Numerical Simulation of Dynamic Systems IV

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

March 5, 2013

Stiff Systems

- We call a linear system *stiff*, if it is stable and its eigenvalues vary a lot in terms of their real parts.

Stiff Systems

- We call a linear system *stiff*, if it is stable and its eigenvalues vary a lot in terms of their real parts.
- Non-linear systems are called stiff, if they are stable and exhibit both fast and slow modes in their behavior. The linearization of such systems leads to stiff linear systems.

Stiff Systems

- We call a linear system *stiff*, if it is stable and its eigenvalues vary a lot in terms of their real parts.
- Non-linear systems are called stiff, if they are stable and exhibit both fast and slow modes in their behavior. The linearization of such systems leads to stiff linear systems.
- These systems cannot be simulated efficiently by means of any explicit RK algorithm, because we would need very small time steps to move those eigenvalues (of either the system itself or of its linearization) that are located most to the left in the complex $\lambda \cdot h$ plane into the numerical stability domain.

Stiff Systems

- We call a linear system *stiff*, if it is stable and its eigenvalues vary a lot in terms of their real parts.
- Non-linear systems are called stiff, if they are stable and exhibit both fast and slow modes in their behavior. The linearization of such systems leads to stiff linear systems.
- These systems cannot be simulated efficiently by means of any explicit RK algorithm, because we would need very small time steps to move those eigenvalues (of either the system itself or of its linearization) that are located most to the left in the complex $\lambda \cdot h$ plane into the numerical stability domain.
- In fact, this is not only a problem of the explicit RK algorithm, but of all explicit numerical ODE solvers in general.

Stiff Systems

- We call a linear system *stiff*, if it is stable and its eigenvalues vary a lot in terms of their real parts.
- Non-linear systems are called stiff, if they are stable and exhibit both fast and slow modes in their behavior. The linearization of such systems leads to stiff linear systems.
- These systems cannot be simulated efficiently by means of any explicit RK algorithm, because we would need very small time steps to move those eigenvalues (of either the system itself or of its linearization) that are located most to the left in the complex $\lambda \cdot h$ plane into the numerical stability domain.
- In fact, this is not only a problem of the explicit RK algorithm, but of all explicit numerical ODE solvers in general.
- **For the efficient simulation of stiff systems, implicit integration algorithms are required.**

Richardson Extrapolation

One way to obtain higher-order algorithms is by combining various approximations of lower-order algorithms.

Let us start with the FE algorithm, repeating the step from t^* until $t^* + h$ four times using different step sizes: h, $h/2$, $h/3$, and $h/4$.

We apply this idea to the linear system, leading to four different *predictors*:

$$\begin{aligned}
\mathbf{x}^{\mathbf{P_1}}(k+1) &= [\mathbf{I}^{(\mathbf{n})} + \mathbf{A} \cdot h] \cdot \mathbf{x}(k) \\
\mathbf{x}^{\mathbf{P_2}}(k+1) &= [\mathbf{I}^{(\mathbf{n})} + \frac{\mathbf{A} \cdot h}{2}]^2 \cdot \mathbf{x}(k) \\
\mathbf{x}^{\mathbf{P_3}}(k+1) &= [\mathbf{I}^{(\mathbf{n})} + \frac{\mathbf{A} \cdot h}{3}]^3 \cdot \mathbf{x}(k) \\
\mathbf{x}^{\mathbf{P_4}}(k+1) &= [\mathbf{I}^{(\mathbf{n})} + \frac{\mathbf{A} \cdot h}{4}]^4 \cdot \mathbf{x}(k)
\end{aligned}$$

We now use the following parameterized *corrector formula*:

$$\mathbf{x}^{\mathbf{C}}(k+1) = \alpha_1 \cdot \mathbf{x}^{\mathbf{P_1}}(k+1) + \alpha_2 \cdot \mathbf{x}^{\mathbf{P_2}}(k+1) + \alpha_3 \cdot \mathbf{x}^{\mathbf{P_3}}(k+1) + \alpha_4 \cdot \mathbf{x}^{\mathbf{P_4}}(k+1)$$

Richardson Extrapolation II

Thus, we obtain:

$$\begin{aligned}\mathbf{x^C}(k+1) =&[(\alpha_1+\alpha_2+\alpha_3+\alpha_4)\cdot\mathbf{I^{(n)}}\\ &+(\alpha_1+\alpha_2+\alpha_3+\alpha_4)\cdot\mathbf{A}\cdot h\\ &+(\frac{\alpha_2}{4}+\frac{\alpha_3}{3}+\frac{3\alpha_4}{8})\cdot(\mathbf{A}\cdot h)^2\\ &+(\frac{\alpha_3}{27}+\frac{\alpha_4}{16})\cdot(\mathbf{A}\cdot h)^3+\frac{\alpha_4}{256}\cdot(\mathbf{A}\cdot h)^4]\cdot\mathbf{x_k}\end{aligned}$$

What we would like to obtain, is:

$$\mathbf{x^C}(k+1)=[\mathbf{I^{(n)}}+\mathbf{A}\cdot h+\frac{(\mathbf{A}\cdot h)^2}{2}+\frac{(\mathbf{A}\cdot h)^3}{6}+\frac{(\mathbf{A}\cdot h)^4}{24}]\cdot\mathbf{x_k}$$

By comparing the parameter values, we obtain four equations in four unknowns:

$$\begin{pmatrix}1 & 1 & 1 & 1\\ 0 & 1/4 & 1/3 & 3/8\\ 0 & 0 & 1/27 & 1/16\\ 0 & 0 & 0 & 1/256\end{pmatrix}\cdot\begin{pmatrix}\alpha_1\\ \alpha_2\\ \alpha_3\\ \alpha_4\end{pmatrix}=\begin{pmatrix}1\\ 1/2\\ 1/6\\ 1/24\end{pmatrix}$$

Therefore:

$$\alpha_1=-\frac{1}{6}\ ;\ \alpha_2=4\ ;\ \alpha_3=-\frac{27}{2}\ ;\ \alpha_4=\frac{32}{3}$$

Richardson Extrapolation III

We just encountered another way to implement an explicit RK4 algorithm, but:

Richardson Extrapolation III

We just encountered another way to implement an explicit RK4 algorithm, but:

- it isn't guaranteed that this algorithm will also be 4^{th}-order accurate when applied to non-linear systems, and

Richardson Extrapolation III

We just encountered another way to implement an explicit RK4 algorithm, but:

- it isn't guaranteed that this algorithm will also be 4^{th}-order accurate when applied to non-linear systems, and
- it isn't an efficient implementation of an RK4 algorithm, as it makes use of 10 stages instead of only 4.

Richardson Extrapolation IV

The approximation order of the algorithm wouldn't be all that important, if we could use very small step sizes.

We can write:

$$\mathbf{x_{k+1}}(\eta) = \mathbf{x_{k+1}} + \mathbf{e_1} \cdot \eta + \mathbf{e_2} \cdot \frac{\eta^2}{2!} + \mathbf{e_3} \cdot \frac{\eta^3}{3!} + \ldots$$

where $\mathbf{x_{k+1}}$ is the *true yet unknown value* of the state vector at time $t^* + h$, whereas $\mathbf{x_{k+1}}(\eta)$ is the *numerical approximation* of the same vector obtained using integration with a step size of η. We develop the numerical value into a Taylor series around the true value. The vectors $\mathbf{e_i}$ are error vectors.

