Stepsize Selection for Ordinary Differential Equations

Fred T. Krogh
Math à la Carte, Inc.

This note offers a new approach based on a least squares fit to past data in order to select the stepsize when solving an ordinary differential equation. The approach used may have applicability to other situations where one wants to repeatedly make short term predictions given somewhat noisy data. Additional ad hoc rules help significantly for reliability and efficiency. Comparisons with some Runge-Kutta codes, an Adams code, and an extrapolation code are also included.

Categories and Subject Descriptors: G.4 [**Mathematical Software**]; G.1.7 [**Numerical Analysis**]: Ordinary Differential Equations

General Terms: Algorithms, Performance

Additional Key Words and Phrases: ODE, stepsize, prediction

1. INTRODUCTION

Historically, most codes for ODE's have selected the next stepsize based on an estimate of the local error. In our work on Adams codes, see [Krogh 1974, p.66] for an early example, we have found it useful to use past history to select the next stepsize based on what the error on the next step is *expected* to be, rather than simply using the error on the current step. The most notable recent work using past history to influence stepsize selection is that of [Söderlind 2003] / [Gustafsson 1991]. We believe what is provided here is easier to understand than either, it has a low computational cost, and as a bonus, appears to work better. Additional rules are suggested which help provide better efficiency and reliability for the particular code examined here. Other work making use of past history include [Zonnefeld 1964] [Deuflhard 1984], [Shampine and Gladwell 1996], and other references cited in these papers.

The work described in this paper was carried out at Math à la Carte., Inc.
Author's current address: Math à la Carte, Inc., P.O. Box 616, Tujunga, CA 91043-0616; email: fkrogh@mathalacarte.com

2. SELECTING THE STEPSIZE

Our model is given by

$$\left|\frac{\text{estimated error}}{\text{requested accuracy}}\right| = \rho_n \approx e^{\phi_n} h_n^p, \tag{1}$$

where p is the order of the method, h_n is the stepsize on step n, and ϕ is a slowly varying function of n. (Note that ϕ_n should be a smoother function of n with this definition for ρ_n, rather than by defining ρ_n as the estimated error; e.g., think of $y' = -y$ where with a relative error test we want a constant stepsize, and thus for ϕ to be constant.) We need ϕ in the exponential to insure that our multiplier does not go negative, but this is also better for tracking rapid changes. We assume that the requested accuracy is the accuracy desired, including any multiplier the code uses to modify the accuracy requested by the user, and thus we desire a value of $\rho_n = 1$. With an estimated value for ϕ_{n+1} we obtain $h_{n+1} = e^{-\phi_{n+1}/p}$.

Here, we derive the formulas to use if ϕ_{n+1-k} is fit by a quadratic or linear polynomial in k. Because error estimates are not perfectly smooth, some type of approximation is desired, and we have used least squares. We assume that the "observations" are to have weights of $w^{(n+1-k)/2}$ for some value of w, $0 < w < 1$, where in a code w is a constant. A smaller value of w gives less weight to the past history while larger values will give the past values a greater influence.

To keep computations independent of n, we solve a least squares system with an infinite number of rows to find an approximation of the form $\phi_{n+1-k} = a + kb + \frac{1}{2}k(k+1)c$. (The factor of c is selected to simplify the updates for r_3 below.) An initialization process gets us started with a system of this form.

$$\begin{bmatrix} 1 & 1 & 1 \\ w^{1/2} & 2w^{1/2} & 3w^{1/2} \\ & \cdots & \\ w^{(k-1)/2} & kw^{(k-1)/2} & \frac{1}{2}k(k+1)w^{(k-1)/2} \\ & \cdots & \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} \cong \begin{bmatrix} \phi_n \\ w^{1/2}\phi_{n-1} \\ \cdots \\ w^{(k-1)/2}\phi_{n-k} \\ \cdots \end{bmatrix}. \tag{2}$$

The normal equations take the form

$$\begin{bmatrix} s_{1,1} & s_{1,2} & s_{1,3} \\ s_{1,2} & s_{2,2} & s_{2,3} \\ s_{1,3} & s_{2,3} & s_{3,3} \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ r_3 \end{bmatrix}, \tag{3}$$

where the r_i are computed as shown in Eq. (7).

