Numerical Simulation of Dynamic Systems II

Prof. Dr. François E. Cellier
Department of Computer Science
ETH Zurich

February 26, 2013

State-space Models

Models of dynamic systems with concentrated parameters are commonly represented using sets of first-order ordinary differential equations (ODEs). We call these models *state-space models*.

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

where $\mathbf{x}$ is the *state vector*, $\mathbf{u}$ is the *input vector*, and t denotes the *time*, the independent variable across which we wish to simulate.

We also require *initial conditions* for the state variables:

$$\mathbf{x}(t = t_0) = \mathbf{x_0}$$

Taylor Series Expansion

The model can be simulated using a *Taylor series expansion*. If we know the state vector at a certain instant of time, t^*, the state vector can be calculated at some later time instant, $t^* + h$ by means of a Taylor series expansion:

$$x_i(t^* + h) = x_i(t^*) + \frac{dx_i(t^*)}{dt} \cdot h + \frac{d^2x_i(t^*)}{dt^2} \cdot \frac{h^2}{2!} + \dots$$

The state-space model is used to compute the first derivative in the Taylor series:

$$x_i(t^* + h) = x_i(t^*) + f_i(t^*) \cdot h + \frac{df_i(t^*)}{dt} \cdot \frac{h^2}{2!} + \dots$$

The different numerical integration methods differ in their numerical approximations of the derivatives of f.

The Truncation Error

Evidently, it is impossible to consider all terms of the Taylor series expansion. All numerical integration methods only approximate a certain number of terms of the Taylor series. This number can be either fixed or variable.

We talk about the *approximation order* of the numerical method. An algorithm that approximates the terms of the Taylor series up to the third derivative:

$$x_i(t^* + h) = x_i(t^*) + f_i(t^*) \cdot h + \frac{df_i(t^*)}{dt} \cdot \frac{h^2}{2!} + \frac{d^2 f_i(t^*)}{dt^2} \cdot \frac{h^3}{3!} + o(h^4)$$

is thus an *algorithm of third-order*.

The *truncation error* of the method grows proportionally with the fourth power of the *integration step size*, h.

The Roundoff Error

There exists a second type of error that results from the finite mantissa of the computer. The effects of this type of error can easily be illustrated graphically:

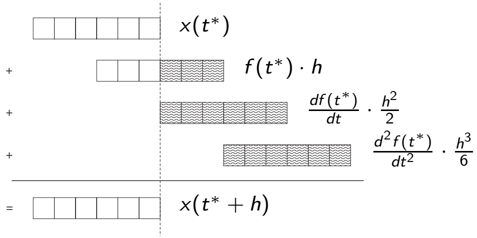
$x(t^*)$
$+$
$f(t^*) \cdot h$
$+$
$\frac{df(t^*)}{dt} \cdot \frac{h^2}{2}$
$+$
$\frac{d^2f(t^*)}{dt^2} \cdot \frac{h^3}{6}$
$=$
$x(t^* + h)$

Figure: Effects of the *roundoff error* in single precision

The Roundoff Error II

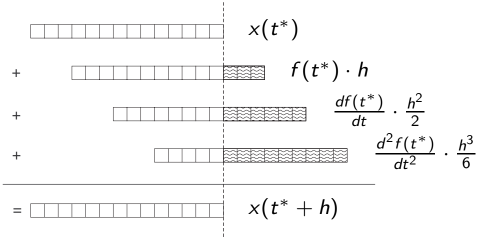
$x(t^*)$
$+$
$f(t^*) \cdot h$
$+$
$\frac{df(t^*)}{dt} \cdot \frac{h^2}{2}$
$+$
$\frac{d^2f(t^*)}{dt^2} \cdot \frac{h^3}{6}$
$=$
$x(t^* + h)$

Figure: Effects of the *roundoff error* in double precision

The Roundoff Error III

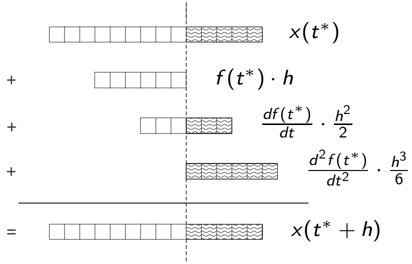
$x(t^*)$
$+$
$f(t^*) \cdot h$
$+$
$\frac{df(t^*)}{dt} \cdot \frac{h^2}{2}$
$+$
$\frac{d^2f(t^*)}{dt^2} \cdot \frac{h^3}{6}$
$=$
$x(t^* + h)$

