The Electronic Journal of Mathematics and Technology, Volume 10, Number 2, ISSN 1933-2823

Exploring Runge-Kutta formulas with a computer algebra system

Alasdair McAndrew
Alasdair.McAndrew@vu.edu.au
College of Engineering and Science
Victoria University
PO Box 14428, Melbourne 8001
Victoria, Australia

Abstract

Runge-Kutta formulas are some of the workhorses of numerical methods for solving differential equations. However, they are extremely difficult to generate; the algebra involved can be very complicated indeed, and so their derivation is not included in undergraduate numerical texts. It is now standard, following with work of Butcher in the 1960's and 70's, to use the combinatorial theory of trees to simplify the algebra. More recently, however, several authors have shown that it is quite feasible to use a computer algebra system to generate Runge-Kutta formulas. This article shows that, using a computer-based approach, the formulas can be generated with very elementary means, using only the tools of elementary calculus and an open-source computer system to handle the messy algebra. This approach brings a formerly difficult operation into the realm of undergraduate mathematics.

1 Introduction

Unless you are teaching an advanced course in numerical analysis, you are not likely to spend much time discussing Runge-Kutta methods for differential equations. You may derive some simple methods (Euler, Heun), and state without proof the standard fourth-order Runge-Kutta method. You will almost certainly not derive this method, and for very good reason—the algebra involved is frighteningly complicated. One text [9] takes at least four pages of dense algebra to produce the result.

As part of intense decades of research into these methods, Butcher [2, 1, 6] developed some highly sophisticated techniques which showed how the equations which characterize these methods are not random, as they at first appear, but are intimately related to the theory of "rooted trees". A *tree* is a graph (in the combinatorial sense) which is connected and contains no circuits, and *rooted* means that one vertex is specified as the "root". Although there had been some indications before Butcher that rooted trees and Runge-Kutta conditions were connected, it was Butcher's work that brought this theory into fruition, and on which much of the current theory of Runge-Kutta conditions is based.

However, the use of a computer algebra system means it is now quite possible to look at Runge-Kutta conditions anew: in particular to restrict the mathematics to elementary undergraduate calculus and algebra. In particular, we can

1. Derive the equations which characterize Runge-Kutta methods.
2. Use the technique of Gröbner bases to simplify the equations (which are all polynomial).
3. Solve the equations, either in a completely general manner, or more particularly with specified values for some of the variables.

Some of this has previously been done [4]. However, we provide a somewhat different approach:

1. We use only open-source software, so that anybody can experiment and verify our results.
2. We show how to determine Runge-Kutta methods for *autonomous equations*.
3. We show how to develop *embedded formulas*, which consist of two methods of different orders and which share the same coefficients. These are the currently preferred methods.

We are concerned with finding the solution to the initial value problem

$$\frac{dy}{dx} = f(x,y), \quad y(x_0) = y_0$$

where the function f, and the initial values (x_0, y_0) are given. A *numerical solution* consists of a sequence of ordered pairs (x_k, y_k), where y_k is an approximation to the exact value $y(x_k)$. One way is to use the Taylor expansion of $y(x)$, using

$$\begin{aligned}
y' &= f(x,y)\\
y'' &= f_x + f_y\frac{dy}{dx} = f_x + f_y f\\
y''' &= (f_x + f_y f)_x + (f_x + f_y f)_y\frac{dy}{dx}\\
&= f_{xx} + f_{yx}f + f_y f_x + (f_{xy} + f_{yy}f + f_y f_y)f\\
&= f_{xx} + 2f_{xy}f + f_x f_y + f_{yy}(f)^2 + (f_y)^2 f
\end{aligned}$$

and then for a suitably small value of h, given (x_k, y_k) and with $x_{k+1} = x_k + h$ compute an approximation to $y_{k+1} \approx y(x_k + h)$ to the exact value $y(x_{k+1})$.

However, this requires the derivatives of f, which in many cases may have to be computed using an approximation, thus introducing a new source of errors.

The insight of Runge[1] and of Kutta[2], was to realize that as the first derivative of y was equal to f, so other derivatives could be computed by judicious nesting.

For example, suppose we truncate the expansion of the Taylor series expansion of f after the first derivative:

$$f(x+h, y+k) \approx f + hf_x + kf_y. \tag{1}$$

[1]Carl David Tolmé Runge, 1856–1927
[2]Martin Wilhelm Kutta, 1867–1944

Also, truncate the Taylor series for y after the second derivative:

$$y(x+h) \approx y + hf + \frac{h^2}{2}(f_x + f_y f). \tag{2}$$

Note that the expression in parentheses on the right of (2) is very close to that on the right of (1), excepting a term of f. But this can be inserted simply by writing (2) as

$$y(x+h) \approx y + \frac{h}{2}f + \frac{h}{2}(f + hf_x + hf_y f). \tag{3}$$

Comparing the final term with (1) we can write

$$y(x+h) \approx y + \frac{h}{2}f + \frac{h}{2}f(x+h, y+hf). \tag{4}$$

This can be written as a sequence of steps, starting with $y_n \approx y(x_n)$ and with $x_{n+1} = x_n + h$:

$$\begin{aligned} k_1 &= f(x,y) \\ k_2 &= f(x+h, y+hk_1) \\ y_{n+1} &= y_n + \frac{h}{2}(k_1 + k_2). \end{aligned}$$

