A Tenth-Order Runge-Kutta Method with Error Estimate

T. Feagin

***Abstract*—A tenth-order explicit Runge-Kutta method with embedded results of order eight is exhibited. The difference between the results of orders eight and ten can be used to estimate the local truncation error and thus to vary the stepsize. Numerical experiments demonstrate that the method compares favorably with other high-order embedded methods.**

***Index Terms*—Embedded Runge-Kutta methods, high-order, initial value problems**

I. Introduction

Obtaining an analytical solution to the first-order system of ordinary differential equations:

$$\frac{dx}{dt} = f(t,x) \tag{1a}$$

subject to the initial condition:

$$x(t_0) = x_0, \tag{1b}$$

is often difficult, especially when $f(t,x)$ is a nonlinear function of x. Numerical methods are frequently used in order to obtain approximate solutions valid over some finite range of interest, $[t_0, t_f]$. It is well known that numerical solutions are subject to errors due to truncation, rounding, and (potentially) numerical instability. Consequently, it is important to have estimates or bounds for these errors. Investigators frequently have reasonable estimates for those regions where numerical instability or rounding errors might dominate the calculations and would naturally employ moderate stepsizes, neither too large or too small, so as to avoid such regions. As a result, the principal problem is obtaining reasonably accurate estimates of the truncation errors that would allow for the selection of an appropriate stepsize for a given error tolerance.

Runge-Kutta methods are particularly advantageous for those problems (1) in which the character of the solution changes so rapidly that a major change in the stepsize must be employed in order to track the solution accurately. Multistep methods, on the other hand, often have difficulty adjusting the stepsize adequately in response to such rapid changes, especially if the strategy is limited to halving and doubling the stepsize.

A number of authors [1, 2, 3, 4, 5, 6] have developed explicit Runge-Kutta methods that provide an estimate for the local truncation error. The estimates are usually provided using the difference between the final m^{th}-order result and the result of an embedded formula of order m-1. Most notably, in the 1960's, Fehlberg [7] developed a number of very efficient, embedded explicit Runge-Kutta methods using this approach. His high-order methods include those of order 5(6), 6(7), 7(8), and 8(9). The first number, in each case, represents the order of the embedded, lower-order result. In the present work, the embedded result is of order m-2 instead of m-1, which allows for a more efficient estimate of the errors [8]. In fact, the method described herein requires only seventeen stages, the minimum known value for the tenth order.

An n-stage, explicit Runge-Kutta method obtains an approximation to $x(t_0 + h)$, the solution of (1) at the point $t_0 + h$, using the formula:

$$x_n = x_0 + h\sum_{k=0}^{n-1} c_k f(t_k, x_k) \tag{2a}$$

where $t_k \equiv t_0 + \alpha_k h$ and the intermediate approximations x_k are computed from

$$x_k = x_0 + h\sum_{j=0}^{k-1} \beta_{kj} f(t_j, x_j) \tag{2b}$$

for $k = 1,\ldots,n-1$. The parameters or coefficients $\{c_k\}_{k=0}^{n-1}$, $\{\alpha_k\}_{k=0}^{n-1}$, and $\{\{\beta_{kj}\}_{j=0}^{k-1}\}_{k=1}^{n-1}$ are simply real constants. Note that $\alpha_0 = 0$ (as is normally assumed); that is, the first evaluation is $f(t_0, x_0)$. The method is said to be of order m if the local truncation error, i.e. the difference between the approximation x_n and the true solution $x(t_0 + h)$, is $O(h^{m+1})$. The above notation follows that of Fehlberg, Bettis, and Horn [1, 7, 9]. The form (2) is also known as the Runge-Kutta *ansatz*.

In order to monitor and control the local truncation error, a lower-order result:

$$\hat{x}_n = x_0 + h\sum_{k=0}^{n-1} \hat{c}_k f(t_k, x_k) \tag{3}$$

is also computed. The difference,

$$\hat{x}_n - x_n = h\sum_{k=0}^{n-1} (\hat{c}_k - c_k) f(t_k, x_k) \tag{4}$$

is then used as an estimate of the local truncation error encountered in the current step. Note that this estimate is of the error in $\hat{x}_n$, not of the error in x_n. Nevertheless, the estimate has been shown to be useful for monitoring the accuracy of the numerical integration and for varying the

Manuscript received December 24, 2006.