Figure: Effects of the *roundoff error* in 1.5-fold precision

The Explicit Euler Integration

The most simple numerical ODE solver is based on the explicit so-called *"Forward Euler" (FE)* formula, a first-order integration method:

$$\begin{aligned} & \mathbf{x}(t^* + h) \approx \mathbf{x}(t^*) + \dot{\mathbf{x}}(t^*) \cdot h \\ \Rightarrow \qquad & \mathbf{x}(t^* + h) \approx \mathbf{x}(t^*) + \mathbf{f}(\mathbf{x}(t^*), t^*) \cdot h \end{aligned}$$

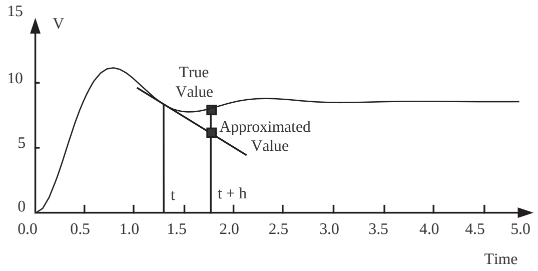

Figure: Numerical integration using the "FE" method

The Explicit Euler Integration II

When using *explicit integration methods*, the simulation doesn't require any *iteration* within an integration step, unless the model contains *algebraic loops*:

$$\begin{array}{lll}
\text{step 1a:} & \dot{\mathbf{x}}(t_0) & = \mathbf{f}(\mathbf{x}(t_0), t_0) \\
\text{step 1b:} & \mathbf{x}(t_0 + h) & = \mathbf{x}(t_0) \; + \; h \cdot \dot{\mathbf{x}}(t_0) \\
& & \\
\text{step 2a:} & \dot{\mathbf{x}}(t_0 + h) & = \mathbf{f}(\mathbf{x}(t_0 + h), t_0 + h) \\
\text{step 2b:} & \mathbf{x}(t_0 + 2h) & = \mathbf{x}(t_0 + h) \; + \; h \cdot \dot{\mathbf{x}}(t_0 + h) \\
& & \\
\text{step 3a:} & \dot{\mathbf{x}}(t_0 + 2h) & = \mathbf{f}(\mathbf{x}(t_0 + 2h), t_0 + 2h) \\
\text{step 3b:} & \mathbf{x}(t_0 + 3h) & = \mathbf{x}(t_0 + 2h) \; + \; h \cdot \dot{\mathbf{x}}(t_0 + 2h)
\end{array}$$

etc.

The Implicit Euler Integration

Another numerical integration method of first order is the *"Backward Euler" (BE)* method:

$$\mathbf{x}(t^* + h) \approx \mathbf{x}(t^*) + \mathbf{f}(\mathbf{x}(t^* + h), t^* + h) \cdot h$$

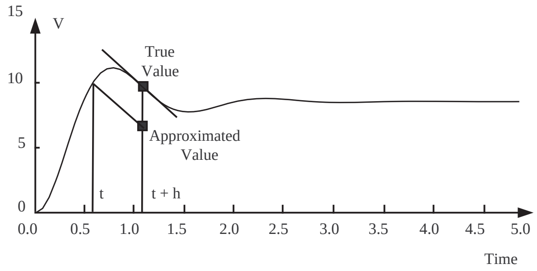

Figure: Numerical integration using the "BE" method

The Numerical Stability Domain

A *linear autonomous time-invariant* system can be represented using the model:

$$\dot{\mathbf{x}} = \mathbf{A} \cdot \mathbf{x} \quad ; \quad \mathbf{x}(t = t_0) = \mathbf{x_0}$$

with the analytical solution:

$$\mathbf{x}(t) = \exp(\mathbf{A} \cdot t) \cdot \mathbf{x_0}$$

The solution is *analytically stable* if:

$$\mathbb{R}\mathrm{e}\{\mathrm{Eig}(\mathbf{A})\} = \mathbb{R}\mathrm{e}\{\lambda\} < 0.0$$

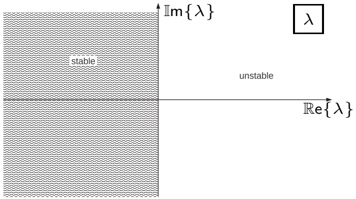
$\mathbb{I}\mathrm{m}\{\lambda\}$
λ
stable
unstable
$\mathbb{R}\mathrm{e}\{\lambda\}$