This is an example of a second-order Runge-Kutta formula, and is equal to a second-order Taylor approximation, but without computing any of the derivatives of f. In general, an n-th order Runge-Kutta formula has the form:

$$\begin{aligned} k_1 &= f(x_n, y_n) \\ k_2 &= f(x + c_2 h, y_n + a_{21} h k_1) \\ k_3 &= f(x + c_3 h, y_n + h(a_{31}k_1 + a_{32}k_2)) \\ &\vdots \\ k_m &= f(x + c_m h, y_n + h(a_{m1}k_1 + a_{m2}k_2 + \cdots + a_{m,m-1}k_{m-1})) \end{aligned}$$

and then

$$y_{n+1} = y_n + h(b_1 k_1 + b_2 k_2 + \cdots + b_m k_m).$$

It is customary to write all the coefficients in a *Butcher array*:

$$\begin{array}{c|ccccc} 0 & & & & & \\ c_2 & a_{21} & & & & \\ c_3 & a_{31} & a_{32} & & & \\ \vdots & & & & & \\ c_m & a_{m1} & a_{m2} & \cdots & a_{m,m-1} & \\ \hline & b_1 & b_2 & \cdots & b_{m-1} & b_m \end{array}$$

These particular Runge-Kutta methods are called *explicit* methods, where at stage i the value k_i is explicitly defined in terms of previously computed values. The above second order method could be written as

$$\begin{array}{c|cc} 0 & & \\ 1 & 1 & \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$$

A very popular fourth-order method (sometimes called "*the* Runge-Kutta method") is given by the array

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

2 Use of Computer Algebra Systems: third-order methods

Before we launch into the use of a CAS, consider a third-order system:

$$\begin{aligned} k_1 &= f(x,y) \\ k_2 &= f(x+c_2h, y+a_{21}hk_1) \\ k_3 &= f(x+c_3h, y+a_{31}hk_1+a_{32}hk_2) \end{aligned}$$

and

$$y_{n+1} = y_n + h(b_1k_1 + b_2k_2 + b_3k_3)$$

and which is to be equivalent to the third-order Taylor polynomial

$$y(x_n+h) \approx y + hf + \frac{h^2}{2}(f_x + f_yf) + \frac{h^3}{6}(f_{xx} + 2f_{xy}f + f_xf_y + f_{yy}(f)^2 + (f_y)^2f).$$

In order to find appropriate coefficients, the expressions for each of the k_i values need to be expanded up to and including the second derivatives; thus:

$$\begin{aligned} k_1 &= f(x,y) \\ k_2 &= f(x,y) + h(c_2f_x + a_{21}k_1f_y) + \frac{h^2}{2}(c_2^2f_{xx} + 2c_2a_{21}k_1f_{xy} + (a_{32}k_1)^2f_{yy}) \\ k_3 &= f(x,y) + h(c_3f_x + (a_{31}k_1 + a_{32}k_2)f_y) \\ &\quad + \frac{h^2}{2}(c_2^2f_{xx} + 2c_2(a_{31}k_1 + a_{32}k_2)f_{xy} + (a_{31}k_1 + a_{32}k_2)^2f_{yy}) \end{aligned}$$

Note that k_3 above is expressed in terms of k_2; this means that the expression for k_2 must be substituted into k_3 wherever it occurs, so that the final expressions for each of the k_i are written using only f and its derivatives, and h.

To equate the Taylor polynomial with the Runge-Kutta values for y_{n+1}, we must have

$$\begin{aligned} y + hf + \frac{h^2}{2}(f_x + f_y f) + \frac{h^3}{6}(f_{xx} + 2f_{xy}f + f_x f_y + f_{yy}(f)^2 + (f_y)^2 f) \\ = y + h(b_1k_1 + b_2k_2 + b_3k_3) \end{aligned}$$

or that

$$b_1k_1 + b_2k_2 + b_3k_3 = f + \frac{h}{2}(f_x + f_y f) + \frac{h^2}{6}(f_{xx} + 2f_{xy}f + f_x f_y + f_{yy}(f)^2 + (f_y)^2 f). \tag{5}$$

We thus need to find values of all the unknown coefficients (the a, b and c values), for which

$$b_1k_1 + b_2k_2 + b_3k_3 - f - \frac{h}{2}(f_x + f_y f) - \frac{h^2}{6}(f_{xx} + 2f_{xy}f + f_x f_y + f_{yy}(f)^2 + (f_y)^2 f) = 0.$$

Collecting all the terms together, and working through all the algebra to expand k_2 and k_3 fully, we end up with

$$\begin{aligned} &\left(\frac{a_{32}^2 b_3}{2} + a_{31}a_{32}b_3 + \frac{a_{31}^2 b_3}{2} + \frac{a_{21}^2 b_2}{2} - \frac{1}{6}\right) f_{yy}h^2 f^2 + \left(a_{21}a_{32}b_3 - \frac{1}{6}\right) f_y^2 h^2 f \\ &\quad + \left(a_{32}b_3c_3 + a_{31}b_3c_3 + a_{21}b_2c_2 - \frac{1}{3}\right) f_{xy}h^2 f \\ &\quad + \left(a_{32}b_3 + a_{31}b_3 + a_{21}b_2 - \frac{1}{2}\right) f_y h f + (b_3 + b_2 + b_1 - 1) f \\ &\quad + \left(a_{32}b_3c_2 - \frac{1}{6}\right) f_x f_y h^2 + \left(\frac{b_3c_3^2}{2} + \frac{b_2c_2^2}{2} - \frac{1}{6}\right) f_{xx}h^2 \\ &\quad + \left(b_3c_3 + b_2c_2 - \frac{1}{2}\right) f_x h \\ &\quad = 0. \end{aligned} \tag{6}$$

