# Teaching the Numerical Solution of Ordinary Differential Equations Using Excel 5.0

Sama Bilbao y León, Robert Ulfig, and James Blanchard
University of Wisconsin - Madison
1500 Johnson Dr.
Madison, WI 53706

**Abstract**

PC-based computational programs have begun to replace procedural programming as the tools of choice for engineering problem-solving. These tools offer ease-of-use along with sufficient computational power to solve realistic problems. Hence, the development time is reduced, while retaining sufficient complexity. These advantages are particularly important in the classroom, allowing students to focus initially on algorithms, with little time spent learning the use of the particular tool. Later, the students can develop more sophisticated solutions using the advanced capabilities of the tool. An example is given, using Microsoft Excel 5.0, implementing algorithms for solving ordinary differential equations. The simple interface of the spreadsheet can be used to learn the fundamentals of the algorithm, and then the macro language (Visual Basic) can be used to produce more powerful equation solvers. The final result is an adaptive algorithm that can easily be used to numerically solve complex systems of differential equations.

## Introduction

Recently, PC-based computational software has begun to replace the use of procedural languages for the solution of engineering problems. Tools such as spreadsheets, equation solvers (such as MathCAD and TK Solver), and symbolic algebra programs (such as Maple and Mathematica) offer ease-of-use and built-in functions that have significant advantages over procedural programming languages. These advantages also carry over into the classroom, allowing more efficient learning with minimal time spent coping with the software itself. For example, it has been traditional for students to take a course in how to program in a procedural language, followed by a course in numerical methods. This is inefficient because problem-solving *per se*, is not considered at length until the second course. In contrast, a student can learn to use a spreadsheet in a few short lessons and can begin to solve problems immediately thereafter. This allows nearly two full courses to focus on numerical methods and problem solving, thus providing significantly more depth to the levels of sophistication achieved by undergraduate engineering students.

This paper gives an example of how a typical, modern computational tool can be used to teach problem-solving. In this case, the Microsoft Excel 5.0 spreadsheet is used to teach the numerical solution of ordinary differential equations. The advantage of the spreadsheet is derived both from its versatility and ease-of-use. The beginner can use the standard spreadsheet interface to implement and test a standard algorithm for solving the

equations. (Here a fourth-order Runge-Kutta algorithm is used, but the conclusions drawn are equally applicable to other algorithms.) This platform allows the student to comprehend the intricacies of the algorithm, but it can be somewhat cumbersome. For instance, it is difficult, using the standard spreadsheet interface, to change from one set of differential equations to another. Thus, as the student advances, the built-in macro language (Visual Basic, in this case) can be used to implement a sophisticated algorithm that is easily adapted to other equations. This tool can then be used by the student in other courses and, ultimately, in their employment.

To demonstrate these principles, this paper provides spreadsheet-based solutions to systems of 1 and 2 ordinary differential equations using the standard spreadsheet interface, a simple function macro that carries out a single time step, and a subroutine (complete with a simple user interface) that carries out the full solution. This is presented in the order that it would be presented in a typical problem-solving course, starting with a straightforward implementation and progressing to more sophisticated techniques that are more generally applicable.

**Using the Standard Spreadsheet Interface**

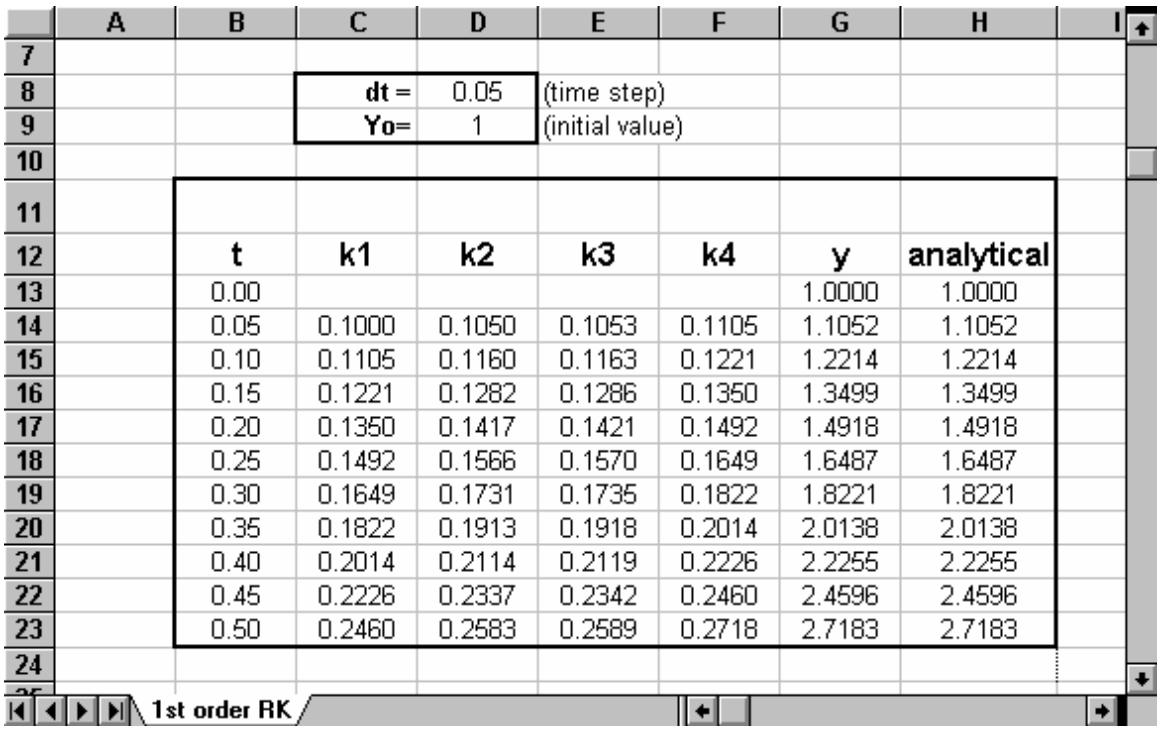
Here the standard spreadsheet interface is used to implement a fourth-order Runge-Kutta scheme in Excel 5 and solve a single, first-order equation of the following form:

$$\frac{dy}{dt} = f(t, y),$$

with the initial condition $y(0)=A$. The fourth-order Runge-Kutta scheme uses the following algorithm to advance a solution from time $t$ to $t+\Delta t$:

$$k_1 = \Delta t * f(t, y)$$
$$k_2 = \Delta t * f(t + \frac{\Delta t}{2}, y + \frac{k_1}{2})$$
$$k_3 = \Delta t * f(t + \frac{\Delta t}{2}, y + \frac{k_2}{2})$$
$$k_4 = \Delta t * f(t + \Delta t, y + k_3)$$
$$y(t + \Delta t) = y(t) + \frac{(k_1 + 2 * (k_2 + k_3) + k_4)}{6}$$

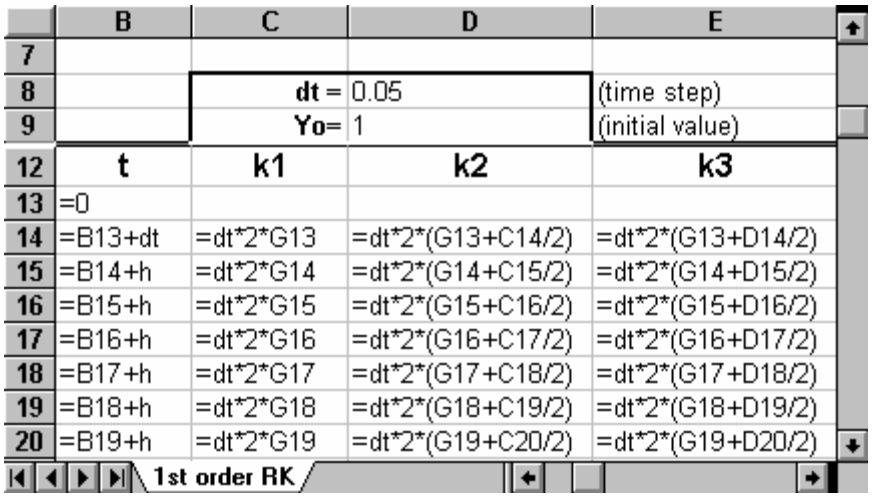
This algorithm is easily implemented in a spreadsheet by setting up columns for each of the quantities in the above equations, with each row representing the values at different times. An example is shown in Figure 1 and the formulas input to achieve the results in this figure are shown in Figures 2 and 3. In this example, $f(t,y)=2y$ and the initial condition is set to $A=1$. The analytical solution to this equation is $y = e^{2t}$.



| | A | B | C | D | E | F | G | H | I |
|---|---|---|---|---|---|---|---|---|---|
| 7 | | | | | | | | | |
| 8 | | | dt = | 0.05 | (time step) | | | | |
| 9 | | | Yo= | 1 | (initial value) | | | | |
| 10 | | | | | | | | | |
| 11 | | | | | | | | | |
| 12 | | t | k1 | k2 | k3 | k4 | y | analytical | |
| 13 | | 0.00 | | | | | 1.0000 | 1.0000 | |
| 14 | | 0.05 | 0.1000 | 0.1050 | 0.1053 | 0.1105 | 1.1052 | 1.1052 | |
| 15 | | 0.10 | 0.1105 | 0.1160 | 0.1163 | 0.1221 | 1.2214 | 1.2214 | |
| 16 | | 0.15 | 0.1221 | 0.1282 | 0.1286 | 0.1350 | 1.3499 | 1.3499 | |
| 17 | | 0.20 | 0.1350 | 0.1417 | 0.1421 | 0.1492 | 1.4918 | 1.4918 | |
| 18 | | 0.25 | 0.1492 | 0.1566 | 0.1570 | 0.1649 | 1.6487 | 1.6487 | |
| 19 | | 0.30 | 0.1649 | 0.1731 | 0.1735 | 0.1822 | 1.8221 | 1.8221 | |
| 20 | | 0.35 | 0.1822 | 0.1913 | 0.1918 | 0.2014 | 2.0138 | 2.0138 | |
| 21 | | 0.40 | 0.2014 | 0.2114 | 0.2119 | 0.2226 | 2.2255 | 2.2255 | |
| 22 | | 0.45 | 0.2226 | 0.2337 | 0.2342 | 0.2460 | 2.4596 | 2.4596 | |
| 23 | | 0.50 | 0.2460 | 0.2583 | 0.2589 | 0.2718 | 2.7183 | 2.7183 | |
| 24 | | | | | | | | | |