Figure: The region of *analytical stability*

The Numerical Stability Domain II

When we use the *FE algorithm*:

$$\begin{aligned} & \mathbf{x}(t^* + h) = \mathbf{x}(t^*) + \mathbf{f}(\mathbf{x}(t^*), t^*) \cdot h \\ \Rightarrow \quad & \mathbf{x}(t^* + h) = \mathbf{x}(t^*) + \mathbf{A} \cdot h \cdot \mathbf{x}(t^*) \\ \Rightarrow \quad & \mathbf{x}(k+1) = [\mathbf{I^{(n)}} + \mathbf{A} \cdot h] \cdot \mathbf{x}(k) \end{aligned}$$

Therefore:

$$\mathbf{x_{k+1}} = \mathbf{F} \cdot \mathbf{x_k}$$

with:

$$\mathbf{F} = \mathbf{I^{(n)}} + \mathbf{A} \cdot h$$

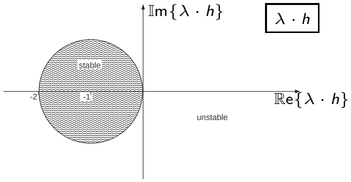
$\mathbb{I}\mathrm{m}\{\lambda \cdot h\}$
$\lambda \cdot h$
stable
-2
-1
$\mathbb{R}\mathrm{e}\{\lambda \cdot h\}$
unstable

Figure: The *numerical stability domain* of the *FE algorithm*

Simulation With the FE Algorithm

When simulating the linear scalar system:

$$\dot{x} = a \cdot x \quad ; \quad x(t = t_0) = x_0$$

using the FE algorithm, we obtain:

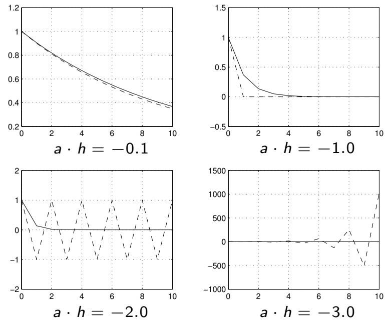

Figure: Simulation of a linear scalar system using the FE algorithm

Computation of the Largest Numerically Stable Integration Step Size for FE

Given a linear system of second order with two complex eigenvalues, λ_1 and λ_2:

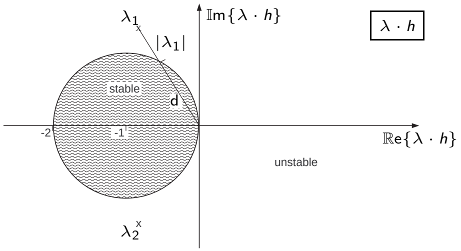
λ_1
$|\lambda_1|$
$\mathbb{I}\text{m}\{\lambda \cdot h\}$
$\lambda \cdot h$
stable
d
-2
-1
$\mathbb{R}\text{e}\{\lambda \cdot h\}$
unstable
λ_2

Figure: Largest numerically stable integration step size for FE

$$\Rightarrow \quad h_{max} = \frac{d}{|\lambda_1|}$$

The Numerical Stability Domain III

When we use the *BE algorithm*:

$$\begin{aligned} & \mathbf{x}(t^* + h) = \mathbf{x}(t^*) + \mathbf{A} \cdot h \cdot \mathbf{x}(t^* + h) \\ \Rightarrow \quad & [\mathbf{I}^{(\mathbf{n})} - \mathbf{A} \cdot h] \cdot \mathbf{x}(t^* + h) = \mathbf{x}(t^*) \\ \Rightarrow \quad & \mathbf{x}(k+1) = [\mathbf{I}^{(\mathbf{n})} - \mathbf{A} \cdot h]^{-1} \cdot \mathbf{x}(k) \end{aligned}$$

Therefore:

$$\mathbf{F} = [\mathbf{I}^{(\mathbf{n})} - \mathbf{A} \cdot h]^{-1}$$

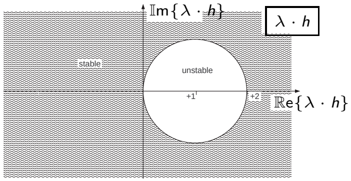
$\mathbb{I}\mathrm{m}\{\lambda \cdot h\}$
$\lambda \cdot h$
stable
unstable
+1
+2
$\mathbb{R}\mathrm{e}\{\lambda \cdot h\}$

