Selçuk J. Appl. Math.
Vol. 2, No. 1, pp. 3–15, 2001

**Selçuk Journal of
Applied Mathematics**

Runge – Kutta Methods, Trees, and Maple

On a Simple Proof of Butcher's Theorem and the Automatic Generation of Order Conditions

Folkmar Bornemann

Center of Mathematical Sciences, Munich University of Technology, 80290 Munich, Germany; e-mail: bornemann@ma.tum.de

February 9, 2001

Summary. This paper presents a simple and elementary proof of Butcher's theorem on the order conditions of Runge-Kutta methods. It is based on a recursive definition of rooted trees and avoids combinatorial tools such as labelings and Faà di Bruno's formula. This strictly recursive approach can easily and elegantly be implemented using modern computer algebra systems like Maple for automatically generating the order conditions. The full, but short source code is presented and applied to some instructive examples.

Key words: numerical solution of ODEs, Runge-Kutta methods, recursive representation of rooted trees, Butcher's theorem, automatic generation of order conditions, computer algebra systems.

Mathematics Subject Classification (1991): 65-01, 65L06, 65Y99

1. Introduction

A first step towards the construction of Runge-Kutta methods is the calculation of the order conditions that the coefficients have to obey. In the old days they were obtained by expanding the error term in a Taylor series by hand, a procedure which for higher orders sooner or later runs into difficulties because of the largely increasing combinatorial complexity. It was a major break-through when Butcher [1] published his result of systematically describing order conditions by rooted trees. The proof of this result has evolved very much in meantime, mainly under the influence of Butcher's later work [3] and the contributions of Hairer and Wanner [7,6]. In this paper we will present a simple and elementary proof of Butcher's theorem by using very consequently the recursive structure of rooted trees. This way we avoid lengthy calculations of combinatorial coefficients, the use of tree-labelings, or Faà di Bruno's formula as in [3,6]. Our proof is very similar in spirit to the presentation of B-series by Hairer in Chapter 2 of his lecture notes [5].

As early as 1976 Jenks [8] posed the problem of automatically generating order conditions for Runge-Kutta methods using computer algebra systems, but no replies were received. In 1988 Keiper of Wolfram Research concluded that the method of automatically calculating Taylor expansions by brute force was bound to be very inefficient. Naturally he turned to the elegant results of Butcher's. Utilizing them, he wrote the Mathematica package `Butcher.m`, which has been available as part of the standard distribution of Mathematica since then. This package was later considerably improved by Sofroniou [9] and offers a lot of sophisticated tools.

While teaching the simple proof of Butcher's result in a first course on numerical ODEs, the author realized that the underlying recursive structure could also be exploited for a simple and elegant computer implementation. This approach differs from the work of Sofroniou in various respects. We will present the full source code in Maple and some applications.

2. Runge-Kutta methods

The Runge-Kutta methods are one-step discretizations of initial-value problems for systems of d ordinary differential equations,

$$x' = f(t,x), \qquad x(t_0) = x_0,$$

where the right-hand side $f : [t_0, T] \times \Omega \subset \mathbb{R} \times \mathbb{R}^d \to \mathbb{R}^d$ is assumed to be sufficiently smooth. The *continuous evolution* $x(t) = \Phi^{t,t_0} x_0$ of the initial-value problem is approximated in steps of length τ by $\Psi^{t+\tau,t}x \approx \Phi^{t+\tau,t}x$. This *discrete evolution* $\Psi^{t+\tau,t}x$ is defined as an approximation of the integral-equation representation

$$\Phi^{t+\tau,t}x = x + \int_t^{t+\tau} f(\sigma, \Phi^{\sigma,t}x)\, d\sigma$$

by appropriate quadrature formulas:

$$(1) \quad \Psi^{t+\tau,t}x = x + \tau \sum_{i=1}^{s} b_i k_i, \qquad k_i = f\left(t + c_i\tau, x + \tau \sum_{j=1}^{s} a_{ij} k_j\right).$$