We perform four separate experiments, using the same four step sizes as before:

$$\begin{aligned}
\mathbf{x}^{\mathbf{P_1}}(\eta_1) &\approx \mathbf{x_{k+1}} + \mathbf{e_1} \cdot h + \frac{\mathbf{e_2}}{2!} \cdot h^2 + \frac{\mathbf{e_3}}{3!} \cdot h^3 \\
\mathbf{x}^{\mathbf{P_2}}(\eta_2) &\approx \mathbf{x_{k+1}} + \mathbf{e_1} \cdot \frac{h}{2} + \frac{\mathbf{e_2}}{2!} \cdot (\frac{h}{2})^2 + \frac{\mathbf{e_3}}{3!} \cdot (\frac{h}{2})^3 \\
\mathbf{x}^{\mathbf{P_3}}(\eta_3) &\approx \mathbf{x_{k+1}} + \mathbf{e_1} \cdot \frac{h}{3} + \frac{\mathbf{e_2}}{2!} \cdot (\frac{h}{3})^2 + \frac{\mathbf{e_3}}{3!} \cdot (\frac{h}{3})^3 \\
\mathbf{x}^{\mathbf{P_4}}(\eta_4) &\approx \mathbf{x_{k+1}} + \mathbf{e_1} \cdot \frac{h}{4} + \frac{\mathbf{e_2}}{2!} \cdot (\frac{h}{4})^2 + \frac{\mathbf{e_3}}{3!} \cdot (\frac{h}{4})^3
\end{aligned}$$

Richardson Extrapolation V

In a matrix-vector form:

$$\begin{pmatrix} \mathbf{x}^{\mathbf{P}_1} \\ \mathbf{x}^{\mathbf{P}_2} \\ \mathbf{x}^{\mathbf{P}_3} \\ \mathbf{x}^{\mathbf{P}_4} \end{pmatrix} \approx \begin{pmatrix} h^0 & h^1 & h^2 & h^3 \\ (h/2)^0 & (h/2)^1 & (h/2)^2 & (h/2)^3 \\ (h/3)^0 & (h/3)^1 & (h/3)^2 & (h/3)^3 \\ (h/4)^0 & (h/4)^1 & (h/4)^2 & (h/4)^3 \end{pmatrix} \cdot \begin{pmatrix} \mathbf{x}_{\mathbf{k}+1} \\ \mathbf{e}_1 \\ \mathbf{e}_2/2 \\ \mathbf{e}_3/6 \end{pmatrix}$$

Solving for the unknown $\mathbf{x}_{\mathbf{k}+1}$, we obtain:

$$\mathbf{x}_{\mathbf{k}+1} \approx \begin{pmatrix} -\frac{1}{6} & 4 & -\frac{27}{2} & \frac{32}{3} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{x}^{\mathbf{P}_1} \\ \mathbf{x}^{\mathbf{P}_2} \\ \mathbf{x}^{\mathbf{P}_3} \\ \mathbf{x}^{\mathbf{P}_4} \end{pmatrix}$$

These are the same parameter values that we obtained before.

Richardson Extrapolation V

In a matrix-vector form:

$$\begin{pmatrix} \mathbf{x}^{\mathbf{P}_1} \\ \mathbf{x}^{\mathbf{P}_2} \\ \mathbf{x}^{\mathbf{P}_3} \\ \mathbf{x}^{\mathbf{P}_4} \end{pmatrix} \approx \begin{pmatrix} h^0 & h^1 & h^2 & h^3 \\ (h/2)^0 & (h/2)^1 & (h/2)^2 & (h/2)^3 \\ (h/3)^0 & (h/3)^1 & (h/3)^2 & (h/3)^3 \\ (h/4)^0 & (h/4)^1 & (h/4)^2 & (h/4)^3 \end{pmatrix} \cdot \begin{pmatrix} \mathbf{x}_{\mathbf{k}+1} \\ \mathbf{e}_1 \\ \mathbf{e}_2/2 \\ \mathbf{e}_3/6 \end{pmatrix}$$

Solving for the unknown $\mathbf{x}_{\mathbf{k}+1}$, we obtain:

$$\mathbf{x}_{\mathbf{k}+1} \approx \begin{pmatrix} -\frac{1}{6} & 4 & -\frac{27}{2} & \frac{32}{3} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{x}^{\mathbf{P}_1} \\ \mathbf{x}^{\mathbf{P}_2} \\ \mathbf{x}^{\mathbf{P}_3} \\ \mathbf{x}^{\mathbf{P}_4} \end{pmatrix}$$

These are the same parameter values that we obtained before.

Each of the two derivations has its advantages and disadvantages.

Richardson Extrapolation V

In a matrix-vector form:

$$\begin{pmatrix} \mathbf{x}^{P_1} \\ \mathbf{x}^{P_2} \\ \mathbf{x}^{P_3} \\ \mathbf{x}^{P_4} \end{pmatrix} \approx \begin{pmatrix} h^0 & h^1 & h^2 & h^3 \\ (h/2)^0 & (h/2)^1 & (h/2)^2 & (h/2)^3 \\ (h/3)^0 & (h/3)^1 & (h/3)^2 & (h/3)^3 \\ (h/4)^0 & (h/4)^1 & (h/4)^2 & (h/4)^3 \end{pmatrix} \cdot \begin{pmatrix} \mathbf{x}_{k+1} \\ \mathbf{e}_1 \\ \mathbf{e}_2/2 \\ \mathbf{e}_3/6 \end{pmatrix}$$

Solving for the unknown $\mathbf{x}_{k+1}$, we obtain:

$$\mathbf{x}_{k+1} \approx \begin{pmatrix} -\frac{1}{6} & 4 & -\frac{27}{2} & \frac{32}{3} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{x}^{P_1} \\ \mathbf{x}^{P_2} \\ \mathbf{x}^{P_3} \\ \mathbf{x}^{P_4} \end{pmatrix}$$

These are the same parameter values that we obtained before.

Each of the two derivations has its advantages and disadvantages.

- The former approach guaranteed that the resulting algorithm is indeed 4^{th}-order accurate (at least for linear systems). The latter technique doesn't offer that guarantee.

Richardson Extrapolation V

In a matrix-vector form:

$$\begin{pmatrix} \mathbf{x}^{P_1} \\ \mathbf{x}^{P_2} \\ \mathbf{x}^{P_3} \\ \mathbf{x}^{P_4} \end{pmatrix} \approx \begin{pmatrix} h^0 & h^1 & h^2 & h^3 \\ (h/2)^0 & (h/2)^1 & (h/2)^2 & (h/2)^3 \\ (h/3)^0 & (h/3)^1 & (h/3)^2 & (h/3)^3 \\ (h/4)^0 & (h/4)^1 & (h/4)^2 & (h/4)^3 \end{pmatrix} \cdot \begin{pmatrix} \mathbf{x}_{k+1} \\ \mathbf{e}_1 \\ \mathbf{e}_2/2 \\ \mathbf{e}_3/6 \end{pmatrix}$$

Solving for the unknown $\mathbf{x}_{k+1}$, we obtain:

$$\mathbf{x}_{k+1} \approx \begin{pmatrix} -\frac{1}{6} & 4 & -\frac{27}{2} & \frac{32}{3} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{x}^{P_1} \\ \mathbf{x}^{P_2} \\ \mathbf{x}^{P_3} \\ \mathbf{x}^{P_4} \end{pmatrix}$$

These are the same parameter values that we obtained before.

