P402 Note: Numerical Solutions of *Second-Order* Differential Equations — The Runge-Kutta Method

Introduction and Review of the Euler Method:

We've learned how to numerically solve a first-order differential equation by using the Euler method. When we speak of a *first-order* differential equation, we mean one where only the first, and no higher derivatives, enter into the equation. Many such equations can be written in the form:

$$y'(x) = F(y(x), x) \quad , \tag{1}$$

where F is any arbitrary combination of the unknown function and the independent variable. We saw that a unique solution of (1) could be obtained numerically provided that we were given the subsidiary initial condition: $y(x_0) = y_0$, i.e., the value of the unknown function, y_0, at some initial value of the independent variable, x_0. We obtained our numerical solution of (1) by applying the iterative Euler method algorithm:

$$\begin{aligned} y_{i+1} &= y_i + \Delta x F(y_i, x_i) \\ x_i &= x_0 + i\Delta x \end{aligned} \tag{2}$$

When F is algebraic (i.e., simple polynomials) the large x solution of (1) generally has exponential behavior (c.f., the population growth and coffee cooling problems). In such cases we could obtain fairly accurate numerical solutions of (1) with a modest Δx size.

Second-Order Differential Equations:

We know from experience that many systems demonstrate *oscillatory* behavior: a mass suspended by a spring and a simple pendulum are common mechanical systems which come to mind. Examples of more complicated systems are sound waves and electromagnetic radiation. How does this oscillatory behavior arise in nature? It's possible to get oscillatory solutions to (1) if F is sufficiently complicated. For example, if $F(y(x), x) = \sin(x)$ then the solution of (1) would oscillate. However, such highly non-linear equations don't generally arise in nature. Typically, systems that display oscillatory behavior do so when the differential equation which describes them involves *second* derivatives of the unknown function. The second derivative of a function is the derivative of the first derivative of the function. Geometrically, this just means the slope of the slope of the curve representing the original function. Most *second-order* differential equations can be written in the form:

$$y''(x) = F(y'(x), y(x), x) \quad , \tag{3}$$

where F is some arbitrary function of $y'(x)$, $y(x)$ and x. $y''(x)$ is the notation we use for the second derivative of the function $y(x)$.

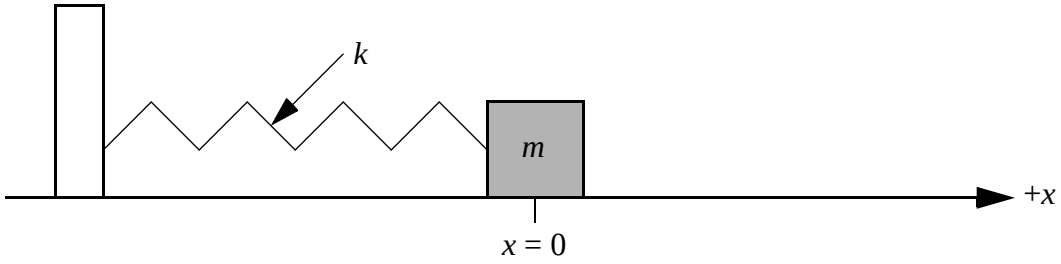
How do equations that have the form of (3) arise in nature? For mechanical systems, Newton's second law of motion is just such an equation. Newton's second law states that the acceleration of an object is proportional to the sum of all the forces acting on the object and inversely proportional to its mass:

$$a = \frac{F}{m}, \tag{4}$$

where a is the acceleration, F is the net force and m is the mass. For motion in two- and three-dimensions, a and F are really vector quantities, but let's not worry about that yet†. Equation (4) is a second-order differential equation because the acceleration is the rate of change of the velocity (the derivative of the velocity with respect to time, $a(t) = v'(t)$), and the velocity, in turn, is the rate of change of the position (the derivative of the position with respect to time, $v(t) = x'(t)$). Therefore the acceleration is the rate of change of the rate of change of the position (the *second* derivative of the position with respect to time, $a(t) = x''(t)$). Therefore, we can rewrite (4) as:

$$x''(t) = \frac{F(v, x, t)}{m} \quad . \tag{5}$$

For complete generality, I've explicitly indicated in (5) the possible dependence of the net force on velocity, position and time. As a simple example, consider a mass m connected to a spring that is free to slide on a frictionless horizontal surface:



The zigzag line is the spring that has a stiffness k. When the mass is at $x = 0$ the spring is completely relaxed (i.e., neither stretched nor compressed). When the mass is moved to the left or to the right it experiences a restoring force due to the spring that is directed towards the right or left, respectively. The magnitude of the force is proportional to the displacement of the mass, the proportionality constant being the stiffness of the spring, k. This is known as Hooke's Law:

†. For now we'll restrict our discussion to one-dimensional systems, i.e., motion along a straight line. Later in the semester we'll discuss the Kepler problem (motion of the planets) and electron-electron scattering, both two-dimensional systems. We'll see then that equation (4) becomes a system of two second-order differential equations which must be solved simultaneously.

$$F = -kx. \tag{6}$$

Substituting (6) into (5) gives the second-order differential equation describing the motion:

$$x''(t) = -(k/m)x(t) \quad . \tag{7}$$

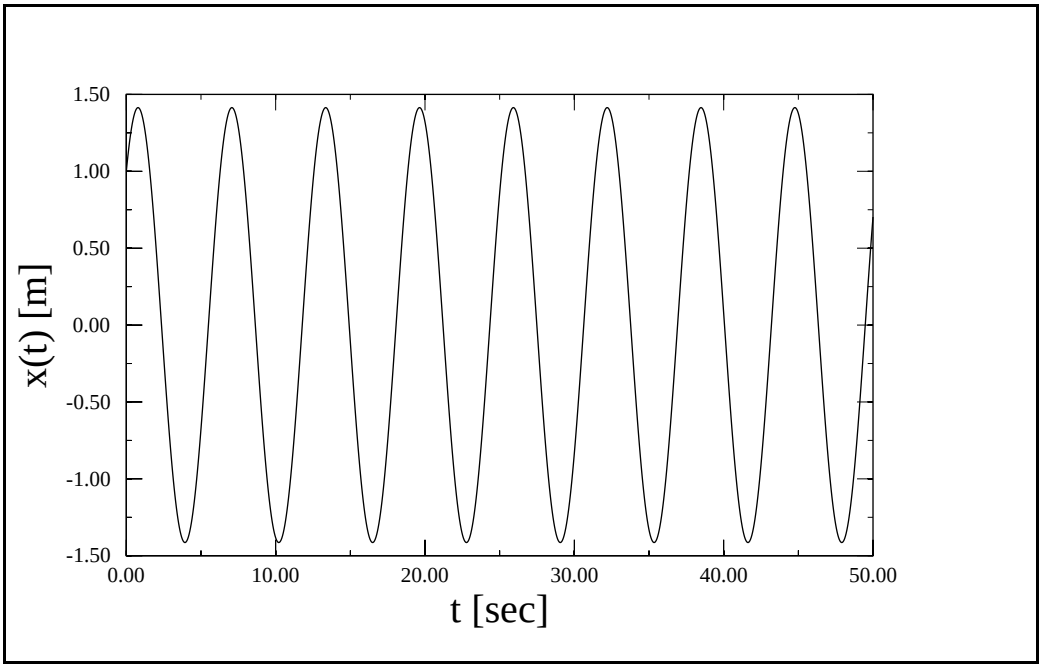
We know that a first-order differential equation has a *unique* solution only when a subsidiary initial condition is supplied. In the case of a second-order differential equation, we must provide two initial conditions to obtain a unique solution. For Newton's second law of motion, we must provide the initial position as well as the initial velocity of the object if we are to determine its position as a function of time: $x(0) = x_0$ and $x'(0) = v_0$. From (7) you can see that the term k/m has units of an inverse time squared. This inverse time is the natural frequency of oscillation of the system: $\omega = \sqrt{k/m}$. The motion of the mass is completely specified by the following:

$$\begin{aligned} x''(t) + \omega^2 x(t) &= 0 \\ x(0) &= x_0 \\ x'(0) &= v_0 \end{aligned} \tag{8}$$

The analytical solution of (8) is easy to obtain, but we'll state it without proof:

$$x(t) = x_0 \cos(\omega t) + \frac{v_0}{\omega} \sin(\omega t) \quad . \tag{9}$$

You can check this result by substituting it back into (8). The important thing to notice is that the solution is oscillatory with an oscillation frequency given by ω. A plot of the solution looks like:



for the case k = 1.0 *N/m*, m = 1.0 *kg*, x_0 = 1.0 *m* and v_0 = 1.0 *m/s*.