T. Feagin is a Professor of Computer Science at the University of Houston-Clear Lake, Houston, TX 77058 USA. (e-mail: feagin@uhcl.edu).

stepsize in such a way that the truncation errors are more uniform [10, pp. 184-196].

For an explicit Runge-Kutta method, the number of stages, n, is the number of times the derivative function $f(t,x)$ must be evaluated during each step of the integration. Obviously, in order to obtain an approximation to the solution $x(t)$ valid over the entire range $[t_0, t_f]$, a number of steps would normally have to be taken. At each step, the formula (2) could be used to provide an approximation to $x(t_0 + h)$ and the formula (4) could be used to estimate the local truncation error. If the estimate of the error were within a user-specified tolerance, then the step would be accepted and the values of t_0 and x_0 would be updated accordingly. If the estimated error were not within the tolerance, the step would be rejected. In either event, the stepsize h would be adjusted in such a way that the expected value of the next estimated error would be within the tolerance. Of course, for the very last step, h would be reduced, if necessary, in order to stop the integration precisely at t_f.

II. Background

The first Runge-Kutta methods were developed many years ago. Euler [11] developed his first-order method in about 1768. In 1895, Runge [12] generalized Euler's method as shown in (2). Kutta [13] further developed the idea, developing fourth-order and fifth-order methods. For many years, the classical fourth-order Runge-Kutta method was perhaps the most common numerical method for solving ordinary differential equations. Sixth-order methods were developed in the 1950's [14]. During these early years, the estimation of the local truncation error at the end of a step was most often accomplished by re-integrating the differential equations by taking two steps of half the stepsize, a process that severely compromised efficiency. Consequently, in the 1960's, the embedded pairs of methods were developed in order to overcome this shortcoming.

During the last several decades, numerous high-order explicit Runge-Kutta methods [4, 7, 8, 15, 16, 17, 18, 19] have been developed. A number of enhancements of the basic methods have also been studied including the abilities to vary the order [9, 20, 21], to estimate the local truncation error [1, 2, 3, 4, 5, 6, 7], and to provide for dense output (i.e., to make accurate, intermediate results available at any point within a given step) [9, 22, 23]. The main focus here is on the estimation of truncation errors.

III. The Coefficients of an Explicit Runge-Kutta Method of Order Ten

A tenth-order Runge-Kutta method requires the solution of 1,205 nonlinear algebraic equations. With the help of a *Mathematica* program , a Runge-Kutta method of order ten with an embedded eighth-order result has been determined with seventeen stages and will be referred to as RK8(10). In comparison, Fehlberg's highest order embedded method, RK8(9), and Hairer's tenth-order method [18] (without an embedded result) also require seventeen stages..

The method is based on the six-point Gauss-Lobatto quadrature formula. The coefficients are given to sixty significant digits in Table 1. Fifty-four of the 170 coefficients (more than 30%) are zero. The maximum absolute value of any coefficient is less than 5.785.

TABLE 1 THE COEFFICIENTS OF RK8(10)

k	α_k
0	0.000000000000000000000000000000000000000000000000000000000000000000000000
1	0.100000000000000000000000000000000000000000000000000000000000000000000000
2	0.539357840802981787532485197881302436857273449701009015505500
3	0.809036761204472681298727796821953655285910174551513523258250
4	0.309036761204472681298727796821953655285910174551513523258250
5	0.981074190219795268254879548310562080489056746118724882027805
6	0.833333333333333333333333333333333333333333333333333333333333333333333333
7	0.354017365856802376329264185948796742115824053807373968324184
8	0.882527661964732346425501486979669075182867844268052119663791
9	0.642615758240322548157075497020439535959501736363212695909875
10	0.357384241759677451842924502979560464040498263636787304090125
11	0.117472338035267653574498513020330924817132155731947880336209
12	0.833333333333333333333333333333333333333333333333333333333333333333333333
13	0.309036761204472681298727796821953655285910174551513523258250
14	0.539357840802981787532485197881302436857273449701009015505500
15	0.100000000000000000000000000000000000000000000000000000000000000000000000
16	1.000000000000000000000000000000000000000000000000000000000000000000000000