1st order RK

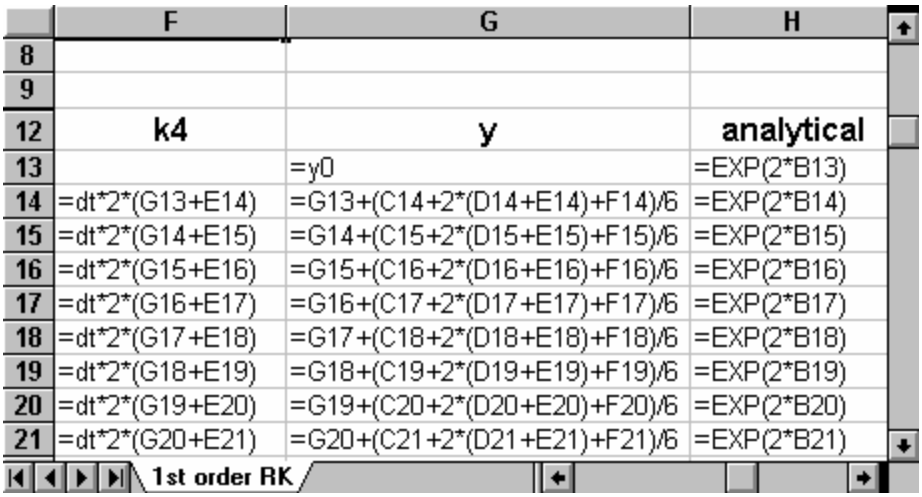
**Figure 1:** A sample spreadsheet using the standard spreadsheet interface to solve a first-order ordinary differential equation



| | B | C | D | E |
|---|---|---|---|---|
| 7 | | | | |
| 8 | | dt = | 0.05 | (time step) |
| 9 | | Yo= | 1 | (initial value) |
| 12 | t | k1 | k2 | k3 |
| 13 | =0 | | | |
| 14 | =B13+dt | =dt*2*G13 | =dt*2*(G13+C14/2) | =dt*2*(G13+D14/2) |
| 15 | =B14+h | =dt*2*G14 | =dt*2*(G14+C15/2) | =dt*2*(G14+D15/2) |
| 16 | =B15+h | =dt*2*G15 | =dt*2*(G15+C16/2) | =dt*2*(G15+D16/2) |
| 17 | =B16+h | =dt*2*G16 | =dt*2*(G16+C17/2) | =dt*2*(G16+D17/2) |
| 18 | =B17+h | =dt*2*G17 | =dt*2*(G17+C18/2) | =dt*2*(G17+D18/2) |
| 19 | =B18+h | =dt*2*G18 | =dt*2*(G18+C19/2) | =dt*2*(G18+D19/2) |
| 20 | =B19+h | =dt*2*G19 | =dt*2*(G19+C20/2) | =dt*2*(G19+D20/2) |

1st order RK

**Figure 2:** A sample spreadsheet, with formulas displayed, using the standard spreadsheet interface to solve a first-order ordinary differential equation. Only a portion of the formulas are shown here. The remainder are shown in Figure 3.



| | F | G | H |
|---|---|---|---|
| 8 | | | |
| 9 | | | |
| 12 | k4 | y | analytical |
| 13 | | =y0 | =EXP(2*B13) |
| 14 | =dt*2*(G13+E14) | =G13+(C14+2*(D14+E14)+F14)/6 | =EXP(2*B14) |
| 15 | =dt*2*(G14+E15) | =G14+(C15+2*(D15+E15)+F15)/6 | =EXP(2*B15) |
| 16 | =dt*2*(G15+E16) | =G15+(C16+2*(D16+E16)+F16)/6 | =EXP(2*B16) |
| 17 | =dt*2*(G16+E17) | =G16+(C17+2*(D17+E17)+F17)/6 | =EXP(2*B17) |
| 18 | =dt*2*(G17+E18) | =G17+(C18+2*(D18+E18)+F18)/6 | =EXP(2*B18) |
| 19 | =dt*2*(G18+E19) | =G18+(C19+2*(D19+E19)+F19)/6 | =EXP(2*B19) |
| 20 | =dt*2*(G19+E20) | =G19+(C20+2*(D20+E20)+F20)/6 | =EXP(2*B20) |
| 21 | =dt*2*(G20+E21) | =G20+(C21+2*(D21+E21)+F21)/6 | =EXP(2*B21) |

1st order RK

**Figure 3:** A sample spreadsheet, with formulas displayed, using the standard spreadsheet interface to solve a first-order ordinary differential equation. Only a portion of the formulas are shown here. The remainder are shown in Figure 4.

