2. Solving Differential Equations by Computer

If a differential equation or an initial value problem is too difficult to solve using the methods of Chapter 1 then we can use a computer to solve it. There are two types of computer methods:

- **Analytical**. This means that the computer program uses precisely the methods of Chapter 1 to solve the problem – it just does it very quickly. Furthermore, the computer expresses the solution in closed form (that is, as a formula involving elementary functions such as $\sin(x)$, e^x, x^2, etc.) exactly as we did in Chapter 1. These programs are also called *computer algebra systems*. Maple is an example.
- **Numerical**. This means that the computer uses numerical approximation as opposed to symbolic manipulation to solve an *initial value problem*. Starting with an initial value, the computer steps forward in small time steps, generating a list of values that can then be plotted in a graph. (Note that numerical methods can only produce particular solutions, not general solutions.)

  Look at the slope field in Fig. 2.1 on page 69. Roughly speaking, a numerical method connects the arrows of the slope field end-to-end as accurately as desired. Because the output is a table of values a spreadsheet program such as Excel is ideal for this method. To use Excel efficiently it can be programmed using Visual Basic.

We first describe how to solve DE's or IVP's using Maple. Then we look at several numerical methods for solving IVP's and show how to implement them in Excel.

2.1 Solving Differential Equations Using Maple

The Maple command to find the general solution of a differential equation is `dsolve(DE, y(x))` and the command to solve an initial value problem is `dsolve({DE, ICs}, y(x))`. DE is the differential equation and ICs are the initial conditions. It is a good idea to define them first and then run the dsolve command.

In the DE use `diff(y(x),x)` to denote dy/dx, `diff(y(x),x,x)` to denote d^2y/dx^2, etc.

In the ICs use `y(0)=3, D(y)(0)=4, (D@@2)(y)(0)=5` to denote $y(0)=3, y'(0)=4, y''(0)=5$, etc.

Let's use Maple to solve question 18 of Problem Set 1.5. First define the DE and the ICs:

```
> de := diff(y(t),t,t) + 2*diff(y(t),t)+y(t) = exp(-t);
> ics := y(0)=0, D(y)(0)=-3;
```

(Note that := means *is defined as*…) Then enter this command to find the general solution:

```
>dsolve(de, y(t));
```

$$y(t) = e^{-t} _C2 + e^{-t} t _C1 + \frac{1}{2} t^2 e^{-t}$$

(Note that in the solution Maple uses the names _C1 and _C2 for the arbitrary constants.) Enter this command to include the ICs and solve the initial value problem:

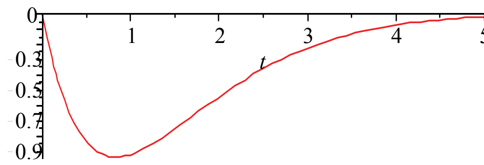
```
>dsolve({de,ics}, y(t));
```

$$y(t) = -3 e^{-t} t + \frac{1}{2} t^2 e^{-t}$$

If you want to plot this function you need to use this command:

```
> plot ( rhs (%), t=0..5);
```

(Here % is defined as *the last thing calculated*, rhs ( ) is a function that returns the right-hand-side of something, and t = 0 .. 5 means plot from 0 to 5.) Here is the plot:



2.2 Solving Differential Equations Numerically

We will first learn Euler's method, which is useful to introduce the basic ideas but is not very accurate. Then we will learn the Runge-Kutta method which is much more accurate and is the most widely used numerical method to solve differential equations.

Euler's Method

We will illustrate Euler's method with an example. Consider the first order differential equation

$$\frac{dy}{dt} = t + y, \tag{2.1}$$

subject to the initial condition that $y=1$ when $t=1$. Recall from Chapter 4 on differentials in *Calculus for Electrical Technology* that dy and dt represent the rise and run of a tangent line to the curve. Let the run have a small constant value h. (From now on we will call h the **time step size**.) Replace the rise by $y_{n+1} - y_n$, where y_n denotes the value of y at the beginning of the time step and y_{n+1} denotes the value of y at the end of the time step. Then Euler's method consists of replacing (2.1) by the so-called difference equation

$$\frac{y_{n+1} - y_n}{h} = t_n + y_n. \tag{2.2}$$

Solving (2.2) for y_{n+1} gives

$$y_{n+1} = y_n + h(t_n + y_n). \tag{2.3}$$

Eq. (2.3) can be interpreted as outputting the value of y at the *end* of the nth timestep when we have input the values of t and y at the *beginning* of the nth timestep. If we write the initial condition as $y_0=1$ at $t_0=1$ and substitute it into the RHS then we see that this formula returns y_1, and if we substitute y_1 and t_1 into the RHS then the formula returns y_2, and so on. This is called an **iterative equation**. We can step forward in time as many timesteps as we want.