Once this has been done, the values we want are the solutions to the non-linear equations:

$$\begin{aligned} a_{32}^2 b_3 + 2a_{31}a_{32}b_3 + a_{31}^2 b_3 + a_{21}^2 b_2 &= 1/3 \\ a_{21}a_{32}b_3 &= 1/6 \\ a_{32}b_3c_3 + a_{31}b_3c_3 + a_{21}b_2c_2 &= 1/3 \\ a_{32}b_3 + a_{31}b_3 + a_{21}b_2 &= 1/2 \\ b_3 + b_2 + b_1 &= 1 \\ a_{32}b_3c_2 &= 1/6 \\ b_3c_3^2 + b_2c_2^2 &= 1/3 \\ b_3c_3 + b_2c_2 &= 1/2 \end{aligned}$$

It can be seen—even without attempting to solve these equations—that the algebra involved is extremely involved, messy, and without any apparent order. Given the apparent lack of structure to these equations, it is all the more remarkable that Butcher was able to relate these equations to a different area of mathematics, and hence bring some order to the chaos.

Our approach, though, will be to simply create the equations from scratch, and solve them, using a computer algebra system. Our choice will be the open-source Sage [10] to perform the algebraic computations. However, much of the initial calculus computations will devolve to the open-source system Maxima [7], which is the current descendant of the venerable system Macsyma; and which has very powerful calculus and algebra functionality. As Sage includes Maxima within it, we can use Maxima initially to create the derivatives and the functions, and use the algebraic power of Sage to solve the equations. We will present our work with monospaced input and typeset output, similar to the appearance of using Sage in a browser-based "notebook" [3] and with Maxima cells..

To start, we need to create a formal function f and its derivatives. In Sage, objects maintain their types, so in a variable assignment such as "z = `f.diff(x)`", assuming `f` to be a function previously defined, the result `z` will be either a Sage object or a Maxima object depending on the type of `f`. We thus start by introducing three Maxima variables:

```
x = maxima('x')
y = maxima('y')
f = maxima('f')
```

Now each of these variables will automatically have access to the Maxima sub-system, and so we can create the derivatives. In order to prevent unnecessary high derivatives of $y(x)$, we shall replace y' with f as soon as it appears.

```
y.depends(x)
f.depends([x,y])
f1 = f.diff(x).subst("diff(y,x)=f")
f2 = f1.diff(x).subst("diff(y,x)=f")
f3 = f2.diff(x).subst("diff(y,x)=f")
```

Sage doesn't automatically display the results of a variable assignment, but we can check out the first two:

```
f1,f2
```

$$f\left(\frac{\partial}{\partial y} f\right) + \frac{\partial}{\partial x} f,$$
$$f\left(f\left(\frac{\partial^2}{\partial y^2} f\right) + \frac{\partial^2}{\partial x\,\partial y} f\right) + \frac{\partial}{\partial y} f\left(f\left(\frac{\partial}{\partial y} f\right) + \frac{\partial}{\partial x} f\right) + \frac{\partial^2}{\partial x^2} f + f\left(\frac{\partial^2}{\partial x\,\partial y} f\right)$$

In order to make the algebra more manageable, we shall subsitute each derivative with a variable name, first introducing those variables into the namespace. Sage is based on the programming language Python, in which any variable must be named before it can be used. These variables will accrue their appropriate types later.

```
var('h,F,Fx,Fy,Fxx,Fxy,Fyy,Fxxx,Fxxy,Fxyy,Fyyy,a21,\
a31,a32,a41,a42,a43,b1,b2,b3,b4,c2,c3,c4,')
dsubs = " 'diff(f,x,3)=Fxxx, 'diff(f,x,2,y,1)=Fxxy,\
  'diff(f,x,1,y,2)=Fxyy,'diff(f,y,3)=Fyyy,\
  'diff(f,x,2)=Fxx, 'diff(f,y,2)=Fyy,\
  'diff(f,x,1,y,1)=Fxy,'diff(f,x,1)=Fx,\
  'diff(f,y,1)=Fy, f=F"
F1 = f1.subst(dsubs)
F2 = f2.subst(dsubs)
F3 = f3.subst(dsubs)
```

As before, their values can be checked:

```
F1, F2, F3
```

$$\begin{aligned}&Fy\,F+Fx\\&F\ (Fyy\,F+Fxy)+Fy\ (Fy\,F+Fx)+Fxy\,F+Fxx\\&F\ (F\ (Fyyy\,F+Fxyy)+Fyy\ (Fy\,F+Fx)+Fxyy\,F+Fxxy)\\&\qquad+Fy\ (F\ (Fyy\,F+Fxy)+Fy\ (Fy\,F+Fx)+Fxy\,F+Fxx)\\&\qquad+2\ (Fy\,F+Fx)\ (Fyy\,F+Fxy)+Fxy\ (Fy\,F+Fx)\\&\qquad+F\ (Fxyy\,F+Fxxy)+Fxxy\,F+Fxxx\end{aligned}$$