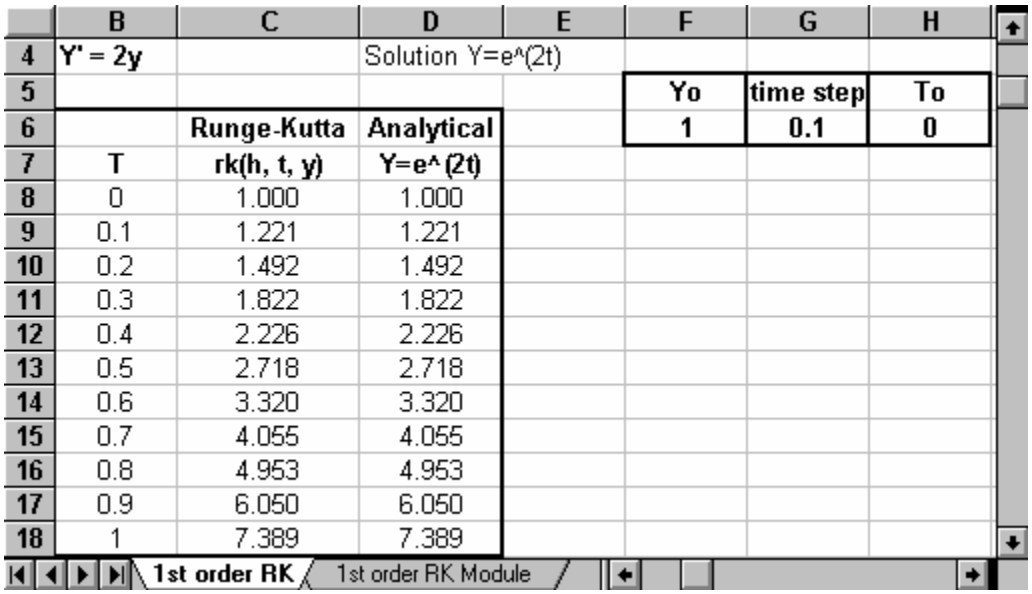
**Using a User-Defined Function**

One of the problems with the above approach to solving differential equations is the clutter on the screen caused by the printing of extraneous information. This is an advantage for the beginner, as it helps to maintain clarity during composition of the solution. But as the student becomes adept at Runge-Kutta solutions, the added values shown on the sheet become a nuisance. A cleaner approach uses a function macro to take a time step and a value for the dependent variable and provides an updated value for the dependent variable. This is implemented in a macro function called *rk*, which has the following form:

```
Function rk(h, t, y)
   k1 = h * f(t, y)
   k2 = h * f(t + h / 2, y + k1 / 2)
   k3 = h * f(t + h / 2, y + k2 / 2)
   k4 = h * f(t + h, y + k3)
   rk = y + (k1 + 2 * (k2 + k3) + k4) / 6
End Function

Function f(t, y)
   f = 2 * y
End Function
```

In the next example, we solved the same problem than before. Here *h* is the time step, *t* is the time at the beginning of the step, and *y* is the dependent variable at the beginning of the step. Repeated calls to this function will easily generate a solution to an initial-value problem. An example implementing this routine is shown in Figure 4, with the formulas shown in Figure 5. Note that in the spreadsheet formulas, *h* is the cell name for the cell holding the time step and *Y0* is the cell name for the cell holding the initial value of the dependent variable.



| | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|
| 4 | Y' = 2y | | Solution Y=e^(2t) | | | | |
| 5 | | | | | Yo | time step | To |
| 6 | | Runge-Kutta | Analytical | | 1 | 0.1 | 0 |
| 7 | T | rk(h, t, y) | Y=e^(2t) | | | | |
| 8 | 0 | 1.000 | 1.000 | | | | |
| 9 | 0.1 | 1.221 | 1.221 | | | | |
| 10 | 0.2 | 1.492 | 1.492 | | | | |
| 11 | 0.3 | 1.822 | 1.822 | | | | |
| 12 | 0.4 | 2.226 | 2.226 | | | | |
| 13 | 0.5 | 2.718 | 2.718 | | | | |
| 14 | 0.6 | 3.320 | 3.320 | | | | |
| 15 | 0.7 | 4.055 | 4.055 | | | | |
| 16 | 0.8 | 4.953 | 4.953 | | | | |
| 17 | 0.9 | 6.050 | 6.050 | | | | |
| 18 | 1 | 7.389 | 7.389 | | | | |

1st order RK | 1st order RK Module

**Figure 4:** A sample spreadsheet using a function macro to solve a first-order ordinary differential equation

| | B | C | D |
|---|---|---|---|
| 4 | Y' = 2y | | Solution Y=e^(2t) |
| 5 | | | |
| 6 | | Runge-Kutta | Analytical |
| 7 | T | rk(h, t, y) | Y=e^ (2t) |
| 8 | 0 | =Y0 | =EXP(2*$B8) |
| 9 | =$B8+$G$6 | =rk(h,$B8,$C8) | =EXP(2*$B9) |
| 10 | =$B9+$G$6 | =rk(h,$B9,$C9) | =EXP(2*$B10) |
| 11 | =$B10+$G$6 | =rk(h,$B10,$C10) | =EXP(2*$B11) |
| 12 | =$B11+$G$6 | =rk(h,$B11,$C11) | =EXP(2*$B12) |
| 13 | =$B12+$G$6 | =rk(h,$B12,$C12) | =EXP(2*$B13) |
| 14 | =$B13+$G$6 | =rk(h,$B13,$C13) | =EXP(2*$B14) |
| 15 | =$B14+$G$6 | =rk(h,$B14,$C14) | =EXP(2*$B15) |
| 16 | =$B15+$G$6 | =rk(h,$B15,$C15) | =EXP(2*$B16) |
| 17 | =$B16+$G$6 | =rk(h,$B16,$C16) | =EXP(2*$B17) |
| 18 | =$B17+$G$6 | =rk(h,$B17,$C17) | =EXP(2*$B18) |

1st order RK | 1st order RK Mod

**Figure 5:** A sample spreadsheet, with formulas displayed, using a function macro to solve a first-order ordinary differential equation

**A System of Two First-Order Ordinary Differential Equations**

The situation is somewhat more complicated when solving a system of two first-order equations (or a second-order equation). Function macros in Visual Basic can only return one value. Hence, it is difficult to solve for two dependent variables simultaneously. However, a function can still be used if the derivative of the dependent variable need not be returned to the spreadsheet. The trick is to use a Static Function to save the value of one of the derivatives between function calls.