The following table lists the values of t in column 2 and Euler's approximation to y in column 3 for two timesteps. (The first row just restates the initial conditions.) We used the timestep size $h=1$. For comparison, the analytical solution for this initial value problem is $y = 3e^{t-1} - t - 1$, and the table lists its values in column 4. We see that Euler's method is not very accurate.

index n	time t_n	Euler's approx. to y_n	Exact solution $y_n = 3e^{t_n-1} - t_n - 1$
0	1	1 (initial condition)	1
1	2	1+1(1+1) = 3	5.2
2	3	3+1(2+3) = 8	18.2

Looking at Fig. 2.1 we can see why. Euler's method uses the slope at the *beginning* of the timestep to calculate the solution forward to the end of the timestep. The slope field (the gray background arrows) shows that the slope changes considerably from the beginning of the timestep to the end. One obvious improvement would be to use a smaller timestep size.

There are two sources of error in Euler's method. The first is called truncation error and is due to approximating Eq. (2.1) by (2.2). The second is called accumulation error and is due to the fact that the method is iterative so the error in each timestep causes a worse error in the next timestep. The following table shows how decreasing the stepsize h reduces the total error in computing forward from $t = 1$ to $t = 3$.

h	steps required	Euler's approx. to $y(3)$ (exact val.=18.2)	Error
1	2	8	10.2
0.5	4	11.2	7.0
0.1	20	16.2	2.0

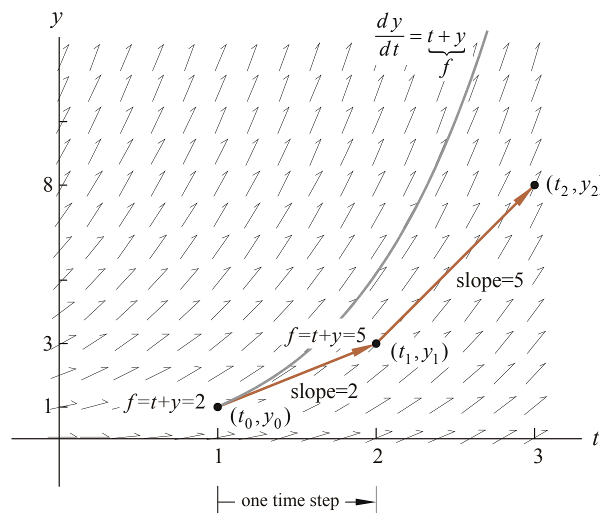


Figure 2.1 Euler's method uses the slope at the initial point (t_0, y_0) to move right by one time step to (t_1, y_1). It then uses the slope at (t_1, y_1) to move right to (t_2, y_2), etc. The gray arrows indicate the slope field. The gray curve is the exact solution.

The Runge-Kutta Method

Let us write our differential equation in the form

$$\frac{dy}{dt} = f(t, y), \tag{2.4}$$

where $f(t, y)$ on the right-hand-side is some function of t and/or y. We can interpret (2.4) as saying that the slope at every point (t, y) on the curve is given by the value of the function f at that point. Euler's method takes the value of f at the beginning of a timestep as the slope with which to move across the timestep. The Runge-Kutta method improves on this by evaluating f at four different places in the timestep and taking a weighted average of them as the slope to use to move across the timestep.

We will illustrate the Runge-Kutta method with the same differential equation as before, namely $dy/dt = t + y$. Refer to Fig. 2.2, which shows one timestep starting at $t=1$, ending at $t=3$ and with stepsize $h=2$. The first evaluation of the function f is at the beginning of the timestep. Call this value f_a. It equals 2. Use this slope of 2 to move halfway across the timestep. Evaluate f here. Call this value f_b. It equals 5. Go back to the beginning of the timestep and use this slope of 5 to again move halfway across the timestep. Evaluate f here. Call this value f_c. It equals 8. Go back to the beginning of the timestep. Use this slope of 8 but this time move all the way across the timestep. Evaluate f here. Call this value f_d. It equals 20. Now take this weighted average:

$$\frac{1}{6}\left(1 f_a + 2 f_b + 2 f_c + 1 f_d\right) \tag{2.5}$$

This is the slope that we finally use to move across the timestep. It equals 8. Thus if the timestep begins at $(t_1 = 1,\ y_1 = 1)$ then it ends at $(t_2 = 3,\ y_2 = 17)$. For comparison the analytical solution gives $y(3) = 18.167$.