The vectors $k_i \in \mathbb{R}^d$, $i = 1, \dots, s$, are called *stages*, s is the stage number. Following the standard notation, we collect the coefficients of the method into a matrix and two vectors

$$\mathcal{A} = (a_{ij})_{ij} \in \mathbb{R}^{s\times s},\ b = (b_1, \dots, b_s)^T \in \mathbb{R}^s,\ c = (c_1, \dots, c_s)^T \in \mathbb{R}^s.$$

The method is *explicit*, if $\mathcal{A}$ is strictly lower triangular. The method has *order* $p \in \mathbb{N}$, if the error term expands to

$$\Phi^{t+\tau,t}x - \Psi^{t+\tau,t}x = O(\tau^{p+1}).$$

In terms of the Taylor expansion of the error $\Phi - \Psi$, the vanishing of all lower order terms in τ just defines the conditions which have to be satisfied by the coefficients $\mathcal{A}$, b and c of a Runge-Kutta method. If we choose

$$c_i = \sum_{j=1}^{s} a_{ij},$$

it can be shown [6], that there is no loss of generality in considering *autonomous* systems only, i.e., those with no dependence of f on t. Doing so, the expressions $\Phi^{t+\tau,t}x$ and $\Psi^{t+\tau,t}x$ are likewise independent of t. We will write $\Phi^\tau x$ and $\Psi^\tau x$ for short, calling them the *flow* and the *discrete flow* of the continuous resp. the discrete system.

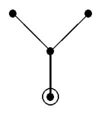
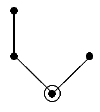
3. Elementary differentials and rooted trees

The Taylor expansions of both the phase flow $\Phi^\tau x$ and the discrete flow $\Psi^\tau x$ are linear combinations of *elementary* differentials like

$$f'''(f'f, f'f, f) = \sum_{ijklm} \frac{\partial^3 f}{\partial x_i \partial x_j \partial x_k} \cdot \frac{\partial f_i}{\partial x_l} f_l \cdot \frac{\partial f_j}{\partial x_m} f_m \cdot f_k.$$

We will use the short multilinear notation of the left hand side for the rest of the paper.

An elementary differential can be expressed uniquely by the structure of how the subterms enter the multilinear maps. For instance, looking at the expression $f'''(f'f, f'f, f)$ we observe that f''' must be a third derivative, since *three* arguments make it *three*-linear. This structure can be expressed in general by *rooted trees*, e.g.,

 expressing $f'f''(f, f)$,  expressing $f''(f'f, f)$.

Every node with n children denotes a nth derivative of f, which is applied as a multilinear map to further elementary differentials, according to the structure of the tree. We start reading off this structure by looking at the root. This defines a *recursive* procedure, if we observe the following: Having removed the root and its edges, a rooted tree β decomposes into rooted subtrees $\beta_1, \dots, \beta_n$ with strictly less nodes. The roots of the subtrees $\beta_1, \dots, \beta_n$ are exactly the n children of β's root. This way a rooted tree β can be defined as the *unordered* list of its successors

$$\beta = [\beta_1, \dots, \beta_n], \qquad \#\beta = 1 + \#\beta_1 + \dots + \#\beta_n. \tag{2}$$

Here, we denote by $\#\beta$ the *order* of a rooted tree β, i.e., the number of its nodes. The root itself can be identified with the *empty* list, $\odot = [\,]$.

