EXPLORATIONS

Chapter 8

Numerical Methods and Series

In this chapter, you will take a closer look at the numerical methods used to solve initial-value differential equations, including the methods used internally by the TI-86. It is important to understand what is being done internally so that you can knowledgeably choose the parameters, such as **difTol**, and so that you have a feel for the accuracy of the displayed values. In a later chapter, you will also be doing numerical computations that the TI-86 does not provide. With an understanding of what the TI-86 can compute, you can use its results as a first step in a more complicated process.

Introduction

Consider the typical format for a single first-order initial-value problem:

$$\frac{dy}{dt} = f\big(t, y(t)\big), \quad y(t_0) = y_0, \quad t_0 \le t \le t_f .$$

Most numerical methods for solving such differential equations have two parts.

- A *local approximation procedure* to successively compute values $y_i \approx y(t_i)$ for some $t_0 < t_1 < t_2 < \cdots < t_m \le t_f$, $i = 1, 2, \ldots, m$.
- An *interpolation procedure* to approximate $y(t)$ for t-values falling between t_{j-1} and t_j for some j.

The simplest method, called *Euler's method*, is now almost always presented in textbooks about differential equations (including many recent calculus books). Suppose you know at some intermediate value $t = a$ that $y(a) \approx s$. The differential equation also tells you that $y'(a) \approx f(a, s)$. Thus you can use a tangential approximation as a local approximation procedure,

$$y(t) \approx y(a) + y'(a)\,(t - a) = s + f(a, s)\,(t - a)$$

to get an estimate for the unknown function at a t-value near a. For Euler's method, you will change t-values by a fixed stepsize h, often by deciding

$$h = (t_f - t_0)\,/\,m$$

so that you know it will take exactly m steps to go from the initial to final t-values.

Start with

$$a = t_0,\ s = y_0 \text{ and } t = t_0 + h = t_1,$$

which gives us the first tangential approximation

$$y(t_1) \approx y_0 + h f(t_0, y_0),$$

which you label y_1. Then the successive tangential approximations are given by

$$\begin{cases} t_{i+1} = t_i + h, \\ y_{i+1} = y_i + h\, f(t_i, y_i), \quad i = 0, 1, \ldots, m-1. \end{cases}$$

For this method, it is traditional to use a piecewise linear interpolation procedure ("connect the dots") to estimate values between steps. In this case, that corresponds to using the same tangential approximation for a "partial step" between t_{j-1} and t_j .

Euler's method is commonly used in textbooks, and this method has been implemented on the TI-86. In the **DifEq** graphing mode, selecting **Euler** on the format screen will cause this numerical method to be used to compute the points for graphing. The viewing window parameters **tMin** and **tMax** correspond to t_0 and t_f, and **tStep** corresponds to h when the last parameter **Estep** = 1. In the connected style, line segments are drawn between the plotted points (the dots are connected).

Example 1: Euler's Method Graphically Explored

Use m = 4, 16, and 64 steps of Euler's method to approximate the solution to

$$\frac{dy}{dt} = \cos(t + y), \quad y(0.2) = 0.1, \quad 0.2 \le t \le 1.8.$$

Plot the results on the same screen to compare how the approximations "improve."

Solution

1. Enter the equation into the differential equation editor as **Q'1**=cos (t+Q1). (Figure 8.1)

Figure 8.1

2. Set the **DifEq** graphing mode, and select **Euler** on the format screen to use Euler's method for computations. (Figure 8.2)

Figure 8.2

3. Set the initial conditions editor (**INITC**) to have **tMin** = 0.2 and **Q1** = 0.1. For **AXES**, select **x** = t and **y** = Q1. For $m = 4$ (Figure 8.3), set the viewing window parameters to

   tMin = 0.2, **tMax** = 1.8, **tStep** = .4,

   tPlot = 0, **xMin** = -0.1, **xMax** = 1.9,

   xScl = 0.1, **yMin** = -0.2, **yMax** = 0.7,

   yScl = 0.1, **Estep** = 1.

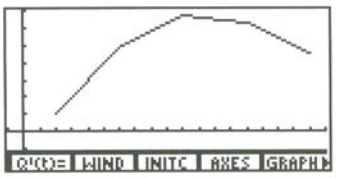

Figure 8.3

4. To compare these computations with the later ones, use the [GRAPH] **STPIC** command ([GRAPH] [MORE] [MORE] [F4]) to store this image as a picture named **P1**. (Figure 8.4)

Figure 8.4

5. Change **tStep** = 0.1 to get the graph of Figure 8.5 where $m = 16$. Store the second plot as a picture named **P2**.