Now we introduce the Taylor polynomial up to the third derivative (this is for a third-order method), and this corresponds to the right hand side of equation (5):

```
T = F + h/2*F1 + h^2/6*F2; T
```

$$\frac{h^2\ (F\ (Fyy\,F+Fxy)+Fy\ (Fy\,F+Fx)+Fxy\,F+Fxx)}{6}+\frac{h\ (Fy\,F+Fx)}{2}+F$$

In order to compute the k_i values, we need a Taylor series expansion up to the second derivative:

$$f(x+a,y+b)=f(x,y)+af_x+bf_y+\frac{1}{2}\left(a^2f_{xx}+2abf_{xy}+b^2f_{yy}\right)$$

where the subscripts represent the usual partial derivatives. Given the above substitutions, we will call this expansion $\mathrm{Tay}(a,b)$. Since we are at the moment dealing with Maxima variables, we will define Tay as a Maxima function:

```
Tay = maxima.function('a,b','F+Fx*a+Fy*b\
  +(Fxx*a^2+2*Fxy*a*b+Fyy*b^2)/2')
```

Since the k_i values are nested, we do not want the powers of h increasing: we are only interested in coefficients for which the powers of h are 2 or less. Maxima has a handy trick here:

```
maxima("tellrat(h^3)")
maxima("algebraic:true")
```

This means that for every rational expansion, all powers of h that are three or more will be set equal to zero. Now we can create the k_i values:

```
k1 = Tay(0, 0)
k2 = Tay(c2*h, a21*h*k1)
k3 = Tay(c3*h, h*a31*k1 + h*a32*k2)
```

(The expressions, certainly for k_3, are too long to display). Now we can create the left hand side of equation (5):

```
RK = b1*k1 + b2*k2 + b3*k3
```

The next step is to create the expression on the left hand side of equation (6); we can do this by collecting all the terms involving F and its derivatives. The Maxima command "`collectterms`" provides just this functionality.

```
d = (T-RK).ratexpand().collectterms(Fyy,Fxy,Fxx,Fy,Fx,F,h)
```

This expression is too long to print, but we can extract the coefficients from it, which are the equations we want:

```
eqs = [xx.inpart(1) for xx in d.args()]
eqs
```

$$\left[-\frac{a_{32}{}^2\,b_3}{2}-a_{31}\,a_{32}\,b_3-\frac{a_{31}{}^2\,b_3}{2}-\frac{a_{21}{}^2\,b_2}{2}+\frac{1}{6},\quad \frac{1}{6}-a_{21}\,a_{32}\,b_3,\right.$$
$$-a_{32}\,b_3\,c_3-a_{31}\,b_3\,c_3-a_{21}\,b_2\,c_2+\frac{1}{3},\quad -a_{32}\,b_3-a_{31}\,b_3-a_{21}\,b_2$$
$$\left.+\frac{1}{2},\quad -b_3-b_2-b_1+1,\frac{1}{6}-a_{32}\,b_3\,c_2,-\frac{b_3\,c_3{}^2}{2}-\frac{b_2\,c_2{}^2}{2}+\frac{1}{6},-b_3\,c_3-b_2\,c_2+\frac{1}{2}\right]$$

Each expression in this list may be considered as the left hand side of an equation which will be set equal to zero. These expressions are all Maxima objects, but to access the algebraic power of Sage, they need to be lifted out of the Maxima sub-system. We will use the Sage "`repr`" command, which produces a string representation of an object, and we will evaluate those strings into Sage expressions.

```
eqs2 = [sage_eval(repr(xx),locals=locals()) for xx in eqs]
```

In Sage an expression can be entered directly in the `solve` command, in which it is automatically assumed to be an equation set equal to zero. These equations can be solved in terms of c_2 and c_3:

```
sols = solve(eqs2,[a21,a31,a32,b1,b2,b3])
sols
```

$$\left[\left[a_{21}=c_2, a_{31}=\frac{3\,(c_2^2-c_2)c_3+c_3^2}{3\,c_2^2-2\,c_2}, a_{32}=\frac{c_2c_3-c_3^2}{3\,c_2^2-2\,c_2}, b_1=\frac{3\,(2\,c_2-1)c_3-3\,c_2+2}{6\,c_2c_3},\right.\right.$$
$$\left.\left.b_2=-\frac{3\,c_3-2}{6\,(c_2^2-c_2c_3)}, b_3=\frac{3\,c_2-2}{6\,(c_2c_3-c_3^2)}\right]\right]$$

These are standard expressions, and are found, for example, in Butcher [1]. In general, Sage solutions are given as a list of lists. In our case there is only one solution, which can be isolated with

```
sols = sols[0]
```

given that in Sage lists are indexed starting at zero. Note that the first item tells us that $a_{21} = c_2$. Adding the next two items shows a similar relation for c_3:

```
(sols[1] + sols[2]).simplify_rational()
```

$$a_{31} + a_{32} = c_3$$

It can in fact be shown that for any Runge-Kutta method, each c_k value is equal to the sum of the corresponding a_{ki} values:

$$c_k = a_{k1} + a_{k2} + \cdots a_{k,k-1}.$$

This is known as the *row-sum condition*, and may be assumed for any computation with Runge-Kutta coefficients.