An application of this procedure shows for the examples above that $f'(f''(f,f))$ is expressed by $[[\odot,\odot]]$ and $f''(f'(f),f)$ is expressed by $[[\odot],\odot]$. The reader will observe the perfect matching of parentheses and commas. In general the relation between a rooted tree $\beta = [\beta_1, \dots, \beta_n]$ and its corresponding elementary differential $f^{(\beta)}(x)$ is recursively defined by

$$f^{(\beta)}(x) = f^{(n)}(x) \cdot \left(f^{(\beta_1)}(x), \dots, f^{(\beta_n)}(x)\right).$$

The dot of multiplication denotes the multilinear application of the derivative to the n given arguments. Due to the symmetry of the n-linear map $f^{(n)}$, the order of the subtrees $\beta_1, \dots, \beta_n$ does not matter, which means, that $f^{(\beta)}$ depends in a well-defined way on β as an unordered list only.

From $\odot = [\,]$ we deduce $f^{(\odot)} = f$. Analogously, each of the recursive definitions in the following will have a well-defined meaning if applied to the single root $\odot = [\,]$, mostly by using the reasonable convention that empty products evaluate to one and empty sums to zero—a convention that is also observed by most computer algebra systems.

4. A simple proof of Butcher's theorem

We are now in a position to calculate and denote the Taylor expansion of the continuous flow Φ^τ in a clear and compact fashion.

Lemma 1. *Given $f \in C^p(\Omega, \mathbb{R}^d)$ the flow $\Phi^\tau x$ expands to*

$$\Phi^\tau x = x + \sum_{\#\beta \le p} \frac{\tau^{\#\beta}}{\beta!} \alpha_\beta \, f^{(\beta)}(x) + O(\tau^{p+1}).$$

The coefficients $\beta!$ and α_β belonging to a rooted tree $\beta = [\beta_1, \dots, \beta_n]$ are recursively defined by

$$\beta! = (\#\beta)\, \beta_1! \cdot \ldots \cdot \beta_n!, \qquad \alpha_\beta = \frac{\delta_\beta}{n!}\, \alpha_{\beta_1} \cdot \ldots \cdot \alpha_{\beta_n}. \tag{3}$$

By δ_β we denote the number of different ordered tuples $(\beta_1, \dots, \beta_n)$ which correspond to the same unordered list $\beta = [\beta_1, \dots, \beta_n]$.

Proof. The assertion is obviously true for $p = 0$. We proceed by induction on p. Using the assertion for p, the multivariate Taylor formula and the multilinearity of the derivatives we obtain

$$\begin{aligned}
f(\Phi^\tau x) &= f\left(x + \sum_{\#\beta \le p} \frac{\tau^{\#\beta}}{\beta!} \alpha_\beta f^{(\beta)} + O(\tau^{p+1})\right) \\
&= \sum_{n=0}^{p} \frac{1}{n!} f^{(n)} \cdot \left(\sum_{\#\beta_1 \le p} \frac{\tau^{\#\beta_1}}{\beta_1!} \alpha_{\beta_1} f^{(\beta_1)}, \dots, \sum_{\#\beta_n \le p} \frac{\tau^{\#\beta_n}}{\beta_n!} \alpha_{\beta_n} f^{(\beta_n)} \right) \\
&\quad + O(\tau^{p+1}) \\
&= \sum_{n=0}^{p} \frac{1}{n!} \sum_{\#\beta_1 + \ldots + \#\beta_n \le p} \frac{\tau^{\#\beta_1 + \ldots + \#\beta_n}}{\beta_1! \cdot \ldots \cdot \beta_n!} \cdot \alpha_{\beta_1} \cdot \ldots \cdot \alpha_{\beta_n} \cdot \\
&\quad f^{(n)} \cdot \left(f^{(\beta_1)}, \dots, f^{(\beta_n)} \right) + O(\tau^{p+1}) \\
&= \sum_{n=0}^{p} \sum_{\substack{\beta = [\beta_1, \dots, \beta_n] \\ \#\beta \le p+1}} \frac{\#\beta \cdot \tau^{\#\beta - 1}}{\beta!} \cdot \underbrace{\frac{\delta_\beta}{n!} \alpha_{\beta_1} \cdot \ldots \cdot \alpha_{\beta_n}}_{= \alpha_\beta} f^{(\beta)} + O(\tau^{p+1}) \\
&= \sum_{\#\beta \le p+1} \frac{\#\beta \cdot \tau^{\#\beta - 1}}{\beta!} \alpha_\beta f^{(\beta)} + O(\tau^{p+1}).
\end{aligned}$$