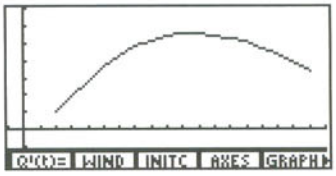

Figure 8.5

6. Set **tStep** = 0.025 to get the graph when $m = 64$. Recall the two previous pictures (with **RCPIC**) to see all three on the same screen. (Figure 8.6)

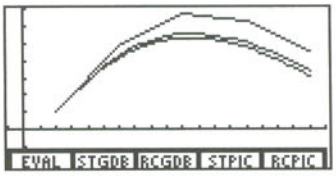

Figure 8.6

It is not hard to understand that making h smaller generally leads to better approximations. Because of this, the TI-86 allows you to have several Euler steps before bothering to plot a point on the graph. The parameter **tStep** sets the change in **t** for plotted points while the parameter **Estep** sets the number of Euler steps to be taken *between plotted points*.

For example, setting **tStep** = 0.1 and **Estep** = 4 (with **tMin**=0.2 and **tMax**=1.8) would also compute $m = 64$ total Euler steps but it would only plot (and allow you to trace) on 16 graphical points. This is important because you might need to take several hundred Euler steps to achieve reasonable accuracy, but you would only want to plot and trace on a fraction of these points.

Series play two roles in the field of numerical differential equations. One is that for certain types of differential equations, you simply accept an infinite power series as the format for a solution. Then for practical computations, you might simply use a finite number of terms in the series.

The second role is that of rating the local approximation procedures by how they compare to a local power series expansion. In general, you consider the *local error* to be the error you make for any one step of the method you are using. For example, the step procedure used in Euler's method is exactly the first two terms of the Taylor series expansion at $t = a$ given below (perhaps in a slightly different notation from the version you have seen before).

$$y(a+h) = y(a) + y'(a)\,h + \frac{y''(a)}{2}h^2 + \frac{y'''(a)}{6}h^3 + \cdots + \frac{y^{(n)}(a)}{n!}h^n + \cdots$$

Using the formula for the remainder, if you truncate the Taylor series to a finite Taylor polynomial, you can prove that the local error is a multiple of the next power of h. For example, the local error using the tangential approximation for estimating $y(a + h)$ in Euler's method is known to be

$$\frac{y''(\xi)}{2}h^2 \quad \text{for some } \xi \text{ between } a \text{ and } a+h \quad \text{OR} \quad K\,h^2.$$

While this is the error estimate for one step, you have taken m steps to reach the final t-value. Generally

$$m = (t_f - t_0)/h$$

so there is some accumulation of the local errors in the final approximation for $y(t_f)$. The total error you have after many steps is called the *global error*. Not surprisingly, the worst global error tends to occur where you stop at $t = t_f$. It can be shown that the global error for Euler's method is a multiple of h (one lower power than in the local error estimate). In numerical analysis, this is called an *order h method*. Its importance is this: if you cut the step size h in half for an *order h method*, you can expect the global error that you make to also be cut in half. In an *order h^2 method*, cutting the step size h in half would result in a global error one-fourth as large as before. You will soon see the advantages of a higher order method.

To experiment numerically with the properties of the error in various numerical methods, you need to work an example where you know the exact answer. Consider

$$\frac{dy}{dt} = 1 + (y-t)^2, \quad y(0) = 0.5, \quad 0 \le t \le 1, \quad \text{which has the exact solution} \quad y = t + \frac{1}{2-t}.$$

If you look at the set of exact solutions for initial conditions **QI1** = {.3, .4, .5, .6, .7}, you can gain a graphical understanding about how hard it is to accurately solve this initial-value problem. See Figure 8.7, where $-0.1 \le x = t \le 1.1$, $-0.3 \le y \le 2.7$, and you have plotted these exact solutions $y = t + 1/(C\text{-}t)$ in **Func** graphing mode. Changing C gives a different solution in the family, and this corresponds to different initial conditions **QI1**, namely **QI1** = $1/C$.

