Numerical Solution to Initial-Value Ordinary Differential Equations

1 Ordinary Differential Equations

These notes will give an introduction to solving initial-value ordinary differential equations (ODEs) on the computer using finite-difference methods. This type of ODE occurs often in nature, and you are familiar with them from equations of motion in mechanics. The ODEs we will study have the following generic form

\[ \dot{x} = f(x, t), \]  

where \( \dot{x} \equiv \frac{dx}{dt} \). One example is the classical equation of motion for a particle

\[ \dot{x} = v, \]  
\[ \dot{v} = a(x, t). \]

Many systems of higher order ODEs can be cast in the form of Eq. (1). For example, Eq. (2)-(3) can be rewritten as

\[ \dot{z} = f(z, t), \]

where \( z = (x, v) \) and \( f(z) = [v, a(x, t)] \). Another example of a system with the form of Eq. (1) is the equation for the nonlinear driven-damped pendulum

\[ \omega = \frac{g}{l} \sin(\theta) - \eta \omega + \epsilon_D \sin(\omega_D t), \]  
\[ \dot{\theta} = \omega, \]

discussed in lecture on Monday 9/22/03 (see Eq. 11.28 in Goldstein 3rd Edition). Yet, other examples are rate equations. Specifically, for radioactive decay

\[ \dot{N} = -\frac{1}{\tau} N, \]

where \( N(t) \) is the number of nuclei of a given isotope at a given point in time and \( \tau \) is the “decay time” of the given isotope. The list of examples is endless, systems of ODEs can be used to approximate weather patterns, highway traffic, financial instruments, robotics, etc.

Shortly, we are going to look at finite-difference methods for solving Eq. (1), however first lets look at three simpler forms of Eq. (1) which are often used for theoretical analysis of numerical error and stability (in these notes and most everywhere else). Often times, the time dependence and the higher dimensionality are dropped to make make the analysis clear and less cluttered with details

\[ \dot{x} = f(x). \]

Do not be too bothered by this since usually the time dependence and higher dimensionality can be kept without too much more difficulty. A much simpler form which is often studied for numerical stability is

\[ \dot{x} = -\lambda x. \]
This is done because the equation is linear, solvable, bounded, and can be analyzed in detail.

With nonlinear equations all bets are off with regard to nice clean solutions and error analysis. By nonlinear we mean the ODE has higher powers of $x$ in it, e.g.

$$\dot{x} = a + bx + cx^2 + dx^3. \quad (10)$$

The first two terms with coefficients $a$ and $b$ are linear terms. The terms with coefficients $c$ and $d$ are nonlinear terms. We call the term with coefficient $c$ a second order nonlinearity and the term with coefficient $d$ a third order nonlinearity, etc. This comes from perturbation theory where $x$ is small, and there is some hope for finding solutions.

We are about to spend some time studying the linear ODE of the form of Eq. (9) which may seem a gross simplification. However, the form of Eq. (9) is fairly general when linearizing a nonlinear system of ODEs. We can expand $f(x)$ in Eq. (8) about an equilibrium which is defined as a point $x_0$ where $f(x_0) = 0$.

$$f(x) = f(x_0) + (x - x_0)f'(x_0) + \frac{1}{2}(x - x_0)^2f''(x_0) + \cdots \quad (11)$$

$$\dot{u} \approx -\lambda u, \text{ where } u = x - x_0 \text{ and } -\lambda = f'(x_0), \quad (12)$$

which shows that for small deviations about the equilibrium point, an equation of the form of Eq. (9) is appropriate.

The “O” symbol means that the error is bounded by some constant times what is in the parentheses. Suppose, we see the equation

$$f(x) = a + b\epsilon x + O(\epsilon^2), \quad (14)$$

then the error is bounded by $C_0\epsilon^2$, where $C_0$ is some constant. This means if we make $\epsilon$ small enough, the error term will be negligible. However, for practical purposes, we need to know how big the error is. That is, we need to know how big we can make $\epsilon$ and still use the approximate equation $f(x) = a + b\epsilon x$. To do this we will have to keep the leading order term in $O(\epsilon^2)$. For example, if $O(\epsilon^2) = c\epsilon^2x^2 + O(\epsilon^3)$ then

$$f(x) = a + b\epsilon x + c\epsilon^2x^2 + O(\epsilon^3). \quad (15)$$

By comparing the second and third terms on the right, we can tell that we need $\epsilon \ll \frac{b}{a}$. If the use of the “O” symbol is confusing to you, it will be much clearer when we use it in the context of the error in finite-difference methods below.