Figure: The *numerical stability domain* of the *BE algorithm*

Simulation With the BE Algorithm

When simulating the linear scalar system:

$$\dot{x} = a \cdot x \quad ; \quad x(t = t_0) = x_0$$

using the BE algorithm, we obtain:

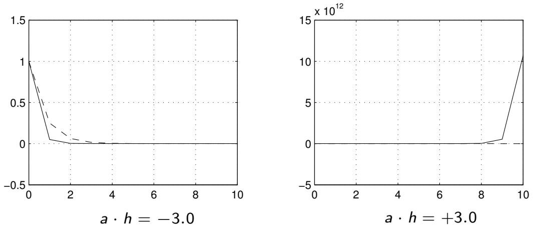

Figure: Simulation of a linear scalar system using the BE algorithm

Numerical Stability Domain Computation

How is the numerical stability domain computed?

We start out with a second-order system with a conjugate complex pair of eigenvalues anywhere on the unit circle. The system with the **A**-matrix:

$$\mathbf{A} = \begin{pmatrix} 0 & 1 \\ -1 & 2\cos(\alpha) \end{pmatrix}$$

has a pair of conjugate complex eigenvalues located on the unit circle. α denotes the angle of one of the two eigenvalues counted in the mathematically positive (i.e., counterclockwise) sense away from the positive real axis.

In Matlab:

```
function [A] = aa (alpha)
    radalpha = alpha * pi/180;
    x = cos(radalpha);
    A = [ 0 , 1 ; -1 , 2 * x ];
return
```

Numerical Stability Domain Computation II

We now compute the **F**-matrix:

```
function [F] = ff(A, h, algor)
    Ah = A * h;
    [n, n] = size(Ah);
    I = eye(n);
    %
    % algor = 1 : Forward Euler
    %
    if algor == 1,
        F = I + Ah;
    end
    %
    % algor = 2 : Backward Euler
    %
    if algor == 2,
        F = inv(I - Ah);
    end
return
```

Numerical Stability Domain Computation III

We now compute the largest possible value of *h*, for which all eigenvalues of **F** are inside the unit circle:

```
function [hmax] = hh(alpha, algor, hlower, hupper)
    A = aa(alpha);
    maxerr = 1.0e-6;
    err = 100;
    while err > maxerr,
        h = (hlower + hupper)/2;
        F = ff(A, h, algor);
        lmax = max(abs(eig(F)));
        err = lmax - 1;
        if err > 0,
            hupper = h;
        else
            hlower = h;
        end,
        err = abs(err);
    end
    hmax = h;
return
```

The hh-function, as shown above, works only for algorithms with stability domains similar to that of the FE algorithm. The logic of the **if**-statement must be reversed for algorithms of the BE type.

Numerical Stability Domain Computation IV

Finally, we need to sweep over a selected range of α values, and plot h_{max} as a function of α in polar coordinates.

There certainly exist more efficient curve tracking algorithms than the one outlined above, but for the time being, this algorithm will suffice.

Fixed-point Iteration

When using implicit numerical integration algorithms, we need to iterate on the solution during each step.

One possible approach to iterating on the solution is to start with a *prediction* followed by many *corrections*.

prediction:
$$\begin{aligned} \dot{\mathbf{x}}_{\mathbf{k}} &= \mathbf{f}(\mathbf{x}_{\mathbf{k}}, t_k) \\ \mathbf{x}^{\mathbf{P}}_{\mathbf{k+1}} &= \mathbf{x}_{\mathbf{k}} + h \cdot \dot{\mathbf{x}}_{\mathbf{k}} \end{aligned}$$

1st correction:
$$\begin{aligned} \dot{\mathbf{x}}^{\mathbf{P}}_{\mathbf{k+1}} &= \mathbf{f}(\mathbf{x}^{\mathbf{P}}_{\mathbf{k+1}}, t_{k+1}) \\ \mathbf{x}^{\mathbf{C1}}_{\mathbf{k+1}} &= \mathbf{x}_{\mathbf{k}} + h \cdot \dot{\mathbf{x}}^{\mathbf{P}}_{\mathbf{k+1}} \end{aligned}$$