k	c_k
0	0.033333333333333333333333333333333333333333333333333333333333333333333333
1	0.025000000000000000000000000000000000000000000000000000000000000000000000
2	0.033333333333333333333333333333333333333333333333333333333333333333333333
3	0.000000000000000000000000000000000000000000000000000000000000000000000000
4	0.050000000000000000000000000000000000000000000000000000000000000000000000
5	0.000000000000000000000000000000000000000000000000000000000000000000000000
6	0.040000000000000000000000000000000000000000000000000000000000000000000000
7	0.000000000000000000000000000000000000000000000000000000000000000000000000
8	0.189237478148923490158306404106012326238162346948625830327194
9	0.277429188517743176508360262560654340428504319718040836339472
10	0.277429188517743176508360262560654340428504319718040836339472
11	0.189237478148923490158306404106012326238162346948625830327194
12	-0.040000000000000000000000000000000000000000000000000000000000000000000000
13	-0.050000000000000000000000000000000000000000000000000000000000000000000000
14	-0.033333333333333333333333333333333333333333333333333333333333333333333333
15	-0.025000000000000000000000000000000000000000000000000000000000000000000000
16	0.033333333333333333333333333333333333333333333333333333333333333333333333

k	j	β_{kj}
1	0	0.100000000000000000000000000000000000000000000000000000000000000000000000
2	0	-0.915176561375291440520015019275342154318951387664369720564660
2	1	1.454534402178273228052500217156644591176224837365378736070 16
3	0	0.202259190301118170324681949205488413821477543637878380814562
3	1	0.000000000000000000000000000000000000000000000000000000000000000000000000
3	2	0.606777570903354510974045847616465241464432630913635142443687
4	0	0.184024714708643575149100693471120664216774047979591417844635
4	1	0.000000000000000000000000000000000000000000000000000000000000000000000000
4	2	0.197966831227192369068141770510388793370637287463360401555746
4	3	-0.072954784731363262918514667159555802301501160891438296142131 1
5	0	0.087900734020668133731977709413212547591888682494454853404137 8
5	1	0.000000000000000000000000000000000000000000000000000000000000000000000000
5	2	0.000000000000000000000000000000000000000000000000000000000000000000000000
5	3	0.410459702520260645318174895920453426088035325902848695210406
5	4	0.482713753678866489204726942976896106809132737721421333413261