Plugging this into the integral form of the initial value problem we obtain

$$\Phi^\tau x = x + \int_0^\tau f(\Phi^\sigma x)\, d\sigma = x + \sum_{\#\beta \le p+1} \frac{\tau^{\#\beta}}{\beta!} \alpha_\beta f^{(\beta)} + O(\tau^{p+2}),$$

which proves the assertion for $p + 1$. □

A likewise clear and compact expression can be calculated for the Taylor expansion of the discrete flow.

Lemma 2. *Given $f \in C^p(\Omega, \mathbb{R}^d)$ the discrete flow $\Psi^\tau x$ expands to*

$$\Psi^\tau x = x + \sum_{\#\beta \le p} \tau^{\#\beta} \alpha_\beta \cdot b^T \mathcal{A}^{(\beta)} f^{(\beta)}(x) + O(\tau^{p+1}).$$

The vector $\mathcal{A}^{(\beta)} \in \mathbb{R}^s$, $\beta = [\beta_1, \dots, \beta_n]$, is recursively defined by

$$\mathcal{A}_i^{(\beta)} = \left(\mathcal{A} \cdot \mathcal{A}^{(\beta_1)}\right)_i \cdot \ldots \cdot \left(\mathcal{A} \cdot \mathcal{A}^{(\beta_n)}\right)_i, \qquad i = 1, \dots, s. \tag{4}$$

Proof. Because of the definition (1)of the discrete flow we have to prove that the stages k_i expand to

$$k_i = \sum_{\#\beta \le p} \tau^{\#\beta - 1} \alpha_\beta \mathcal{A}_i^{(\beta)} f^{(\beta)} + O(\tau^p).$$

This is obviously the case for $p = 0$. We proceed by induction on p. Using the assertion for p, the definition of the stages k_i, the multivariate Taylor formula and the multilinearity of the derivatives we obtain

$$\begin{aligned}
k_i &= f\left(x + \tau\left(\sum_{\#\beta \le p} \tau^{\#\beta-1} \alpha_\beta \left(\mathcal{A} \cdot \mathcal{A}^{(\beta)}\right)_i f^{(\beta)} + O(\tau^p)\right)\right) \\
&= \sum_{n=0}^{p} \frac{1}{n!} f^{(n)} \cdot \left(\sum_{\#\beta_1 \le p} \tau^{\#\beta_1} \alpha_{\beta_1} \left(\mathcal{A} \cdot \mathcal{A}^{(\beta_1)}\right)_i f^{(\beta_1)}, \dots \right. \\
&\qquad\qquad \left. \dots, \sum_{\#\beta_n \le p} \tau^{\#\beta_n} \alpha_{\beta_n} \left(\mathcal{A} \cdot \mathcal{A}^{(\beta_n)}\right)_i f^{(\beta_n)}\right) + O(\tau^{p+1}) \\
&= \sum_{n=0}^{p} \frac{1}{n!} \sum_{\#\beta_1 + \ldots + \#\beta_n \le p} \tau^{\#\beta_1 + \ldots + \#\beta_n} \cdot \alpha_{\beta_1} \cdot \ldots \cdot \alpha_{\beta_n} \cdot \left(\mathcal{A} \cdot \mathcal{A}^{(\beta_1)}\right)_i \cdot \\
&\qquad \ldots \cdot \left(\mathcal{A} \cdot \mathcal{A}^{(\beta_n)}\right)_i f^{(n)} \cdot \left(f^{(\beta_1)}, \dots, f^{(\beta_n)}\right) + O(\tau^{p+1}) \\
&= \sum_{n=0}^{p} \sum_{\substack{\beta = [\beta_1, \dots, \beta_n] \\ \#\beta \le p+1}} \tau^{\#\beta - 1} \cdot \underbrace{\frac{\delta_\beta}{n!} \alpha_{\beta_1} \cdot \ldots \cdot \alpha_{\beta_n}}_{= \alpha_\beta} \cdot \mathcal{A}_i^{(\beta)} f^{(\beta)} + O(\tau^{p+1}) \\
&= \sum_{\#\beta \le p+1} \tau^{\#\beta - 1} \alpha_\beta \cdot \mathcal{A}_i^{(\beta)} f^{(\beta)} + O(\tau^{p+1}),
\end{aligned}$$