2nd correction:
$$\begin{aligned} \dot{\mathbf{x}}^{\mathbf{C1}}_{\mathbf{k+1}} &= \mathbf{f}(\mathbf{x}^{\mathbf{C1}}_{\mathbf{k+1}}, t_{k+1}) \\ \mathbf{x}^{\mathbf{C2}}_{\mathbf{k+1}} &= \mathbf{x}_{\mathbf{k}} + h \cdot \dot{\mathbf{x}}^{\mathbf{C1}}_{\mathbf{k+1}} \end{aligned}$$

3rd correction:
$$\begin{aligned} \dot{\mathbf{x}}^{\mathbf{C2}}_{\mathbf{k+1}} &= \mathbf{f}(\mathbf{x}^{\mathbf{C2}}_{\mathbf{k+1}}, t_{k+1}) \\ \mathbf{x}^{\mathbf{C3}}_{\mathbf{k+1}} &= \mathbf{x}_{\mathbf{k}} + h \cdot \dot{\mathbf{x}}^{\mathbf{C2}}_{\mathbf{k+1}} \end{aligned}$$

etc.

This type of iteration method is called *fixed-point iteration*.

Fixed-point Iteration II

When we apply fixed-point iteration to the linear system, we obtain:

$$\begin{aligned}
\mathbf{F}^{\mathbf{P}} &= \mathbf{I}^{(\mathbf{n})} + \mathbf{A}\cdot h \\
\mathbf{F}^{\mathbf{C1}} &= \mathbf{I}^{(\mathbf{n})} + \mathbf{A}\cdot h + (\mathbf{A}\cdot h)^2 \\
\mathbf{F}^{\mathbf{C2}} &= \mathbf{I}^{(\mathbf{n})} + \mathbf{A}\cdot h + (\mathbf{A}\cdot h)^2 + (\mathbf{A}\cdot h)^3 \\
\mathbf{F}^{\mathbf{C3}} &= \mathbf{I}^{(\mathbf{n})} + \mathbf{A}\cdot h + (\mathbf{A}\cdot h)^2 + (\mathbf{A}\cdot h)^3 + (\mathbf{A}\cdot h)^4
\end{aligned}$$

After an infinitely large number of iterations:

$$\mathbf{F} = \mathbf{I}^{(\mathbf{n})} + \mathbf{A}\cdot h + (\mathbf{A}\cdot h)^2 + (\mathbf{A}\cdot h)^3 + \dots$$

Therefore:

$$(\mathbf{A}\cdot h)\cdot\mathbf{F} = \mathbf{A}\cdot h + (\mathbf{A}\cdot h)^2 + (\mathbf{A}\cdot h)^3 + (\mathbf{A}\cdot h)^4 + \dots$$

Subtracting one from the other:

$$[\mathbf{I}^{(\mathbf{n})} - \mathbf{A}\cdot h]\cdot\mathbf{F} = \mathbf{I}^{(\mathbf{n})}$$

we obtain:

$$\mathbf{F} = [\mathbf{I}^{(\mathbf{n})} - \mathbf{A}\cdot h]^{-1}$$

Seemingly we obtain the same **F** matrix as in the case of the BE algorithm.

Fixed-point Iteration III

Let us draw the numerical stability domain of this algorithm:

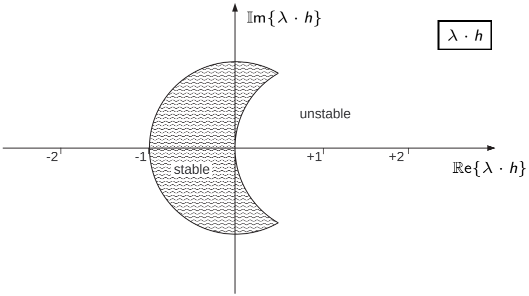
$\mathbb{I}m\{\lambda \cdot h\}$
$\lambda \cdot h$
unstable
-2
-1
+1
+2
$\mathbb{R}e\{\lambda \cdot h\}$
stable

Figure: Numerical stability domain of *predictor-corrector FE-BE technique*

Fixed-point Iteration III

Let us draw the numerical stability domain of this algorithm:

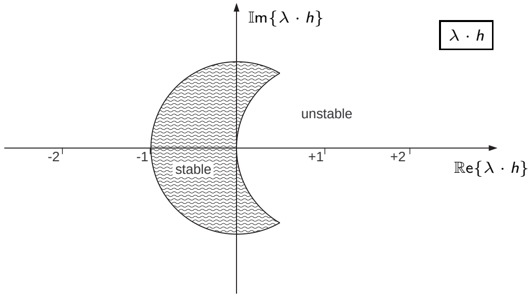
$\mathbb{I}\text{m}\{\lambda \cdot h\}$
$\boxed{\lambda \cdot h}$
unstable
-2
-1
+1
+2
stable
$\mathbb{R}\text{e}\{\lambda \cdot h\}$

Figure: Numerical stability domain of *predictor-corrector FE-BE technique*

This evidently didn't work very well.