A third equation that is often used to test and analyze numerics is the simple harmonic oscillator

$$\ddot{x} = -\omega_0^2 x. \quad (16)$$

This equation has an exact conserved energy, which should not damp away or grow due to numerical error. Many physical problems involve oscillatory motion (waves), so this is often a better test than Eq. (9) which has pure exponential damping. As a physicist, the first thing you should ask is “Can the numerical method at least accurately solve the harmonic oscillator problem?”

### 2 Euler Method, Accuracy, Stability

We now begin our discussion of finite-difference methods. The classic (graduate) text on this subject is the book *Numerical Initial-Value Problems in Ordinary Differential Equations*, by C. W. Gear
which is available in the Math/Physics Library. *Gear* is thorough, but hard to use as a reference. I do not know of a good, to the point reference on this topic. I feel *Numerical Recipes* coverage of the accuracy and stability is too sparse to help you access the range of methods you may encounter in computational physics. The mathematical theory for finite-difference methods for initial value ODEs is pretty well worked out and covered well by *Gear*. No such claims can be made for finite-difference methods for partial differential equations (PDEs). Here, we will learn the important basics for ODEs (i.e. what we need to know to solve nonlinear dynamical systems efficiently).

Three important mathematical concepts are accuracy, stability and convergence, important for numerical solution of both ODEs and PDEs. Gear puts much effort into defining these concepts and proving the following:

**Accuracy + Stability ⇒ Convergence**

*Convergence* means that if we make the timestep smaller, the numerical solution will converge to the exact solution. *Accuracy* means that we have some bound on the error from one timestep$^1$ to the next, and that it gets smaller if we make the timestep smaller.$^2$ *Stability* means that the numerical solution will not blow up for long times, given the exact solution does not blow up. More precisely, that it is bounded for large times ($t \to \infty$).

The simplest numerical solution to $\dot{x} = f(x)$ is Euler’s method (pronounced “oilier”), which simply comes from a Taylor series expansion about $x(t^n)$

$$x(t^{n+1}) = x(t^n) + \Delta t \dot{x}[x(t^n)] + \frac{\Delta t^2}{2} \ddot{x}[x(t^n)] + O(\Delta t^3),$$

(17)

where $t^n = n\Delta t$. We write the numerical scheme as

$$x^{n+1} = x^n + \Delta t f(x^n).$$

(18)

There is an important point about notation here. $x^n$ is the numerical approximation to the exact $x(t^n)$. Sometimes, $f(x^n)$ or $f(x^n, t^n)$ will be written simply as $f^n.3$

The local truncation error is the exact solution minus the approximate solution

$$e^{n+1} = x^{n+1} - x(t^{n+1})$$

(19)

with all the earlier values of $x^n$ set to their exact values $x(t^n)$. From Eq. (17), we see that $e^{n+1} = -\frac{1}{2} \Delta t^2 \ddot{x}[x(t^n)] + O(\Delta t^3)$ for the Euler method. We say that “$e^n$ is order $\Delta t^2$,” or “second order in $\Delta t$.” However, this local truncation error adds up from timestep to timestep (or accumulates), so the total error (called the global truncation error) is first order, or $O(\Delta t)$. We can see this by assuming $\frac{1}{2} \ddot{x}$ less than some constant $C_0$ for all times, then the global error $E^n$ is less then $NC_0 \Delta t^2$, where $N$ is the total number of timesteps we take. The total time is then $T = N\Delta t$. If we take the total time fixed, but increase $N$ (or decrease $\Delta t$) we see that the error is bounded by $C_0 T \Delta t$, or $O(\Delta t)$.

---

$^1$We are using the term time step to mean both the time interval between discrete steps and the discrete steps themselves, don’t get confused!

$^2$To be specific, accuracy, as used here, is the local truncation error and it must be at least $O(\Delta t^2)$, as will be defined below (do not get hung up on this statement).