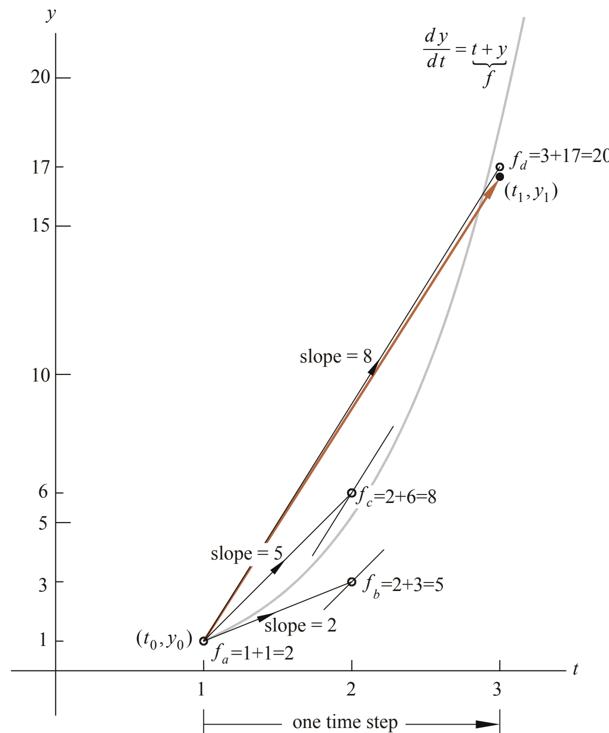


Figure 2.2 The Runge-Kutta method finds four points in sequence: at the beginning, middle and end of the time step, as indicated by the open circles. It evaluates the function *f* on the RHS of Eq. (2.4) at these points. These are the slopes f_a, f_b, f_c, f_d at these locations. It then uses a weighted average of them to move across the timestep from (t_0, y_0) to (t_1, y_1).

On the following page is a function *RungeKutta*, written in Visual Basic, that implements one timestep of the Runge-Kutta method and a subroutine, *Wrapper*, that calls *RungeKutta* repeatedly to step forward through multiple timesteps and print the output to an Excel spreadsheet. There is also a function, *RHS*, which is the right-hand-side of the differential equation to be solved. Do the following:

- Start Excel. Look for the Developer tab. (If it's not there then click on File>Options>Customize ribbon and check the Developer checkbox on the right side of the dialog box.)
- On the Developer tab click the Visual Basic button to open Excel's Visual Basic Editor.
- In the Editor click on Insert>Module to open a module into which the code can be typed.
- Type (or copy) *RHS*, *RungeKutta* and *Wrapper* into the module. Because *Wrapper* is declared "public" it appears in the list of macros that can be run in Excel. Run it.

It prints the following output to a spreadsheet:

index n	time t	y_{approx}	error
0	1.0	1	0
1	1.5	2.445	0.0009
2	2.0	5.152	0.0028
3	2.5	9.938	0.0069
4	3.0	18.152	0.0153

The program is presently set to solve the differential equation $dy/dt = t + y$ subject to the initial condition $y=1$ when $t=1$. Columns 2 and 3 are a table of values for the solution. The right-hand-side of the differential equation, presently $t+y$, is set in the line labelled [1]. The timestep size is set in line [2]. The initial condition is set in line [3]. The number of iterations (timesteps) is set in line [4]. For comparison purposes the program also prints out the error which is the difference between the Runge-Kutta solution and the known exact solution, $y = 3e^{t-1} - t - 1$.