Each of the two derivations has its advantages and disadvantages.

- The former approach guaranteed that the resulting algorithm is indeed 4^{th}-order accurate (at least for linear systems). The latter technique doesn't offer that guarantee.
- On the other hand, the former approach was derived explicitly using the FE algorithm for its internal stages. The latter technique doesn't make this assumption. The same parameter values will result for *any* numerical ODE solver used for its internal stages.

The IEX4 Method

We can use Richardson extrapolation to develop an *implicit 4^{th}-order RK algorithm* by employing BE steps for its internal stages:

$$
\begin{array}{lll}
1^{\text{st}} \text{ predictor:} & \mathbf{k_1} & = \mathbf{x_k} + h \cdot \mathbf{f}(\mathbf{k_1}, t_{k+1}) \\
2^{\text{nd}} \text{ predictor:} & \mathbf{k_{2a}} & = \mathbf{x_k} + \frac{h}{2} \cdot \mathbf{f}(\mathbf{k_{2a}}, t_{k+\frac{1}{2}}) \\
 & \mathbf{k_2} & = \mathbf{k_{2a}} + \frac{h}{2} \cdot \mathbf{f}(\mathbf{k_2}, t_{k+1}) \\
3^{\text{rd}} \text{ predictor:} & \mathbf{k_{3a}} & = \mathbf{x_k} + \frac{h}{3} \cdot \mathbf{f}(\mathbf{k_{3a}}, t_{k+\frac{1}{3}}) \\
 & \mathbf{k_{3b}} & = \mathbf{k_{3a}} + \frac{h}{3} \cdot \mathbf{f}(\mathbf{k_{3b}}, t_{k+\frac{2}{3}}) \\
 & \mathbf{k_3} & = \mathbf{k_{3b}} + \frac{h}{3} \cdot \mathbf{f}(\mathbf{k_3}, t_{k+1}) \\
4^{\text{th}} \text{ predictor:} & \mathbf{k_{4a}} & = \mathbf{x_k} + \frac{h}{4} \cdot \mathbf{f}(\mathbf{k_{4a}}, t_{k+\frac{1}{4}}) \\
 & \mathbf{k_{4b}} & = \mathbf{k_{4a}} + \frac{h}{4} \cdot \mathbf{f}(\mathbf{k_{4b}}, t_{k+\frac{1}{2}}) \\
 & \mathbf{k_{4c}} & = \mathbf{k_{4b}} + \frac{h}{4} \cdot \mathbf{f}(\mathbf{k_{4c}}, t_{k+\frac{3}{4}}) \\
 & \mathbf{k_4} & = \mathbf{k_{4c}} + \frac{h}{4} \cdot \mathbf{f}(\mathbf{k_4}, t_{k+1}) \\
\text{corrector:} & \mathbf{x_{k+1}} & = -\frac{1}{6} \cdot \mathbf{k_1} + 4 \cdot \mathbf{k_2} - \frac{27}{2} \cdot \mathbf{k_3} + \frac{32}{3} \cdot \mathbf{k_4}
\end{array}
$$

We shall call this method IEX4.

The IEX4 Method II

Let us draw the *numerical stability domain of the IEX4 method*:

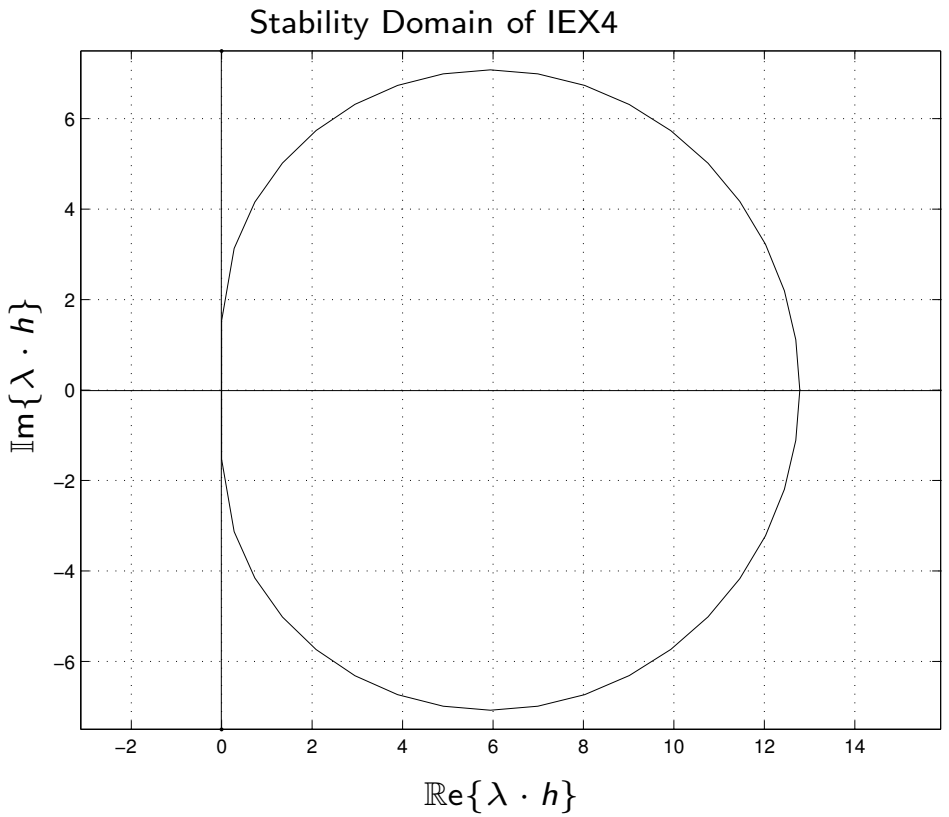

Figure: Numerical stability domain of IEX4 method

Marginally Stable Systems

If the *dominant eigenvalues* of a system are complex and are located *near the imaginary axis*, we'll run into problems with the simulation.

On the one hand, explicit algorithms require very small step sizes to include these eigenvalues in the numerical stability domain of the method. On the other hand, stiffly-stable implicit algorithms, like IEX4, will dampen the contributions of these eigenvalues out too fast.

What is needed are algorithms, whose numerical stability domain coincides with the analytical stability domain, i.e., whose border of numerical stability coincides with the imaginary axis.

We call such algorithms F-stable (faithfully stable).

Backinterpolation Methods

Let us consider once more the BE algorithm:

$$\mathbf{x_{k+1}} = \mathbf{x_k} + h \cdot \dot{\mathbf{x}}_{\mathbf{k+1}}$$

It is possible the reformulate the BE algorithm in the following way:

$$\mathbf{x_k} = \mathbf{x_{k+1}} - h \cdot \dot{\mathbf{x}}_{\mathbf{k+1}}$$

This means that using the BE algorithm with a step size of h is identical to using the FE algorithm with a step size of $-h$.