$^3$If you are wondering why we have used superscripts for time levels, e.g. $x^n$, whereas some other references do not do so, the reason is that with partial differential equations, we will have spatial grid points and the subscript will be used to specify the grid index. Also, for large dynamical systems (e.g. N-body problems, particle simulations), the subscript is used as a “particle” index.
Let’s define the \textit{global truncation error} precisely as

\[ E^{n+1} = x^{n+1} - x(t^{n+1}), \]

where, now, \( x^{n+1} \) is numerical solution the solution \textit{integrated over all n timesteps}. We can almost always get a handle on the local truncation error of a given finite-difference scheme, so it is a very important measure of accuracy. Even though the global truncation error is what we are really interested in, it is usually much harder or impossible to estimate, except for simple linear problems we have the exact answer for. Let’s now study the global truncation error of the Euler method using our very simple linear equation with \( f(x) = -\lambda x \), with \( x(t = 0) = x^0 \). The exact solution is

\[ x(t^n) = x^0 \exp(-\lambda n \Delta t). \]

The finite-difference approximation is

\[ x^{n+1} = x^n + \Delta t f(x^n) = x^n (1 - \lambda \Delta t). \]

So, the numerical solution is exactly

\[ x^n = x^0 (1 - \lambda \Delta t)^n. \]

The global truncation error is

\[
E^n = x^n - x(t^n) = x^0 \left[ (1 - \lambda \Delta t)^n - \exp(\lambda n \Delta t) \right] \\
= x^0 \left\{ (1 - \lambda n \Delta t + \frac{n(n+1)}{2} (\lambda \Delta t)^2 + \cdots - (1 - \lambda n \Delta t + \frac{n^2}{2} (\lambda \Delta t)^2 + \cdots ) \right\} \\
= x^0 \frac{n}{2} (\lambda \Delta t)^2 + O(n \lambda^3 \Delta t^3) \\
= x^0 \frac{\lambda^2 t^n \Delta t}{2} + O(\lambda^3 t^n \Delta t^2).
\]

Keeping the rate constant and the total time fixed, we see that the the global truncation error is \( O(\Delta t) \).

To analyze stability, we first note that, the numerical solution is amplified or damped by \( (1 - \lambda \Delta t) \) each timestep. Therefore, to ensure the numerical solution does not blow up, we require

\[ |1 - \lambda \Delta t| \leq 1, \]

or \( \lambda \Delta t \leq 2 \) for \( \lambda > 0 \).

To generalize to \( \lambda < 0 \), or exponentially growing physical solutions, stability has to be defined as an amplification of an introduced error to have meaning, see Gear. The difficulty is that any introduced error increases from timestep to timestep regardless of \( \Delta t \). The important condition is that the the error does not grow faster than the physical solution. Gear defines absolute stability as a condition when an introduced error does not increase, which in this case is when \(|1 - \lambda \Delta t| \leq 1\) for all (complex) values of \( \lambda \).

\section{3 Runge-Kutta Methods}

The Euler method is seldom used except as a teaching tool, because it is only first order accurate \([O(\Delta t^2)]\) local truncation error. We can improve on the Euler method by evaluating \( f \) at an
intermediate location. The \textit{midpoint} method is
\begin{equation}
  \begin{aligned}
  \dot{x} &= x^n + \frac{\Delta t}{2} f(x^n, t^n), \\
  x^{n+1} &= x^n + \Delta t f(\dot{x}, t^n + \frac{\Delta t}{2}).
  \end{aligned}
\end{equation}

Often, this is called the second-order Runge-Kutta method. This should seem better then Euler’s method to you. The reason is that it is more \textit{time-centered}. Being \textit{time-centered} is a good general rule to follow, and not doing so will usually reduce accuracy. More importantly, being \textit{de-centered} introduces damping to problems involving oscillatory motion. This is another question to always ask, “Is the method time-centered?” In general, for both ODEs and PDEs, the only time one sacrifices being time-centered, is if 1) there is no other way to solve some part of the calculation, 2) it is the only way to get a stable solution (shock problems), 3) there is some advantage to enhanced damping, i.e. damp out spurious high frequencies introduced by other parts of the calculation. To calculate the local truncation error, we substitute the exact values $x(t^n)$ for $x^n$ in Eq. (26).
\begin{equation}
  e^{n+1} = x(t^{n+1}) - x^{n+1} = x(t^{n+1}) - x(t^n) \\
  - \Delta t f \left( x(t^n) + \frac{\Delta t}{2} f(x(t^n), t^n), t^n + \frac{\Delta t}{2} \right).
\end{equation}