Of course the Runge-Kutta method has truncation and accumulation error too. The following table shows how decreasing the stepsize h reduces the total error in computing forward from $t = 1$ to $t = 3$.

h	steps required	Runge-Kutta approx. to $y(3)$	Error
1	2	18.005	0.16
0.5	4	18.152	0.015
0.25	8	18.166	0.0012

The accuracy of the two methods can be compared using **big *O*** notation. It can be shown that the total error of Euler's method is $O(h)$ (spoken as "*of order h*"), meaning that $\lim_{h\to 0}(Error) \le M\,h$, where M is some constant. The total error of Runge and Kutta's method is $O(h^4)$ (spoken as "*of order h to the fourth*"), meaning that $\lim_{h\to 0}(Error) \le M\,h^4$. What this means is that, assuming h is small, cutting h in half in Euler's method will cut the error in half, but cutting h in half in the Runge-Kutta method will reduce the error by a factor of 16 (since $\left(\frac{1}{2}\right)^4 = \frac{1}{16}$). That is much better!

A Visual Basic Program for the Runge-Kutta Method

```
Function RHS(T, Y)                    'The RHS of the differential equation as defined in Eq. (2.4)
   RHS = T + Y                        '[1]  change this line to the right-hand-side of your DE
End Function

Function RungeKutta(T, Y, H)     'Input:  values of t and y at beginning of timestep, and the stepsize h
                                 'Output: value of y at end of timestep calculated by Runge-Kutta method
   Dim Fa, Fb, Fc, Fd, YTemp                               'Note:   This function calls function RHS

   Fa = RHS(T, Y)                     ' fa, fb, fc, fd are as described in the paragraph before Eq. (2.5)

   YTemp = Y + 0.5 * H * Fa                     'use the slope of point a to find height of point b
   Fb = RHS(T + 0.5 * H, YTemp)                                               'find slope at point b

   YTemp = Y + 0.5 * H * Fb                     'use the slope of point b to find height of point c
   Fc = RHS(T + 0.5 * H, YTemp)                                               'find slope at point c

   YTemp = Y + H * Fc                           'use the slope of point c to find height of point d
   Fd = RHS(T + H, YTemp)                                                     'find slope at point d

   RungeKutta = Y + H * (Fa + 2 * Fb + 2 * Fc + Fd) / 6     'use the weighted average defined in (2.5)
End Function                                       'and return the y value at the end of the timestep

Public Sub Wrapper( )          'User calls this macro.  It prints the solution of the IVP as a table of values.
   Const H = 0.5                              '[2]  The step size; smaller h results in better accuracy
   Dim T, Y, YOut, N

   T = 1: Y = 1                                                           '[3]  Set the initial conditions

                                       'print the heading of the table and first row (the initial conditions)
   Cells(1, 1) = "index n":   Cells(1, 2) = "time t":   Cells(1, 3) = "y approx":   Cells(1, 4) = "error"
   Cells(2, 1) = 0:           Cells(2, 2) = T:          Cells(2, 3) = Y:            Cells(2, 4) = 0

   For N = 1 To 4                          '[4]  Each time through the loop calculate and print one timestep.
      YOut = RungeKutta(T, Y, H)
      Y = YOut                                                                                  'update y
      T = T + H                                                                                 'update t
      Cells(N + 2, 1) = N: Cells(N + 2, 2) = T:   Cells(N + 2, 3) = Y         '[5] print the rest of the table
      Cells(N + 2, 4) = (3 * Exp(T - 1) - T - 1) - Y
   Next N
End Sub
```

Systems of Differential Equations and Higher Order Differential Equations

Systems of differential equations arise naturally in many models. An example is the Lotka-Volterra or predator-prey model. Imagine an isolated place inhabited only by rabbits (the prey) and foxes (the predators). First imagine that there are rabbits but no foxes. The growth rate of the rabbit population is proportional to the number of rabbits present. Represent this by the equation $dr/dt = \alpha r$ where r represents the number of rabbits. We know that the result is exponential growth, with rate α. Now imagine that there are foxes but no rabbits. The foxes, being carnivores, will die out at a rate proportional to the number remaining. Represent this by the equation $df/dt = -\beta f$ where f represents the number of foxes. We know that the result is exponential decay, with rate β. Now imagine both rabbits and foxes present. The resulting interaction can be most simply modeled with this system of differential equations:

$$\begin{cases} \dfrac{dr}{dt} = \alpha r - \gamma r \cdot f \\ \dfrac{df}{dt} = -\beta f + \delta r \cdot f \end{cases} \tag{2.6}$$

The interaction term, $r \cdot f$, is large when there are both many foxes eating and many rabbits being eaten. The interaction is negative (literally) for the unlucky rabbits and positive for the foxes.