With k_+ defined as $\frac{1}{2}k(k+1)$, the s's are

$$\begin{aligned} s_{1,1} &= \sum_{k=1}^{\infty} w^{k-1} &&= \frac{1}{1-w} & \qquad s_{2,2} &= \sum_{k=1}^{\infty} k^2 w^{k-1} &&= \frac{1+w}{(1-w)^3} \\ s_{1,2} &= \sum_{k=1}^{\infty} k w^{k-1} &&= \frac{1}{(1-w)^2} & \qquad s_{2,3} &= \sum_{k=1}^{\infty} k k_+ w^{k-1} &&= \frac{1+2w}{(1-w)^4} \\ s_{1,3} &= \sum_{k=1}^{\infty} k_+ w^{k-1} &&= \frac{1}{(1-w)^3} & \qquad s_{3,3} &= \sum_{k=1}^{\infty} (k_+)^2 w^{k-1} &&= \frac{w^2+4w+1}{(1-w)^5}. \end{aligned} \tag{4}$$

All symbolic results given here were obtained using the free code maxima, ["Maxima" 2008]. The code used is given in the appendix.

We have defined indexes so that a gives $\hat{\phi}_{n+1}$, the extrapolated value for ϕ_{n+1}, and from the above we have for the quadratic model

$$a = \frac{1-w}{w^2}\left[(1+w+w^2)r_1 + (-2+w+w^2)r_2 + (1-2w+w^2)r_3\right], \tag{5}$$

and for the linear model

$$a = \frac{1-w^2}{w}r_1 - \frac{(1-w)^2}{w}r_2. \tag{6}$$

Of course, all coefficients involving w can be precomputed in parameter statements.

We believe the quadratic model may work better for a multistep code or low order Runge-Kutta code, but for the high order Runge-Kutta code we have been working with here, the linear model does a better job.

The inner product of column i of the matrix in (2) with the right hand side of that equation, gives r_i. Once initialized, these can be updated at step n as follows:

$$\begin{aligned} r_1 &= \phi_n + wr_1 \\ r_2 &= r_1 + wr_2 \\ r_3 &= r_2 + wr_3, \end{aligned} \tag{7}$$

where the "=" sign is being used for assignment, $\phi_n = \log \rho_n - p \log h_n$, and r_3 is not needed in the linear case. Despite the complicated looking formulas, there is very little computation to be done, and much of the work is needed just for the unavoidable $h_{n+1} = e^{-\hat{\phi}_{n+1}/p}$.

All that is left is to get the r's initialized. After the first step, there is no information for anything fancy, and we propose to use the usual simple rule that $h_2 = h_1 * \rho_1^{-1/p}$. Upon completing the second step, we would like to get the r's initialized. At this point, we assume that ϕ is a linear function of k, thus $\phi_{3-k} = \phi_2 - (k-1)(\phi_2 - \phi_1)$. Forming the inner product under this assumption gives the initialization values:

$$\begin{aligned} r_1 &= \sum_{k=1}^{\infty} w^{k-1}\left[\phi_2 - (\phi_2-\phi_1)(k-1)\right] &&= \frac{w\phi_1 + (1-2w)\phi_2}{(1-w)^2} \\ r_2 &= \sum_{k=1}^{\infty} k w^{k-1}\left[\phi_2 - (\phi_2-\phi_1)(k-1)\right] &&= \frac{2w\phi_1 + (1-3w)\phi_2}{(1-w)^3} \\ r_3 &= \sum_{k=1}^{\infty} \frac{k(k+1)}{2} w^{k-1}\left[\phi_2 - (\phi_2-\phi_1)(k-1)\right] &&= \frac{3w\phi_1 + (1-4w)\phi_2}{(1-w)^4}. \end{aligned} \tag{8}$$

We also do this type of initialization after a rejected step.

3. THE AD HOC STUFF

What we have presented above is the easy part of the strategy. For the hard part we used an approach to developing heuristics described in [Krogh 2006], and that sort of approach is more important than the details here. Although what we have seems to work, some of the choices are difficult to justify. What we present here should be taken in the spirit of "this is what we did" rather than "this is what should be used". That said, some sort of extra logic beyond a model such as we have used is likely necessary for best performance.