To solve two ordinary differential equations of the following type:

$$\frac{dy}{dt} = f(t, y, z)$$
$$\frac{dz}{dt} = g(t, y, z)$$

the previous algorithm can be extended to the solution of two first-order equations. This gives:

$$k_1 = \Delta t * f(t, y, z)$$
$$l_1 = \Delta t * g(t, y, z)$$
$$k_2 = \Delta t * f(t + \frac{\Delta t}{2}, y + \frac{k_1}{2}, z + \frac{l_1}{2})$$
$$l_2 = \Delta t * g(t + \frac{\Delta t}{2}, y + \frac{k_1}{2}, z + \frac{l_1}{2})$$
$$k_3 = \Delta t * f(t + \frac{\Delta t}{2}, y + \frac{k_2}{2}, z + \frac{l_2}{2})$$
$$l_3 = \Delta t * g(t + \frac{\Delta t}{2}, y + \frac{k_2}{2}, z + \frac{l_2}{2})$$
$$k_4 = \Delta t * f(t + \Delta t, y + k_3, z + l_3)$$
$$l_4 = \Delta t * g(t + \Delta t, y + k_3, z + l_3)$$
$$y(t + \Delta t) = y(t) + \frac{(k_1 + 2 * (k_2 + k_3) + k_4)}{6}$$
$$z(t + \Delta t) = z(t) + \frac{(l_1 + 2 * (l_2 + l_3) + l_4)}{6}$$

To implement this algorithm in a spreadsheet, one uses the following macro:

```
Static Function RK(h, t, y)
   If t = 0 Then z = 0
   k1 = h * g(t, y, z)
   l1 = h * f(t, y, z)
   k2 = h * g(t + h / 2, y + k1 / 2, z + l1 / 2)
   l2 = h * f(t + h / 2, y + k1 / 2, z + l1 / 2)
   k3 = h * g(t + h / 2, y + k2 / 2, z + l2 / 2)
   l3 = h * f(t + h / 2, y + k2 / 2, z + l2 / 2)
   k4 = h * g(t + h, Y + k3, z + l3)
   l4 = h * f(t + h, Y + k3, z + l3)
   z = z + (l1 + 2 * (l2 + l3) + l4) / 6
   RK = y + (k1 + 2 * (k2 + k3) + k4) / 6
End Function

Function g(t, y, z)
   g = z
End Function

Function f(t, y, z)
   f = -4 * y
End Function
```

The same approach can be taken to solve a second order differential equation

$$\frac{d^2y}{dt^2} = -4y$$

by breaking the equation into two first order equations:

$$\frac{dy}{dt} = z$$
$$\frac{dz}{dt} = -4y$$

One complication with this approach is that one must be able to treat the initial value for the dependent variable which is not returned to the spreadsheet (in this case, *z*). This is more difficult than in the single-variable case because the current value of *z* is not passed to the function as an argument. The first line of the subroutine solves this problem by setting the initial value for *z* when *t=0.*

**Using a Subroutine to Solve the Equation**

One drawback of this solution is that you must copy the appropriate formula into as many cells as is needed to generate a solution. This can be cumbersome, so a more fully automated alternative would be desirable. This can be achieved using a subroutine. In this case the subroutine carries out all the steps necessary to solve the problem. This has the added benefit that the user can choose to only print (in the spreadsheet) a subset of the steps. A subroutine developed for implementing this algorithm to solve second-order differential equations is:

```
Sub RungekuttaSR2call()
    [b14].Select
    steps = Range("steps")
    tnot = Range("tnot")
    tend = Range("tend")
    Yo = Range("Yo")
    Zo = Range("Zo")
    h = (tend - tnot) / steps
    T = 0
    Y = Yo
    z = Zo
    For i = 1 To steps
      Call rk2sr(h, T, Y, z, ynew, znew)
      ActiveCell.Offset(i - 1, 0).Value = T
      ActiveCell.Offset(i - 1, 1).Value = Y
      ActiveCell.Offset(i - 1, 2).Value = z
      T = T + h
      Y = ynew
      z = znew
   Next
End Sub
```

This subroutine takes values for the number of time steps (*steps*), the beginning time (*tnot)*, and the finishing time (tend), along with initial values for both dependent variables (*Yo* and *Zo*) and then makes repeated calls to a stepping routine (*rk2sr*) to advance the solution. The *Range* command is used to read input values from the spreadsheet and the *ActiveCell.Offset(i, j).Value* command is used to write the results back to the spreadsheet, offsetting the values to write the results for each time step on a different line. The stepping routine and necessary functions are:

```
Sub rk2sr(h, t, y, z, ynew, znew)
    k1 = h * g(t, y, z)
    l1 = h * f(t, y, z)
    k2 = h * g(t + h / 2, y + k1 / 2, z + l1 / 2)
    l2 = h * f(t + h / 2, y + k1 / 2, z + l1 / 2)
    k3 = h * g(t + h / 2, y + k2 / 2, z + l2 / 2)
    l3 = h * f(t + h / 2, y + k2 / 2, z + l2 / 2)
    k4 = h * g(t + h, y + k3, z + l3)
    l4 = h * f(t + h, y + k3, z + l3)
    znew = z + (l1 + 2 * (l2 + l3) + l4) / 6
    ynew = y + (k1 + 2 * (k2 + k3) + k4) / 6
End Sub

Function g(t, y, z)
g = z
End Function

Function f(t, y, z)
f = -4 * y
End Function
```

The stepping subroutine *rk2sr* takes values for the time step, time, and both dependent variables and calculates new values for the dependent variables using a fourth-order Runge-Kutta scheme. The two functions define the desired differential equation. An example of this type of spreadsheet is shown in Figure 6. The user merely inputs the initial values, the number of steps desired, and the time interval, and then the solution is obtained by pressing the button on the sheet.