Any n^{th} order differential equation can also be expressed as a system of n first order differential equations. For example the second order differential equation for the spring,

$$m\frac{d^2x}{dt^2} + R\frac{dx}{dt} + kx = 0, \tag{2.7}$$

can be replaced by a system of two first order differential equations

$$\begin{cases} \dfrac{dx}{dt} = v \\ \dfrac{dv}{dt} = -\dfrac{R}{m}v - \dfrac{k}{m}x \end{cases} \tag{2.8}$$

where the first equation defines a new *"auxiliary"* variable v. (It is just the velocity.) Thus we now have two variables, x and v, that are both dependent on t.

The usual way to choose the auxiliary variables is to simply let them be derivatives of the original variable. The standard form for a system of differential equations has: (1) a first order differential equation for each dependent variable, and (2) the derivative isolated on the LHS as we have in (2.6) and (2.8). An initial value problem also includes n initial conditions, one for each dependent variable. For the spring we have to specify the initial position x and initial velocity v.

Thus the standard form for an initial value problem with a system of two first order differential equations is as shown below in (2.9). This form generalizes to larger systems.

$$\begin{cases} \dfrac{dy_1}{dt} = f_1(t, y_1, y_2) \\ \dfrac{dy_2}{dt} = f_2(t, y_1, y_2) \end{cases} \quad \text{and} \quad \begin{cases} y_1(0) = c_1 \\ y_2(0) = c_2 \end{cases} \tag{2.9}$$

Example 2.1: Consider the third order differential equation $x''' + 2x'' + x' - 2x = 5 + t$ subject to the initial conditions $x(0) = 1,\ x'(0) = 2,\ x''(0) = 3$. Rewrite this as a system of three first order equations.

Solution: Define the sequence of derivatives: $x \xrightarrow[\text{differentiate}]{} \underbrace{y}_{=x'} \xrightarrow[\text{differentiate}]{} \underbrace{z}_{=x''}$.

In other words define $x' = y$ and $y' = z$. Next move all the terms of the original equation to the right *except the highest derivative*, giving $x''' = -2x'' - x' + 2x + 5 + t$, and notice that this can be rewritten as $z' = -2z - y + 2x + 5 + t$. Next rewrite the initial conditions as $x(0) = 1,\ y(0) = 2,\ z(0) = 3$. Thus the system consists of the definitions of y and z and the original equation rewritten in terms of them:

$$\begin{cases} x' = y, & x(0) = 1 \\ y' = z, & y(0) = 2 \\ z' = -2z - y + 2x + 5 + t, & z(0) = 3 \end{cases}$$

A Visual Basic Program for Solving a System of First Order Differential Equations with the Runge-Kutta Method

The program on the next page solves an initial value problem consisting of a system of n first order differential equations of the form

$$\begin{cases} dy_1/dt = f_1(t,\ y_1,\ y_2,\ \cdots,\ y_n) \\ dy_2/dt = f_2(t,\ y_1,\ y_2,\ \cdots,\ y_n) \\ \qquad \vdots \\ dy_n/dt = f_n(t,\ y_1,\ y_2,\ \cdots,\ y_n) \end{cases} \tag{2.10}$$

and n initial conditions, $y_1(0) = c_1,\ y_2(0) = c_2,\ \cdots,\ y_n(0) = c_n$. The program is very similar to the one on page 72 for a single first order differential equation so we will mainly just describe the differences. The number of dependent variables or differential equations, n, is stored in the global constant NumVar. This number is set in the line labelled [1]. The values of the dependent variables $y_1,\ y_2,\ \cdots,\ y_n$ are stored in an array called Y of length NumVar. Thus Y(1) holds y_1, Y(2) holds y_2, etc. The contents of array Y change with each timestep.

The right-hand-sides of the n differential equations, namely $f_1,\ f_2,\ \cdots,\ f_n$, are expressed in the lines labelled [2] in subroutine *EvaluateRHS*. Thus the right-hand-side of the first equation, f_1, is expressed as RHS(1), and so on.[1]

The stepsize h is set in line [3]. The initial conditions for t and $y_1, y_2, \ldots, y_n$ are specified in line [4]. The call to subroutine *RungeKutta* in the loop [6] carries out one timestep. Upon entry to *RungeKutta* the values of $y_1, y_2, \ldots, y_n$ are in array Y and upon exit they are in array YOut.