k	j	β_{kj}
6	0	0.0859700504902460302188480225945808401411132615636600222593880
6	1	0.0000000000000000000000000000000000000000000000000000000000000000
6	2	0.0000000000000000000000000000000000000000000000000000000000000000
6	3	0.3308859630407221839488840576587531736482401548384020334486 32
6	4	0.4896629573094501928445070111358982011780154784337900971 10790
6	5	-0.0731856375070850736789057580558988816340355615025188195854775
7	0	0.1209304491253337206603788549276689539589389969997036788 12621
7	1	0.0000000000000000000000000000000000000000000000000000000000000000
7	2	0.0000000000000000000000000000000000000000000000000000000000000000
7	3	0.0000000000000000000000000000000000000000000000000000000000000000
7	4	0.2601246757582956228090076178383351743681087564846933618878 39
7	5	0.0325402621549091330158899334391231259332716675992700000776101
7	6	-0.0595780211817361001560122025633051214449536727629300724538856
8	0	0.1108543795803914835089361710102184419094257801686565595807038
8	1	0.0000000000000000000000000000000000000000000000000000000000000000
8	2	0.0000000000000000000000000000000000000000000000000000000000000000
8	3	0.0000000000000000000000000000000000000000000000000000000000000000
8	4	0.0000000000000000000000000000000000000000000000000000000000000000
8	5	-0.0605761488255005587620924953655516875526344415354339234619466
8	6	0.3217637056017783901008987990498789040814043686030771292511 10
8	7	0.5104857256080630315777590122851234167446721370317525345 67590
9	0	0.1120544147528790048297150027618023630037176111581722293293 93
9	1	0.0000000000000000000000000000000000000000000000000000000000000000
9	2	0.0000000000000000000000000000000000000000000000000000000000000000
9	3	0.0000000000000000000000000000000000000000000000000000000000000000
9	4	0.0000000000000000000000000000000000000000000000000000000000000000
9	5	-0.1449427759028659156723498283409807771816684997485068388761 85
9	6	-0.3332697190962567065897052114157468717094674239921 1549796872 4
9	7	0.4992692295568800613533168439699785678602768165926732012403 32
9	8	0.5095046089296861042360986900453862539866432323529896021850 60
10	0	0.1139767839641859861380041867369011638907247525414868316403 41
10	1	0.0000000000000000000000000000000000000000000000000000000000000000
10	2	0.0000000000000000000000000000000000000000000000000000000000000000
10	3	0.0000000000000000000000000000000000000000000000000000000000000000
10	4	0.0000000000000000000000000000000000000000000000000000000000000000
10	5	-0.0768813364203356938586214289120895270821349023390922987406384
10	6	0.2395273603243906491077114552718823730197413112010041193395 63
10	7	0.3977746623680946390478304624889521045647164163434546399026 13
10	8	0.0107558956873607455550609147441477450257136782823280838547024
10	9	-0.3277691241640188741470610873502333953782629923923940719064 57
11	0	0.0798314528280196046351426864486400322758737630423413945356284
11	1	0.0000000000000000000000000000000000000000000000000000000000000000
11	2	0.0000000000000000000000000000000000000000000000000000000000000000
11	3	0.0000000000000000000000000000000000000000000000000000000000000000
11	4	0.0000000000000000000000000000000000000000000000000000000000000000
11	5	-0.0520329686800603076514949887612959068721311443881683526937298
11	6	-0.0576954146168548881732784355283433509066159287152968723021864
11	7	0.1947819157121041649763062621473828711561429213544093647380 90
11	8	0.1453849231883250697275248259770711948592034675682365238668 82
11	9	-0.0782942710351670777553986729725692447252077047239160551335016
11	10	-0.1145032993610989121843031642905546709701332184056581226746 74
12	0	0.9851156101648572801200415003065172784136466773141955595205 29
12	1	0.0000000000000000000000000000000000000000000000000000000000000000
12	2	0.0000000000000000000000000000000000000000000000000000000000000000
12	3	0.3308859630407221839488840576587531736482401548384020334486 32
12	4	0.4896629573094501928445070111358982011780154784337900971 10790
12	5	-1.3789648657484356758211272093075190235390432714855947152639 7
12	6	-0.8611641950276356666739169996655345733510260609874270933144 12
12	7	5.7842881363753722002299978548657843600687278968949917260185 6
12	8	3.2880776198510356689046061593731480547726825290334235658192 5
12	9	-2.3863390509313638401342232521552786614840146597595410458580 7
12	10	-3.2547934248364391865458936758778872674771150467478068026991 1
12	11	-2.1634354168642298235395421130005482088967803642010999915488 7
13	0	0.8950802957716328910496131323365851381481562792415613459917 10
13	1	0.0000000000000000000000000000000000000000000000000000000000000000
13	2	0.1979668312271923690681417705103887933706372874633606405555 746
13	3	-0.0729547847313632629185146671595558023015011608914382961421311
13	4	0.0000000000000000000000000000000000000000000000000000000000000000
13	5	-0.8512362396620076197390493714459667932893597228757022271661 05
13	6	0.3983201123185333017197186141743736433364809181037739042318 56
13	7	3.6393726318103560602941292004709004413202738789397780417622 9
13	8	1.5482287703983032236530166307517456491998173634897349631306 5
13	9	-2.1222171470405371602606242746042726102531846114626012440156 1
13	10	-1.5835039854532617271338434962575321275726918893443423797529 1
13	11	-1.7156160828593626492203181975134909891261588082755199297303 4
13	12	-0.0244036405750127452135415444412216875465593593837091056606913 2
14	0	-0.9151765613752914405200150192753421543189513876643697205646 60
14	1	1.4545344021782732280525002171566445911762248373653787360701 6
14	2	0.0000000000000000000000000000000000000000000000000000000000000000
14	3	0.0000000000000000000000000000000000000000000000000000000000000000
14	4	-0.7773336436449682335389312285753021378033510536295472863344 69
14	5	0.0000000000000000000000000000000000000000000000000000000000000000
14	6	-0.0910895662155176069593203555807484200111889091770101799647985
14	7	0.0000000000000000000000000000000000000000000000000000000000000000
14	8	0.0000000000000000000000000000000000000000000000000000000000000000
14	9	0.0000000000000000000000000000000000000000000000000000000000000000
14	10	0.0000000000000000000000000000000000000000000000000000000000000000
14	11	0.0000000000000000000000000000000000000000000000000000000000000000
14	12	0.0910895662155176069593203555807484200111889091770101799647985
14	13	0.7773336436449682335389312285753021378033510536295472863344 69
15	0	0.1000000000000000000000000000000000000000000000000000000000000000
15	1	0.0000000000000000000000000000000000000000000000000000000000000000
15	2	-0.1571786657997711633670589982731289218671837541267094194096 54
15	3	0.0000000000000000000000000000000000000000000000000000000000000000
15	4	0.0000000000000000000000000000000000000000000000000000000000000000
15	5	0.0000000000000000000000000000000000000000000000000000000000000000
15	6	0.0000000000000000000000000000000000000000000000000000000000000000
15	7	0.0000000000000000000000000000000000000000000000000000000000000000
15	8	0.0000000000000000000000000000000000000000000000000000000000000000
15	9	0.0000000000000000000000000000000000000000000000000000000000000000
15	10	0.0000000000000000000000000000000000000000000000000000000000000000
15	11	0.0000000000000000000000000000000000000000000000000000000000000000
15	12	0.0000000000000000000000000000000000000000000000000000000000000000
15	13	0.0000000000000000000000000000000000000000000000000000000000000000
15	14	0.1571786657997711633670589982731289218671837541267094194096 54
16	0	0.1817813007000952838884720625822623796504438314631995216649 45
16	1	0.6750000000000000000000000000000000000000000000000000000000000000
16	2	0.3427581598471898399422205534138508717423387347039589199372 60
16	3	0.0000000000000000000000000000000000000000000000000000000000000000
16	4	0.2591112145483227445129770761917673792677836845431824287781 56
16	5	-0.3582789667179520890489612767219793977397506346736288024842 71
16	6	-1.0459489594088330609505006875640990513158812317237848928608 0
16	7	0.9303278454156269832923005644324287771376016511829657946803 97
16	8	1.7795095943170810244614210679482445392627574324332779053600 0
16	9	0.1000000000000000000000000000000000000000000000000000000000000000
16	10	-0.2825475695390440816124777852222872764084893759762111899528 77
16	11	-0.1593273501199725491692619843734858592780315421275519314618 21
16	12	-0.1455158946470015108609919610810841113086501305786264049455 71
16	13	-0.2591112145483227445129770761917673792677836845431824287781 56
16	14	-0.3427581598471898399422205534138508717423387347039589199372 60
16	15	-0.6750000000000000000000000000000000000000000000000000000000000000