Using a Taylor’s series expansion (this step left to you) we find the local truncation error to be third order $O(\Delta t^3)$. The stability of the midpoint method can be examined using $\dot{x} = -\lambda x$
\begin{equation}
  \begin{aligned}
  x^{n+1} &= x^n - \Delta t \lambda (x^n - \frac{\Delta t}{2} \lambda x^n) \\
  x^{n+1} &= (1 - \Delta t \lambda + \frac{\Delta t^2 \lambda^2}{2}) x^n
  \end{aligned}
\end{equation}

So, for stability, we require
\begin{equation}
  |1 - \Delta t \lambda + \frac{\Delta t^2 \lambda^2}{2}| \leq 1.
\end{equation}

More intermediate steps can be taken leading to higher-order accuracy. Eventually, the speed of a simpler scheme taking smaller timesteps wins over evaluating more and more intermediate steps. The “workhorse in the industry” is the \textit{fourth-order Runge-Kutta method}
\begin{equation}
  \begin{aligned}
  k_1 &= \Delta t f(x^n, t^n), \\
  k_2 &= \Delta t f(x^n + \frac{k_1}{2}, t^n + \frac{\Delta t}{2}), \\
  k_3 &= \Delta t f(x^n + \frac{k_2}{2}, t^n + \frac{\Delta t}{2}), \\
  k_4 &= \Delta t f(x^n + k_3, t^n + \Delta t), \\
  x^{n+1} &= x^n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6}.
  \end{aligned}
\end{equation}

This scheme has a $O(\Delta t^5)$ local truncation error.

4 Leapfrog Method and Multi-Step Methods

Often situations arise where $f(x, t)$ is computationally time consuming to evaluate. For, example it may be defined on a spatial grid and evaluation at a given position $x$ requires an interpolation or spline fit. In such cases, one wants to minimize evaluating $f(x, t)$, yet still maintain high accuracy.
For equations of motion that are canonical (this means they conserve energy and can be derived from a Hamiltonian), the standard leapfrog method is very often used for solving $\dot{x} = a(x, t)$, or

$$\dot{v} = a(x, t), \quad \dot{x} = v.$$  \hfill (31)

The leapfrog method can be written in the following time-centered looking form

$$v^{n+1/2} = v^{n-1/2} + \Delta t a(x^n, t^n),$$  \hfill (32)

$$x^{n+1} = x^n + \Delta t v^{n+1/2}.$$  We can re-write this scheme as

$$\frac{x^{n+1} - 2x^n + x^{n-1}}{\Delta t^2} = a(x^n, t^n).$$  \hfill (33)

Sometimes this is called this the Verlet method, in reference to molecular dynamics simulations. We will analyze the error and stability shortly, but first, there is a problem with starting off this method for initial value problems because you only know $x^0 = x(t = 0)$ at the start, but you also need to know either $x^{-1}$ or $x^1$ to advance forward in time. This does not cause any practical problem, since you can simply use another scheme such as Runge-Kutta to predict $x^1$ and then advance from there. This is an example of a multi-step method. That is, one that uses multiple values of $x$ (at old time intervals), to advance forward in time. The Euler and Runge-Kutta methods are single-step methods. Once you advance to the next timestep, all past information except $x^n$ is lost and not used again.

To analyze the leapfrog method we can calculate the local truncation error $e^{n+1} = x^{n+1} - x(t^{n+1})$ where $x(t^{n+1})$ is evaluated using Eq. (33), with $x^n$ set to $x(t^n)$ and $x^{n-1}$ set to $x(t^{n-1})$

$$e^{n+1} = 2x(t^{n}) - x(t^{n-1}) + \Delta t^2 a(x(t^n), t^{n+1}),$$  \hfill (34)

$$= O(\Delta t^2)$$

For canonical equations of this form it is often more natural to analyze both accuracy and stability using the following approach. We use the harmonic oscillator test problem

$$\ddot{x} = -\omega_0^2 x.$$  \hfill (35)

This has the following solution for the initial condition $x^0 = x(t = 0)$

$$x(t) = x^0 \exp(-i\omega_0 t)$$  \hfill (36)