---

[1] RHS is not an actual array; it is a *reference* to an array. In the Runge-Kutta method each of the variables, $y_1, y_2, \ldots, y_n$, has its slope (rate of change) evaluated at four locations in the timestep. These values were called f_a, f_b, f_c, f_d, in the previous program. In this program these are arrays of length n, and RHS is a reference to one of them.

```
Const NumVar = 2                                  '[1] Set to the number of dependent variables or DEs.

Sub EvaluateRHS(T, Y, RHS)
   RHS(1) = Y(2)                     '[2]  Set these lines to the right-hand-sides of your system of DEs
   RHS(2) = -Sin(Y(1))                              'with the DE's as defined in Eq. (2.10)
End Sub

Sub RungeKutta(T, Y, H, YOut)          'Input: time t, array y at beginning of timestep, stepsize h
   Dim I As Integer                                 'Output: yout , the array y at end of timestep
   Dim Fa(1 To NumVar), Fb(1 To NumVar)                    'Define arrays Fa, Fb, Fc, Fd, YTemp
   Dim Fc(1 To NumVar), Fd(1 To NumVar), YTemp(1 To NumVar)

   Call EvaluateRHS(T, Y, Fa)
   For I = 1 To NumVar
      YTemp(I) = Y(I) + 0.5 * H * Fa(I)
   Next I

   Call EvaluateRHS(T + 0.5 * H, YTemp, Fb)
   For I = 1 To NumVar
      YTemp(I) = Y(I) + 0.5 * H * Fb(I)
   Next I

   Call EvaluateRHS(T + 0.5 * H, YTemp, Fc)
   For I = 1 To NumVar
      YTemp(I) = Y(I) + H * Fc(I)
   Next I

   Call EvaluateRHS(T + H, YTemp, Fd)

   For I = 1 To NumVar
      YOut(I) = Y(I) + H * (Fa(I) + 2 * Fb(I) + 2 * Fc(I) + Fd(I)) / 6
   Next I
End Sub

Sub WrapperSysEq( )                  'User calls this macro.  It prints the solution as a table of values.
   Const H = 0.5                                                        '[3]  Set the step size h.
   Dim T, Y(1 To NumVar), YOut(1 To NumVar), N, I

   T = 0: Y(1) = 3.14: Y(2) = 0                                    '[4]  Set the initial conditions.

   Cells(1, 1) = "time": Cells(1, 2) = "theta"                     '[5]  Print the heading of the table
   Cells(2, 1) = T: Cells(2, 2) = Y(1)                          'and the first row (the initial conditions)

   For N = 1 To 60                   '[6]  Each time through the loop calculate and print one timestep.
      Call RungeKutta(T, Y, H, YOut)
      For I = 1 To NumVar
         Y(I) = YOut(I)                                                              'update array y
      Next I
      T = T + H                                                                      'update time t
      Cells(N + 2, 1) = T: Cells(N + 2, 2) = Y(1)          '[7]  This line is presently set to print t and θ
   Next N

End Sub
```

The program is presently set up to model the pendulum, which is described by this system of first order DE's.

$$\begin{cases} d\theta/dt = y_2 \\ dy_2/dt = -\frac{g}{L}\sin(\theta) \end{cases} \tag{2.11}$$

(This system is gotten by starting with the second order differential equation for the pendulum given on page 14, namely

$$L\frac{d^2\theta}{dt^2} = -g\sin(\theta), \tag{2.12}$$

and defining $d\theta/dt \equiv y_2$. Since θ is the angle of the pendulum, y_2 is the angular velocity of the pendulum.) In the program we let Y(1) represent θ, Y(2) represent y_2 and we let $g/L = 1$. This means that the system of DE's in (2.12) is expressed in subroutine *EvaluateRHS* as

$$\begin{cases} \text{RHS}(1) = \text{Y}(2) \\ \text{RHS}(2) = -\sin(\text{Y}(1)) \end{cases}$$

For initial conditions we used $\theta(0) = 3.14$ and $y_2(0) = 0$. This means that the pendulum starts almost upside-down, from rest. We set line [6] to print out a table of values for t and θ and used it to plot a scatter graph. The result is the rather square-looking, large-amplitude oscillation in Fig. 2.3. We also used two other initial conditions $\theta(0) = 1$, $y_2(0) = 0$ and $\theta(0) = 0.5$, $y_2(0) = 0$ to produce the two smaller-amplitude, more sinusoidal shaped oscillations in the graph.