Euler Method Solution of Second-Order Differential Equations:

Could we have obtained this solution numerically, perhaps with the Euler method? The answer is yes but not very accurately, at least if we want to invest only a modest amount of computer time in obtaining the solution. Here's how:

In general, you can write any second-order differential equation as a pair of first-order equations. We can rewrite equation (5) as:

$$\begin{aligned} v'(t) &= \frac{F}{m} \\ x'(t) &= v(t) \end{aligned} \tag{10}$$

Now write the left-hand-sides of (10) using the secant approximation for first derivatives:

$$\begin{aligned} \frac{v(t+\Delta t)-v(t)}{\Delta t} &= \frac{F}{m} \\ \frac{x(t+\Delta t)-x(t)}{\Delta t} &= v(t) \end{aligned} \tag{11}$$

Rearranging (11) gives:

$$\begin{aligned} v(t+\Delta t) &= v(t) + \Delta t\frac{F}{m} \\ x(t+\Delta t) &= x(t) + \Delta t v(t) \end{aligned} \tag{12}$$

The pair of equations in (12) can be simultaneous iterated beginning from the initial conditions. This is Euler's method for a second-order differential equation. Using the subscript notation of equation (2), we can write (12) as:

$$\begin{aligned} v_{i+1} &= v_i + \Delta t\frac{F(x_i)}{m} \\ x_{i+1} &= x_i + \Delta t v_i \\ t_i &= t_0 + i\Delta t \end{aligned} \tag{13}$$

For simplicity, we typically take $t_0 = 0$. Starting from the initial conditions ($i = 0$), the left-hand-sides of (13) are completely specified. This allows us to compute v_1 and x_1. Knowing these latter values we can then compute v_2 and x_2, and so forth. This iterative procedure is repeated until the solution duration is reached.

Here's a program to solve the mass-spring problem using the Euler Method:

```
/*
 * spring.c: program that solves the spring-mass problem using Euler's method
 *           to solve a second order differential equation.
 */

#include <stdio.h>
#include <math.h>

#define M 1.0     /* mass */
#define K 1.0     /* spring constant */
#define OUTFILE "spring.out" /* output file name */

void
main()
{
      double x, v, x_new, v_new, tmax, t, dt, f(double, double, double);
      int n, i;
      FILE *fp;

      fp = fopen(OUTFILE, "w");

      printf("\nEnter X0, V0, Tmax and N: ");
      scanf("%le %le", &x, &v, &tmax, &n);

      dt = tmax/(n-1);

      for (t = 0.0, i = 0; i < n; i++) {

            fprintf(fp, "%le %le %le\n", t, x, v);

            v_new = v + dt * f(t, x, v);
            x_new = x + dt * v;

            v = v_new;
            x = x_new;

            t += dt;

      }

      fclose(fp);
      exit(0);
}

double
f(double t, double x, double v)
{
      return -K*x/M;
}
```

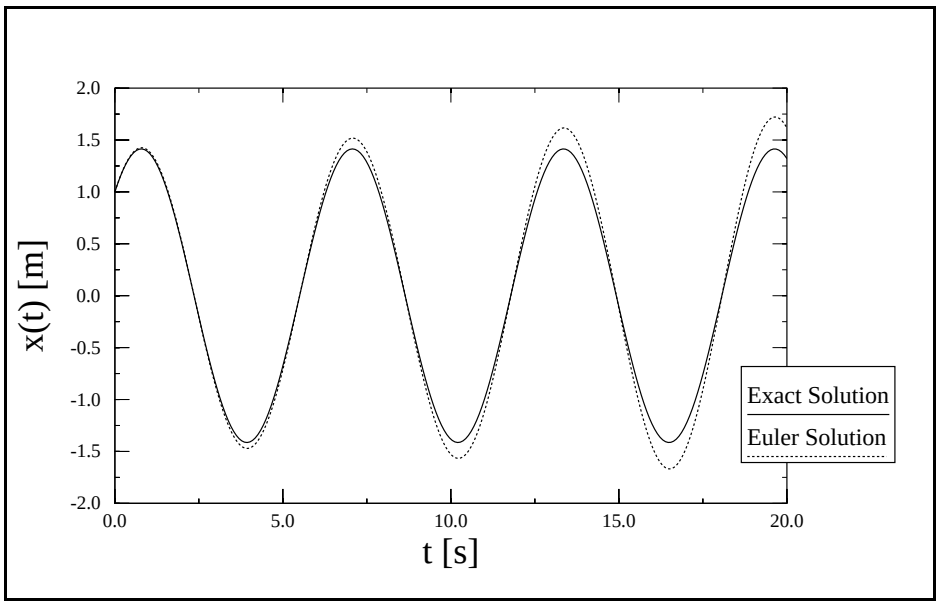
```
egs:ed:362% spring

Enter X0, V0, Tmax and N: 1.0 1.0 20.0 1000

egs:ed:363% cat spring.out

0.000000e+00 1.000000e+00 1.000000e+00
```

```
2.002002e-02 1.020020e+00 9.799800e-01
```