which proves the assertion for $p + 1$. □

Comparing the coefficients of the elementary differentials in the expansions of both the phase flow and the discrete flow, we immediately obtain Butcher's theorem [1].

Theorem 1 (Butcher 1963). *A Runge-Kutta-method $(b, \mathcal{A})$ is of order $p \in \mathbb{N}$ for all $f \in C^p(\Omega, \mathbb{R}^d)$, if the order conditions*

$$b^T \mathcal{A}^{(\beta)} = 1/\beta!$$

are satisfied for all rooted trees β of order $\#\beta \leq p$.

5. Generating order conditions with Maple

The recursive constructions underlying the proof of Butcher's theorem can easily be realized using modern computer algebra systems like Maple. We assume that the reader is familiar with this particular package.

The recursive data-structure of a rooted tree itself can be realized by self-reference as a list of rooted trees,

```
> `type/tree`:=list(tree): f:=[]:
```

Conveniently, we have given a name to the root $\odot = []$. Because Maple's lists are *ordered*, we have to sort the trees recursively before testing for equality:

```
TreeSort:=proc(beta::tree)
  sort(map(TreeSort,beta));
end: # TreeSort
```

Now, here is an example for the tree representing the elementary differential $f''(f''(f, f'(f)), f) = f''(f, f''(f'(f), f))$. The corresponding tree is obtained by erasing all the derivatives $f^{(n)}$ from the expression and replacing parentheses $(\dots)$ by $[\dots]$:

```
> beta[1]:=[[f,[f]],f]: beta[2]:=[f,[[f],f]]:
> evalb(TreeSort(beta[1])=TreeSort(beta[2]));
                          true
```

The definition (2)of the order $\#\beta$ is simply expressed by the recursive procedure

```
TreeOrder:=proc(beta::tree)
  option remember;
  1+`+`(op(map(TreeOrder,beta)));
end: # TreeOrder
```

As an example, we take the tree that represents the elementary differential $f''(f'''(f'(f), f'(f), f), f)$:

```
> beta:=[[[f],[f],f],f]: TreeOrder(beta);
                          8
```

The definition (3)of $\beta!$ is analogously expressed by:

```
TreeFactorial:=proc(beta::tree)
  option remember;
  TreeOrder(beta)*`*`(op(map(TreeFactorial,beta)));
end: # TreeFactorial
```

The above used tree β of order 8 gives:

```
  >  TreeFactorial(beta);
                                 192
```

For the sake of completeness we also express the recursive definition (3)of α_β using Maple:

```
TreeAlpha:=proc(beta::tree)
  option remember;
  local n;
  n:=nops(beta);
  nops(combinat[permute](beta,n))/n!*
      `*`(op(map(TreeAlpha,beta)));
end: # TreeAlpha
```