Eq. (2.12) is an example of a non-linear equation that can only be solved numerically. However if the oscillations are kept small then we can approximate $\sin(\theta) \approx \theta$ and then (2.12) reduces to

$$L\frac{d^2\theta}{dt^2} = -g\,\theta, \tag{2.13}$$

which is a linear equation that we learned to solve analytically in Chapter 1. Its solution is

$$\theta = A\sin\left(\sqrt{\frac{g}{L}}\,t + \varphi\right),$$

where A and φ are arbitrary constants.

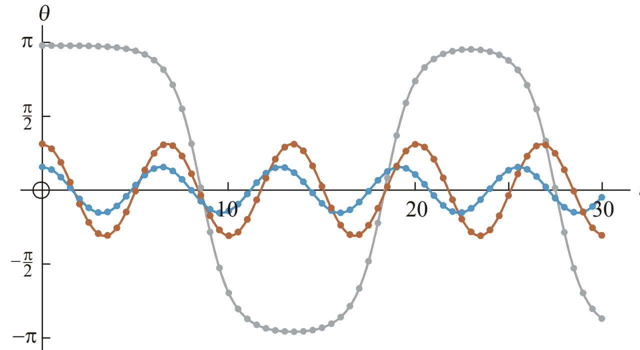


Figure 2.3 Solutions of the pendulum model. θ is the angle of the pendulum. The small amplitude oscillations are almost sinusoidal with period 2π. The large amplitude oscillation is square shaped because when the pendulum is upside-down, at $\theta = \pi$ and $-\pi$, it "hangs there" for a long time before moving.

Example 2.2: Solve the predator/prey equations. Assume that α, β, γ and δ are all equal to 1.

Solution: The equations are:

$$\begin{cases} dr/dt = r - r\cdot f \\ df/dt = -f + r\cdot f \end{cases}, \tag{2.14}$$

where r is the rabbit population and f is the fox population (in some unspecified units). Because of the presence of the product $r\cdot f$ they are non-linear and can only be solved numerically, using the Runge-Kutta program listed on page 75. We must reset lines [2], [3], [4], [5] and [7] as follows:

[2] The right-hand-sides of the differential equations: We will let Y(1) represent r and Y(2) represent f. Thus we must change lines [2] to read:

```
RHS(1) = Y(1) - Y(1) * Y(2)
RHS(2) = -Y(2) + Y(1) * Y(2)
```

[3] The stepsize H: This requires some experimentation. If we run the program with H = 0.5 and H = 0.25 we see almost no change in the results so we will leave this at H = 0.5.

[4] The initial conditions: We will use $r(0) = 1$ and $f(0) = 0.5$ so we will change line [4] to read:

```
T = 0: Y(1) = 1: Y(2) = 0.5
```

[5] and [7] Printing the results: We will print a table of values with three columns: time, rabbit population and fox population, so we must change the appropriate lines to:

```
Cells(1, 1) = "time":  Cells(1, 2) = "rabbits":   Cells(1, 3) = "foxes"
Cells(2, 1) = T:       Cells(2, 2) = Y(1):        Cells(2, 3) = Y(2)
Cells(N + 2, 1) = T:   Cells(N + 2, 2) = Y(1):    Cells(N + 2, 3) = Y(2)
```

From the table of values we can make two types of graphs. The first kind is the usual graph of the evolution of the population in time, as shown in Fig. 2.4 (a). The other kind is a plot of the fox population versus the rabbit population as shown in Fig. 2.4 (b). Time doesn't appear directly in this graph. The arrow on the curve shows the direction in which time increases and the dots show the time at various points along the curve. This is called a **state space plot** because a point on this plot completely describes the state of the system at that time. (Also if we are given the state at any one time (i.e. a value of r and a value of f) then we can calculate the state at all times in the future.) The fact that the path retraces over top of itself shows that this system is periodic. Fig. 2.4 (b) also shows several other paths that the system would follow starting from other initial conditions (the gray curves). If this model is accurate then cycles in animal populations are normal. What we humans have to be careful of is not to drive a species to extinction when its numbers are low.