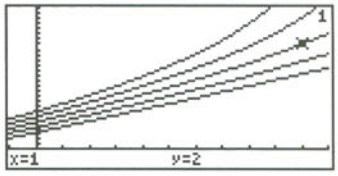

Figure 8.7

Notice in Figure 8.7 how a very small change in the initial value $y(0)$ at t=0 may result in a much larger change in the true solution $y(1)$ at t=1. All numerical methods effectively "get a little off" of the true solution curve onto a "nearby" member of the family of all solutions as they step. In this family, "getting a little off" near $t = 0$ can result in "being off quite a bit" when you get to $t = 1$.

Example 2: Euler's Method Numerically Explored

Numerically compute the solution to the known problem on the previous pages using Euler's method with different values for the stepsize h. Store the results, and use list operations to confirm the expected relationship between the size of h and the size of the final t-value 1. Estimate the number of Euler steps needed to have a final accuracy of 1E-7.

Solution

1. Enter the equation into the differential equation editor as **Q'1**=1+(Q1-t)^2. Select **Euler** in the format screen. Set the initial conditions editor to have **tMin** = 0 and **Q1** = 0.5. For **AXES**, select **x** = t and **y** = Q1.

   Set the viewing window parameters initially to be

   tMin = 0, **tMax** = 1, **tStep** = .1,

   tPlot = 0, **xMin** = ⁻0.1, **xMax** = 1.1,

   xScl = 0.1, **yMin** = ⁻0.3, **yMax** = 2.7,

   yScl = 0.1, **Estep** = 1.

2. Move back to the home screen with [2nd] [QUIT]. Set **y1** to be the exact solution (with t replaced by x) by the command **y1 = x + 1/(2-x)**. Then evaluate the true solution at 0.1 and 1. (Figure 8.8)

3. You are going to check six different choices for the value of h, and you can do this in the home screen by repeatedly changing **tStep** and using the command **eval** followed by the desired t-value. (To enter the **eval** command, press [2nd] [MATH] [F5] (**MISC**) [MORE] [F5].)

   See Figure 8.9 for the computations using the current **tStep** = 0.1. Note that the command **eval** evaluates the selected graphing object (using the selected computational method **Euler**) at the given t-value.

```
y1=x+1/(2-x)
                        Done
y1(.1)
              .626315789474
y1(1)
                           2
```

Figure 8.8

```
                           2
eval .1
                       (.625)
eval 1
            (1.94220484186)
 NUM  PROB  ANGLE  HYP  MISC
▶Frac  %   ▶Eval   ^r   eval
```

Figure 8.9

```
            (1.94220484186)
.01→tStep
                         .01
eval .01
                     (.5125)
eval 1
            (1.99320755515)
```

Figure 8.10

It is tedious to repeatedly do this by hand as shown in Figure 8.10, and then to build up the lists of results with the **aug** command from the [2nd] [LIST] [F5] (**OPS**) menu. Instead, use a short program (which can be modified for later investigation of other methods) to do this process.

```
PROGRAM:ERRORS
:Disp "Assumes DEmethod set"
:Disp "and exact soln is y1"
:seq(.5/(5^J),J,1,6)→HH
:tMax→TF
:HH(1)→tStep
:(eval tStep)→LE
:(eval TF)→GE
:For(J,2,dimL HH,1)
:HH(J)→tStep
:aug(LE,eval tStep)→LE
:aug(GE,eval TF)→GE
:End
:LE-y1(HH)→LE
:GE-y1(TF)→GE
:Disp HH,LE,GE
```

Running this program for this example takes quite a while (about 20 minutes)! The output is displayed in Figure 8.11.

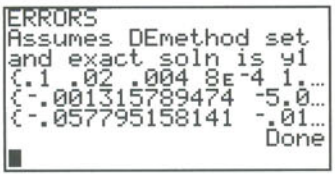

Figure 8.11

4. These lists are best displayed in the list editor. From the home screen, use the command **SetLEdit, HH,LE,GE** to put these lists in the list editor. (You can find this command in the CATALOG under S, by pressing [2nd] [LIST] [F5] (**OPS**) [MORE] [MORE] [MORE] [F3], or you can simply type the letters **SetLEdit** from the keyboard.)

5. Press [2nd] [LIST] [F4] (**EDIT**) to bring up the list editor to view these computed lists. (Figure 8.12)

   To test the claim that the local errors (**LE**) behave like a constant times h^2, create a new list where each entry is an entry from **LE** divided by the square of the corresponding entry from **HH**.