(*996 more lines of output*)

```
1.997998e+01 1.627015e+00 -5.798540e-01
2.000000e+01 1.615407e+00 -6.124269e-01
```

Results of the above program with $x_0 = 1$, $v_0 = 1$, $\omega = 1$, $t_{max} = 20$, and $\Delta t = 2\times10^{-2}$ are rather disappointing:



As can be seen in the above figure, the Euler method solution departs from the exact solution with a magnitude of error increasing with time. The problem lies in the secant approximation of the derivative. Recall that for a second-order differential equations we are effectively approximating the second derivative of the unknown function by applying the secant method to the curve representing the *first derivative* of the unknown function. But the first derivative *itself* was approximated using the secant method. In other words we're approximating an approximation! It's not at all surprising that the error accumulates quickly. Another problem is that the solution has turning points (places where the sign of the slope changes) where the second derivative is large. You can see from the figure that these are precisely the places where the Euler method is poorest. To obtain 4 significant figure accuracy in the above solution, we would need to decrease the time-step to $\Delta t = 2\times10^{-4}$, or 100,000 Euler steps!

Is there a more efficient way? Yes, it's called the *Runge-Kutta* method.

The Runge-Kutta Method of Solving First-Order Differential Equations:

The Runge-Kutta (pronounced rung-ah cut-ah) method for solving differential equations is fairly difficult to understand and rather involved algebraically to derive. For these reasons we will only give a qualitative description of its inner workings and state, without proof‡, the few simple equations that make up the method.

The gist of the method is to approximate the tangent to the curve not with a single secant (as is done in the Euler method) but with a weighted average of some small number of secants. The weighting factors and the position of the secants are chosen in such a way as *to minimize the error in the estimate of the slope of the true tangent*. The number of secants employed is called the *order* of the method. Thus if two secants are used it is a *second-order* Runge-Kutta approximation (the term "order", as used here, has nothing to do with the order of the differential equation being solved). There is, however, a law of diminishing returns: the computational effort increases linearly with the order of the method, but the increase in the solution accuracy does not. For most problems the fourth-order Runge-Kutta method gives the highest accuracy-effort ratio.

The fourth-order Runge-Kutta solution of the first-order differential equation of the form:

$$y'(x) = F(x, y) \tag{14}$$

is found by iterating the equation:

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) , \tag{15}$$

where the constants k_1, k_2, k_3 and k_4 are given by:

$$\begin{aligned} k_1 &= \Delta x F(x_n, y_n) \\ k_2 &= \Delta x F\left(x_n + \frac{\Delta x}{2}, y_n + \frac{k_1}{2}\right) \\ k_3 &= \Delta x F\left(x_n + \frac{\Delta x}{2}, y_n + \frac{k_2}{2}\right) \\ k_4 &= \Delta x F(x_n + \Delta x, y_n + k_3) \end{aligned} \tag{16}$$

On examination of (16), one sees that the constant k_1 depends on only x_n and y_n, while k_2 depends on only x_n and k_1, and so forth. Therefore, the right-hand-side of (15) ultimately depends only upon x_n and y_n. This was also the property of the Euler method equation (equation (2)) that allowed it to be iterated. Thus, the fourth-order Runge-Kutta solution of (14) is obtained by iterating (15), beginning at the known initial condition $y_0 = y(x_0)$.

‡. For a detailed description and analysis of the method, see Press et al.'s *Numerical Recipes in C*.