We can now find particular solutions by substituting values for c_2 and c_3 (such that the denominators are all non-zero, which means that c_2 and c_3 must be different), for example:

```
[xx.subs(c2=-1,c3=1) for xx in sols]
```

$$\left[a_{21} = (-1), a_{31} = \left(\frac{7}{5}\right), a_{32} = \left(-\frac{2}{5}\right), b_1 = \left(\frac{2}{3}\right), b_2 = \left(-\frac{1}{12}\right), b_3 = \left(\frac{5}{12}\right)\right]$$

Two other substitutions are:

```
[xx.subs(c2=1/2,c3=1) for xx in sols]
```

$$\left[a_{21} = \left(\frac{1}{2}\right), a_{31} = (-1), a_{32} = 2, b_1 = \left(\frac{1}{6}\right), b_2 = \left(\frac{2}{3}\right), b_3 = \left(\frac{1}{6}\right)\right]$$

```
[xx.subs(c2=1/2,c3=1) for xx in sols]
```

$$\left[a_{21} = \left(\frac{1}{3}\right), a_{31} = 0, a_{32} = \left(\frac{2}{3}\right), b_1 = \left(\frac{1}{4}\right), b_2 = 0, b_3 = \left(\frac{3}{4}\right)\right]$$

These last two can be written into Butcher arrays as follows:

$$\begin{array}{c|ccc} 0 & & & \\ \frac{1}{2} & \frac{1}{2} & & \\ 1 & -1 & 2 & \\ \hline & \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \end{array}, \qquad \begin{array}{c|ccc} 0 & & & \\ \frac{1}{3} & \frac{1}{3} & & \\ \frac{2}{3} & 0 & \frac{2}{3} & \\ \hline & \frac{1}{3} & 0 & \frac{3}{4} \end{array}$$

and are known as *Kutta's third-order method* and *Heun's third-order method* respectively.

3 Fourth-order methods

Having set up the ground work, fourth-order methods can be found similarly; with suitable changes to some of the entries to allow for the higher order. With each of `f1, f2, f3, F1, F2, F3` as before, the commands (given with no outputs) will be:

```
maxima("tellrat(h^4)")
T = F + h/2*F1 + h^2/6*F2 + h^3/24*F3
Tay = maxima.function('a,b','F+Fx*a+Fy*b\
      +(Fxx*a^2+2*Fxy*a*b+Fyy*b^2)/2\
      +(Fxxx*a^3+3*Fxxy*a^2*b+3*Fxyy*a*b^2+Fyyy*b^3)/6')
k1 = Tay(0, 0)
k2 = Tay(c2*h, a21*h*k1)
k3 = Tay(c3*h, h*a31*k1 + h*a32*k2)
k4 = Tay(c4*h, h*a41*k1 + h*a42*k2 + h*a43*k3)

RK = b1*k1 + b2*k2 + b3*k3 + b4*k4
d = (T-RK).ratexpand()\
     .collectterms(Fyyy,Fxyy,Fxxy,Fxxx,Fyy,Fxy,Fxx,Fy,Fx,F,h)
eqs = [xx.inpart(1) for xx in d.args()]
```

At this stage, with no simplification, we will have a list of 19 equations. Before attempting to simplify the equations, we first introduce the row-sum condition:

```
eqs2 = [xx.subst("a21=c2,a31=c3-a32,a41=c4-a42-a43").expand()\
     .collectterms(b1,b2,b3,b4) for xx in eqs]
```

and transform the set of equations out of Maxima and into Sage:

```
eqss = [sage_eval(repr(xx),locals=locals()) for xx in eqs2]
```

Now we can create a polynomial ring in which all the computations will be done, and in the polynomial ring compute the reduced Gröbner basis of the ideal generated by the equations:

```
R = PolynomialRing(QQ,'a21,a31,a32,a41,a42,a43,b1,b2,b3,b4,c2,
...: c3,c4',order='lex')
Id = R.ideal(eqss)
ib = Id.interreduced_basis()
```

The "`interreduced_basis`" command simplifies the 19 equations obtained from the initial computations to a set of eight equations, which is a minimal Gröbner basis in the formal sense that no smaller set can generate the initial polynomial ideal. See for example [11] for an elementary and readable account of Gröbner bases and reduced Gröbner bases. The eight equations thus obtained are as follows:

$$a_{32}a_{43}b_4c_2 - \frac{1}{24} \tag{7.1}$$

$$a_{32}b_3c_2 + a_{42}b_4c_2 + a_{43}b_4c_3 - \frac{1}{6} \tag{7.2}$$

$$a_{42}b_4c_2c_3 - a_{42}b_4c_2c_4 + a_{43}b_4c_3^2 - a_{43}b_4c_3c_4 - \frac{1}{6}c_3 + \frac{1}{8} \tag{7.3}$$

$$a_{43}b_4c_2c_3 - a_{43}b_4c_3^2 - \frac{1}{6}c_2 + \frac{1}{12} \tag{7.4}$$

$$b_1 + b_2 + b_3 + b_4 - 1 \tag{7.5}$$

$$b_2c_2 + b_3c_3 + b_4c_4 - \frac{1}{2} \tag{7.6}$$

$$b_3c_2c_3 - b_3c_3^2 + b_4c_2c_4 - b_4c_4^2 - \frac{1}{2}c_2 + \frac{1}{3} \tag{7.7}$$

$$b_4c_2c_3c_4 - b_4c_2c_4^2 - b_4c_3c_4^2 + b_4c_4^3 - \frac{1}{2}c_2c_3 + \frac{1}{3}c_2 + \frac{1}{3}c_3 - \frac{1}{4} \tag{7.8}$$