6. With the cursor positioned at the very top of the screen in the list-name area, move to the top to an empty column and press [ENTER]. Then type the name of this new list, **H2** (Figure 8.13), and press [ENTER].

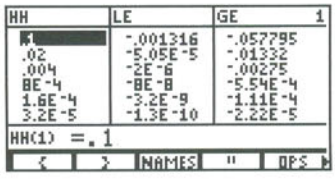

Figure 8.12

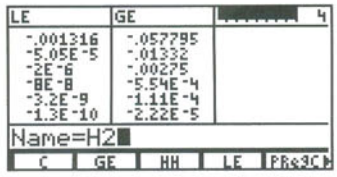

Figure 8.13

7. Press ENTER again with the new name highlighted to attach a formula to this new listname **H2**. The formula is typed inside quotations marks. (Figure 8.14)

   In this way, the formula inside the quotation marks is attached to the list **H2**, and the list **H2** is locked (see the icon in the column heading area) so that it always stays defined according to the formula.

   As you see in Figure 8.15, the claim proves to be correct with the constant about ⁻0.125.

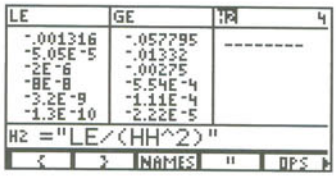

Figure 8.14

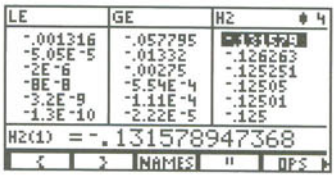

Figure 8.15

8. In a similar fashion, you create the new list **H1** with the formula GE/HH to see that the global errors for approximating $y(1)$ are roughly equal to the constant ⁻0.693 times h. (Figure 8.16)

   Working backwards, if you wish to know $y(1)$ with an accuracy of 1E-7, then you need to choose $h < 1.44\text{E-}7$ or you need to take approximately m=6,930,000 steps of Euler's method.

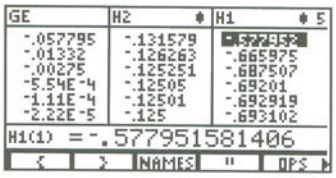

Figure 8.16

It is possible to generalize Euler's method (and all the methods in this chapter) to vector differential equations. Thus when you enter a system of the form

$$\begin{bmatrix} Q'1 \\ Q'2 \\ Q'3 \end{bmatrix} = \begin{bmatrix} f_1(t, Q1, Q2, Q3) \\ f_2(t, Q1, Q2, Q3) \\ f_3(t, Q1, Q2, Q3) \end{bmatrix},$$

it can be considered as a vector equation

$$\mathbf{Q}' = \mathbf{F}(t, \mathbf{Q}) \text{ where } \mathbf{Q} = \begin{bmatrix} Q1 \\ Q2 \\ Q3 \end{bmatrix}.$$

Euler's method for such a system then becomes

$$\begin{cases} t_{i+1} = t_i + h, \\ \mathbf{Q}_{i+1} = \mathbf{Q}_i + h\,\mathbf{F}(t_i, \mathbf{Q}) \end{cases} \text{ or } \begin{cases} t_{i+1} = t_i + h, \\ Q1_{i+1} = Q1_i + h\,f_1(t_i, Q1_i, Q2_i, Q3_i), \\ Q2_{i+1} = Q2_i + h\,f_2(t_i, Q1_i, Q2_i, Q3_i), \\ Q3_{i+1} = Q3_i + h\,f_3(t_i, Q1_i, Q2_i, Q3_i), \quad i = 0, 1, \ldots, m-1. \end{cases}$$

You will not pursue the theory behind numerical methods for systems any further in this chapter, but the TI-86 has been programmed to handle this generalization.

Euler's method is easy to understand (and easy to program if you wish to do so), but it is not computationally efficient. To obtain reasonable accuracy, you must take a large number of small steps.

It is much better to use a higher order local approximation method which will give better accuracy with fewer steps.

Next, you will look at a further series expansion, both to better understand series solutions and to gain some insight into the second method implemented on the TI-86. If you have not previously worked very much with series, the next example might be a little hard to follow. You can go on without understanding every detail.

Example 3: Series Solution Explored

Find the series expansions for $y(t)$, the solution to

$$\frac{dy}{dt} = 1+(y-t)^2, \quad y(0) = 0.5,$$

out at least to the cubic terms in the expansion.

