differentiate"

Use Algorithmic Differentiation (AD) to differentiate each step of the integration scheme. Illustration: AD for Euler

$$\dot{x} = f(x)$$

$$\Downarrow \text{ integrate}$$

$$x_{k+1} = x_k + h\, f(x_k)$$

$$S_{k+1} = S_k + h\, \frac{\partial f(x_k)}{\partial x} S_k$$

Up to machine precision 100 % identical with derivative of numerical solution $\Phi(\cdot)$, but:

- tailored implementations needed (e.g. ACADO) ...
- or integrator and right-hand side $f(\cdot)$ need to be compatible codes (e.g. C++ when using ADOL-C)

Simulation methods: software

Simulation for optimization:

- *SUNDIALS*: BDF and Adams in CVODE(S) + BDF in IDA(S)
- *SolvIND*: BDF in DAESOL-II + RK in RKFSWT
- *ACADO Toolkit*: BDF and (I)RK methods
- ...

Summary

- High order schemes preferable for smooth problems

Summary

- High order schemes preferable for smooth problems
- Explicit methods are good for non-stiff systems

Summary

- High order schemes preferable for smooth problems
- Explicit methods are good for non-stiff systems
- For stiff and/or implicit models, the use of implicit methods (BDF, IRK, ...) is highly recommended

References

- E. Hairer, S.P. Nørsett, and G. Wanner: Solving Ordinary Differential Equations I, Springer Series in Computational Mathematics, Berlin, 1993.
- E. Hairer and G. Wanner: Solving Ordinary Differential Equations II Stiff and Differential-Algebraic Problems, Springer, Berlin Heidelberg, 1996.
- K.E. Brenan, S.L. Campbell, and L.R. Petzold: The Numerical Solution of Initial Value Problems in Differential-Algebraic Equations, SIAM Classics Series, 1996.
- U.M. Ascher and L.R. Petzold: Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations. SIAM, 1998.