As we have seen previously, we may consider each of these expressions as the left hand sides of equations set equal to zero. And this new system of equations can be easily solved in terms of c_i. We first note that $c_4 = 1$; although this can be shown analytically, we can easily demonstrate it using our reduced Gröbner basis:

```
(ib*R).reduce(c4-1)
```

$$0$$

The b_i values can now be obtained by noting that the last four equations (7.5), (7.6), (7.7), (7.8) are linear in b_i:

```
bsols = solve([ib[i].subs(c4=1) for i in [4,5,6,7]],[b1,b2,b3,b4],\
    solution_dict=True)[0]
bsols = {xx: factor(yy) for xx, yy in bsols.items()}
```

$$\left\{b_2 : \frac{2\,c_3 - 1}{12\,(c_2 - c_3)(c_2 - 1)c_2}, b_1 : \frac{6\,c_2c_3 - 2\,c_2 - 2\,c_3 + 1}{12\,c_2c_3},\right.$$
$$\left. b_4 : \frac{6\,c_2c_3 - 4\,c_2 - 4\,c_3 + 3}{12\,(c_2 - 1)(c_3 - 1)}, b_3 : -\frac{2\,c_2 - 1}{12\,(c_2 - c_3)(c_3 - 1)c_3}\right\}$$

(Note that in Sage lists the indices start at 0, so that our equation numbers (7.5), (7.6), (7.7), (7.8) correspond to Sage list indices 4, 5, 6, 7.)

These are standard results [1]. From these we can use the equations (7.2, (7.3) and (7.4) to compute the a_{ij} values:

```
asols = solve([SR(ib[i]).subs(c4=1).subs(bsols) for i in [1,2,3]],\
    [a32,a42,a43],solution_dict=True)[0]
asols = {xx: factor(yy) for xx, yy in asols.items()}
```

$$\left\{a_{43} : \frac{(2\,c_2-1)(c_2-1)(c_3-1)}{(6\,c_2c_3-4\,c_2-4\,c_3+3)(c_2-c_3)c_3}, \right.$$
$$\left. a_{42} : -\frac{(4\,c_3^2-c_2-5\,c_3+2)(c_2-1)}{2\,(6\,c_2c_3-4\,c_2-4\,c_3+3)(c_2-c_3)c_2}, a_{32} : \frac{(c_2-c_3)c_3}{2\,(2\,c_2-1)c_2}\right\}$$

These values satisfy the first equation (7.1):

```
(asols[a32]*asols[a43]*bsols[b4]*c2).rational_simplify()
```

$$\frac{1}{24}$$

Note that because of the factor $c_2 - c_3$ in the denominators of some of these expressions, we cannot substitute equal values for c_2 and c_3. In order to develop a fourth-order method for which $c_2 = c_3$ we need to go back a few steps:

```
var('u')
eqss = [sage_eval(repr(xx),locals=locals()).subs(c2=u,c3=u,c4=1)
...:  for xx in eqs2]
R.<a21,a31,a32,a41,a42,a43,b1,b2,b3,b4,u> = PolynomialRing(QQ)
Id = R.ideal(eqss)
ib = Id.interreduced_basis(); ib
```

$$\left[a_{32}a_{43}-\frac{1}{2}, a_{32}b_3-\frac{1}{6}, a_{42}+a_{43}-1, b_1-\frac{1}{6}, b_2+b_3-\frac{2}{3}, b_4-\frac{1}{6}, u-\frac{1}{2}\right]$$

These can be solved to produce:

$$u=\frac{1}{2},\quad b_1=\frac{1}{6},\quad b_2=r_1,\quad b_3=\frac{2}{3}-r_1,\quad b_4=\frac{1}{6},\quad a_{32}=\frac{1}{2(2-3r_1)},$$
$$a_{42}=3r_1-1,\quad a_{43}=2-3r_1.$$

These values can be written into the following Butcher array:

$$\begin{array}{c|cccc}
0 & & & & \\
\frac{1}{2} & \frac{1}{2} & & & \\
\frac{1}{2} & 0 & \frac{1}{2} & & \\
1 & 0 & 3r_1-1 & 2-3r_1 & \\
\hline
 & \frac{1}{6} & r_1 & \frac{2}{3}-r_1 & \frac{1}{6}
\end{array}$$

Putting $r_1 = 1/3$ produces the classic Runge-Kutta fourth-order method.

4 Use of autonomy

Much of the computations in the previous sections can be simplified by noting that we do not in fact need to include the c_i values in any equations list, as their values can be determined from the a_{ij} values. Since c_i only appear in the computation of k_i, all the computations can be simplified by considering only differential equations for the form

$$y' = f(y)$$

in which f does not depend explicitly on x; such differential equations are said to be *autonomous*. This leads to greatly simplified forms for the higher derivatives of f:

```
f.depends(y)
```

Create f_i and F_i as before, but note the values of F_i:

```
F1, F2, F3
```

$$\left(Fy\,F, \quad Fyy\,F^2 + Fy^2\,F, \quad Fyyy\,F^3 + 4\,Fy\,Fyy\,F^2 + Fy^3\,F\right)$$

Since we need not consider any partial derivatives of f which include x, the following commands can be used:

```
maxima("tellrat(h^4)")
T = F + h/2*F1 + h^2/6*F2 + h^3/24*F3
Tay = maxima.function('a,b','F+Fy*b+Fyy*b^2/2+Fyyy*b^3/6')
k1 = Tay(0, 0)
k2 = Tay(0, a21*h*k1)
k3 = Tay(0, h*a31*k1 + h*a32*k2)
k4 = Tay(0, h*a41*k1 + h*a42*k2 + h*a43*k3)
```