Solution

There are two different methods for finding the terms in the series. First, you can (repeatedly) differentiate both sides of the differential equation implicitly to get higher derivatives. Do this once carefully.

$$\frac{d}{dt}\left(\frac{dy}{dt}\right) = \frac{d}{dt}\left(1+(y-t)^2\right) = 0+2(y-t)\frac{d}{dt}(y-t) = 2(y-t)\left(\frac{dy}{dt}-1\right)$$

Thus you can know that the following is true:

$$\frac{d^2y}{dt^2} = 2(y-t)\left(\frac{dy}{dt}-1\right), \quad \frac{d^3y}{dt^3} = 2\left(\frac{dy}{dt}-1\right)^2 + 2(y-t)\frac{d^2y}{dt^2}, \quad \ldots$$

You can evaluate these formulas (and the differential equation) at t=0 knowing that $y(0)$=0.5.

$$y(0) = 0.5,$$
$$y'(0) = 1+(0.5-0)^2 = 1.25,$$
$$y''(0) = 2(0.5-0)(1.25-1) = 0.25,$$
$$y'''(0) = 2(1.25-1)^2 + 2(0.5-0)(0.25)^2 = 0.375$$

Knowing these derivatives at t=0, you can write the Taylor series expansion for the solution.

The second method leads directly to the series. Suppose the solution has the form

$$y(t) = \sum_{n=0}^{\infty} a_n t^n .$$

Assuming this series can be differentiated term-by-term, substitute the series expressions for y and y' into the differential equation.

$$\sum_{n=1}^{\infty} n a_n t^{n-1} = 1 + \left(\sum_{n=0}^{\infty} a_n t^n - t \right)^2$$

$$a_1 + 2a_2 t + 3a_3 t^2 + 4a_4 t^3 + \cdots$$

$$= 1 + \left(a_0 + [a_1 - 1]t + a_2 t^2 + a_3 t^3 + \cdots \right)^2$$

$$= 1 + a_0 \left(a_0 + [a_1 - 1]t + a_2 t^2 + \cdots \right) + [a_1 - 1]t \left(a_0 + [a_1 - 1]t + a_2 t^2 + \cdots \right) + a_3 t^3 \left(a_0 + \cdots \right) + \cdots$$

$$= \left\{ 1 + a_0{}^2 \right\} + \left\{ 2a_0 [a_1 - 1] \right\} t + \left\{ 2a_0 a_2 + [a_1 - 1]^2 \right\} t^2 + \left\{ 2a_0 a_3 + 2[a_1 - 1] a_2 \right\} t^3 + \cdots$$

Setting the coefficients of like powers of t to be equal on both sides of this equation, you find that this gives equations relating each coefficient a_n to coefficients you already know. For example, the constant a_1 on the left side must equal the constant $\{1+a_0{}^2\}$ on the right side. Then $2a_2 = \{2a_0[a_1 - 1]\}$ matching linear terms. You start out knowing $a_0 = 0.5$ from the given initial condition. You find

$$a_0 = y(0) = \tfrac{1}{2}, \quad a_1 = \tfrac{5}{4}, \quad a_2 = \tfrac{1}{8}, \quad a_3 = \tfrac{1}{16}, \quad a_4 = \tfrac{1}{32}.$$

These are the same coefficients that you got in the first method (as far as you went).

$$y(0) = 0.5, \quad y'(0) = 1.25, \quad y''(0)/2! = 0.125, \quad y'''(0)/3! = 0.0625$$

In particular, you could use as a first local approximation any partial sum of the series

$$y(0+h) = \tfrac{1}{2} + \tfrac{5}{4}h + \tfrac{1}{8}h^2 + \tfrac{1}{16}h^3 + \tfrac{1}{32}h^4 + \cdots.$$

Unfortunately, the methods you used to find a series expansion in Example 3 do not lend themselves to easy implementation for successive steps as in Euler's method. There is just too much symbolic work involved in getting the series expansion for each step. The local approximation procedures used in Runge-Kutta methods match more terms of the series expansion but are much easier to compute numerically.