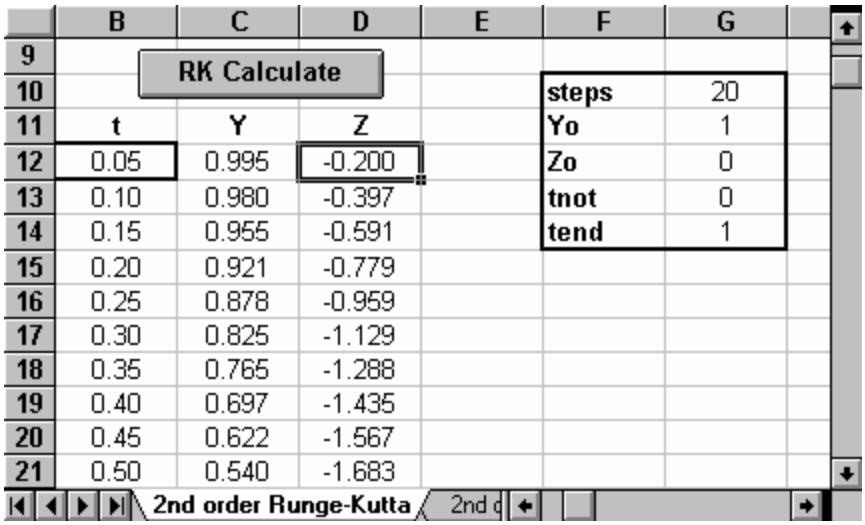



**Figure 6:** A sample spreadsheet using a subroutine macro to solve a second-order ordinary differential equation. There are no formulas, because the macro writes the solution directly to the spreadsheet.

**An Adaptive Method for First-Order Equations**

The techniques described above are useful because they are easily implemented, but they are inherently inefficient. When the step size is fixed, one must choose it such that the error induced in regions where the solution changes most rapidly is below some desired value. Obviously, this step size will be smaller than necessary in all other regions. A more efficient method will adjust the time step according to the local variation in the solution, requiring that this local solution achieve a prescribed accuracy.

To build an adaptive time step solver, the algorithm must return information about its progress and an estimate of its truncation error. In this case, our algorithm is based on the Runge-Kutta-Fehlberg[1] method, which uses the embedded Runge-Kutta formulas to adjust the step. The general form of a fifth-order Runge-Kutta formula is

$$k_1 = h \cdot f\left(x_n, y_n\right)$$
$$k_2 = h \cdot f\left(x_n + a_2 h, y_n + b_{21} k_1\right)$$
$$\cdots$$
$$k_6 = h \cdot f\left(x_n + a_6 h, y_n + b_{61} k_1 + \cdots + b_{65} k_5\right)$$

$$y_{n+1} = y_n + c_1 k_1 + c_2 k_2 + c_3 k_3 + c_4 k_4 + c_5 k_5 + c_6 k_6 + O\left(h^6\right)$$

The embedded fourth-order formula is

$$y^*_{n+1} = y_n + c^*_1 k_1 + c^*_2 k_2 + c^*_3 k_3 + c^*_4 k_4 + c^*_5 k_5 + c^*_6 k_6 + O\left(h^5\right)$$

The values of the needed constants that are used are given in the following table. These are not Fehlberg's original values, but those found by Cash and Karp[2], who provide a more effective method with better error properties[1].

| i | $a_i$ | $b_{ij}$ | | | | | $c_i$ | $c_i^*$ |
|---|---|---|---|---|---|---|---|---|
| 1 | | | | | | | $\frac{37}{378}$ | $\frac{2825}{27648}$ |
| 2 | $\frac{1}{5}$ | $\frac{1}{5}$ | | | | | 0 | 0 |
| 3 | $\frac{3}{10}$ | $\frac{3}{40}$ | $\frac{9}{40}$ | | | | $\frac{250}{621}$ | $\frac{18575}{48384}$ |
| 4 | $\frac{3}{5}$ | $\frac{3}{10}$ | $-\frac{9}{10}$ | $\frac{6}{5}$ | | | $\frac{125}{594}$ | $\frac{13525}{55296}$ |
| 5 | 1 | $-\frac{11}{54}$ | $\frac{5}{2}$ | $-\frac{70}{27}$ | $\frac{35}{27}$ | | 0 | $\frac{277}{14336}$ |
| 6 | $\frac{7}{8}$ | $\frac{1631}{55296}$ | $\frac{175}{512}$ | $\frac{575}{13824}$ | $\frac{44275}{110592}$ | $\frac{253}{4096}$ | $\frac{512}{1771}$ | $\frac{1}{4}$ |
| j | = | 1 | 2 | 3 | 4 | 5 | | |

**Table 1**: Cash-Karp Parameters for the Embedded Runge-Kutta Method

The error is then calculated as

$$error_y = y_{n+1} - y^*_{n+1} = \sum_{i=1}^{6} \left(c_i - c^*_i\right) \cdot k_i$$

From these equations it is found that the error progresses as $h^5$. Thus, if an error $(error_y)_1$ is produced when taking a step $h_1$, one could easily infer what step $h_0$ would need to take in order to produce an error $(error_y)_0$.

$$h_0 = h_1 \cdot \left| \frac{(error_y)_0}{(error_y)_1} \right|^{0.2}$$

Now, by calling $(error_y)_0$ the desired accuracy, this equation can be used in two ways:

1. If the desired is larger than the calculated error, it determines how much the time step must be decreased when the current step is repeated.

2. If the desired error is smaller than the calculated error, it determines how much the time step can safely be increased when the next step is calculated.