Note that we need to include an extra dummy variable in the “Tay” function; the systems in their current forms prevent a Maxima function of one variable being used in this way. The next few commands are similar to those above.

```
RK = b1*k1 + b2*k2 + b3*k3 + b4*k4
d = (T-RK).ratexpand().collectterms(Fyyy,Fyy,Fy,F,h)
eqs = [xx.inpart(1) for xx in d.args()]
eqss = [sage_eval(repr(xx),locals=locals()) for xx in eqs]
```

The equations are still quite long and complicated; as before they can be simplified by introducing the row-sum conditions, and putting $c_4 = 1$:

```
eqs2 = [xx.subs(a31=c3-a32,a41=1-a42-a43).expand() for xx in eqss]
```

to produce:

$$
\begin{aligned}
&-\frac{1}{6}\,b_2c_2^3-\frac{1}{6}\,b_3c_3^3-\frac{1}{6}\,b_4c_4^3+\frac{1}{24}\\
&-\frac{1}{2}\,a_{32}b_3c_2^2-\frac{1}{2}\,a_{42}b_4c_2^2-a_{32}b_3c_2c_3-\frac{1}{2}\,a_{43}b_4c_3^2-a_{42}b_4c_2c_4-a_{43}b_4c_3c_4+\frac{1}{6}\\
&-\frac{1}{2}\,b_2c_2^2-\frac{1}{2}\,b_3c_3^2-\frac{1}{2}\,b_4c_4^2+\frac{1}{6}\\
&-a_{32}a_{43}b_4c_2+\frac{1}{24}\\
&-a_{32}b_3c_2-a_{42}b_4c_2-a_{43}b_4c_3+\frac{1}{6}\\
&-b_2c_2-b_3c_3-b_4c_4+\frac{1}{2}\\
&-b_1-b_2-b_3-b_4+1
\end{aligned}
$$

These expressions are not quite the same as the expressions (7.1) to (7.8) from the previous section, but they can be solved similarly as equations set equal to zero:

```
bsols = solve([eqs2[i] for i in [0,2,5,6]],[b1,b2,b3,b4],\
    solution_dict=True)[0]
asols = solve([eqs2[i].subs(bsols) for i in [1,3,4]],[a32,a42,a43],\
    solution_dict=True)[1]
```

to produce the same results as before.

For this autonomous approach, there has been no need to simplify a large set of nineteen equations to a smaller set by involving the machinery of Gröbner bases; the equation set was optimally small at the start.

To solve these equations with $c_2 = c_3$, we need to be a bit careful; the attempt

```
var('u')
eqs2 = [xx.subs(a21=u,a31=u-a32,a41=1-a42-a43).expand()
...:  for xx in eqss]
bsols = solve([eqs2[i] for i in [0,2,5,6]],[b1,b2,b3,b4],
...: solution_dict=True)
```

will not work: as b_2 and b_3 have the same coefficients in all the equations, the determinant of the matrix of coefficients is zero. So we leave b_2 out:

```
bsols = solve([eqs2[i] for i in [0,2,6]],[b1,b3,b4],\
....: solution_dict=True)[0]
asols = solve([eqs2[i].subs(bsols) for i in [1,3,4]],[a32,a42,a43],\
....: solution_dict=True)[0]
```

The results will be expressed in terms of the parameters b_2 and u. Substituting $b_2 = 1/3$ and $u = 1/2$ will produce the standard fourth-order method.

5 Embedded formulas

Many applications now use *embedded Runge-Kutta methods*, in which two methods share the same coefficients. Generally the order of the methods differs by one, so we might have a fifth order method, from which the coefficients can be used to build a fourth-order method. Then the differences between the results of these methods can be used to adjust the step size h for the next iteration.

Methods of order $4(3)$ and the theory behind them are well known [1, 5], but we can show that it is very easy to construct such a method. Starting with Kutta's 3/8 method, we need to find coefficients $\hat{b}_1, \hat{b}_2, \hat{b}_3, \hat{b}_4, \hat{b}_5$ so that with the extra stage

$$k_5 = f(x_n + c_5 h, h(b_1 k_1 + b_2 k_2 + b_3 k_3 + b_4 k_4))$$

the value of y_{n+1} obtained with

$$y_{n+1} = y_n + h(\hat{b}_1 k_1 + \hat{b}_2 k_2 + \hat{b}_3 k_3 + \hat{b}_4 k_4 + \hat{b}_5 k_5)$$

will be accurate to order three.

This is easily done, assuming the autonomous approach. We enter the fourth-order values, and for simplicity we use s_i in place of $\hat{b}_i$.

```
a21,a31,a32,a41,a42,a43 = 1/3,-1/3,1,1,-1,1
b1,b2,b3,b4 = 1/8,3/8,3/8,1/8
var('s1,s2,s3,s4,s5')
```

We set up the third-order conditions as previously, but this time with five stages:

```
T = F + h/2*F1 + h^2/6*F2
maxima("tellrat(h^3)")
maxima("algebraic:true")
Tay = maxima.function('a,b','F+Fy*b+Fyy*b^2/2')

k1 = Tay(0, 0)
k2 = Tay(0, h*a21*k1)
k3 = Tay(0, h*a31*k1 + h*a32*k2)
k4 = Tay(0, h*a41*k1 + h*a42*k2 + h*a43*k3)
k5 = Tay(0, h*b1*k1 + h*b2*k2 + h*b3*k3 + h*b4*k4)
RK = s1*k1 + s2*k2 + s3*k3 + s4*k4 + s5*k5
```