IV. Estimation of Local Truncation Errors

Originally, the tenth-order method of RK8(10), was developed without consideration for estimating the local truncation errors. However, the method was observed to admit an eighth-order embedded result without additional evaluations of $f(t,x)$.

Let $\hat{x}_n$ represent an approximation to $x(t_0+h)$ of order $\hat{m} < m$ such that

$$\hat{x}_n = x_0 + h\sum_{k=0}^{n-1} \hat{c}_k f(t_k, x_k) . \qquad (5)$$

The coefficients $\{\hat{c}_k\}_{k=0}^{n-1}$ represent new weights for an embedded lower-order method. For RK8(10), the largest value of $\hat{m}$ that is possible is eight. In other words, any

method of order nine that uses the same values for the coefficients $\{\alpha_k\}_{k=0}^{n-1}$ and $\{\{\beta_{kj}\}_{j=0}^{k-1}\}_{k=1}^{n-1}$ must also have $\hat{c}_k = c_k$ for $k = 0,...,n-1$ and consequently would not be distinct from the tenth-order result..

The eighth-order result for $\hat{x}_n$ is obtained by taking $\hat{c}_k = c_k$ for all values of k except for $k = 1$ and $k = 15$. Choosing, for example, the values $\hat{c}_1 = 1/36$ and $\hat{c}_{15} = -1/36$ (in contrast to $c_1 = 1/40$ and $c_{15} = -1/40$ for the tenth-order result) causes $\hat{x}_n$ to be an eighth-order result. The difference

$$\hat{x}_n - x_n = \frac{1}{360} h\big(f(t_1, x_1) - f(t_{15}, x_{15})\big) \qquad (6)$$

is then used as an estimate of the local truncation error of the eighth-order result, which in turn is used to adjust the stepsize of the numerical integration and to provide for a more efficient algorithm overall. However, note that (like Fehlberg's high-order methods) the difference $\hat{x}_n - x_n$ vanishes for problems in which $f = f(t)$ only (i.e., for quadrature problems). Therefore, the error estimate must be used with caution.