This idea brings about a new way of implementing implicit algorithms. We start with an estimate of the value $\mathbf{x_{k+1}}$ and integrate backward through time with a step size of $-h$, iterating on the value $\mathbf{x_k}$ using Newton iteration.

This is another way to obtain *higher-order implicit algorithms* based on the set of FRK algorithms. **We call these algorithms BRK.**

The BRK Algorithms

The BRK algorithms of orders 1..4 are characterized by:

$$\mathbf{F_1} = [\mathbf{I^{(n)}} - \mathbf{A} \cdot h]^{-1}$$

$$\mathbf{F_2} = [\mathbf{I^{(n)}} - \mathbf{A} \cdot h + \frac{(\mathbf{A} \cdot h)^2}{2!}]^{-1}$$

$$\mathbf{F_3} = [\mathbf{I^{(n)}} - \mathbf{A} \cdot h + \frac{(\mathbf{A} \cdot h)^2}{2!} - \frac{(\mathbf{A} \cdot h)^3}{3!}]^{-1}$$

$$\mathbf{F_4} = [\mathbf{I^{(n)}} - \mathbf{A} \cdot h + \frac{(\mathbf{A} \cdot h)^2}{2!} - \frac{(\mathbf{A} \cdot h)^3}{3!} + \frac{(\mathbf{A} \cdot h)^4}{4!}]^{-1}$$

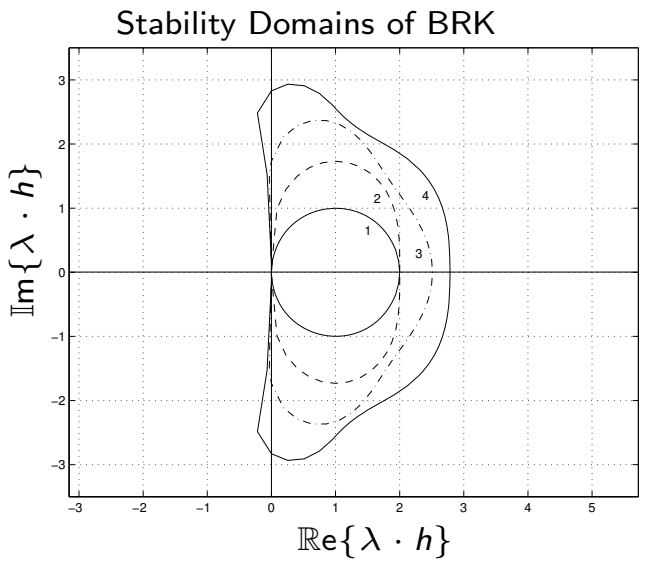

Figure: Numerical stability domains of BRK algorithms

The BRK Algorithms II

The BRK4 algorithm is usually considered superior to the IEX4 algorithm, because:

The BRK Algorithms II

The BRK4 algorithm is usually considered superior to the IEX4 algorithm, because:

- it only requires 4 stages instead of 10,

The BRK Algorithms II

The BRK4 algorithm is usually considered superior to the IEX4 algorithm, because:

- it only requires 4 stages instead of 10,
- it is guaranteed to preserve its order of approximation accuracy also in the case of non-linear systems, and

The BRK Algorithms II

The BRK4 algorithm is usually considered superior to the IEX4 algorithm, because:

- it only requires 4 stages instead of 10,
- it is guaranteed to preserve its order of approximation accuracy also in the case of non-linear systems, and
- it doesn't require the development of a new code, as any existing FRK algorithm can be used inside the Newton iteration.

The BRK Algorithms II

The BRK4 algorithm is usually considered superior to the IEX4 algorithm, because:

- it only requires 4 stages instead of 10,
- it is guaranteed to preserve its order of approximation accuracy also in the case of non-linear systems, and
- it doesn't require the development of a new code, as any existing FRK algorithm can be used inside the Newton iteration.

The IEX4 algorithm may still have its place in a multi-processor architecture, as the four predictor steps of the method can be easily parallelized and computed on four separate processors in parallel. In this way, we only need to compute four stages in series, and furthermore, each of these stages uses only a simple BE algorithm.

The BI Algorithms

The backinterpolation idea can also be implemented differently. To this end, we develop a *cyclic method* consisting in a *half-step of FRK* followed by a *half-step of BRK*. We iterate on the difference between the two values $\mathbf{x}_{\mathbf{k}+\frac{1}{2}}$:

$$\mathcal{F}(\mathbf{x}_{\mathbf{k}+1}) = \mathbf{x}^{\text{right}}_{\mathbf{k}+\frac{1}{2}} - \mathbf{x}^{\text{left}}_{\mathbf{k}+\frac{1}{2}} = 0.0$$

using Newton iteration.

The BI1 algorithm can be implemented in the following way:

$$1^{\text{st}} \text{ stage:} \quad \mathbf{x}_{\mathbf{k}+\frac{1}{2}} = \mathbf{x}_{\mathbf{k}} + \frac{h}{2} \cdot \dot{\mathbf{x}}_{\mathbf{k}}$$

$$2^{\text{nd}} \text{ stage:} \quad \mathbf{x}_{\mathbf{k}+1} = \mathbf{x}_{\mathbf{k}+\frac{1}{2}} + \frac{h}{2} \cdot \dot{\mathbf{x}}_{\mathbf{k}+1}$$

This algorithm is also known under the name *trapezoidal rule*. The method can be represented by:

$$\mathbf{F}_{\mathbf{TR}} = [\mathbf{I}^{(\mathbf{n})} - \mathbf{A} \cdot \frac{h}{2}]^{-1} \cdot [\mathbf{I}^{(\mathbf{n})} + \mathbf{A} \cdot \frac{h}{2}]$$

Just by chance, the *trapezoidal rule* happens to be 2^{nd}*-order accurate*.

The BI Algorithms II

In general, the BI algorithms can be characterized by:

$$\mathbf{F_{TR}} = [\mathbf{I^{(n)}} - \mathbf{A} \cdot \frac{h}{2}]^{-1} \cdot [\mathbf{I^{(n)}} + \mathbf{A} \cdot \frac{h}{2}]$$

$$\mathbf{F_{BI2}} = [\mathbf{I^{(n)}} - \mathbf{A} \cdot \frac{h}{2} + \frac{(\mathbf{A} \cdot h)^2}{8}]^{-1} \cdot [\mathbf{I^{(n)}} + \mathbf{A} \cdot \frac{h}{2} + \frac{(\mathbf{A} \cdot h)^2}{8}]$$

$$\mathbf{F_{BI3}} = [\mathbf{I^{(n)}} - \mathbf{A} \cdot \frac{h}{2} + \frac{(\mathbf{A} \cdot h)^2}{8} - \frac{(\mathbf{A} \cdot h)^3}{48}]^{-1} \cdot [\mathbf{I^{(n)}} + \mathbf{A} \cdot \frac{h}{2} + \frac{(\mathbf{A} \cdot h)^2}{8} + \frac{(\mathbf{A} \cdot h)^3}{48}]$$

$$\mathbf{F_{BI4}} = [\mathbf{I^{(n)}} - \mathbf{A} \cdot \frac{h}{2} + \frac{(\mathbf{A} \cdot h)^2}{8} - \frac{(\mathbf{A} \cdot h)^3}{48} + \frac{(\mathbf{A} \cdot h)^4}{384}]^{-1} \cdot [\mathbf{I^{(n)}} + \mathbf{A} \cdot \frac{h}{2} + \frac{(\mathbf{A} \cdot h)^2}{8} + \frac{(\mathbf{A} \cdot h)^3}{48} + \frac{(\mathbf{A} \cdot h)^4}{384}]$$

All of the BI algorithms are F-stable.