Now we extract the coefficients of `T-RK` as equations to be solved.

```
d = (T-RK).ratexpand().collectterms(Fyy,Fy,F,h)
eqs = [xx.subst('h=1,F=1,Fy=1,Fyy=1') for xx in d.args()]
eqs2 = [sage_eval(repr(xx),locals=locals()) for xx in eqs]
```

These equations are easily solved:

```
solve(eqs2,[s1,s2,s3,s4,s5])
```

$$\left[\left[s_1 = -\frac{1}{4}r_1 + \frac{1}{8}, s_2 = \frac{3}{4}r_1 + \frac{3}{8}, s_3 = -\frac{3}{4}r_1 + \frac{3}{8}, s_4 = -\frac{3}{4}r_1 + \frac{1}{8}, s_5 = r_1\right]\right]$$

and the extra parameter can be set to any value we like, for example $r_1 = 1$.

```
[xx.subs(r1=1) for xx in sols[0]]
```

$$\left[s_1 = \left(-\frac{1}{8}\right), s_2 = \left(\frac{9}{8}\right), s_3 = \left(-\frac{3}{8}\right), s_4 = \left(-\frac{5}{8}\right), s_5 = 1\right]$$

This values can be written into a Butcher array as follows:

$$\begin{array}{c|ccccc}
0 & & & & & \\
\frac{1}{3} & \frac{1}{3} & & & & \\
\frac{2}{3} & -\frac{1}{3} & 1 & & & \\
1 & 1 & -1 & 1 & & \\
\hline
 & \frac{1}{8} & \frac{3}{8} & \frac{3}{8} & \frac{1}{8} & \\
 & -\frac{1}{8} & \frac{9}{8} & -\frac{3}{8} & -\frac{5}{8} & 1
\end{array}$$

for an embedded $4(3)$ method. If we choose the parameter r_1 so that $s_4 = 0$; that is $r_1 = 1/6$, we obtain the following values:

$$\left[s_1 = \left(\frac{1}{12}\right), s_2 = \left(\frac{1}{2}\right), s_3 = \left(\frac{1}{4}\right), s_4 = 0, s_5 = \left(\frac{1}{6}\right)\right]$$

6 Conclusions

The literature on Runge-Kutta methods and associated mathematics is vast, Butcher [1] lists many hundreds of references. However, much of this material is directed at the specialist researcher. The various articles which use computer algebra systems in an attempt to sidestep the specialist material have tended to use commercial systems, which puts the material out of bounds for people who don't use (or can't afford) those systems. As long ago as 1993, Joachim Neubüser, the creator of the GAP package for group theory (and which is part of Sage) [8] deplored the fact that mathematical theorems are open to everybody to use, but mathematics using a computer system was not, unless that system was open-source. Our article has attempted to allow the general non-specialist reader to experiment and explore some of the basic properties of Runge-Kutta methods, using only open-source systems.

Since the computer algebra system handles all the algebraic hard work, the difficulty is mainly setting up the system at the beginning, and eliminating unnecessary higher derivatives when they are not needed. The level of calculus and algebra is not in fact particularly difficult—what is difficult is the complexity of the expressions, and the huge equations which derive from them. Using a CAS to do the "heavy lifting" means that much of this work falls within the purview of undergraduate material, and indeed could make a good undergraduate project, or an exploration for the interested non-specialist.

References

[1] John C. Butcher. *Numerical Methods for Ordinary Differential Equations*. John Wiley & Sons, 2008.

[2] John C. Butcher. *The Numerical Analysis of Ordinary Differential Equations: Runge-Kutta and General Linear Methods*. Wiley-Interscience, 1987.

[3] Burçin Eröcal and William A. Stein. "The Sage project: unifying free mathematical software to create a viable alternative to Magma, Maple, Mathematica and MATLAB". In: *Mathematical Software–ICMS 2010*. Springer, 2010, pp. 12–27.

[4] Walter Gander and Dominik Gruntz. "Derivation of numerical methods using computer algebra". In: *SIAM review* 41.3 (1999), pp. 577–593.

[5] Ernst Hairer, Syvert P. Nørsett, and Gerhard Wanner. *Solving Ordinary Differential Equations I: Non-Stiff Systems*. Springer-Verlag, Berlin, 1987.

[6] John D. Lambert. *Numerical Methods for Ordinary Differential Systems: the Initial Value Problem*. John Wiley & Sons, Inc., 1991.

[7] Maxima. *Maxima, a Computer Algebra System. Version 5.30.0*. `http://maxima.sourceforge.net/`, 2013.

[8] Joachim Neubüser. "An invitation to computational group theory". In: *Groups' 93–Galway/St. Andrews, volume 212 of London Math. Soc. Lecture Note Ser*. Citeseer. Aug. 1995, pp. 457–475.

[9] Antony Ralston and Philip Rabinowitz. *A First Course in Numerical Analysis*. Dover Publications, 1965.

[10] William A. Stein et al. *Sage Mathematics Software (Version 6.0)*. `http://www.sagemath.org`. The Sage Development Team. 2014.

[11] Bernd Sturmfels. "What is a Gröbner basis?" In: *Notices of the AMS* 52.10 (Nov. 2005), pp. 2–3.