We have tested with elliptic two body problems as we believe that doing a good job over a range of eccentricities and tolerances on these will carry over to other problems. We have not concerned ourselves with the kinds of discontinuities that arise from changing the analytic definition of how derivatives are computed. Our code provides what we call G-Stops, an interface for finding zeros of arbitrary functions of the dependent (and independent) variables, and this can be used to restart the integration where derivative definitions change. This approach is much to be preferred, since simply trying to integrate over such discontinuities is inherently unreliable.

3.1 Internal Scaling, Defining ρ

Let

$$\rho = \left| \frac{\text{estimated error}}{\text{requested accuracy}} \right| = \beta \left| \frac{\text{estimated error}}{\text{accuracy requested by the user}} \right|. \tag{9}$$

It is this ρ that we attempt to keep close to 1, and a step is rejected if $\rho > \gamma$. To get the same global accuracy an increase in γ requires an increase in β. We have not done a lot of testing for various choices but have settled on $\beta = 100$, and $\gamma = 6$. In the case of our Adams codes we have used $\beta = 10$, and $\gamma = 3$. The larger value of γ for the high order Runge-Kutta case results from the fact that rejected steps are more expensive in that case.

3.2 Extra Cautions and Stiffness Checking

In looking at intermediate results we found, as we expected, that when equations are getting stiff, the fifth order error estimate, O_5, tends to get bigger than the third order estimate, O_3. We also noticed that when O_5/O_3 was getting a little big it served as an indication that our usual stepsize selection was likely to pick a bigger stepsize than desired. This stiffness test works very well for $y' = -cy$ for both large and small values of c..

Initially set $\kappa = 0$. Just prior to what is in the next section, we have,

```
κ = κ − 1
if (O5 > O3) then
    φr = φn + .75 log(.01 O5/O3)
    if (φr > φ̂n+1) then
        if ((φr > φn) .and. (ρ > 10^−4)) φ̂n+1 = φr
        κ = max(0, κ + 2)
        if (κ == 5) then
            κ = 2001
            Give a one time warning of stiffness
        end if
    end if
end if
```

3.3 Minimal use of learning

If a given stepsize has resulted in a rejected step, this is a sign that anything close to that size should be used with caution. We assume (although the code does not) that the stepsize h is positive. Let

H_u	The user maximal stepsize (if any), or ∞.
H_M	A secondary maximal stepsize.
H_m	The current maximum stepsize.
$h_n,\ h_{n+1}$	The previous, next h.

Initially H_m and H_M are set to H_u.

```
if (step is rejected) then
    if (just after a previous rejection)
        h_n = 0
        h = e^{φ_{n+1}}
    else
        h = e^{.75φ_{n+1}+.25φ_n}
    end if
    H_m = max(h_n, h)
end if
```

Just after h_{n+1} has been selected

```
if (accepted step just after a rejection) then
    H_m = h_n
    if (H_M == H_u) H_M = H_m
end if
if (h_{n+1} > H_m) then
    if (h_n ≥ H_m) H_m = sqrt(h_{n+1} H_m)
    h_{n+1} = H_m
else
    if (H_m < H_M) then
        H_m = H_M
    else
        H_M = H_m
    end if
end if
```

4. RESULTS

The first problem set consists of 2 body problems with eccentricity $\mathbf{e}$, solved for t from 0 to 16π, with errors only computed at multiples of π

2 Body Problems – Points Where Solution Checked

i	f_i	$y_i(2k\pi)$	$y_i((2k+1)\pi)$
1	y_2	$1-\mathbf{e}$	$-1-\mathbf{e}$
2	$-y_1/\sqrt{y_1^2+y_3^2}$	0	0
3	y_4	0	0
4	$-y_3/\sqrt{y_1^2+y_3^2}$	$\sqrt{(1+\mathbf{e})/(1-\mathbf{e})}$	$-\sqrt{(1-\mathbf{e})/(1+\mathbf{e})}$,

The second set is Euler equations solved from $t = 0$ to $t = 28c$, where $c = 1.86264080233273855203028122057 9$, and errors only computed at multiples of c.