Fixed-point Iteration IV

What Went Wrong?

The approach didn't work, because the infinite series:

$$\mathbf{F} = \mathbf{I}^{(\mathbf{n})} + \mathbf{A} \cdot h + (\mathbf{A} \cdot h)^2 + (\mathbf{A} \cdot h)^3 + \ldots$$

only converges, if all of the eigenvalues of $\mathbf{A} \cdot h$ lie inside the *unit circle*. If this is not the case, the subtraction is invalid.

Inside the unit circle, the numerical stability domain of the predictor-corrector method is identical to that of the BE algorithm, but outside the unit circle, the method is unstable everywhere.

For this reason, the *fixed-point iteration method* is useless.

Newton Iteration

Newton iteration can be used to determine the zero-crossings of a function:

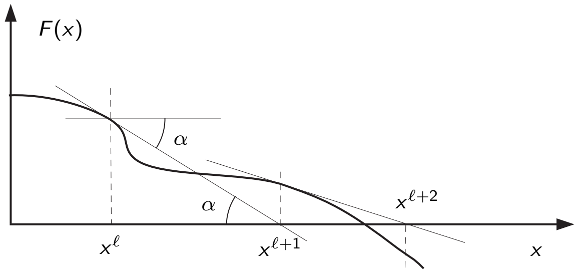

Figure: Newton iteration on a single zero-crossing function

$$\tan\alpha = \frac{\partial\mathcal{F}^{\ell}}{\partial x} = \frac{\mathcal{F}^{\ell}}{x^{\ell} - x^{\ell+1}} \quad \Rightarrow \quad x^{\ell+1} = x^{\ell} - \frac{\mathcal{F}^{\ell}}{\partial\mathcal{F}^{\ell}/\partial x}$$

Newton Iteration II

The BE algorithm applied to a scalar differential equation can be formulated as follows:

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

Therefore:

$$\mathcal{F}(x_{k+1}) = x_k + h \cdot f(x_{k+1}, t_{k+1}) - x_{k+1} = 0.0$$

Now, Newton iteration can be applied:

$$x_{k+1}^{\ell+1} = x_{k+1}^{\ell} - \frac{x_k + h \cdot f(x_{k+1}^{\ell}, t_{k+1}) - x_{k+1}^{\ell}}{h \cdot \partial f(x_{k+1}^{\ell}, t_{k+1})/\partial x - 1.0}$$

Newton Iteration III

In the case of a state vector, we can write:

$$\mathbf{x}^{\ell+1} = \mathbf{x}^{\ell} - \left(\mathcal{H}^{\ell}\right)^{-1} \cdot \mathcal{F}^{\ell}$$

where:

$$\mathcal{H} = \frac{\partial \mathcal{F}}{\partial \mathbf{x}} = \begin{pmatrix} \partial\mathcal{F}_1/\partial x_1 & \partial\mathcal{F}_1/\partial x_2 & \dots & \partial\mathcal{F}_1/\partial x_n \\ \partial\mathcal{F}_2/\partial x_1 & \partial\mathcal{F}_2/\partial x_2 & \dots & \partial\mathcal{F}_2/\partial x_n \\ \vdots & \vdots & \ddots & \vdots \\ \partial\mathcal{F}_n/\partial x_1 & \partial\mathcal{F}_n/\partial x_2 & \dots & \partial\mathcal{F}_n/\partial x_n \end{pmatrix}$$

is the *Hessian matrix* of the Newton iteration.