As it turns out, the eighth-order result described above satisfies all but twenty of the 286 additional ninth-order equations. Also note that it is the *tenth-order* result, x_n, that is used to propagate the solution to the next step.

V. Some Numerical Experiments

The tenth-order method RK8(10), given in the previous section, has been applied to a number of initial value problems of the form (1). A couple of such applications are described in this section. The performance of the method with respect to its ability to obtain an accurate numerical solution over the range $[t_0, t_f]$ for these problems has been compared with that of several other methods using efficiency diagrams. For a given method displayed in these diagrams, the negative logarithm (base 10) of the *absolute value of the largest component of the error* (denoted by *E*) is plotted as a function of the logarithm (base 10) of the *number of evaluations of* $f(t,x)$ *required* (denoted by *NF*). For problems in which $f(t,x)$ is computationally expensive, the time of computation is proportional to *NF*. On such a diagram, the curve connecting the individual experimental results is often a nearly straight line with slope proportional to the order of the method for regions where the largest source of error is truncation error.

Figure 1 exhibits an efficiency diagram for the initial value problem (1) where the differential equations are the equations of motion for the two-body problem:

$$\frac{dx_1}{dt} = x_2\,,\ \frac{dx_2}{dt} = -x_1(x_1^2 + x_3^2)^{-\frac{3}{2}}\,,$$
$$\frac{dx_3}{dt} = x_4\,,\ \frac{dx_4}{dt} = -x_3(x_1^2 + x_3^2)^{-\frac{3}{2}} \qquad (7a)$$

subject to the initial conditions:

$$x_1(0) = 1\,,\ \ x_2(0) = e\,,$$
$$x_3(0) = 0\,,\ \ x_4(0) = \sqrt{1-e^2} \qquad (7b)$$

where the eccentricity $e = 0.4$. For such an eccentricity, it is essential to be able to adjust the stepsize of the integration in order to preserve a fairly uniform truncation error.

For each experimental result, the error is measured at $t = 4\pi$. In the diagram, the results for RK8(10) are connected by the blue line, the results for RK7(8) due to Fehlberg [7] are shown by the purple line, and the results for Fehlberg's RK8(9) are shown by the green line. Each line shown in the diagram connects the points of more than twenty individual experimental results (which are fairly evenly distributed).

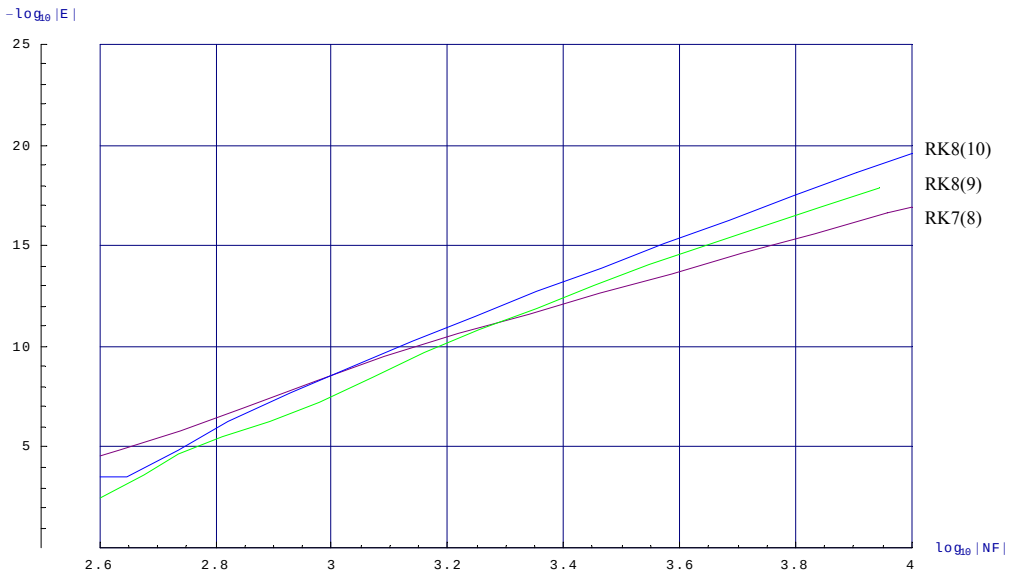

Fig. 1. Efficiency Diagram – Two-Body Problem (e=0.4)

It can be seen from the diagram that as *NF* increases (i.e., as the stepsize decreases), the RK8(10) method is the most efficient of the methods shown when an accuracy above nine significant digits is desired. For lower accuracy, Fehlberg's RK7(8) provides the most efficient results.