We assume a numerical solution for $x^n$ that has the following form

$$x^n = x^0 \exp(-i\omega n \Delta t) = x^0 z^n,$$  \hfill (37)

where $z = \exp(-i\omega \Delta t)$. $z$ is called the amplification factor because if $|z| > 1$ the scheme will be unstable. We then substitute Eqs. (35) and (37) into the numerical scheme and we get an algebraic equation for $z$ in terms of $\Delta t$ and $\omega_0$. The basic idea is the following: We have some physical oscillations at the natural frequency $\omega_0$, what numerical frequency $\omega$ will the numerical simulation produce? Let’s do this for the leapfrog method using Eq. (33), we get

$$z^{n+1} - 2z^n + z^{n-1} = -\omega_0^2 \Delta t^2 \Delta t z^n$$  \hfill (38)
Further simplification, and using the definition of the amplification factor $z$, we obtain the relationship for $z(\omega_0 \Delta t)$ and $\omega(\omega_0 \Delta t)$

$$(\omega_0 \Delta t)^2 = \frac{(z-1)^2}{z} = 4 \sin^2(\omega \Delta t / 2).$$

(39)

It is not totally obvious from Eq. (39), however, the roots of $z$ lie on the unit circle for $\omega_0 \Delta t \leq 2$. For $\omega_0 \Delta t > 2$, the leap frog scheme is unstable and $|z| > 1$. The easiest thing to do to obtain a solution to Eq. (39), is to use Mathematica and solve for the roots, which was done and the results are shown below. The quite amazing thing about the leapfrog method is that $|z| = 1$ for $\omega_0 \Delta t \leq 2$.

![Graph of the amplification factor $|z|$ versus $(\omega_0 \Delta t)^2$ for the leapfrog scheme solving Eq. (39).](image)

Figure 1: Plot of the amplification factor $|z|$ versus $(\omega_0 \Delta t)^2$ for the leapfrog scheme solving Eq. (39).

This means that its ability to conserve energy in the simple harmonic oscillator problem is quite good when compared to other schemes. Figure 1 is an example of a bifurcation diagram. As one varies the parameter $\omega \Delta t$, the solution $x = x^0 z^n$ transitions from a purely oscillating solution to a growing and damped solution. Figure 1 shows this transition. It is important to realize that these growing and damped solutions are purely numerical.

We can get a measure of the error using this type of analysis as well. From Eq. (39) we see that
Figure 2: Plot of the roots of Eq. (39), $\omega \Delta t$ versus $\omega_0 \Delta t$. Real and imaginary parts are shown, both are plotted as solid lines.

for $\omega_0 \Delta t \ll 2$,

$$\frac{\omega \Delta t}{2} = \frac{\omega_0 \Delta t}{2} \left\{ 1 + \frac{1}{6} \left( \frac{\omega_0 \Delta t}{2} \right)^2 + \cdots \right\}$$

(40)

So, the frequency is

$$\omega = \omega_0 \left( 1 + \frac{\omega_0^2 \Delta t^2}{24} \right) + O(\Delta t^4).$$

(41)

This type of simple analysis is seldom found in the numerical methods literature, but, is a quick way to get at what the qualitative behavior of a given scheme will be for problems involving oscillatory motion.

5 Implicit Schemes

Often times we are solving systems of equations that have multiple types of oscillations or rate constants at disparate timescales. However, we may only be interested in the slowest timescale behavior. This presents a problem since all the schemes we have have looked at so far have timestep constraints, i.e. $\lambda \Delta t < O(1)$ or $\omega \Delta t < O(1)$. The implication is that your timestep has to be small
enough to resolve the smallest physical timescale of the problem regardless if you are interested in it or not. Otherwise, your simulation blows up! All schemes we have looked at so far are explicit. That is, the new value depends only on previously calculated values. It is generally true that explicit methods always have such a timestep constraint.

Let’s take a look at another scheme, the trapezoidal method for solving \( \dot{x} = f(x) \)

\[
x^{n+1} = x^n + \frac{\Delta t}{2} (f(x^n) + f(x^{n+1})).
\]

This is an implicit scheme because \( f(x^{n+1}) \) must be evaluated before \( x^{n+1} \) is known. Typically, implicit methods are hard to solve, and they involve inverting the operator \( 1 + f \). However, in some applications implicit methods are superior, because they are unconditionally stable. We will show the unconditional stability using \( \dot{x} = -\lambda x \), with \( \lambda > 0 \)

\[
x^{n+1} = x^n + \frac{\Delta t}{2} (-\lambda x^n - \lambda x^{n+1}),
\]

\[
x^{n+1} = \frac{(1 - \frac{\lambda \Delta t}{2})}{(1 + \frac{\lambda \Delta t}{2})} x^n,
\]

which is stable for all \( \Delta t \). In reality, full-implicit methods are rarely used. However, semi-implicit methods are fairly common. Semi-implicit methods linearize the operator \( 1 + f \), making the inversion a matrix operation in higher dimensional systems of equations. The linearized solution for \( x^{n+1} \) is then used to the evaluate of \( f(x^{n+1}) \).