An application to the above example yields:

```
  >  TreeAlpha(beta);
```

$$\frac{1}{2}$$

We are now ready to map the recursive definition (4)of $\mathcal{A}^{(\beta)}$ and of the order condition $b^T\mathcal{A}^{(\beta)} = 1/\beta!$ to Maple:

```
TreeA:=proc(beta::tree,no::posint)
  option remember;
  local vars,val,son;
  vars:=[i,j,k,l,m,p,q,r,u,v,w];
  val:=1;
  for son in beta do
  val:= val*Sum(a[vars[no],vars[no+1]]*
      TreeA(son,no+1),vars[no+1]=1..s);
  od;
  val;
end: # TreeA

TreeOrderCondition:=proc(beta::tree)
  Sum(b[i]*TreeA(beta,1),i=1..s)=
      1/TreeFactorial(beta);
end: # TreeOrderCondition
```

The coordinate index for the vector $\mathcal{A}^{(\beta)}$ can be chosen from the list `[i,j,k,l,m,p,q,r,u,v,w]` and is passed by number as the second argument to `TreeA`. The order condition belonging to the above example is obtained as follows:

```
> TreeOrderCondition(beta);
```

$$\sum_{i=1}^{s} b_i \left(\sum_{j=1}^{s} a_{i,j} \left(\sum_{k=1}^{s} a_{j,k} (\sum_{l=1}^{s} a_{k,l})\right)^2 (\sum_{k=1}^{s} a_{j,k})\right) (\sum_{j=1}^{s} a_{i,j}) = \frac{1}{192}$$

Even the typesetting of this formula was done completely automatically, using Maple's ability to generate TEX-sources.

To generate all the order conditions for a given order p, we need a device that constructs the set of all trees β with $\#\beta \le p$. There are, in principle, two different recursive approaches:

- *root-oriented*: generate all trees β of order $\#\beta = p$ by first, listing all integer partitions $p - 1 = p_1 + \ldots + p_n$, $n = 1, \ldots, p - 1$, and next, setting $\beta = [\beta_1, \ldots, \beta_n]$ for all trees $\beta_1, \ldots, \beta_n$ of order $\#\beta_1 = p_1 < p, \ldots, \#\beta = p_n < p$. These trees have already been generated by the recursion.
- *leaf-oriented*: Add a leaf to each node of the trees $\hat{\beta} = [\beta_1, \ldots, \beta_n]$ of order $\#\hat{\beta} = p - 1$, increasing thereby the order exactly by one. This can be done recursively by adding a leaf to every node of the subtrees $\beta_1, \ldots, \beta_n$.

The root-oriented approach was chosen by Sofroniou [9] in his Mathematica package `Butcher.m`. It requires an efficient integer partition package and the handling of cartesian products. The leaf-oriented approach is as least as efficient as the other one, but much easier to code:

```
Trees:=proc(order::posint)
  option remember;
  local Replace,AddLeaf,all,trees;

  Replace:=proc(new::tree,old::posint,beta::tree)
    sort(subsop(old=new,beta)); # order-independent...
  end: # Replace

  AddLeaf:=proc(beta::tree)
    option remember;
    local val,child,new;
    val:={sort([[],op(beta)])};
    for child from 1 to nops(beta) do
      new:=AddLeaf(beta[child]);
      val:=val union map(Replace,new,child,beta);
    od;
```

```
      val;
    end: # AddLeaf

    trees:={[]}; all:=[trees];
    to order-1 do
      trees:=`union`(op(map(AddLeaf,trees)));
      all:=[op(all),trees];
    od:
    all;
  end: # Trees
```

Given an order p this procedure generates a list of the sets of trees for each order $q \leq p$, e.g.,

```
> Trees(4);
```

$$[\{[]\}, \{[[]]\}, \{[[[]]], [[], []]\}, \{[[], [], []], [[[[]]]], [[], [[]]], [[[]], []]\}]$$