Figure 2 exhibits an efficiency diagram for the initial value problem (1) where the differential equations are the well-known predator-prey equations:

$$\frac{dx_1}{dt} = x_1(2 - x_2)\,, \quad \frac{dx_2}{dt} = x_2(x_1 - 1)\,, \tag{8a}$$

subject to the initial conditions:

$$x_1(0) = 2\,, \quad x_2(0) = 2\,. \tag{8b}$$

For each experimental result in the diagram, the error is measured at the final point, $t = 4$.

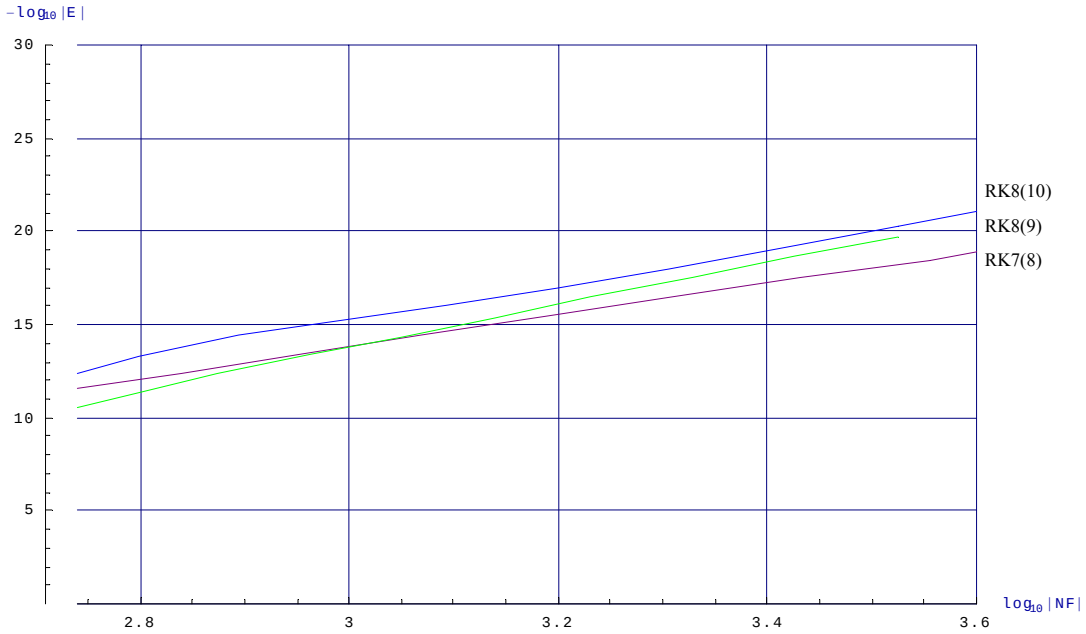

Fig. 2. Efficiency Diagram – Predator-Prey Problem

Once again, each line shown in the diagram connects the points of more than twenty individual experimental results. It can be seen from the diagram that when highly accurate solutions are required, the RK8(10) method is the most efficient of the methods shown for this problem as well.

VI. Conclusions

A new explicit Runge-Kutta method of order ten (with seventeen stages) has been exhibited. The method contains an embedded method of order eight. The difference between the eighth- and tenth-order results at the end of a step can be used as an estimate of the local truncation error. This estimate can be used to monitor the accuracy of the numerical integration and to control the stepsize in a meaningful way. Numerical experiments illustrates the effectiveness of the technique and indicates that the new method offers promise for those problems requiring high accuracy.

VII. References

[1] D. G. Bettis, *Efficient embedded Runge-Kutta methods,* in *Numerical Treatment of Differential Equations*: *Proceedings Oberwolfach, 1976*, *Lecture Notes in Mathematics, No. 631,* R. Bulirsch, Ed., Springer, Berlin (1978), pp. 9-18.

