VBA PROGRAMMING TECHNIQUES IN EXCEL FOR A CROSS DISCIPLINE MATHEMATICS, PHYSICS AND CHEMISTRY COURSE

David Nawrocki
Albright College
13th and Bern Streets
Reading, Pennsylvania 19612
dnawrocki@alb.edu

In the spring of 2003 the physics, mathematics and chemistry departments at Albright College developed an interdisciplinary course that used visual basic programming in Microsoft Excel as a problem-solving tool. The decision to use Microsoft Excel over other packages, such as, Matlab, Mathematica, and Maple was made for several reasons. It had become apparent that many of our students were using Excel as a problem-solving tool upon graduation. Excel is relatively cheap to implement on a limited budget. Most students already have limited experience using Excel, and the programming tools included within Excel give students valuable programming experience that they are not exposed to in other courses. Although the course included numerous topics from physics and chemistry, we will only discuss a few of the numerical techniques used as part of the mathematical component of the course.

There are two ways to approach numerical problems in Excel. The first technique we will discuss is the cell-formula technique. This technique gives students the ability to solve relatively complex problems quickly without the need for advanced programming skills. Numerical techniques for solving differential equations can be illustrated nicely using cell formulas. For example, recall that Euler's method is described iteratively by the equation,

$$y_{n+1} = y_n + hf(x_n, y_n)$$

and the improved Euler method is described by,

$$y_{n+1} = y_n + h\frac{f(x_n, y_n) + f(x_{n+1}, y_n + hf(x_n, y_n))}{2}.$$

Figure 1 below shows how cell formulas can be used to solve and compare numerical solutions to $y' = xy$, $y(0)=1$ with step sizes $h=0.1$. By selecting cells (A4:C4), you can grab the cells handle (the small rectangle graphic at the lower right corner) with the mouse and drag a copy of the cells downward on the spreadsheet to yield the results shown in Figure 2. Notice that in Figure 2 we have included an exact solution column. This can be included by entering the cell formula =*EXP(A2^2/2)* in cell D2 and copying.

	A	B	C
1	x	y (Euler)	y (Improved Euler)
2	0	1	1
3	=A2+0.1	=B2+0.1*(A2*B2)	=C2+0.1/2*((A2*C2)+(A3*(C2+0.1*(A2*C2))))

Figure 1: Euler and Improved Euler Cell Formulas

	A	B	C	D
1	x	y (Euler)	y (Improved Euler)	Exact Solution
2	0	1	1	1
3	0.1	1	1.005	1.005012521
4	0.2	1.01	1.0201755	1.02020134
⋮	⋮	⋮	⋮	⋮
10	0.8	1.314229016	1.376773106	1.377127764
11	0.9	1.419367338	1.498755203	1.4993025
12	1	1.547110398	1.647881346	1.648721271

Figure 2: Euler and Improved Euler Outputs

Using this technique it is easy to compare convergence rates of various numerical approximation methods. Using Excel's charting feature it is possible to display a graphical comparison of the methods as is shown in Figure 3.