The BI2 algorithm isn't very useful, as already the TR algorithm, which is more economical, is 2^{nd}-order accurate.

Implementation of BI1 (TR) Algorithm

$$
\begin{aligned}
\mathbf{x}^{\text{left}}_{\mathbf{k}+\frac{1}{2}} &= \mathrm{FE}(\mathbf{x_k}, t_k, \tfrac{h}{2}) \\
\mathbf{x}^{0}_{\mathbf{k}+1} &= \mathbf{x}^{\text{left}}_{\mathbf{k}+\frac{1}{2}} \\
\\
\mathbf{J}^{0}_{\mathbf{k}+1} &= \mathcal{J}(\mathbf{x}^{0}_{\mathbf{k}+1}, t_{k+1}) \\
\mathbf{x}^{\text{right\_1}}_{\mathbf{k}+\frac{1}{2}} &= \mathrm{FE}(\mathbf{x}^{0}_{\mathbf{k}+1}, t_{k+1}, -\tfrac{h}{2}) \\
\mathbf{H}^{1} &= \mathbf{I}^{(n)} - \tfrac{h}{2} \cdot \mathbf{J}^{0}_{\mathbf{k}+1} \\
\mathbf{x}^{1}_{\mathbf{k}+1} &= \mathbf{x}^{0}_{\mathbf{k}+1} - \mathbf{H}^{1\,-1} \cdot (\mathbf{x}^{\text{right\_1}}_{\mathbf{k}+\frac{1}{2}} - \mathbf{x}^{\text{left}}_{\mathbf{k}+\frac{1}{2}}) \\
\varepsilon^{1}_{\mathbf{k}+1} &= \|\mathbf{x}^{1}_{\mathbf{k}+1} - \mathbf{x}^{0}_{\mathbf{k}+1}\|\infty \\
\\
\mathbf{J}^{1}_{\mathbf{k}+1} &= \mathcal{J}(\mathbf{x}^{1}_{\mathbf{k}+1}, t_{k+1}) \\
\mathbf{x}^{\text{right\_2}}_{\mathbf{k}+\frac{1}{2}} &= \mathrm{FE}(\mathbf{x}^{1}_{\mathbf{k}+1}, t_{k+1}, -\tfrac{h}{2}) \\
\mathbf{H}^{2} &= \mathbf{I}^{(n)} - \tfrac{h}{2} \cdot \mathbf{J}^{1}_{\mathbf{k}+1} \\
\mathbf{x}^{2}_{\mathbf{k}+1} &= \mathbf{x}^{1}_{\mathbf{k}+1} - \mathbf{H}^{2\,-1} \cdot (\mathbf{x}^{\text{right\_2}}_{\mathbf{k}+\frac{1}{2}} - \mathbf{x}^{\text{left}}_{\mathbf{k}+\frac{1}{2}}) \\
\varepsilon^{2}_{\mathbf{k}+1} &= \|\mathbf{x}^{2}_{\mathbf{k}+1} - \mathbf{x}^{1}_{\mathbf{k}+1}\|\infty \\
\\
&\text{etc.}
\end{aligned}
$$

where $\mathcal{J}$ denotes the Jacobian.

Implementation of BI2 Algorithm

$$
\begin{aligned}
\mathbf{x}_{\mathbf{k}+\frac{1}{2}}^{\text{left}} &= \text{Heun}(\mathbf{x}_{\mathbf{k}}, t_k, \tfrac{h}{2}) \\
\mathbf{x}_{\mathbf{k}+1}^{0} &= \mathbf{x}_{\mathbf{k}+\frac{1}{2}}^{\text{left}} \\
\\
\mathbf{J}_{\mathbf{k}+1}^{0} &= \mathcal{J}(\mathbf{x}_{\mathbf{k}+1}^{0}, t_{k+1}) \\
\mathbf{x}_{\mathbf{k}+\frac{1}{2}}^{\text{right\_1}} &= \text{Heun}(\mathbf{x}_{\mathbf{k}+1}^{0}, t_{k+1}, -\tfrac{h}{2}) \\
\mathbf{J}_{\mathbf{k}+\frac{1}{2}}^{0} &= \mathcal{J}(\mathbf{x}_{\mathbf{k}+\frac{1}{2}}^{\text{right\_1}}, t_{k+\frac{1}{2}}) \\
\mathbf{H}^{1} &= \mathbf{I}^{(\mathbf{n})} - \tfrac{h}{4} \cdot (\mathbf{J}_{\mathbf{k}+1}^{0} + \mathbf{J}_{\mathbf{k}+\frac{1}{2}}^{0} \cdot (\mathbf{I}^{(\mathbf{n})} - \tfrac{h}{2} \cdot \mathbf{J}_{\mathbf{k}+1}^{0})) \\
\mathbf{x}_{\mathbf{k}+1}^{1} &= \mathbf{x}_{\mathbf{k}+1}^{0} - \mathbf{H}^{1\,-1} \cdot (\mathbf{x}_{\mathbf{k}+\frac{1}{2}}^{\text{right\_1}} - \mathbf{x}_{\mathbf{k}+\frac{1}{2}}^{\text{left}}) \\
\varepsilon_{\mathbf{k}+1}^{1} &= \|\mathbf{x}_{\mathbf{k}+1}^{1} - \mathbf{x}_{\mathbf{k}+1}^{0}\|\infty \\
\\
\mathbf{J}_{\mathbf{k}+1}^{1} &= \mathcal{J}(\mathbf{x}_{\mathbf{k}+1}^{1}, t_{k+1}) \\
\mathbf{x}_{\mathbf{k}+\frac{1}{2}}^{\text{right\_2}} &= \text{Heun}(\mathbf{x}_{\mathbf{k}+1}^{1}, t_{k+1}, -\tfrac{h}{2}) \\
\mathbf{J}_{\mathbf{k}+\frac{1}{2}}^{1} &= \mathcal{J}(\mathbf{x}_{\mathbf{k}+\frac{1}{2}}^{\text{right\_2}}, t_{k+\frac{1}{2}}) \\
\mathbf{H}^{2} &= \mathbf{I}^{(\mathbf{n})} - \tfrac{h}{4} \cdot (\mathbf{J}_{\mathbf{k}+1}^{1} + \mathbf{J}_{\mathbf{k}+\frac{1}{2}}^{1} \cdot (\mathbf{I}^{(\mathbf{n})} - \tfrac{h}{2} \cdot \mathbf{J}_{\mathbf{k}+1}^{1})) \\
\mathbf{x}_{\mathbf{k}+1}^{2} &= \mathbf{x}_{\mathbf{k}+1}^{1} - \mathbf{H}^{2\,-1} \cdot (\mathbf{x}_{\mathbf{k}+\frac{1}{2}}^{\text{right\_2}} - \mathbf{x}_{\mathbf{k}+\frac{1}{2}}^{\text{left}}) \\
\varepsilon_{\mathbf{k}+1}^{2} &= \|\mathbf{x}_{\mathbf{k}+1}^{2} - \mathbf{x}_{\mathbf{k}+1}^{1}\|\infty
\end{aligned}
$$

etc.