For instance, the number of trees for each order $p \leq 10$ is given by the entries of the following list:

```
> map(nops,Trees(10));
```

$$[1, 1, 2, 4, 9, 20, 48, 115, 286, 719]$$

The number of order conditions for the order $p = 10$ can thus be obtained by:

```
> `+`(op(%));
```

$$1205$$

Finally, for concrete calculations one has to specify the number s of stages. The following procedure then generates the specific set of equations for *explicit* Runge-Kutta methods:

```
  OrderConditions:=proc(order::posint,stages::posint)
    option remember;
    local eqs,vars,auto,explicit;
    explicit:=seq(seq(a[i,j]=0,j=i..stages),
        i=1..stages);
    vars:=eval(seq(b[i],i=1..stages),
        seq(seq(a[i,j],j=1..stages),i=1..stages),
        seq(c[i],i=1..stages),explicit) minus {0};
    auto:=eval(seq(sum(a[i,j],j=1..stages)=c[i],
        i=1..stages),explicit);
    eqs:=value(Eval(map(TreeOrderCondition,
        `union`(op(Trees(order)))),s=stages));
    eqs:=eval(eval(eqs,explicit),auto);
    eqs,auto,vars;
  end: # OrderConditions
```

This way, we can automatically generate and typeset the order conditions for the classical explicit 4-stage Runge-Kutta methods of order 4:

```
> OrderConditions(4,4): %[1];
```

$$
\begin{aligned}
&\{b_1 + b_2 + b_3 + b_4 = 1,\, b_4\, a_{4,3}\, a_{3,2}\, c_2 = \frac{1}{24},\\
&b_2\, c_2 + b_3\, c_3 + b_4\, c_4 = \frac{1}{2},\, b_2\, {c_2}^3 + b_3\, {c_3}^3 + b_4\, {c_4}^3 = \frac{1}{4},\\
&b_3\, a_{3,2}\, c_2 + b_4\, (a_{4,2}\, c_2 + a_{4,3}\, c_3) = \frac{1}{6},\\
&b_3\, c_3\, a_{3,2}\, c_2 + b_4\, c_4\, (a_{4,2}\, c_2 + a_{4,3}\, c_3) = \frac{1}{8},\\
&b_3\, a_{3,2}\, {c_2}^2 + b_4\, (a_{4,2}\, {c_2}^2 + a_{4,3}\, {c_3}^2) = \frac{1}{12},\\
&b_2\, {c_2}^2 + b_3\, {c_3}^2 + b_4\, {c_4}^2 = \frac{1}{3}\}
\end{aligned}
$$

Remark 1. Even in the more recent literature one can find examples like [4], where order conditions for Runge-Kutta methods are generated by using a computer algebra system to calculate the Taylor expansions of the flow and the discrete flow directly. This approach is typically bound to *scalar* non-autonomous equations, i.e., $d = 1$. Besides being inefficient for higher orders, it is well-known [2] that for $p \geq 5$ additional order conditions for general systems make an appearance, which do not show up in the scalar case.

6. Examples of usage

The following simple procedure tempts to solve the order conditions for a given order p and stage number s by using brute force, i.e., Maple's `solve`-command. To simplify the task, the user is allowed to supply a set `pre` of a priori chosen additional equations and assignments that he thinks to be helpful.

```
RungeKuttaMethod:=proc(p::posint,s::posint,pre::set)
  # explicit methods only
  local conds,auto,vars,eqs,sols,sol,val;
  conds,auto,vars:=OrderConditions(p,s);
  eqs:=conds union auto union pre;
  sols:=solve(eqs,vars);
  val:=NULL;
  for sol in sols do
```

```
      val:=val,[
        a=matrix([seq([seq(eval(a[i,j],sol),j=1..i-1),
                  seq(0,j=i..s)],i=1..s)]),
        b=vector([seq(eval(b[i],sol),i=1..s)]),
        c=vector([seq(eval(c[i],sol),i=1..s)])];
    od:
    val;
  end: # RungeKuttaMethod
```