Euler Equations – Points Where Solution Checked

i	f_i	$y_i(4kc)$	$y_i((4k+1)c)$	$y_i((4k+2)c)$	$y_i((4k+3)c)$
1	y_2y_3	0	1	0	-1
2	$-y_1y_3$	1	0	-1	0
3	$-.51y_1y_2$	1	.7	1	.7,

Since different codes use a different internal scaling of the user's input tolerance, we have multiplied the value of the tolerance passed to the codes so that results are more comparable. These factors are given after the code name in the form (multiplier used for the two body problems, multiplier used for the Euler equations). An absolute error tolerance is used in all cases. (We are interested primarily in checking the effectiveness of the codes with identical inputs and relative error tests are done differently in different codes.) The codes tested are

dop853(.1, .4): This code from [Hairer, Nørsett, and Wanner 1993].

dxrk8.n(1, 1): Dxrk8 is a code, [Krogh 1997], derived from dop853, using the stepsize selection introduced here with $w = .n$. We have settled on $w = .1$.

rksuite8(.08, .03): The 8^{th} order Fortran 77 version of the code described in [Brankin, Gladwell, and Shampine 1993] and [Brankin and Gladwell 1997].

rksuite5(.01, .01): As for rksuite8, but 5^{th} order

dxrk8s(1, 1): Like dxrk8, but using the stepsize selection described in [Söderlind 2003] / [Gustafsson 1991]. We used Eq. (31) in [Söderlind 2003], with $k = 8$ and $b = 4$.

diva(.2, .5): A variable order Adams code of ours similar to what is described in [Krogh 1974].

diva2(.2): Diva with the two-body problems modeled as second order equations.

odex(.07, .003): This code from [Hairer, Nørsett, and Wanner 1993].

odex$_e$(.07, .003): As for odex, but with no interpolation, and thus errors only computed at the end point.

In the tables, E is the largest value seen over the problems run for (error seen at the points specified above) / tolerance, $\overline{\text{NF}}$ is the average number of function evaluations required for a single case, Secs is the running time and the remaining columns give the counts for (global error) / tolerance for the ranges indicated. The tolerance referred to here is always the tolerance prior to it being modified for input to a code. All of the codes provide an interpolation facility to get the results at the selected points except for rksuite8, which integrates to such points.

Results for odex were so bad that we thought it worth trying the codes on a problem where it might perform better. Based on the evidence in [Hairer, Nørsett, and Wanner 1993, p.252], we selected problem LRNZ (pp. 245, 120) where odex appeared to be very slightly better at high accuracy. We obtained the true result in quadruple precision (using g95 with the "-r16" option and slightly modified versions of dop853 and dxrk8), to get results for 60 tolerances (equal relative and absolute tolerances to match with the computations in [Hairer, Nørsett, and Wanner 1993]) spaced between $10^{-11} - 10^{-12}$, $10^{-12} - 10^{-13}$, and $10^{-13} - 10^{-14}$ with other details similar to the above. We chose different multipliers on the stated tolerances to make

clear which methods perform best. Where E is more than a factor of 2 greater than the results for dxrk8, a smaller factor on the tolerance did not reduce E further. Because of the huge error growth on this problem, the E values are $10^{-6}\times$ the error at the final point/ tolerance. (Note, no interpolation needed).

All runs were made on a 64 bit Linux system with gfortran 4.4.1 (optimization level $-$O2) on a 1.6 GHz dual Opteron system.

Elliptical Two Body Problems, for
tol=$10^{-3} \times .96^j,\ j = 0, \ldots, 400$ $\quad$ **e** $= .1 + .01j,\ j = 0, \ldots, 80$

code	**E**	$\overline{\textbf{NF}}$	**Secs**	10^0-10^1	10^1-10^2	10^2-10^3	10^3-10^4	10^4-10^5
dop853	15188	3197	13.8	499	7920	19009	5406	48
dxrk8.1	12951	2283	12.0	842	10086	18057	3790	107
dxrk8.01	14861	2287	12.1	837	10105	17618	4148	174
dxrk8.4	9044	2306	12.2	1032	13420	15171	3259	0
dxrk8s	81864	2746	15.0	421	1129	14653	13593	3086
rksuite8	13205	2692	29.9	3092	13844	12986	2549	10
rksuite5	12339	5447	111.6	15248	6867	7149	3204	13
diva	14750	1950	137.7	4423	14747	8232	5365	115
diva2	14973	1763	130.9	6383	14742	6678	4585	93
odex	16619	6018	66.1	1435	14875	14278	2260	34
odex$_e$	17972	4743	28.9	11457	14236	5782	1403	4