Implementation of BI Algorithms

If we assume that the Jacobian remains basically unchanged during one integration step (*modified Newton iteration*), we can compute both the Jacobian and the Hessian at the beginning of the step, and we find for BI1:

$$\begin{aligned} \mathbf{J} &= \mathcal{J}(\mathbf{x_k}, t_k) \\ \mathbf{H} &= \mathbf{I}^{(\mathbf{n})} - \frac{h}{2} \cdot \mathbf{J} \end{aligned}$$

and for BI2:

$$\begin{aligned} \mathbf{J} &= \mathcal{J}(\mathbf{x_k}, t_k) \\ \mathbf{H} &= \mathbf{I}^{(\mathbf{n})} - \frac{h}{2} \cdot \mathbf{J} + \frac{h^2}{8} \cdot \mathbf{J}^2 \end{aligned}$$

Thus:

$$\begin{aligned} \mathbf{H_1} &= \mathbf{I}^{(\mathbf{n})} - \mathbf{J} \cdot \frac{h}{2} \\ \mathbf{H_2} &= \mathbf{I}^{(\mathbf{n})} - \mathbf{J} \cdot \frac{h}{2} + \frac{(\mathbf{J} \cdot h)^2}{8} \\ \mathbf{H_3} &= \mathbf{I}^{(\mathbf{n})} - \mathbf{J} \cdot \frac{h}{2} + \frac{(\mathbf{J} \cdot h)^2}{8} - \frac{(\mathbf{J} \cdot h)^3}{48} \\ \mathbf{H_4} &= \mathbf{I}^{(\mathbf{n})} - \mathbf{J} \cdot \frac{h}{2} + \frac{(\mathbf{J} \cdot h)^2}{8} - \frac{(\mathbf{J} \cdot h)^3}{48} + \frac{(\mathbf{J} \cdot h)^4}{384} \end{aligned}$$

One-legged Algorithms

The sequence in which we execute the forward and the backward semi-steps can be interchanged. The **F**-matrix of the interchanged BI1 algorithm is:

$$\mathbf{F_{MP}} = [\mathbf{I^{(n)}} + \mathbf{A} \cdot \frac{h}{2}] \cdot [\mathbf{I^{(n)}} - \mathbf{A} \cdot \frac{h}{2}]^{-1}$$

which corresponds to the algorithm:

$$\mathbf{x_{k+1}} = \mathbf{x_k} + h \cdot \dot{\mathbf{x}}_{\mathbf{k+\frac{1}{2}}}$$

which is the well-known *implicit midpoint rule*, the one-legged twin of the *trapezoidal rule*.

One-legged Algorithms

The sequence in which we execute the forward and the backward semi-steps can be interchanged. The **F**-matrix of the interchanged BI1 algorithm is:

$$\mathbf{F_{MP}} = [\mathbf{I^{(n)}} + \mathbf{A} \cdot \frac{h}{2}] \cdot [\mathbf{I^{(n)}} - \mathbf{A} \cdot \frac{h}{2}]^{-1}$$

which corresponds to the algorithm:

$$\mathbf{x_{k+1}} = \mathbf{x_k} + h \cdot \dot{\mathbf{x}}_{\mathbf{k+\frac{1}{2}}}$$

which is the well-known *implicit midpoint rule*, the one-legged twin of the *trapezoidal rule*.

In the same manner, it is possible to generate algorithms of higher orders as well. The two twins are identical in their linear properties, but they behave differently with respect to their non-linear characteristics. The original BI algorithms are a little *more accurate* than their one-legged twins, since we read out the value of the state at the end of the iteration rather than after the forward semi-step. On the other hand, the one-legged variety has somewhat better *non-linear stability* (*contractivity*) properties.

Tustin's Method

Given the linear autonomous time-invariant system:

$$\dot{\mathbf{x}} = \mathbf{A} \cdot \mathbf{x}$$

The analytical solution is:

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

Tustin's Method

Given the linear autonomous time-invariant system:

$$\dot{\mathbf{x}} = \mathbf{A} \cdot \mathbf{x}$$

The analytical solution is:

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

The analytical solution after one time step, h, is:

$$\mathbf{x_1} = \exp(\mathbf{A} \cdot h) \cdot \mathbf{x_0}$$

After two time steps:

$$\begin{aligned} \mathbf{x_2} &= \exp(\mathbf{A} \cdot 2h) \cdot \mathbf{x_0} \\ &= (\exp(\mathbf{A} \cdot h))^2 \cdot \mathbf{x_0} \\ &= \exp(\mathbf{A} \cdot h) \cdot \exp(\mathbf{A} \cdot h) \cdot \mathbf{x_0} \\ &= \exp(\mathbf{A} \cdot h) \cdot \mathbf{x_1} \end{aligned}$$

Tustin's Method II

Thus:

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

where:

$$\mathbf{F_{anal}} = \exp(\mathbf{A} \cdot h)$$

This method (only applicable to linear systems) is often referred to as *Tustin's method*.

For a linear time-invariant autonomous system, Tustin's method generates the *exact analytical solution* at the *sampling instants*.

Padé Approximations

Padé approximated numerically the $\mathbf{F}_{\mathbf{anal}}$ matrix. He applied the following technique.

$$\begin{aligned}\mathbf{F}_{\mathbf{anal}} &= \exp(\mathbf{A}\cdot h)\\ &= \exp(\mathbf{A}\frac{h}{2})\cdot\exp(\mathbf{A}\frac{h}{2})\\ &= [\exp(\mathbf{A}(-\frac{h}{2}))]^{-1}\cdot\exp(\mathbf{A}\frac{h}{2})\end{aligned}$$

He found polynomial approximations for the two expressions:

$$\mathbf{F}_{\mathbf{anal}} \approx \mathbf{D}(p,q)^{-1}\cdot\mathbf{N}(p,q)$$

where:

$$\begin{aligned}\mathbf{D}(p,q) &= \sum_{j=0}^{q}\frac{(p+q-j)!\ q!}{(p+q)!\ j!\ (q-j)!}\cdot(-\mathbf{A}h)^j\\ \mathbf{N}(p,q) &= \sum_{j=0}^{p}\frac{(p+q-j)!\ p!}{(p+q)!\ j!\ (p-j)!}\cdot(\mathbf{A}h)^j\end{aligned}$$