In addition, a reference value of the dependent variable is needed for comparison with the calculated error. For example, in some cases one would like to obtain constant fractional errors at each step and in other occasions one is interested in keeping low the global accumulation of errors. The form of the reference error value in our algorithm is

$$value_{ref} = Abs(y) + Abs\left(h \cdot \frac{dy}{dt}\right)$$

The first term of this equation is valid when the solution reaches an asymptotic value different from zero. The second term is valid when the solution passes through zero with a certain slope. This form, however, would cause problems when the function passes through zero with zero slope. Moreover, because of this new definition of the error, which has an implicit scaling with the time step, the exponent 0.25 must be used instead of 0.20 when decreasing the time step. That is:

$$h_0 = h_1 \cdot \left| \frac{(error_y)_0}{(error_y)_1} \right|^{a} \qquad with \qquad \begin{array}{llll} a = 0.20 & stepsize & increase \\ a = 0.25 & stepsize & decrease \end{array}$$

Then, to solve an ordinary differential equation of the form

$$\frac{dy}{dt} = f(t, y)$$

with the initial condition

$$y(0) = A,$$

the macro described below can be used. The central portion of the macro loops through a series of optimized time steps until the requested time is reached. Our version of this macro is:

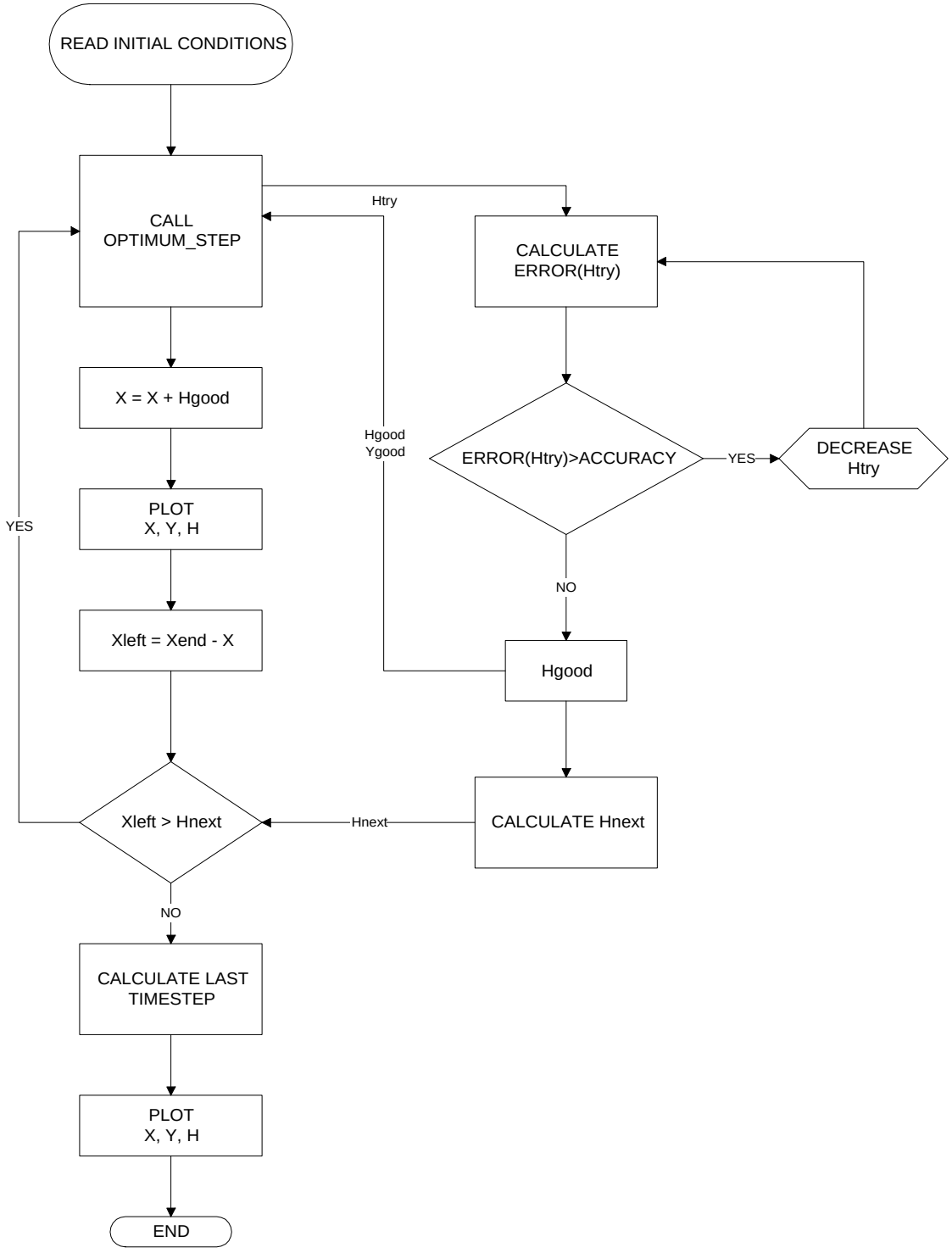
```
Sub adaptive_rungekutta()
    [F13].Select
    htry = Range("htry").Value
    ynot = Range("ynot").Value
    tnot = Range("tnot").Value
    tend = Range("tend").Value
    h = htry
    t = tnot
    y = ynot
    dt = tend - tnot
    hnew = htry
    ActiveCell.Value = t
    ActiveCell.Offset(0, 1).Value = y
    ActiveCell.Offset(0, 2).Select
    Do
        Call optimize(h, t, y, hnew)
        t = t + h
        ActiveCell.Value = h
        ActiveCell.Offset(1, -2).Value = t
        ActiveCell.Offset(1, -1).Value = y
        ActiveCell.Offset(1, 0).Select
        h = hnew
        tnew = t
        dt = tend - tnew
    Loop Until dt < hnew
    h = dt
    Call rkfive(h, t, y, ynew, ynewstar)
    y = ynew
    t = t + dt
    ActiveCell.Value = dt
    ActiveCell.Offset(1, -2).Value = t
    ActiveCell.Offset(1, -1).Value = y
End Sub
```