This way we can simply generate the general explicit 2-stage Runge-Kutta method of order $p = 2$:

```
 >  RungeKuttaMethod(2,2,{b[2]=theta});
```

$$\left[a = \begin{bmatrix} 0 & 0 \\ \frac{1}{2}\frac{1}{\theta} & 0 \end{bmatrix},\ b = [-\theta + 1,\ \theta],\ c = \left[0,\ \frac{1}{2}\frac{1}{\theta}\right]\right]$$

The next example is more demanding. In his book [3, p. 199] Butcher describes an algorithm for the construction of explicit 6-stage methods of order $p = 5$. The choices $c_6 = 1$ and $b_2 = 0$ together with the free parameters c_2, c_3, c_4, c_5 and a_{43} yield a unique method. Butcher provides a two-parameter example by choosing $c_2 = u$, $c_3 = 1/4$, $c_4 = 1/2$, $c_5 = 3/4$, $a_{43} = v$. By just passing this additional information to Maple's `solve`-command we obtain the following solution

```
 >  pre:={c[2]=u,c[3]=1/4,c[4]=1/2,c[5]=3/4,c[6]=1,
 >  b[2]=0,a[4,3]=v}:
 >  RungeKuttaMethod(5,6,pre): %[1];
 >  %%[2],%%[3];
```

$$a = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ u & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{32}\frac{-1+8u}{u} & \frac{1}{32}\frac{1}{u} & 0 & 0 & 0 & 0 \\ -\frac{1}{8}\frac{-2v+1+8vu-4u}{u} & -\frac{1}{8}\frac{2v-1}{u} & v & 0 & 0 & 0 \\ \frac{3}{16}\frac{-v+1-3u+4vu}{u} & \frac{3}{16}\frac{v-1}{u} & \frac{3}{4}-\frac{3}{4}v & \frac{9}{16} & 0 & 0 \\ -\frac{1}{14}\frac{-6v+7+24vu-22u}{u} & -\frac{1}{14}\frac{6v-7}{u} & \frac{12}{7}v & \frac{-12}{7} & \frac{8}{7} & 0 \end{bmatrix}$$

$$b = \left[\frac{7}{90},\ 0,\ \frac{16}{45},\ \frac{2}{15},\ \frac{16}{45},\ \frac{7}{90}\right],\ c = \left[0,\ u,\ \frac{1}{4},\ \frac{1}{2},\ \frac{3}{4},\ 1\right]$$

This result shows that the coefficients a_{51} and a_{52} of Butcher's solution [3, p. 199] are in error, a fact that was already observed by Sofroniou [9] using the Mathematica package `Butcher.m`.

References

1. Butcher, J. C. (1963): Coefficients for the study of Runge-Kutta integration processes, J. Austral. Math. Soc. **3**, 185–201.
2. Butcher, J. C. (1963): On the integration processes of A. Huta, J. Austral. Math. Soc. **3**, 202–206.
3. Butcher, J. C. (1987): *The Numerical Analysis of Ordinary Differential Equations. Runge-Kutta Methods and General Linear Methods*, John Wiley & Sons, Chichester.
4. Gander, W. and Gruntz, D. (1999): Derivation of numerical methods using computer algebra, SIAM Review **41**, 577–593.
5. Hairer, E. (1999): *Numerical Geometric Integration*, Lecture notes of a course given in 1998/99, `http://www.unige.ch/math/folks/hairer/polycop.html`.
6. Hairer, E., Nørsett, S. P., and Wanner, G. (1993): *Solving Ordinary Differential Equations I. Nonstiff Problems*, 2nd Edition, Springer-Verlag, Berlin, Heidelberg, New York.
7. Hairer, E. and Wanner, G. (1974): On the Butcher group and general multi-value methods, Computing **13**, 1–15.
8. Jenks, R. J., (1976): Problem # 11: Generation of Runge-Kutta equations, SIGSAM Bulletin **10**, 6.
9. Sofroniou, M. (1994): Symbolic derivation of Runge-Kutta methods, J. Symb. Comp. **18**, 265–296.