Padé Approximations II

For $p = q$, we obtain the algorithms:

$$\mathbf{F_{TR}} = [\mathbf{I^{(n)}} - \mathbf{A} \cdot \frac{h}{2}]^{-1} \cdot [\mathbf{I^{(n)}} + \mathbf{A} \cdot \frac{h}{2}]$$

$$\mathbf{F_{P4}} = [\mathbf{I^{(n)}} - \mathbf{A} \cdot \frac{h}{2} + \frac{(\mathbf{A} \cdot h)^2}{12}]^{-1} \cdot [\mathbf{I^{(n)}} + \mathbf{A} \cdot \frac{h}{2} + \frac{(\mathbf{A} \cdot h)^2}{12}]$$

$$\mathbf{F_{P6}} = [\mathbf{I^{(n)}} - \mathbf{A} \cdot \frac{h}{2} + \frac{(\mathbf{A} \cdot h)^2}{10} - \frac{(\mathbf{A} \cdot h)^3}{120}]^{-1} \cdot [\mathbf{I^{(n)}} + \mathbf{A} \cdot \frac{h}{2} + \frac{(\mathbf{A} \cdot h)^2}{10} + \frac{(\mathbf{A} \cdot h)^3}{120}]$$

$$\mathbf{F_{P8}} = [\mathbf{I^{(n)}} - \mathbf{A} \cdot \frac{h}{2} + \frac{3(\mathbf{A} \cdot h)^2}{28} - \frac{(\mathbf{A} \cdot h)^3}{84} + \frac{(\mathbf{A} \cdot h)^4}{1680}]^{-1} \cdot [\mathbf{I^{(n)}} + \mathbf{A} \cdot \frac{h}{2} + \frac{3(\mathbf{A} \cdot h)^2}{28} + \frac{(\mathbf{A} \cdot h)^3}{84} + \frac{(\mathbf{A} \cdot h)^4}{1680}]$$

The *Padé algorithms* are very similar to the *BI algorithms*. All of these algorithms are also F-stable. For the *same computational effort*, it is possible to *double the order of approximation accuracy* of this type of algorithm.

Padé Approximations III

One might be tempted to conclude that the Padé algorithms are superior to the BI algorithms due to their higher order of approximation accuracy for equal computational effort.

Unfortunately, this is not true. Whereas the BI4 algorithm is 4^{th}-order accurate for all systems, the P8 algorithm is 8^{th}-order accurate for linear systems, but only 2^{nd}-order accurate for non-linear systems.

For this reason, Padé algorithms (just like Tustin's algorithm) should be used for the simulation of linear systems only.

Algorithms for the Simulation of Stiff Systems

An eigenvalue:

$$\lambda = \sigma + j \cdot \omega$$

causes a trajectory with a *damping factor* of $-\sigma$ and a *oscillation frequency* of ω.

For this reason, the trajectory contributions resulting from eigenvalues located far to the left of the imaginary axis of the complex $\boxed{\lambda \cdot h}$ plane should rapidly decay.

Algorithms for the Simulation of Stiff Systems

An eigenvalue:

$$\lambda = \sigma + j \cdot \omega$$

causes a trajectory with a *damping factor* of $-\sigma$ and a *oscillation frequency* of ω.

For this reason, the trajectory contributions resulting from eigenvalues located far to the left of the imaginary axis of the complex $\boxed{\lambda \cdot h}$ plane should rapidly decay.

- Unfortunately, *F-stable algorithms* don't exhibit this behavior. These algorithms are marginally stable along the imaginary axis all the way to infinity, and therefore, they exhibit marginal stability everywhere at infinity, irrespective of the direction of approach.

Algorithms for the Simulation of Stiff Systems

An eigenvalue:

$$\lambda = \sigma + j \cdot \omega$$

causes a trajectory with a *damping factor* of $-\sigma$ and a *oscillation frequency* of ω.

For this reason, the trajectory contributions resulting from eigenvalues located far to the left of the imaginary axis of the complex $\boxed{\lambda \cdot h}$ plane should rapidly decay.

- Unfortunately, *F-stable algorithms* don't exhibit this behavior. These algorithms are marginally stable along the imaginary axis all the way to infinity, and therefore, they exhibit marginal stability everywhere at infinity, irrespective of the direction of approach.
- Instead of offering very large damping for eigenvalues far out to the left, they offer hardly any damping at all.

Algorithms for the Simulation of Stiff Systems

An eigenvalue:

$$\lambda = \sigma + j \cdot \omega$$

causes a trajectory with a *damping factor* of $-\sigma$ and a *oscillation frequency* of ω.

For this reason, the trajectory contributions resulting from eigenvalues located far to the left of the imaginary axis of the complex $\boxed{\lambda \cdot h}$ plane should rapidly decay.

- Unfortunately, *F-stable algorithms* don't exhibit this behavior. These algorithms are marginally stable along the imaginary axis all the way to infinity, and therefore, they exhibit marginal stability everywhere at infinity, irrespective of the direction of approach.
- Instead of offering very large damping for eigenvalues far out to the left, they offer hardly any damping at all.
- **For this reason, F-stable algorithms are not suitable for the simulation of stiff systems.**

Properties of Numerical Stability

Algorithms that work well for the simulation of stiff systems should have numerical stability domains similar to that of the BE algorithm, i.e., the stability domain should be closed in the right half complex $\lambda \cdot h$ plane.

Properties of Numerical Stability

Algorithms that work well for the simulation of stiff systems should have numerical stability domains similar to that of the BE algorithm, i.e., the stability domain should be closed in the right half complex $\lambda \cdot h$ plane.

- **An algorithm that contains the entire left half complex $\lambda \cdot h$ plane in its numerical stability domain is called A-stable (absolute stable).**

Properties of Numerical Stability

Algorithms that work well for the simulation of stiff systems should have numerical stability domains similar to that of the BE algorithm, i.e., the stability domain should be closed in the right half complex $\lambda \cdot h$ plane.

- **An algorithm that contains the entire left half complex $\lambda \cdot h$ plane in its numerical stability domain is called A-stable (absolute stable).**
- Algorithms that work well for the simulation of stiff systems should be A-stable. However, this property is only necessary, but not sufficient.

Properties of Numerical Stability

Algorithms that work well for the simulation of stiff systems should have numerical stability domains similar to that of the BE algorithm, i.e., the stability domain should be closed in the right half complex $\lambda \cdot h$ plane.

- **An algorithm that contains the entire left half complex $\lambda \cdot h$ plane in its numerical stability domain is called A-stable (absolute stable).**
- Algorithms that work well for the simulation of stiff systems should be A-stable. However, this property is only necessary, but not sufficient.
- **An algorithm that is A-stable and furthermore exhibits an infinitely large damping factor at infinity is called L-stable.**

Properties of Numerical Stability

Algorithms that work well for the simulation of stiff systems should have numerical stability domains similar to that of the BE algorithm, i.e., the stability domain should be closed in the right half complex $\lambda \cdot h$ plane.

- **An algorithm that contains the entire left half complex $\lambda \cdot h$ plane in its numerical stability domain is called A-stable (absolute stable).**
- Algorithms that work well for the simulation of stiff systems should be A-stable. However, this property is only necessary, but not sufficient.
- **An algorithm that is A-stable and furthermore exhibits an infinitely large damping factor at infinity is called L-stable.**
- F-stable algorithms are also A-stable, but they are never L-stable.