The central call here is the call to the procedure *optimum_step*, which chooses the step size needed to achieve a desired accuracy. This procedure is:

```
Sub optimize(h, t, y, hnew)
    accuracy = Range("accuracy").Value
    safety = Range("safety").Value
    expup = -0.2
    expdown = -0.25
    limerr = (5 / safety) ^ (1 / expup)
    dummy = 1
    Do
        Call rkfive(h, t, y, ynew, ynewstar)
        yerr = Abs(ynew - ynewstar)
        yscal = Abs(ynew) + Abs(h * func(t, y))
        ratio = Abs(yerr / yscal)
        maxerr = ratio / accuracy
        If maxerr > 1 Then
            hnext = safety * h * (maxerr ^ expdown)
            If hnext < (0.1 * h) Then hnext = 0.1 * h
            h = hnext
            dummy = 1
        Else
            y = ynew
            dummy = 0
        End If
    Loop Until dummy = 0
        If maxerr > limerr Then
            hnew = safety * h * (maxerr ^ expup)
        Else
            hnew = 5 * h
        End If
 End Sub
```

This procedure calls other procedures which are not displayed here. The *rkfive* routine carries out single fifth-order and embedded-fourth-order Runge-Kutta steps, returning the values as *ynew* and *ynewstar*, respectively.

The logic of these procedures is described by the flow-chart in Figure 7.




.

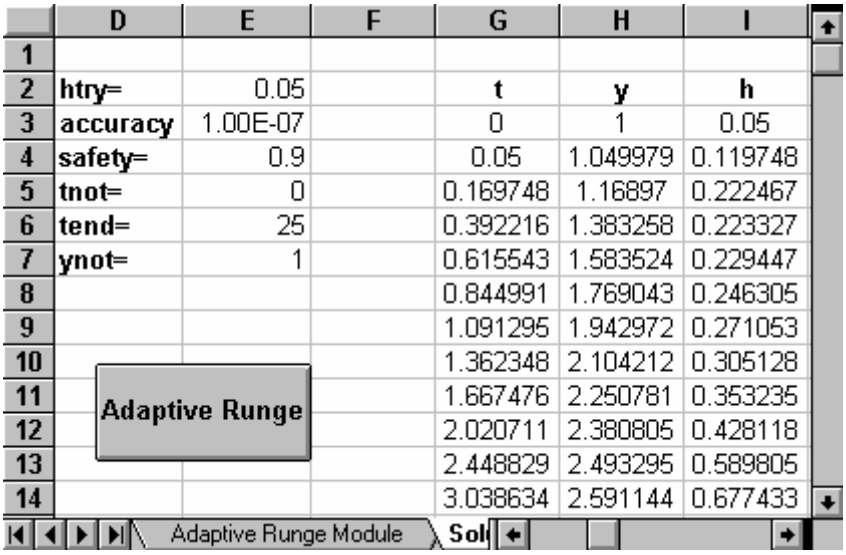
**Figure 7:** Flow chart for adaptive Runge-Kutta method.

The general idea behind an adaptive time step method for multidimensional problems is the same as the method seen for one-dimensional equations. The major complication is deciding how to define the appropriate error for a multi-dimensional system. The approach adopted here is to consider the errors in each dependent variable separately and choose an appropriate time step to maintain the desired error for each variable at each time step. That is, we deal only with the maximum error in each step. An implementation of this scheme for systems of two first-order ordinary differential equations is included in the enclosed diskette.

An example of such a solution is shown in Figure 8. This spreadsheet solves the first-order, ordinary differential equation:

$$\frac{dy}{dt} = y * e^{-t}$$

with the initial condition *y(0)=1*. As is shown, the step size is increased as the solution approaches the steady state solution, thus maximizing the efficiency of the solution. A second spreadsheet, designed to solve a system of two equations using the same algorithm is provided with the enclosed floppy disk.




| | D | E | F | G | H | I |
|---|---|---|---|---|---|---|
| 1 | | | | | | |
| 2 | htry= | 0.05 | | t | y | h |
| 3 | accuracy | 1.00E-07 | | 0 | 1 | 0.05 |
| 4 | safety= | 0.9 | | 0.05 | 1.049979 | 0.119748 |
| 5 | tnot= | 0 | | 0.169748 | 1.16897 | 0.222467 |
| 6 | tend= | 25 | | 0.392216 | 1.383258 | 0.223327 |
| 7 | ynot= | 1 | | 0.615543 | 1.583524 | 0.229447 |
| 8 | | | | 0.844991 | 1.769043 | 0.246305 |
| 9 | | | | 1.091295 | 1.942972 | 0.271053 |
| 10 | | | | 1.362348 | 2.104212 | 0.305128 |
| 11 | | | | 1.667476 | 2.250781 | 0.353235 |
| 12 | | | | 2.020711 | 2.380805 | 0.428118 |
| 13 | | | | 2.448829 | 2.493295 | 0.589805 |
| 14 | | | | 3.038634 | 2.591144 | 0.677433 |

**Figure 8:** A sample spreadsheet using a subroutine macro to solve a first-order ordinary differential equation. The step size is adjusted to achieve the prescribed accuracy.

**Conclusions**

A modern computational tool such as Microsoft Excel 5.0 can be an excellent platform for teaching engineering computation. It has a simple interface, allowing focus on various numerical algorithms, but it also has a powerful macro language that allows the implementation of more sophisticated algorithms. An example implementing fourth-order Runge-Kutta algorithms for the solution of ordinary differential equations demonstrates these principles.

**References**

[1] W. H. Press, S. A. Teukolsky, W. T. Vettering, B. P. Flannery. *"Numerical Recipes in Fortran. The Art of Scientific Computing."*. 2nd Edition. Cambridge University Press. 1992.

[2] P. Kaps, P. Rentrop. *"Numerische Mathematik"* vol. 8, 1979, pp. 93-113.