6 Predictor-Corrector Schemes

The modified trapezoidal method is an explicit scheme has the same local truncation error \( O(\Delta t^3) \) as the implicit trapezoidal method

\[
\tilde{x}^{n+1} = x^n + \Delta t f(x^n, t^n),
\]

\[
x^{n+1} = x^n + \frac{\Delta t}{2} (f(x^n) + f(x^{n+1})).
\]

However, because it is explicit, it has a timestep constraint from the stability analysis

\[
x^{n+1} = x^n + \frac{\Delta t}{2} (-\lambda x^n - \lambda x^{n+1}),
\]

\[
x^{n+1} = (1 - \lambda \Delta t + \lambda^2 \Delta t^2) x^n.
\]

This is the same stability constraint as for the midpoint method, see Eq. (29). The modified trapezoidal method is a form of a predictor-corrector method. The first step in the algorithm is a predicted value \( \tilde{x}^{n+1} \) and the next step is a corrected value \( x^{n+1} \). The predictor-corrector step can be iterated (i.e. done multiple times)

\[
\tilde{x}^{n+1}_0 = x^n + \Delta t f(x^n, t^n),
\]

\[
\tilde{x}^{n+1}_m = x^n + \frac{\Delta t}{2} (f(x^n) + f(\tilde{x}^{n+1}_m)).
\]

where \( m = 0, 2, 3, ..., M - 1 \), and \( x^{n+1} = \tilde{x}^{n+1}_M \). This is not a very high order predictor-corrector method, and there are better choices. We simply use it as an illustration of the idea behind predictor-corrector schemes. Often, and in the case here with Eq. (46), taking multiple corrector steps greatly improves time-centering and greatly reduces damping of oscillatory motion.
7 Higher Dimensional Problems

In the homework, and in life, you will need to solve higher dimensional problems. The higher dimensional extension of the various schemes is fairly straightforward. Let’s take the midpoint method as an example.

\[ \tilde{x} = x^n + \frac{\Delta t}{2}f(x^n, t^n), \]

\[ x^{n+1} = x^n + \Delta tf(\tilde{x}, t^n + \frac{\Delta t}{2}). \]

Specifically, let’s apply the midpoint method to an equation of the form of the nonlinear pendulum problem

\[ \dot{\theta} = \omega, \]

\[ \ddot{\omega} = f(\theta, t). \]

The numerical solution will be the following

\[ \dot{\theta} = \theta^n + \frac{\Delta t}{2}\omega^n \]

\[ \ddot{\omega} = \omega^n + \frac{\Delta t}{2}f(\theta^n, t^n) \]

\[ \theta^{n+1} = \theta^n + \Delta t\ddot{\omega} \]

\[ \omega^{n+1} = \omega^n + \Delta tf(\dot{\theta}, t + \frac{\Delta t}{2}) \]

8 Some Comments on Programming Re-usable Bug-free Procedural Code

Here are some key points to good procedural (Fortran or C programming, not object-oriented) programming. Following these tips will help make your code easy to understand, which in turn, makes it less bug-prone and more re-usable. If you can’t understand what you have done, there is little hope of finding a programming error. Or, you cannot easily pick up a snippet of code from your past, and re-use it in a new application or extend it. Finally, others will have nothing to do with your code if it is not readable.

Some good general rules to follow are:

- **Make your code very clear and easy to read.** Make it look like the mathematical equations you are solving. When you program for efficiency (e.g. on a massively parallel computer), you may find the programming gets quite convoluted. You may find it useful to keep around the original easy-to-read version in the comments, just to remind yourself what is really going on in the code. This will make bugs easier to find, make your code more portable, and re-usable.

- **Clear and simple comments.** Keeping comments up-to-date is critical. Outdated comments are worse than no comments at all.

- **Use procedures to organize the program structure and provide for future re-use.** Do not get carried away with many nested procedures that might confuse someone else who may use your program in the future.
• Put in a little extra time to make your output and graphics clear and readable. If you don’t do this when you first write your diagnostic/graphic routines, you probably never will go back and clean them up. Also, your results will be confusing and lack professionalism.

We are now ready to solve most any nonlinear ODE. Please take some time getting familiar with Mathcad.