Numerical Simulation of Dynamic Systems III

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

March 5, 2013

Analysis of Truncation Error

We would like to perform an *analysis of the truncation error* of the explicit numerical integration algorithm consisting in a prediction step using FE followed by a single correction step using BE:

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

$$\begin{aligned} \text{correction:} \quad \dot{\mathbf{x}}^P_{k+1} &= \mathbf{f}(\mathbf{x}^P_{k+1}, t_{k+1}) \\ \mathbf{x}^C_{k+1} &= \mathbf{x}_k + h \cdot \dot{\mathbf{x}}^P_{k+1} \end{aligned}$$

We obtain:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + h \cdot \mathbf{f}(\mathbf{x}_k + h \cdot \mathbf{f}_k, t_k + h)$$

We wish to develop the non-linear expression into a multi-dimensional Taylor series:

$$f(x + \Delta x, y + \Delta y) \approx f(x, y) + \frac{\partial f(x, y)}{\partial x} \cdot \Delta x + \frac{\partial f(x, y)}{\partial y} \cdot \Delta y$$

Therefore:

$$\mathbf{f}(\mathbf{x}_k + h \cdot \mathbf{f}_k, t_k + h) \approx \mathbf{f}(\mathbf{x}_k, t_k) + \frac{\partial \mathbf{f}(\mathbf{x}_k, t_k)}{\partial \mathbf{x}} \cdot (h \cdot \mathbf{f}_k) + \frac{\partial \mathbf{f}(\mathbf{x}_k, t_k)}{\partial t} \cdot h$$

Analysis of Truncation Error II

We obtain the approximation:

$$\mathbf{x}_{k+1} \approx \mathbf{x}_k + h \cdot \mathbf{f}(\mathbf{x}_k, t_k) + h^2 \cdot \left(\frac{\partial \mathbf{f}(\mathbf{x}_k, t_k)}{\partial \mathbf{x}} \cdot \mathbf{f}_k + \frac{\partial \mathbf{f}(\mathbf{x}_k, t_k)}{\partial t}\right)$$

Let us compare this approximation to the Taylor series truncated after the quadratic term:

$$\mathbf{x}_{k+1} \approx \mathbf{x}_k + h \cdot \mathbf{f}(\mathbf{x}_k, t_k) + \frac{h^2}{2} \cdot \dot{\mathbf{f}}(\mathbf{x}_k, t_k)$$

where:

$$\dot{\mathbf{f}}(\mathbf{x}_k, t_k) = \frac{d\mathbf{f}(\mathbf{x}_k, t_k)}{dt} = \frac{\partial \mathbf{f}(\mathbf{x}_k, t_k)}{\partial \mathbf{x}} \cdot \frac{d\mathbf{x}_k}{dt} + \frac{\partial \mathbf{f}(\mathbf{x}_k, t_k)}{\partial t}$$

y:

$$\frac{d\mathbf{x}_k}{dt} = \dot{\mathbf{x}}_k = \mathbf{f}_k$$

Therefore:

$$\mathbf{x}_{PC}(k+1) \approx \mathbf{x}_k + h \cdot \mathbf{f}(\mathbf{x}_k, t_k) + h^2 \cdot \dot{\mathbf{f}}(\mathbf{x}_k, t_k)$$

The Heun Integration Algorithm

Comparing the two approximations of FE and of PC:

$$\begin{aligned} \mathbf{x}_{FE}(k+1) &\approx \mathbf{x}_k + h \cdot \mathbf{f}(\mathbf{x}_k, t_k) \\ \mathbf{x}_{PC}(k+1) &\approx \mathbf{x}_k + h \cdot \mathbf{f}(\mathbf{x}_k, t_k) + h^2 \cdot \dot{\mathbf{f}}(\mathbf{x}_k, t_k) \end{aligned}$$

we notice that these two approximations can be easily combined in such a way that a second-order approximation results:

$$\mathbf{x}(k+1) = 0.5 \cdot (\mathbf{x}_{PC}(k+1) + \mathbf{x}_{FE}(k+1))$$

that is:

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

$$\begin{aligned} \text{correction:} \quad \dot{\mathbf{x}}^P_{k+1} &= \mathbf{f}(\mathbf{x}^P_{k+1}, t_{k+1}) \\ \mathbf{x}^C_{k+1} &= \mathbf{x}_k + 0.5 \cdot h \cdot (\dot{\mathbf{x}}_k + \dot{\mathbf{x}}^P_{k+1}) \end{aligned}$$

This numerical integration method is called *Heun integration algorithm*.

Second-order Explicit Runge-Kutta Methods

We can check if it is possible to combine more than two different approximations with the aim of obtaining *higher-order numerical integration algorithms*.

Let us start with a single correction term as before, but this time around parameterized as follows:

$$\begin{aligned} \text{prediction:} \quad & \dot{\mathbf{x}}_k = \mathbf{f}(\mathbf{x}_k, t_k) \\ & \mathbf{x}^P = \mathbf{x}_k + h \cdot \beta_{11} \cdot \dot{\mathbf{x}}_k \\ \text{correction:} \quad & \dot{\mathbf{x}}^P = \mathbf{f}(\mathbf{x}^P, t_k + \alpha_1 \cdot h) \\ & \mathbf{x}^C_{k+1} = \mathbf{x}_k + h \cdot (\beta_{21} \cdot \dot{\mathbf{x}}_k + \beta_{22} \cdot \dot{\mathbf{x}}^P) \end{aligned}$$

Developing into a Taylor series as before, we obtain:

$$\mathbf{x}^C_{k+1} = \mathbf{x}_k + h \cdot (\beta_{21} + \beta_{22}) \cdot \mathbf{f}_k + \frac{h^2}{2} \cdot [2 \cdot \beta_{11} \cdot \beta_{22} \cdot \frac{\partial \mathbf{f}_k}{\partial \mathbf{x}} \cdot \mathbf{f}_k + 2 \cdot \alpha_1 \cdot \beta_{22} \cdot \frac{\partial \mathbf{f}_k}{\partial t}]$$

Second-order Explicit Runge-Kutta Methods II

This approximation:

$$\mathbf{x}^C_{k+1} = \mathbf{x}_k + h \cdot (\beta_{21} + \beta_{22}) \cdot \mathbf{f}_k + \frac{h^2}{2} \cdot [2 \cdot \beta_{11} \cdot \beta_{22} \cdot \frac{\partial \mathbf{f}_k}{\partial \mathbf{x}} \cdot \mathbf{f}_k + 2 \cdot \alpha_1 \cdot \beta_{22} \cdot \frac{\partial \mathbf{f}_k}{\partial t}]$$

can be compared to the Taylor series expansion truncated after the quadratic term:

$$\mathbf{x}_{k+1} \approx \mathbf{x}_k + h \cdot \mathbf{f}_k + \frac{h^2}{2} \cdot [\frac{\partial \mathbf{f}_k}{\partial \mathbf{x}} \cdot \mathbf{f}_k + \frac{\partial \mathbf{f}_k}{\partial t}]$$

In this fashion, we obtain general conditions that guarantee that the resulting algorithms are *second-order accurate methods*.

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

Second-order Explicit Runge-Kutta Methods III

Evidently, the *Heun algorithm* with:

$$\alpha = \begin{pmatrix} 1 \\ 1 \end{pmatrix}; \quad \beta = \begin{pmatrix} 1 & 0 \\ 0.5 & 0.5 \end{pmatrix}$$

satisfies the three non-linear equations in four unknowns. α_2 represents the time at which the correction is to be evaluated. Evidently, this must always be 1, as the integration step must end at $t^* + h$.