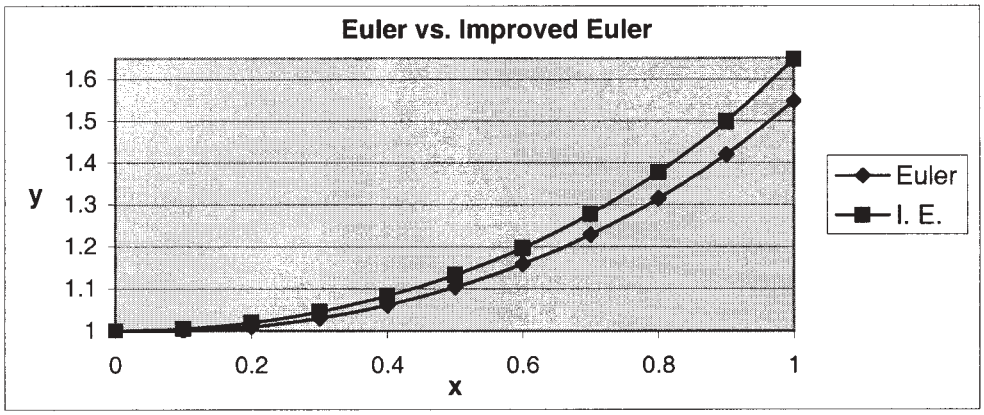

Figure 3: Euler and Improved Euler Output Graphs

Since it is a bit cumbersome to use the cell formulas *=B2+0.1*(A2*B2)* and *=C2+0.1/2*((A2*C2)+(A3*(C2+0.1*(A2*C2))))*, as shown in Figure 1, to represent the Euler and improved Euler iterations, the Visual Basic for Applications (VBA) editor can be used to create user-defined functions that make the cell formulas easier to work with and more robust. The VBA editor is a built in feature that comes with Excel and can be accessed via the menu selection Tools | Macro | Visual Basic Editor. This will open the VBA editor or return you to it if it is already open. Alternatively, pressing *ALT + F11* will also open the editor. After opening the editor you will see several windows. If you are working with a new Excel document the project window on the left side of the screen will contain a folder named *Microsoft Excel Objects*. From the menu select Insert | Module. This will create a new module and another folder (*Modules*) will appear in the project window. Double click on the new Module (*Module1*) to select it and in the editor window to the right type the following lines of code,

```
1       ' A user defined function
2       Function f(x, y)
3           f = x * y
```

```
4       End Function
```

Do not to type the line numbers as they are used only for reference in this paper. Line 1 is a comment line, and line 2 is the function declaration. This particular function name is *f* and it takes two arguments, *x* and *y*. Line 3 defines the return value for the function determined by the arguments *x* and *y* passed to the function from line 2. Line 4 defines the end of the function declaration.

Returning to the worksheet, Figure 4 illustrates the use of the user-defined function, *f(x,y)*, just created. Copying row 3 will yield the same results as shown in Figure 1, but it is now much easier to identify the contents of cells B3 and C3 with the Euler and improved Euler methods.

	A	B	C
1	x	y (Euler)	y (Improved Euler)
2	0	1	1
3	=A2+0.1	=B2+0.1*f(A2,B2)	=C2+0.1/2*(f(A2,C2)+f(A3,(C2+0.1*f(A2,C2))))

Figure 4: Euler and Improved Euler Cell Formulas with User-Defined Function Calls

Although the Runge-Kutta method can also be implemented via cell formulas, to illustrate more advanced programming techniques, we will use Visual Basic. Directly below the code defining the user-defined function *f(x,y)* created earlier, enter the following lines of code,

```
1       ' The Runge-Kutta Method
2       Sub RK()
3         h = Val(InputBox("Enter the Step Size h", "Runge-Kutta", 0.1))
4         numIter = Val(InputBox("Enter the Number of Iterations", "Runge-Kutta",
                100))
5         x = Val(InputBox("Enter the Initial Value for x", "Runge-Kutta", 0))
6         y = Val(InputBox("Enter the Initial Value for y", "Runge-Kutta", 0))
7         r = ActiveCell.Row
8         c = ActiveCell.Column
9         Cells(r, c).Value = x
10        Cells(r, c + 1).Value = y
11        For i = 1 To numIter
12          k1 = h * f(x, y)
13          k2 = h * f(x + h / 2, y + k1 / 2)
14          k3 = h * f(x + h / 2, y + k2 / 2)
15          k4 = h * f(x + h, y + k3)
16          x = x + h
17          y = y + (k1 + 2 * k2 + 2 * k3 + k4) / 6
18          Cells(r + i, c).Value = x
19          Cells(r + i, c + 1).Value = y
20        Next
21      End Sub
```

When entering the code, do not break up line 4. Notice that this section of code initiates what is called a subroutine. Like the function initiated earlier, a subroutine is a procedure that executes specific statements, but it does not return a value. Notice within the body of the function the line $f = x * y$ where f is the name of the function. The subroutine does not have such a line, hence, the subroutine does not return a value. Although subroutines do not return values they can perform other tasks that are not allowed within functions.