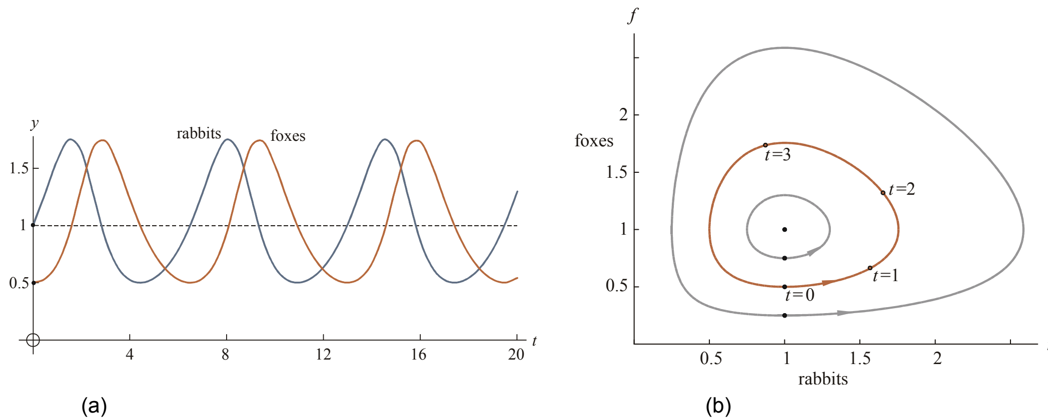


Figure 2.4 Two ways to plot solutions of the predator-prey model. (a) is called a time evolution plot. (b) is called a state space plot.

Problem Set 2.1

1. Modify the single-variable Runge-Kutta program on page 72 to solve Example 1.25. In this case the time constant is $RL = 0.5$ sec and the period of the alternating voltage produced by the generator is 2 sec. About 20 timesteps per cycle should be sufficient to accurately track the alternating voltage. Thus use h=0.1. Hint:

```
Function RHS(T, Y)
   RHS = -2 * Y
   If T >= 4 And T < 8 Then RHS = -2 * Y + 8
   If T >= 12 And T < 16 Then RHS = -2 * Y + 8 * Sin(3.14159 * T)
End Function
```

2. Modify the multivariable Runge-Kutta program on page 75 to solve this system of differential equations:

$$\begin{cases} dy_1/dt = y_1(3 - y_1 - y_2) \\ dy_2/dt = y_2(y_1 - 1) \end{cases}$$

Try several initial conditions and several values of the timestep h. Make a phase space plot. Your results should look like Fig. 2.5.

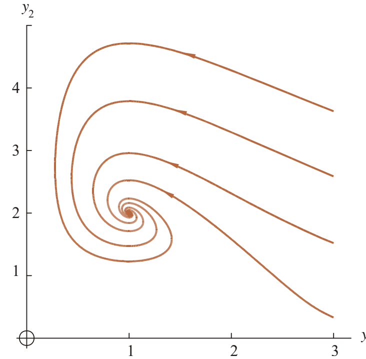


Figure 2.5

3. Modify the multivariable Runge-Kutta program to solve the electric circuit problem in Example 1.24 on page 50. If you look at the solution graphs on page 52 you will notice that the current initially changes very rapidly. This is because one of the transients has a time constant of 0.2 sec. To accurately track it use a timestep at least 4 times smaller, say h=0.05 sec. Make graphs of q vs. t and i vs. t. Your graphs should look like Fig. 1.25.

4. Use the multivariable Runge-Kutta program to solve the spring equations, (2.8). Experiment with different values of m, k, R and the timestep h. Make a plot of x vs. t as well as a phase space plot, v vs. x.

5. Solve the van der Pol equation, $\dfrac{d^2y}{dt^2} + \alpha\left(y^2 - 1\right)\dfrac{dy}{dt} + y = 0$, subject to the initial condition that y=0 and $dy/dt = 1$ at time t=0 using the Runge-Kutta method. Assume that $\alpha = 8$. Your solution should look like Fig. 2.6.

This equation describes an oscillatory system having variable damping, as can be seen by comparing it to the equation for a normal spring, $m\,d^2y/dt^2 + R\,dy/dt + k\,y = 0$. If the displacement y is smaller than 1 then the damping term is negative and if the displacement y is larger than 1 then the damping is positive. This equation was used by van der Pol to describe the operation of certain types of electronic oscillators.

Figure 2.6