Euler Equations, for tol=$10^{-3} \times .96^j,\ j = 0, \ldots,\ 400$

code	**E**	$\overline{\textbf{NF}}$	**Secs**	10^0-10^1	10^1-10^2
dop853	9.2	1541	0.06	401	0
dxrk8.1	9.2	1514	0.07	401	0
dxrk8.01	7.6	1521	0.07	401	0
dxrk8.4	12.2	1512	0.07	380	21
dxrk8s	9.1	1516	0.08	401	0
rksuite8	11.5	1936	0.24	397	4
rksuite5	8.4	3088	30.38	401	0
diva	7.9	1200	0.67	401	0
odex	84.8	3727	0.43	399	2
odex$_e$	10.6	2527	0.14	399	2

LRNZ Problem

	10^{-11} – 10^{-12}			10^{-12} – 10^{-13}			10^{-13} – 10^{-14}		
code	E	$\overline{\textbf{NF}}$	**Secs**	E	$\overline{\textbf{NF}}$	**Secs**	E	$\overline{\textbf{NF}}$	**Secs**
dxrk8.1	14.4	11399	0.08	16.3	15133	0.10	3.6	20150	0.13
dop853	15.1	12918	0.07	16.3	16339	0.09	9.0	26893	0.14
diva	9.9	6874	0.70	7.5	7617	0.72	239.1	8076	0.96
odex	16.0	15547	0.12	57.5	18213	0.14	617.7	21599	0.16

4.1 Observations and Remarks

(1) Comparing dop853 and dxrk8, we see a better control of the stepsize makes a very noticeable difference, particularly on the two body problems.

(2) Different values of w do not have a great influence on the results using dxrk8.

(3) The comparison of dxrk8 and dxrk8s, suggests that the least squares error control used in dxrk8 works better than the digital filtering approach of dxrk8s when big stepsize changes are needed. The two are essentially the same for the Euler equations. (These codes were as close as we could make them in terms of the additional restrictions placed on the stepsize.) We believe any code that makes some use of past history will work better than the same code without such logic. The differences between rksuite8 and dop853 on the two-body problems and the Euler equations is further evidence of this.

(4) Dxrk8 does better than rksuite8 in terms of function evaluations, and much better on overhead. The overhead for the rksuite codes is surprisingly high. Rksuite8 does not have an interpolation facility, and thus it could not provide the G-Stop feature which is provided by dxrk8. Rksuite5 is only included to show the value of high order. In other experiments we have done, rksuite5 is inferior to rksuite8 even for large tolerances.

(5) Comparison of dxrk8, diva, and odex, gives (unneeded?) further evidence that Adams methods are best when the cost of derivative evaluations dominate, Runge-Kutta methods shine when this is not the case, and extrapolation methods are unlikely to be a preferred choice in any situation. Odex despite our best efforts to make it look good, is always slower than dxrk8 and dop853, and takes many more function evaluations for a given accuracy. We were surprised that in the high accuracy test, it does not even get as good accuracy as an Adams code, which in turn, as expected, can not reach the same level of accuracy as the Runge-Kutta codes.

5. STEPSIZE SELECTION FOR VARIABLE STEP MULTISTEP METHODS

A high-order Runge-Kutta algorithm is challenging for a stepsize selection algorithm, since what we have called ϕ can have large changes from one step to the next. Multistep methods which interpolate to past values, have a simpler job in this respect. But there are other problems: the order can change; the error estimates bounce about much more since these methods tend to select the order very close to the boundary of the region of relative stability; and the expected error on the next step depends on the past stepsize history and not just on the change on the current step.