Lines 2 and 21 declare and end the subroutine, and lines 3-6 instantiate dialog boxes that allow the user to input values for the execution of the Runge-Kutta method. Lines 7 and 8 store the location of the active cell when the subroutine is run. The subroutine uses this information to determine where within the worksheet to write the output of the subroutine. Lines 9 and 10 write the initial x condition for the problem into the active cell on the active worksheet (*Cell(r,c)*) and the initial y condition for the problem into the cell to the right of the active cell on the active worksheet (*Cell(r,c+1)*). Lines 11-20 then execute the Runge-Kutta method and lines 18 and 19 write the output in the rows directly below the cells where the initial conditions were written.

To execute the Runge-Kutta subroutine, set up a worksheet as shown in Figure 5 and select cell A2.

	A	B
1	x	y (Runge-Kutta)
2		

Figure 5: Runge-Kutta Set Up

From the menu select Tools | Macro | Macros (or press *Alt + F8*). You will see the subroutine *RK* as a macro selection. Select it and press Run. The output in Figure 6 will be generated.

	A	B
1	x	y (Runge-Kutta)
2	0	1
3	0.1	1.00501252083333
4	0.2	1.02020133975837
⋮	⋮	⋮
10	0.8	1.37712769490194
11	0.9	1.49930236244832
12	1	1.6487210070534

Figure 6: Runge-Kutta Output

It is also possible to use Excel to explore numerical solutions to systems of differential equations. The following code will execute the Runge-Kutta method for the Lotka-Volterra predator-prey model,

$$u_1'(t) = 2u_1(t) - 1.2u_1(t)u_2(t)$$
$$u_2'(t) = -u_1(t) + 1.2u_1(t)u_2(t).$$

```
1     'The following two user defined functions are used for the Runge-Kutta Method
2     Function gr(t, u1, u2)
3       gr = 2 * u1 - 1.2 * u1 * u2
4     End Function
5     Function fr(t, u1, u2)
6       fr = -u2 + 1.2 * u1 * u2
7     End Function
8     ' The Runge-Kutta method for a system of 2 differential equations
9     Sub RKS()
10      h = Val(InputBox("Enter the Step Size h", "Runge-Kutta", 0.1))
11      numIter = Val(InputBox("Enter the Number of Iterations", "Runge-Kutta",
              100))
12      t = Val(InputBox("Enter the Initial Value for t", "Runge-Kutta", 0))
13      u1 = Val(InputBox("Enter the Initial Value for u1", "Runge-Kutta", 0))
14      u2 = Val(InputBox("Enter the Initial Value for u2", "Runge-Kutta", 0))
15      r = ActiveCell.Row
16      c = ActiveCell.Column
17      Cells(r, c).Value = t
18      Cells(r, c + 1).Value = u1
19      Cells(r, c + 2).Value = u2
20      For i = 1 To numIter
21        k1 = h * gr(t, u1, u2)
22        q1 = h * fr(t, u1, u2)
23        k2 = h * gr(t + h / 2, u1 + k1 / 2, u2 + q1 / 2)
24        q2 = h * fr(t + h / 2, u1 + k1 / 2, u2 + q1 / 2)
25        k3 = h * gr(t + h / 2, u1 + k2 / 2, u2 + q2 / 2)
26        q3 = h * fr(t + h / 2, u1 + k2 / 2, u2 + q2 / 2)
27        k4 = h * gr(t + h, u1 + k3, u2 + q3)
28        q4 = h * fr(t + h, u1 + k3, u2 + q3)
29        t = t + h
30        u1 = u1 + (k1 + 2 * k2 + 2 * k3 + k4) / 6
31        u2 = u2 + (q1 + 2 * q2 + 2 * q3 + q4) / 6
32        Cells(r + i, c).Value = t
33        Cells(r + i, c + 1).Value = u1
34        Cells(r + i, c + 2).Value = u2
35      Next
36    End Sub
```

References

1. Burden, Richard and Faires, Douglas. *Numerical Analysis*. Pacific Grove, CA: Brooks/Cole Publishing Company, 1997.
2. Liengme, Bernard. *A Guide to Microsoft Excel 2002 for Scientist and Engineers*. Woburn, MA: Butterworth-Heinemann, 2002.
3. Roman, Steven. *Writing Excel Macros with VBA*. Sabastopol, CA: O'Reilly & Associates, 2002.