The Runge-Kutta methods may alternatively be characterized by a so-called *Butcher tableau*:

0	0	0
1	1	0
x	1/2	1/2

where the first row represents the initial evaluation of the derivative at time t^*, the second row denotes the prediction step, i.e., the first stage of the algorithm, whereas the last row denotes the correction step, i.e., the approximation of the value of the state vector at time $t^* + h$. The column to the left indicates the time instants of each stage, whereas the additional columns specify the weights used in each stage.

The Explicit Midpoint Rule

Another algorithm that satisfies the three equations is characterized by:

$$\alpha = \begin{pmatrix} 0.5 \\ 1 \end{pmatrix}; \quad \beta = \begin{pmatrix} 0.5 & 0 \\ 0 & 1 \end{pmatrix}$$

that is be the Butcher tableau:

0	0	0
1/2	1/2	0
x	0	1

This algorithm is called the *explicit midpoint rule*.

The algorithm can be implemented in the following fashion:

$$\begin{aligned} \text{prediction:} \quad & \dot{\mathbf{x}}_k = \mathbf{f}(\mathbf{x}_k, t_k) \\ & \mathbf{x}^P_{k+\frac{1}{2}} = \mathbf{x}_k + \tfrac{h}{2} \cdot \dot{\mathbf{x}}_k \\ \text{correction:} \quad & \dot{\mathbf{x}}^P_{k+\frac{1}{2}} = \mathbf{f}(\mathbf{x}^P_{k+\frac{1}{2}}, t_{k+\frac{1}{2}}) \\ & \mathbf{x}^C_{k+1} = \mathbf{x}_k + h \cdot \dot{\mathbf{x}}^P_{k+\frac{1}{2}} \end{aligned}$$

This method is a bit more economical than the Heun algorithm, because its Butcher tableau contains one additional zero entry.

The Family of Explicit Runge-Kutta Methods

If we allow more than one prediction stage, it is possible to obtain *higher-order integration algorithms*:

$$\begin{aligned}
&\text{stage 0:} && \dot{\mathbf{x}}^{\mathbf{P_0}} = \mathbf{f}(\mathbf{x_k}, t_k) \\
&\text{stage j:} && \mathbf{x}^{\mathbf{P_j}} = \mathbf{x_k} + h \cdot \sum_{i=1}^{j} \beta_{ji} \cdot \dot{\mathbf{x}}^{\mathbf{P_{i-1}}} \\
& && \dot{\mathbf{x}}^{\mathbf{P_j}} = \mathbf{f}(\mathbf{x}^{\mathbf{P_j}}, t_k + \alpha_j \cdot h) \\
&\text{last stage:} && \mathbf{x_{k+1}} = \mathbf{x_k} + h \cdot \sum_{i=1}^{\ell} \beta_{\ell i} \cdot \dot{\mathbf{x}}^{\mathbf{P_{i-1}}}
\end{aligned}$$

The best known algorithm from this class of numerical integration methods is the 4^{th}*-order accurate Runge-Kutta (RK4) algorithm* characterized by:

$$\alpha = \begin{pmatrix} 1/2 \\ 1/2 \\ 1 \\ 1 \end{pmatrix} \; ; \; \beta = \begin{pmatrix} 1/2 & 0 & 0 & 0 \\ 0 & 1/2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1/6 & 1/3 & 1/3 & 1/6 \end{pmatrix}$$

The RK4 Algorithm

Therefore:

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

The *RK4 algorithm* can be implemented in the following way:

$$\begin{aligned}
&\text{stage 0:} && \dot{\mathbf{x}}_\mathbf{k} = \mathbf{f}(\mathbf{x_k}, t_k) \\
&\text{stage 1:} && \mathbf{x}^{\mathbf{P_1}} = \mathbf{x_k} + \tfrac{h}{2} \cdot \dot{\mathbf{x}}_\mathbf{k} \\
& && \dot{\mathbf{x}}^{\mathbf{P_1}} = \mathbf{f}(\mathbf{x}^{\mathbf{P_1}}, t_{k+\frac{1}{2}}) \\
&\text{stage 2:} && \mathbf{x}^{\mathbf{P_2}} = \mathbf{x_k} + \tfrac{h}{2} \cdot \dot{\mathbf{x}}^{\mathbf{P_1}} \\
& && \dot{\mathbf{x}}^{\mathbf{P_2}} = \mathbf{f}(\mathbf{x}^{\mathbf{P_2}}, t_{k+\frac{1}{2}}) \\
&\text{stage 3:} && \mathbf{x}^{\mathbf{P_3}} = \mathbf{x_k} + h \cdot \dot{\mathbf{x}}^{\mathbf{P_2}} \\
& && \dot{\mathbf{x}}^{\mathbf{P_3}} = \mathbf{f}(\mathbf{x}^{\mathbf{P_3}}, t_{k+1}) \\
&\text{stage 4:} && \mathbf{x_{k+1}} = \mathbf{x_k} + \tfrac{h}{6} \cdot [\dot{\mathbf{x}}_\mathbf{k} + 2 \cdot \dot{\mathbf{x}}^{\mathbf{P_1}} + 2 \cdot \dot{\mathbf{x}}^{\mathbf{P_2}} + \dot{\mathbf{x}}^{\mathbf{P_3}}]
\end{aligned}$$

It is therefore an *explicit* 4^{th}*-order accurate single-step method in four stages.*

History of Explicit Runge-Kutta Methods

Developer	Year	Order	# of Stages
Euler	1768	1	1
Runge	1895	4	4
Heun	1900	2	2
Kutta	1901	5	6
Huťa	1956	6	8
Shanks	1966	7	9
Curtis	1970	8	11

Table: History of Explicit Runge-Kutta Algorithms

- The number of non-linear equations grows rapidly with the order of the methods. Already for RK methods of order 5, there no longer exists a solution in 5 stages. More stages must be added in order to increase the number of parameters.
- Because of the many non-linear equations to be solved, it took a long time before higher-order RK methods were found.
- In recent years, a sequence of yet higher-order RK methods were developed quite rapidly using computer algebra methods (*Maple*, *Mathematica*).

Additional Constraints

We may wish to impose more constraints on the parameters characterizing desirable RK methods.

- Obviously, we want to request that, in an ℓ–stage algorithm:
$$\alpha_\ell = 1.0$$
since we wish to end the step at t_{k+1}.
- Also, we usually want to make sure that:
$$\alpha_i \in [0.0, 1.0] \;\; ; \;\; i = \{1, 2, \ldots, \ell\}$$
that is, all function evaluations are performed at times between t_k and t_{k+1}.
- If we wish to prevent the algorithm from ever "integrating backward through time," we shall add the constraint that:
$$\alpha_j \geq \alpha_i \;\; ; \;\; j \geq i$$
- If we want to disallow micro-steps of length 0, we make this condition even more stringent:
$$\alpha_j > \alpha_i \;\; ; \;\; j > i$$
The previously introduced classical RK4 algorithm violates the latter constraint.

Higher Derivatives

While we were able to develop Heun's method using a matrix-vector notation, this technique won't work anymore as we proceed to third-order algorithms.

We found that:

$$\frac{d\mathbf{f}}{dt} = \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \cdot \frac{d\mathbf{x}}{dt} + \frac{\partial \mathbf{f}}{\partial t}$$

or, in shorthand notation:

$$\dot{\mathbf{f}} = \mathbf{f}_\mathbf{x} \cdot \mathbf{f} + \mathbf{f}_t$$

When we proceed to third-order algorithms, we need an expression for the second absolute derivative of $\mathbf{f}$ with respect to time. Thus, we are inclined to write formally:

$$\begin{aligned}
\ddot{\mathbf{f}} &= (\mathbf{f}_\mathbf{x} \cdot \mathbf{f} + \mathbf{f}_t)^{\cdot} \\
&= \dot{\mathbf{f}}_\mathbf{x} \cdot \mathbf{f} + \mathbf{f}_\mathbf{x} \cdot \dot{\mathbf{f}} + \dot{\mathbf{f}}_t \\
&= (\dot{\mathbf{f}})_\mathbf{x} \cdot \mathbf{f} + \mathbf{f}_\mathbf{x} \cdot (\dot{\mathbf{f}}) + (\dot{\mathbf{f}})_t \\
&= (\mathbf{f}_\mathbf{x} \cdot \mathbf{f} + \mathbf{f}_t)_\mathbf{x} \cdot \mathbf{f} + \mathbf{f}_\mathbf{x} \cdot (\mathbf{f}_\mathbf{x} \cdot \mathbf{f} + \mathbf{f}_t) + (\mathbf{f}_\mathbf{x} \cdot \mathbf{f} + \mathbf{f}_t)_t \\
&= \mathbf{f}_{\mathbf{xx}} \cdot (\mathbf{f})^2 + 2 \cdot (\mathbf{f}_\mathbf{x})^2 \cdot \mathbf{f} + 2 \cdot \mathbf{f}_{\mathbf{x}t} \cdot \mathbf{f} + 2 \cdot \mathbf{f}_\mathbf{x} \cdot \mathbf{f}_t + \mathbf{f}_{tt}
\end{aligned}$$

Higher Derivatives II

Unfortunately, it is not clear, what this is supposed to mean. Obviously, $\ddot{\mathbf{f}}$ and $\mathbf{f}_{tt}$ are vectors, but what is $\mathbf{f}_{\mathbf{xx}} \cdot (\mathbf{f})^2$ supposed to mean? Is it a tensor multiplied by the square of a vector? Quite obviously, the formal differentiation mechanism doesn't extend to higher derivatives in the sense of familiar matrix-vector multiplications. Evidently, we must treat the expression $\mathbf{f}_{\mathbf{xx}} \cdot (\mathbf{f})^2$ differently.

John Butcher developed a new syntax and a set of rules for how these higher derivatives must be interpreted. In essence, it turns out that, in this new syntax:

1. sums remain commutative and associative,
2. derivatives can still be computed in any order, i.e., $(\dot{\mathbf{f}})_\mathbf{x} = (\mathbf{f}_\mathbf{x})^{\cdot}$, and
3. the multiplication rule can be generalized, thus: $(\mathbf{f}_\mathbf{x} \cdot \mathbf{f})_\mathbf{x} = \mathbf{f}_{\mathbf{xx}} \cdot \mathbf{f} + (\mathbf{f}_\mathbf{x})^2$.

It is not necessary for us to learn Butcher's new syntax. It is sufficient to know that we can basically proceed as before, but must abstain from interpreting terms involving higher derivatives as consisting of factors that are combined by means of the familiar matrix-vector multiplication.

Higher Derivatives III

Prior to Butcher's work, all higher-order RK algorithms had simply been derived for the scalar case, and were then blindly applied to integrate entire state vectors. Butcher discovered that several of the previously developed and popular higher-order RK algorithms drop one or several orders of accuracy when applied to a state vector instead of a scalar state variable.

The reason for this somewhat surprising discovery is very simple. Already when computing the third absolute derivative of $\mathbf{f}$ with respect to time, the two terms $\mathbf{f}_\mathbf{x} \cdot \mathbf{f}_{\mathbf{xx}} \cdot (\mathbf{f})^2$ and $\mathbf{f}_{\mathbf{xx}} \cdot \mathbf{f} \cdot \mathbf{f}_\mathbf{x} \cdot \mathbf{f}$ appear in the derivation. In the scalar case, these two terms are identical and can be combined, since:

$$a \cdot b = b \cdot a$$

Unfortunately this rule does not extend to the vector case.