[2] M. Calvo, J. I. Montijano, L. Rández, *A new embedded pair of Runge-Kutta formulas or orders 5 and 6*, Computers Math. Applic., 20 (1990), pp. 15-24.

[3] R. England, *Error estimates for Runge-Kutta type solutions to systems of ordinary differential equations*, Comp. Jour., 12 (1969), pp. 166-170.

[4] P. J. Prince and J. R. Dormand, *High-order embedded Runge-Kutta formulae*, J Comput. Appl. Math., 7 (1981), pp. 67-76.

[5] J. H. VERNER, *Explicit Runge-Kutta methods with estimates of the local truncation error*, SIAM J. Numer. Anal., (1978), pp. 772-790.

[6] J. H. VERNER AND P. W. SHARP, *Completely imbedded Runge-Kutta formula pairs,* Dept. Mathematics and Statistics, Queen's University, Kingston, Preprint 1991-01, 24 pages.

[7] E. FEHLBERG, *Classical Fifth-, Sixth-, Seventh- , and Eighth-Order Runge-Kutta Formulas with Stepsize Control*, NASA TR R-287, (1968).

[8] C. H. TSITOURAS AND S. N. PAPAKOSTAS, *Cheap Error Estimation for Runge Kutta methods,* SIAM J. Sci. Comput., (1999), pp. 2067-2088.

[9] M. K. HORN, *Fourth- and fifth-order, scaled Runge-Kutta algorithms for treating dense output*, SIAM J. Numer. Anal. 20 (1983), pp. 558-568.

[10] J. C. BUTCHER, *Numerical Methods for Ordinary Differential Equations*, Wiley and Sons, New York, (2003), pp.123-196.

[11] L. EULER, *De integratione aequationum differentialium per approximationem*, in Opera Omnia, 1st series, Vol. 11, Institutiones Calculi Integralis, Teubner, Leipzig and Berlin, (1913), pp.424-234.

[12] C. RUNGE, *Über die numerische Auflösung von Differentialgleichungen*, Math. Ann., 46 (1895), pp. 167-178.

[13] W. KUTTA, *Beitrag zur näherungsweisen Integration total Differentialgleichungen*, ZAMP 46, (1901) pp. 435-453.

[14] A. HUŤA, *Contribution à la formule de sixième ordre dans la méthode de Runge-Kutta-Nyström*, Acta Fac. Nat. Univ. Comenian. Math., 2 (1957), pp. 21-24.

[15] G. J. COOPER AND J. H. VERNER, *Some explicit Runge-Kutta methods of high order*, SIAM J. Numer. Anal., 9 (1972), pp. 389-405.

[16] A.R. CURTIS, *An eighth order Runge-Kutta process with eleven function evaluations per step*, Numer. Math., 16 (1970), pp. 268-277.

[17] A.R. CURTIS, *High-order explicit Runge-Kutta formulae, their uses, and limitations*, J. Inst. Maths. Applic., 16 (1975), pp. 35-55.

[18] E. HAIRER, *A Runge-Kutta method of order 10*, J. Inst. Math. Applics. 21 (1978), pp. 47-59.

[19] E. BAYLIS SHANKS, *Solutions of Differential Equations by Evaluations of Functions*, Math. Comp. 20, No. 93 (1966), pp. 21-38.

[20] J. R. CASH, *Block Runge-Kutta Methods for the Numerical Integration of Initial Value Problems in Ordinary Differential Equations Part I. The Nonstiff Case*, Math. Comp. 40 (1983), pp. 175-191.

[21] J. R. CASH AND ALAN H. KARP, *A variable order Runge-Kutta method for initial value problems with rapidly varying right-hand sides*, ACM Trans. Math. Software, 16, Issue 3 (1990), pp. 201-222.

[22] LAWRENCE F. SHAMPINE, *Interpolation for Runge-Kutta Methods*, SIAM J. Numer. Anal. 22 (1985), pp. 1014-1027.

[23] P. W. SHARP AND J. H. VERNER, *Generation of high-order interpolants for explicit Runge-Kutta pairs*, ACM Trans. Math. Software, 24, Issue 1 (1998), pp. 13-29.