Properties of Numerical Stability

Algorithms that work well for the simulation of stiff systems should have numerical stability domains similar to that of the BE algorithm, i.e., the stability domain should be closed in the right half complex $\lambda \cdot h$ plane.

- **An algorithm that contains the entire left half complex $\lambda \cdot h$ plane in its numerical stability domain is called A-stable (absolute stable).**
- Algorithms that work well for the simulation of stiff systems should be A-stable. However, this property is only necessary, but not sufficient.
- **An algorithm that is A-stable and furthermore exhibits an infinitely large damping factor at infinity is called L-stable.**
- F-stable algorithms are also A-stable, but they are never L-stable.
- **We need L-stable algorithms for the simulation of stiff systems.**

BI Algorithms for the Simulation of Stiff Systems

It is possible to generate *algorithms of the BI class with numerical stability domains similar to that of the BE algorithm*.

Instead of computing the FRK semi-step all the way to the center followed by a BRK semi-step of equal length, we can reduce the length of the FRK semi-step to $\vartheta \cdot h$ with $\vartheta < 0.5$. The shortened FRK semi-step is then followed by an extended BRK semi-step of length $(1 - \vartheta) \cdot h$:

$$\mathbf{F_{BI1}}(\vartheta) = [\mathbf{I^{(n)}} - \mathbf{A}(1-\vartheta)h]^{-1} \cdot [\mathbf{I^{(n)}} + \mathbf{A}\vartheta h]$$

$$\mathbf{F_{BI2}}(\vartheta) = [\mathbf{I^{(n)}} - \mathbf{A}(1-\vartheta)h + \frac{(\mathbf{A}(1-\vartheta)h)^2}{2!}]^{-1} \cdot [\mathbf{I^{(n)}} + \mathbf{A}\vartheta h + \frac{(\mathbf{A}\vartheta h)^2}{2!}]$$

$$\mathbf{F_{BI3}}(\vartheta) = [\mathbf{I^{(n)}} - \mathbf{A}(1-\vartheta)h + \frac{(\mathbf{A}(1-\vartheta)h)^2}{2!} - \frac{(\mathbf{A}(1-\vartheta)h)^3}{3!}]^{-1} \cdot [\mathbf{I^{(n)}} + \mathbf{A}\vartheta h + \frac{(\mathbf{A}\vartheta h)^2}{2!} + \frac{(\mathbf{A}\vartheta h)^3}{3!}]$$

$$\mathbf{F_{BI4}}(\vartheta) = [\mathbf{I^{(n)}} - \mathbf{A}(1-\vartheta)h + \frac{(\mathbf{A}(1-\vartheta)h)^2}{2!} - \frac{(\mathbf{A}(1-\vartheta)h)^3}{3!} + \frac{(\mathbf{A}(1-\vartheta)h)^4}{4!}]^{-1} \cdot [\mathbf{I^{(n)}} + \mathbf{A}\vartheta h + \frac{(\mathbf{A}\vartheta h)^2}{2!} + \frac{(\mathbf{A}\vartheta h)^3}{3!} + \frac{(\mathbf{A}\vartheta h)^4}{4!}]$$

BI Algorithms for the Simulation of Stiff Systems II

With:

$$\vartheta = 0.4$$

we obtain the algorithms:

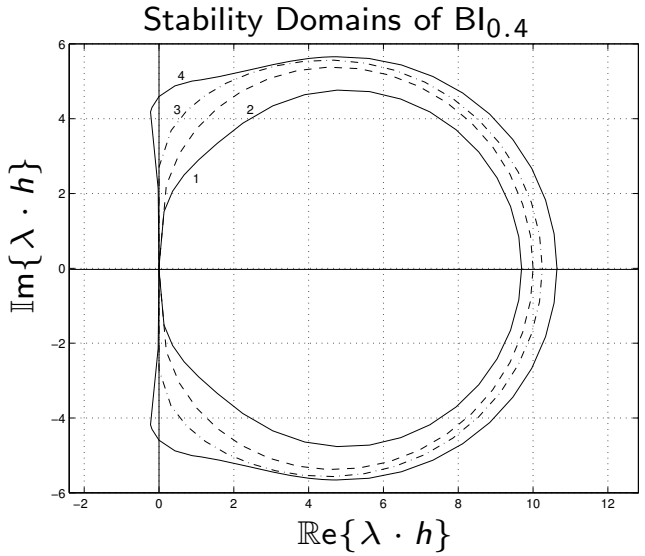

Figure: Numerical stability domains of $BI_{0.4}$ algorithms

All of these algorithms with the exception of $BI4_{0.4}$ are A-stable, but unfortunately, none of them is L-stable.

BI Algorithms for the Simulation of Stiff Systems III

To obtain L-stable algorithms, it is necessary to let the order of the denominator polynomial of the F-matrix be at least one higher than that of the numerator polynomial.

One way to obtain L-stable BI algorithms is to increase the order of approximation accuracy of the BRK algorithm by one.

A very good method of this type is the $\mathrm{BI4}/5_{0.45}$ algorithm. This algorithm starts with an FRK4 semi-step of length $0.45 \cdot h$ followed by a BRK5 semi-step of length $0.55 \cdot h$.

Conclusions

- In this presentation, we introduced the *Runge-Kutta algorithms*. These are single-step methods with multiple function evaluations within one integration step.

Conclusions

- In this presentation, we introduced the *Runge-Kutta algorithms*. These are single-step methods with multiple function evaluations within one integration step.
- Some RK algorithms are *explicit* whereas others are *implicit*.

Conclusions

- In this presentation, we introduced the *Runge-Kutta algorithms*. These are single-step methods with multiple function evaluations within one integration step.
- Some RK algorithms are *explicit* whereas others are *implicit*.
- We talked about the *numerical stability properties* of simulation algorithms, such as *F-stability*, *A-stability*, and *L-stability*.

Conclusions

- In this presentation, we introduced the *Runge-Kutta algorithms*. These are single-step methods with multiple function evaluations within one integration step.
- Some RK algorithms are *explicit* whereas others are *implicit*.
- We talked about the *numerical stability properties* of simulation algorithms, such as *F-stability*, *A-stability*, and *L-stability*.
- We introduced a number of *blended algorithms* that consist in multiple FRK and/or BRK partial steps, such as the *Richardson extrapolation* methods.

Conclusions

- In this presentation, we introduced the *Runge-Kutta algorithms*. These are single-step methods with multiple function evaluations within one integration step.
- Some RK algorithms are *explicit* whereas others are *implicit*.
- We talked about the *numerical stability properties* of simulation algorithms, such as *F-stability*, *A-stability*, and *L-stability*.
- We introduced a number of *blended algorithms* that consist in multiple FRK and/or BRK partial steps, such as the *Richardson extrapolation* methods.
- We finally introduced a class of *cyclic algorithms* that consist also in multiple FRK and/or BRK partial steps, such as the *backinterpolation* methods.

References

1. Xie, Wei (1995), *Backinterpolation Methods for the Numerical Solution of Ordinary Differential Equations and Applications*, MS Thesis, Dept. of Electrical & Computer Engineering, University of Arizona, Tucson, AZ.