Our new animals in the mathematical zoo of data structures and operations exhibit a property that we are already quite familiar with from matrix calculus, namely that multiplications are usually not commutative:

$$\mathbf{A} \cdot \mathbf{B} = (\mathbf{B}' \cdot \mathbf{A}')' \neq \mathbf{B} \cdot \mathbf{A}$$

Numerical Stability Domains of Explicit RK Methods

Let us start by applying the Heun algorithm to a linear system:

$$\begin{aligned}
\text{prediction:} \quad \dot{\mathbf{x}}_\mathbf{k} &= \mathbf{A} \cdot \mathbf{x}_\mathbf{k} \\
\mathbf{x}^\mathbf{P}_{\mathbf{k}+1} &= \mathbf{x}_\mathbf{k} + h \cdot \dot{\mathbf{x}}_\mathbf{k} \\
\text{correction:} \quad \dot{\mathbf{x}}^\mathbf{P}_{\mathbf{k}+1} &= \mathbf{A} \cdot \mathbf{x}^\mathbf{P}_{\mathbf{k}+1} \\
\mathbf{x}^\mathbf{C}_{\mathbf{k}+1} &= \mathbf{x}_\mathbf{k} + 0.5 \cdot h \cdot (\dot{\mathbf{x}}_\mathbf{k} + \dot{\mathbf{x}}^\mathbf{P}_{\mathbf{k}+1})
\end{aligned}$$

that is:

$$\mathbf{x}^\mathbf{C}_{\mathbf{k}+1} = [\mathbf{I}^{(\mathbf{n})} + \mathbf{A} \cdot h + \frac{(\mathbf{A} \cdot h)^2}{2}] \cdot \mathbf{x}_\mathbf{k}$$

Therefore:

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

Numerical Stability Domains of Explicit RK Methods II

The algorithms must approximate the *analytical solution*:

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

up to the term that corresponds to the approximation order of the method. Therefore, all n^{th}-order methods in n stages have identical stability domains.

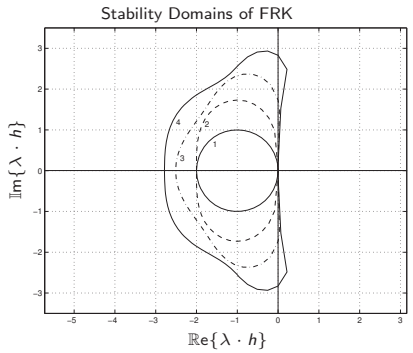

Figure: Numerical Stability Domains of Explicit RK Methods

Numerical Stability Domains of Explicit RK Methods III

Starting from the fifth-order RK methods, different RK algorithms of the same order may have slightly different stability domains. The reason is that these algorithms use additional stages.

An RK5 algorithm in 6 stages contains in its **F**-matrix a term in $(\mathbf{A} \cdot h)^6$, albeit with the incorrect coefficient. This term contributes to the stability domain. Different 6-stage RK5 algorithms differ in their coefficients of the term in $(\mathbf{A} \cdot h)^6$, and consequently, their stability domains differ as well.

Some of these higher-order RK algorithms exhibit small stable islands somewhere in their right-half complex $\lambda \cdot h$ plane.

Conclusions

- Explicit Runge-Kutta algorithms of various orders of approximation accuracy were developed.
- All FRK algorithms except FE are multi-stage algorithms that require internal function evaluations.
- FRK algorithms do not preserve any information across multiple steps, i.e., these algorithms are self-starting and start afresh with each new integration step.
- The class of explicit RK algorithms are among the most widely used numerical ODE solvers on the market today.
- For most engineering problems, 4^{th}-order FRK algorithms offer a good compromise between the needed accuracy and the economy of simulating across a single step.
- Professional FRK codes usually offer step-size control, i.e., they adjust the step size from one integration step to the next.