Newton Iteration IV

We can apply the Hessian matrix to the BE algorithm:

$$\mathbf{x}_{\mathbf{k+1}}^{\ell+\mathbf{1}} = \mathbf{x}_{\mathbf{k+1}}^{\ell} - [h \cdot \mathcal{J}_{\mathbf{k+1}}^{\ell} - \mathbf{I}^{(\mathbf{n})}]^{-1} \cdot [\mathbf{x_k} + h \cdot \mathbf{f}(\mathbf{x}_{\mathbf{k+1}}^{\ell}, t_{k+1}) - \mathbf{x}_{\mathbf{k+1}}^{\ell}]$$

where:

$$\mathcal{J} = \frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \begin{pmatrix} \partial f_1/\partial x_1 & \partial f_1/\partial x_2 & \dots & \partial f_1/\partial x_n \\ \partial f_2/\partial x_1 & \partial f_2/\partial x_2 & \dots & \partial f_2/\partial x_n \\ \vdots & \vdots & \ddots & \vdots \\ \partial f_n/\partial x_1 & \partial f_n/\partial x_2 & \dots & \partial f_n/\partial x_n \end{pmatrix}$$

is the *Jacobian matrix* of the dynamic system.

Newton Iteration V

If the system is linear:

$$\mathcal{J} = \mathbf{A}$$

Therefore:

$$\begin{aligned} & \mathbf{x}_{\mathbf{k+1}}^{\ell+1} = \mathbf{x}_{\mathbf{k+1}}^{\ell} - [\mathbf{A} \cdot h - \mathbf{I}^{(\mathbf{n})}]^{-1} \cdot [(\mathbf{A} \cdot h - \mathbf{I}^{(\mathbf{n})}) \cdot \mathbf{x}_{\mathbf{k+1}}^{\ell} + \mathbf{x}_{\mathbf{k}}] \\ \Rightarrow \quad & \mathbf{x}_{\mathbf{k+1}}^{\ell+1} = [\mathbf{I}^{(\mathbf{n})} - \mathbf{A} \cdot h]^{-1} \cdot \mathbf{x}_{\mathbf{k}} \end{aligned}$$

Newton iteration does not ever change the numerical stability domain of an ODE solver. This is true not only for the BE algorithm, but rather for all numerical ODE solvers.

Conclusions

- In the analysis of numerical ODE solvers, the *numerical stability of the algorithm* must always be taken into consideration.

Conclusions

- In the analysis of numerical ODE solvers, the *numerical stability of the algorithm* must always be taken into consideration.
- The numerical stability of most ODE solvers can be represented by a *numerical stability domain* drawn in the complex $\boxed{\lambda \cdot h}$ plane.

Conclusions

- In the analysis of numerical ODE solvers, the *numerical stability of the algorithm* must always be taken into consideration.
- The numerical stability of most ODE solvers can be represented by a *numerical stability domain* drawn in the complex $\boxed{\lambda \cdot h}$ plane.
- The numerical stability of ODE solvers is usually analyzed for *linear autonomous time-invariant* systems only.

Conclusions

- In the analysis of numerical ODE solvers, the *numerical stability of the algorithm* must always be taken into consideration.
- The numerical stability of most ODE solvers can be represented by a *numerical stability domain* drawn in the complex $\boxed{\lambda \cdot h}$ plane.
- The numerical stability of ODE solvers is usually analyzed for *linear autonomous time-invariant* systems only.
- There exists also a *theory of non-linear stability*, but this theory is quite involved, and it is usually not necessary to use it, because the numerical stability of a linearized system is the same as that of the original non-linear system.

Conclusions II

- In the analysis of numerical ODE solvers, it is also important to consider the *approximation accuracy of the algorithm*.

Conclusions II

- In the analysis of numerical ODE solvers, it is also important to consider the *approximation accuracy of the algorithm*.
- The numerical accuracy of an ODE solver is subject to a number of error types, such as the *truncation error*, the *roundoff error*, and the *accumulation error*.

Conclusions II

- In the analysis of numerical ODE solvers, it is also important to consider the *approximation accuracy of the algorithm*.
- The numerical accuracy of an ODE solver is subject to a number of error types, such as the *truncation error*, the *roundoff error*, and the *accumulation error*.
- Most important among these error types is the *truncation error* that is characterized by the *order of approximation accuracy* of the solver.

Conclusions II

- In the analysis of numerical ODE solvers, it is also important to consider the *approximation accuracy of the algorithm*.
- The numerical accuracy of an ODE solver is subject to a number of error types, such as the *truncation error*, the *roundoff error*, and the *accumulation error*.
- Most important among these error types is the *truncation error* that is characterized by the *order of approximation accuracy* of the solver.
- It is important to analyze the order of approximation accuracy also for *non-linear* and *multi-state* systems, because it can happen that the order of approximation accuracy is higher for linear than for non-linear systems and possibly also higher for scalar than for multi-state systems.