Recall the general first-order initial-value problem you wish to solve.

$$\frac{dy}{dt} = f\left(t, y(t)\right), \quad y(t_0) = y_0, \quad t_0 \le t \le t_f.$$

The following is a two-stage Runge-Kutta method that is of order h^2 (that is, its expansion will effectively match terms in the series expansion out to the h^2 but generally differ after that).

$$\begin{cases} t_{i+1} = t_i + h, \\ \quad k_1 = h\, f(t_i, y_i) \\ \quad k_2 = h\, f(t_i + h, y_i + k_1) \\ y_{i+1} = y_i + \dfrac{k_1 + k_2}{2}, \quad i = 0, 1, \ldots, m-1. \end{cases}$$

This two-stage Runge-Kutta method is sometimes called the *improved Euler method*, and it is possible to interpret it as an average slope method (since $f(t_i, y_i)$ and $f(t_i+h, y_i+k_1)$ can be considered as slopes at both ends of the interval $t_i \le t \le t_i + h$).

Briefly, the justification that this method has order h^2 involves two steps. First, you use the idea of implicit differentiation demonstrated in Example 3 to expand the solution in general.

$$y(t_i + h) \approx y_i + f(t_i, y_i)h + \tfrac{1}{2}\left\{\frac{\partial f}{\partial t} + \frac{\partial f}{\partial y} f\right\}\Bigg|_{(t_i, y_i)} h^2 + \cdots$$

Then you expand the approximation given by this Runge-Kutta method (using Taylor series in two variables on k_2).

$$\begin{aligned} y(t_i + h) &\approx y_i + \frac{k_1 + k_2}{2} = y_i + \tfrac{1}{2} h f(t_i, y_i) + \tfrac{1}{2} h f(t_i + h, y_i + h f(t_i, y_i)) \\ &= y_i + \tfrac{1}{2} h f(t_i, y_i) + \tfrac{1}{2} h \left\{ f(t_i, y_i) + \frac{\partial f}{\partial t}(t_i, y_i) + \frac{\partial f}{\partial t}(t_i, y_i) f(t_i, y_i) + \cdots \right\} \end{aligned}$$

Terms shown in each infinite expansion match, but the next terms in the expansion do not.

The general form for a three-stage Runge-Kutta method is:

$$\begin{cases} t_{i+1} = t_i + h, \\ \quad k_1 = h f\left(t_i, y_i\right) \\ \quad k_2 = h f\left(t_i + \alpha_2 h, y_i + \beta_{2,1} k_1\right) \\ \quad k_3 = h f\left(t_i + \alpha_3 h, y_i + \beta_{3,1} k_1 + \beta_{3,2} k_2\right) \\ y_{i+1} = y_i + w_1 k_1 + w_2 k_2 + w_3 k_3, \quad i = 0, 1, \ldots, m-1 \end{cases}$$

The coefficients

$$\alpha_2, \alpha_3, \beta_{2,1}, \beta_{3,1}, \beta_{3,2}, w_1, w_2, \text{ and } w_3$$

are chosen so that there is the highest order of match possible with the series expansion. Expanding in a fashion similar to that used for the specific second-order method above, it can be shown that this will be order h^3 if and only if

$$\begin{aligned} &w_1 + w_2 + w_3 = 1, \qquad w_2 \alpha_2 + w_3 \alpha_3 = \tfrac{1}{2}, \qquad w_2 {\alpha_2}^2 + w_3 {\alpha_3}^2 = \tfrac{1}{3}, \\ &\beta_{2,1} = \alpha_2, \qquad \beta_{3,1} + \beta_{3,2} = \alpha_3 \end{aligned}$$

There are many ways to solve this nonlinear system of equations for the needed constants in such a third-order Runge-Kutta method. The following come from the reference by *Iserles* where the idea of a *RK tableaux* is introduced to neatly display these parameters.

$$\begin{array}{c|cc} \alpha_2 & \beta_{2,1} & \\ \alpha_3 & \beta_{3,1} & \beta_{3,2} \\ \hline & w_1 & w_2 \quad w_3 \end{array} \qquad \begin{array}{c|cc} \frac{1}{2} & \frac{1}{2} & \\ 1 & -1 & 2 \\ \hline & \frac{1}{6} & \frac{2}{3} \quad \frac{1}{6} \end{array} \qquad \begin{array}{c|cc} \frac{2}{3} & \frac{2}{3} & \\ \frac{2}{3} & 0 & \frac{2}{3} \\ \hline & \frac{1}{4} & \frac{3}{8} \quad \frac{3}{8} \end{array}$$

RK Tableaux Classical RK Nystrom Scheme

For 2, 3, and 4 stage Runge-Kutta methods, it is possible to achieve the orders h^2, h^3, and h^4, respectively. For each, it turns out that there are many ways achieve the highest possible order. It takes at least six stages to achieve order h^5. Also it becomes more difficult to achieve orders higher than h^4 in the vector case.