One can assume as is usually done that $\rho = e^{\phi}h^p$, but the following approach, although somewhat speculative, will probably do a better job since it matches more closely the nature of the actual errors in the formulas.

Since the p^{th} divided difference approximates the p^{th} derivative we see from [Krogh 1974, p. 24, Eq. 2.1] that a more exact model would have the error ratio given by $\rho_n = h_n(h_n+h_{n-1})\cdots(h_n+\cdots+h_{n-p+1})e^{\phi_n}$. Since including the effect of the past stepsizes on the integration coefficients would severely complicate things, and since this effect is smaller we have thrown this effect into e^{ϕ_n}.

It is worth emphasizing that what is said here applies to any Adams or BDF code whether some form of divided differences, the Nordsieck (Taylor's series), or the Lagrangian (saving past derivative values) representation is used for the past history. The representation does not effect the error propagation from previous steps, and an estimate for the derivative in the error term is available from the estimated error. (They all use the same interpolating polynomial; it's just that the computations are arranged differently. The Nordsieck form has the disadvantage of requiring work quadratic in the order for each equation, while the less frequently used Lagrangian form requires more work to compute the integration coefficients, probably introduces slightly more rounding errors, requires extra work to generate information for order selection, and has no compensating advantages.) Methods that do not integrate an interpolating polynomial through previous values have serious stability problems reducing stepsize at high order, see e.g. [Krogh 1973], and are not recommended. (Some early implementations of Adams methods used an interpolating polynomial with an update process that in effect integrated a polynomial that interpolated points on previous interpolating polynomials.)

When we want to solve for h_{n+1} we have computed our extrapolated value $\hat{\phi}_{n+1}$, and are looking for a solution to

$$F(h_{n+1}) = h_{n+1}(h_{n+1} + h_n) \cdots (h_{n+1} + h_n + \cdots + h_{n-p+2}) e^{\hat{\phi}_{n+1}} - 1 = 0 \quad (10)$$

Of course $F(0) = -1$, and clearly F is a convex up, strictly increasing function of h_{n+1} for $h_{n+1} > 0$. $F'(0)$ is also available with little computation, since to compute ϕ_n at the end of the last step we needed to form $h_h(h_n+h_{n-1}) \cdots (h_n+\cdots+h_{n-p+1})$, a quantity that was computed solving for the value of h_n when starting the previous step (with minor adjustments if the order has changed). If we start with a Newton method, it will give an h_{n+1} that is too large. Experience from the iteration on the previous step might suggest what fraction of a Newton step to take, or one might use the minimum of the Newton step, and $2h_n - h_{n-1}$. In either case, the following iterate can be obtained by solving for the positive 0 of the quadratic fitting $F(0)$, $F'(0)$, and $F(h_{n+1})$ computed with the first guess for h_{n+1}. The next iterate could fit a quadratic to the last two $F's$, and $F'(0)$. Later iterates, if needed, would use the last three F values to get the quadratic.

It is probably adequate to stop with a residual of .2(?), which we expect to occur very quickly. The overhead should be small relative to the other computations. needed. Some small adjustments may be needed when the order changes.

6. ACKNOWLEDGMENTS

This work would not have been done if Philip Sharp had not been working with me on a new Adams/BDF code; Philip also pushed me into learning to use maxima and had some helpful suggestions while preparing the final draft. The work by Söderlind and Gustaffson pushed me into looking further into this issue, which led to this work. Ernst Hairer provided a (slightly) new version of dop853, which simplified the comparisons with it, and corrected some errors I had introduced in using his code. Results with an extrapolation method would not have been included if it were not for Peter Deuflhard urging me to include such a code. And finally, thanks to the referees for their careful reading of the manuscript, for their suggestions which have led to a better paper, and for the references [Zonnefeld 1964], [Deuflhard 1984], and [Shampine and Gladwell 1996].

References

Brankin, R. W. and Gladwell, I. 1997. Algorithm 771. `rksuite_90`: Fortran software for ordinary differential equation initial value problems. *ACM Transactions on Mathematical Software 23,* 3 (Sept.), 402–415.