The Runge-Kutta Method of Solving Second-Order Differential Equations:

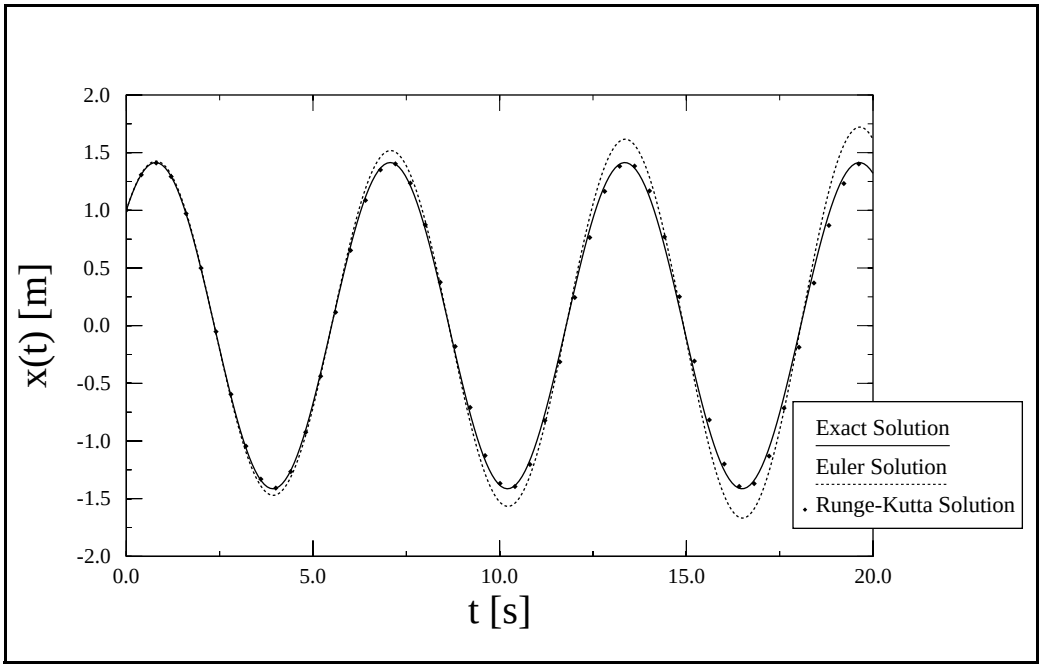
The Runge-Kutta method is not restricted to solving only first-order differential equations of the form given by (14). We can do the same splitting of (14) into two first-order equations as was done above in the discussion of solving second-order equations with the Euler method. Recall that we were able to re-write the single second-order equation given in (5) as the pair of first-order equations given in (10). We can simultaneously iterate the pair of equations in (10) using the fourth-order Runge-Kutta algorithm embodied in equation (15). Here's a program that does so:

```
/*
 * spring2.c: program to solve the spring-mass problem using 4th-order
 * Runge-Kutta method to solve a second order differential equation.
 */

#include <stdio.h>
#include <math.h>

#define M 1.0 /* mass */
#define K 1.0 /* spring constant */
#define OUTFILE "spring.out" /* output file name */

void
main()
{
      double x, v, tmax, t, dt, k1, k2, k3, k4, f(double, double);
      int n, i;
      FILE *fp;

      fp = fopen(OUTFILE, "w");

      printf("\nEnter X0 and V0: ");
      scanf("%le %le", &x, &v);

      printf("Enter Tmax and N: ");
      scanf("%le %d", &tmax, &n);

      dt = tmax/(n-1);

      for (t = 0.0, i = 0; i < n; i++) {

            fprintf(fp, "%le %le %le\n", t, x, v);

            k1 = dt*f(t, x);
            k2 = dt*f(t+dt/2, x+k1/2);
            k3 = dt*f(t+dt/2, x+k2/2);
            k4 = dt*f(t+dt, x+k3);

            v += (k1+2*k2+2*k3+k4)/6;

            k1 = k2 = k3 = k4 = dt*v;

            x += (k1+2*k2+2*k3+k4)/6;

            t += dt;

      }

      fclose(fp);
      exit(0);
}
```

```
double
f(double t, double x)
{
      return -K*x/M;
}
```

Take note of a few things about this program: (*i*) the variable *v* is updated immediately and the new value is used in updating *x*, (*ii*) the right-hand-side of the Runge-Kutta equation for updating *x* depends only on *v* which does not explicitly depend on any other variables, hence we have the statement `k1 = k2 = k3 = k4 = dt*v`. We could therefore have simplified the program slightly by writing `x += dt*v`, omitting the `k1 = k2 = k3 = k4 = dt*v` statement.

The Runge-Kutta solution for the same system parameters and value of Δt looks like:



You can see the far better agreement between the Runge-Kutta solution and the exact solution than that between the Euler method and the exact solutions.