Example 4: Runge-Kutta Method Numerically Explored

Write a short program to implement the classical third-order Runge-Kutta method to solve again

$$\frac{dy}{dt} = 1 + (y - t)^2, \quad y(0) = 0.5, \quad 0 \le t \le 1, \quad \text{which has the exact solution} \quad y = t + \frac{1}{2 - t}.$$

Include in the program the creation of lists consisting of the local error (for the first step) and the global error (for $y(t_f)$) for a variety of stepsizes.

Solution

The following program aims more for simplicity than elegance and general use. The differential equation and the exact solution appear explicitly in the code (for example, $y(1) = 2$).

The only user option is the number of steps m. You will need to modify the code (or redesign the first part) to use it on other problems with a known solution. To use the code on a problem where you do not know the exact solution, you will need to eliminate the creation of the error lists.

To highlight the simplicity of the Runge-Kutta method, the subprogram RK3 implements one step of the classical third-order method. This subprogram assumes that the expression for the right side of the differential equation $f(T, Y)$ is stored in $y2$. (You can use a function slot to store an expression involving several variables. If you store different values to T and Y, evaluating $y2$ uses the most recently stored values.) Every time RK3 is called, it expects the variables named **TI**, **YI**, and **H** to contain the current t_i, y_i, and h. The subprogram "returns" the values t_{i+1} and y_{i+1} by storing these in **TI** and **YI** (thus updating the variables for the next step).

Note that the use of **DelVar** at the end deletes all the temporary variables created in the program. There is no way to declare a program variable as *local*, but at least this action "cleans up" a little to keep your variable list down in size to save memory. The variables **TI** and **YI** remain because you might want to do something further with the final approximation outside the program.

```
PROGRAM:EX4
:y1=T+1/(2-T)
:y2=1+(Y-T)²
:0→A:1→B:0.5→S:0→C
:Lbl A
:Input "No. steps=",M
:(B-A)/M→H:Disp H:Pause
:A→TI:S→YI:RK3
:If C=0:Then
:{H}→HH:{YI-y1}→LE
:Else
:aug(HH,{H})→HH
:aug(LE,{YI-y1})→LE
:End
:For(I,2,M,1)
:RK3
:End
:If C=0:Then
:{YI-y1}→GE
:Else:aug(GE,{YI-y1})→GE
:End
:1→C :Menu(1,"NEWM",A,5,"QUIT",E)
:Lbl E
:DelVar(y1):DelVar(y2):DelVar(A)
:DelVar(B):DelVar(S):DelVar(C)
:DelVar(H):DelVar(M):DelVar(T):DelVar(Y):DelVa
r(K1):DelVar(K2)
:DelVar(K3)
```

```
PROGRAM:RK3
:TI→T:YI→Y:H*y2→K1
:TI+H/2→T:YI+K1/2→Y:H*y2→K2
:TI+H→T:YI-K1+2*K2→Y:H*y2→K3
:T→TI:YI+(K1+4*K2+K3)/6→YI
:Disp TI,YI
```

1. Enter and run the program, picking the number of steps **M** to be, for example, 5, 10, 20, 40, 100, and 125. This will create a list of step sizes **HH**={.2, .1, .05, .025, .01, .008}. The first pass with **M=5** will first pause to show you **H**.

2. Press [ENTER] to have the program continue, ending as in Figure 8.17 with the displayed menu allowing the users to pick a new **M** or to quit.

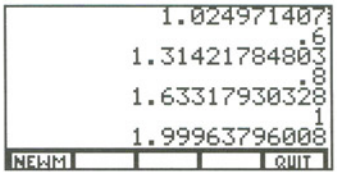

Figure 8.17

3. Press [F1] and make another pass through the program for another number **M**. The last pass through the program with **M=125** will end as in Figure 8.18. Press [F5] to quit the program.

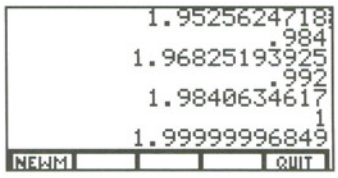

Figure 8.18