Brankin, R. W., Gladwell, I., and Shampine, L. F. 1993. RKSUITE: A suite of explicit Runge-Kutta codes. In *Contributions to Numerical Mathematics*, R. P. Agarwal, Ed. WSIAA, vol. 2. World Scientific Publishing Co., River Edge, NJ, 41–53.

Deuflhard, P. 1984. Order and stepsize control in extrapolation methods. *Numer. Math. 41*, 399–422.

Gustafsson, K. 1991. Control theoretic techniques for stepsize selection in explicit Runge-Kutta methods. *ACM Transactions on Mathematical Software 17,* 4 (Dec.), 533–554.

Hairer, E., Nørsett, S. P., and Wanner, G. 1993. *Solving Ordinary Differential Equations I*, Second Revised ed. Springer Verlag, Berlin. Codes at `http://www.unige.ch/~hairer/software.html`.

Krogh, F. T. 1973. Algorithms for changing the stepsize. *SIAM Journal on Numerical Analysis 10,* 5 (Oct.), 949–965.

Krogh, F. T. 1974. Changing stepsize in the integration of differential equations using modified divided differences. In *Proceedings of the Conference on the Numerical Solution of Ordinary Differential Equations*, D. G. Bettis, Ed. Number 362 in Lecture Notes in Mathematics. Springer Verlag, Berlin, 22–71.

Krogh, F. T. 1997. Explicit Runge-Kutta method for ordinary differential equations (dxrk8). Tech. rep., Math à la Carte, Inc., Tujunga, CA. Feb. With registration, available from `http://mathalacarte.com/cb/mom.fcg/ya65`.

Krogh, F. T. 2006. On developing mathematical software. *J. Computational and Applied Mathematics 185*, 196–202. Preprint at `http://mathalacarte.com/fkrogh`.

"Maxima". 2008. Maxima, a computer algebra system. url: http://maxima.sourceforge.net/.

Shampine, L. and Gladwell, I. 1996. Software based on explicit RK formulas. *Appl. Numer. Math. 22*, 293–308.

Söderlind, G. 2003. Digital filters in adaptive time-stepping. *ACM Transactions on Mathematical Software 29,* 1 (Mar.), 1–26.

Zonnefeld, J. A. 1964. Automatic numerical integration. Ph.D. thesis, Math. Centre Tracts 8. CWI, Amsterdam, The Netherlands.

APPENDIX

For those interested in checking the work, or perhaps going beyond the quadratic model, the input fed to maxima is given here.

```
assume(w>0, w<1); /* Necessary for getting sums */
s11:limit(nusum(w^(k-1),k,1,n),n, inf); /* Get matrix coefficients */
s12:limit(nusum(k*w^(k-1),k,1,n),n, inf);
s22:limit(nusum(k*k*w^(k-1),k,1,n),n, inf);
s13:limit(nusum((k*(k+1)/2)*w^(k-1),k,1,n),n, inf);
s23:limit(nusum((k*k*(k+1)/2)*w^(k-1),k,1,n),n, inf);
s33:limit(nusum(((k^2*(k+1)^2)/4)*w^(k-1),k,1,n),n, inf);
e1:s11*a + s12*b + s13*c - r1; /* The equations */
e2:s12*a + s22*b + s23*c - r2; e3:s13*a + s23*b + s33*c - r3;
sq: algsys([e1,e2,e3],[a,b,c]);
aq: rhs(factor(sq)[1][1]); /* The solutions for the quadratic case */
bq: rhs(factor(sq)[1][2]); cq: rhs(factor(sq)[1][3]);
f1:s11*c + s12*d - r1; f2:s12*c + s22*d - r2;
sl: algsys([f1,f2],[c,d]);
al: rhs(factor(sl)[1][1]); /* The solutions for the linear case */
bl: rhs(factor(sl)[1][2]); /* Next get data for setting things up initially */
ri1:factor(limit(nusum((w^(k-1))*(p2 - (k-1)*(p2-p1)),k,1,n),n, inf));
ri2:factor(limit(nusum((k*w^(k-1)) *(p2 - (k-1)*(p2-p1)),k,1,n),n, inf));
ri3:factor(limit(nusum((k*(k+1)*w^(k-1)/2)*(p2 - (k-1)*(p2-p1)),k,1,n),n, inf));
disp("Done")$
```