10. INITIAL VALUE ORDINARY DIFFERENTIAL EQUATIONS

ivode_2013.docx

10.1 PRELIMINARIES

"The most important mathematical model for physical phenomena is the differential equation" ([p. 265] Rice 1983).

"We now turn to the study of differential equations, that is, equations involving derivative of the unknown solution. We have previously considered only algebraic equations for which the unknown solution is a discrete vector in finite-dimensional space. For the differential equation on the other hand, the unknown solution is a continuous function in infinite-dimensional space" ([p. 297] Heath 1997).

"Most physical systems change with time. One of the motivating problems for the invention of differential calculus was to characterize the motion of planets and earthly projectiles so that their future locations could be predicted. Differential equations provide a mathematical language for describing continuous change. Beginning with Newton's laws of motion, most fundamental laws of science can be expressed as differential equations" ([p. 382] Heath 2002).

When there is only one independent variable such as time, then all derivatives of the dependent variables are with respect to that independent variable, and we have an ordinary differential equation, or ODE" ([p. 383] Heath 2002).

"The highest-order derivative appearing in an ODE determines the order of the ODE. For example, Newton's second law is a second-order ODE" ([p. 383] Heath 2002).

"A k^{th} order ODE is said to be ***explicit*** *if it can be written in the form*

$$y^{(k)} = f(t, y, y', y'', \ldots, y^{(k-1)}) = 0\text{"}$$ ([p. 383] Heath 2002).

"An ODE $y' = f(t, y)$ does not by itself determine a unique solution function because only the slopes $y'(t)$ of the solution components are prescribed by the ODE for any value of t, not the solution value $y(t)$ itself, so there is usually an infinite family of functions that satisfy the ODE. To single out a particular solution, we must specify the value of the solution function, denoted by y_o, for some value of t, denoted by t_o Thus part of the given problem data is the requirement that $\mathbf{y}(t_o) = \mathbf{y}_o$. Under reasonable assumptions, this additional requirement determines the unique solution to the given ODE. Because the independent variable t often represents time, we think of t_o as the initial time and $\mathbf{y}_o$ as the initial value of the state vector. Accordingly, the requirement $\mathbf{y}(t_o) = \mathbf{y}_o$ is called an initial condition and an ODE together with the initial condition is called the an initial value problem or IVP" ([p. 386] Heath 2002).

"Most initial value problems which are encountered in practice cannot be solved analytically" ([p. 280] Kahaner, Moler, & Nash 1989).

"If the derivatives of f relative to its arguments are continuous over the domain encompassing its initial values, then a solution exists and is ***unique*** *for some time t"* ([p 38] Borelli & Coleman 1987).

Stiffness:
"When a differential equation is modeling two interrelated phenomena, one changing rapidly, and one changing slowly, difficulties can arise … Some numerical methods have difficulty with stiff problems since they slavishly follow the rapid motions even when they are less important than the general trend in the solution " ([p.

274] Kahaner, Moler, & Nash 1989). *"Physically, stiffness corresponds to a process whose components have very disparate time scales, or a process whose time scale is very short compared to the interval over which it is being studied"* ([p. 401] Heath 2002).

Partial Differential Equations:
"In systems of more than one independent variable, so that partial derivatives are required, and we have a partial differential equation or PDE ([p. 383] Heath, 2002). *Some PDEs are: Maxwell's equations, Navier-Stokes equation governing the behavior of fluids, elasticity, and Schrodinger equation of quantum. mechanics"* ([p. 447] Heath 2002).

Computational solutions to ODEs are not a panacea. Standard numerical methods apply to ODEs defined by smooth functions that are solved over finite intervals. When physical problems involve singular points or infinite intervals, asymptotic expansions must be combined with computational methods to solve such difficulties ([p. 5] Shampine, Gladwell, & Thompson 2003).

9.2 STABILITY OF ORDINARY DIFFERENTIAL EQUATIONS

Consider a first-order ODE

$$y' = dy/dt = f(t, y),$$

where t is the independent variable, f(t, y) is the given ODE, and y(t) is the unknown function we are attempting to determine. If a solution to a ODE problem is perturbed at some point, and the perturbed solution separates from the unperturbed solution as t increases, the ODE is termed *unstable*; alternatively, it the two trajectories come together as t progresses, the ODE is called *stable* ([p. 280] Kahaner, Moler, & Nash 1989; [p. 388] Heath 2002). Note however, the concept of the stability of an ordinary differential equation depends on the solution family and not on a particular solution or choice of initial condition. Also both stable and unstable behaviors can exist in different regions of the problem domain for an ODE ([p. 272] Heath 1997). At a fixed t the ODE, f(t, y), changes with y, and such changes are controlled by $\partial f/\partial y$, which in turn governs the stability of the ODE; the quantity is termed the *Jacobian*, *J* ([p. 281] Kahaner, Moler, & Nash 1989). *Stable ODEs possess negative J values* ([p. 281] Kahaner, Moler, & Nash 1989). For a system of n ODEs J is an n-by-n matrix, whose ik elements are

$$J_{ik} = \frac{\partial f_i}{\partial y_j}.$$

Stability of a ODE system is connected to the *eigenvalues* of J, which are the values, $\lambda_1, \lambda_2, \ldots, \lambda_n$ which force the matrix $(J - \lambda_k I)$ to be singular $(\det(J - \lambda I) \equiv 0)$, where I represents the n-by-n identity matrix ([p. 283] Kahaner, Moler, & Nash 1989). Usually, eigenvalues are complex numbers, whose values depend on t and y ([p. 283] Kahaner, Moler, & Nash 1989).

10.3 NUMERICAL TECHNIQUES

A *system* of ordinary differential equations has the form

$$y' = f(t, y)$$

where t is a real variable and y is a vector-valued function of t and $y' \equiv dy/dt$, represents derivatives with respect to t –

$$\begin{bmatrix} y_1' \\ y_2' \\ \vdots \\ \vdots \\ y_N' \end{bmatrix} \equiv \begin{bmatrix} dy_1/dt \\ dy_2/dt \\ \vdots \\ \vdots \\ dy_N/dt \end{bmatrix}$$

([p. 269] Heath 1997). Consequently, one has a system of *known* coupled differential equations and one wants to derive the *unknown* function, y(t) ([p. 269] Heath 1997).

We focus for the present on solving scalar first-order initial-value ordinary differential equations (IVODE) of the form –

$$\frac{dy}{dt} = f(t, y(t)), \quad y(a) = y_0, \quad \text{for } a \le t \le b,$$

and we will "solve" this problem computationally. The standard approach here is to discretize the problem and compute a solution y(t) on a finite set of discrete times –

$$a \equiv t_o < t_1 < t_2 < \cdots < t_{n-1} < t_n \equiv b,$$

([p. 273] Kahaner, Moler, & Nash 1989). To represent the approximations of y(t) and f(t, y(t)) at the kth grid point, t_k, the notation $y(t_k) \approx y_k$ and $f(t_k, y(t_k)) \approx f_k$ is employed. To determine the approximate solution at a general point, t, some form of interpolation must be used in conjunction with the grid of solution points. All the algorithms to be discussed are based on replacing an IVODE by a difference equation whose solution is determined by stepping discretely in time from a to b.

One-step algorithms integrate f(t, y(t)) from t_k to t_{k+1}. In one-step algorithms information about y_k and $f(t_k, y_k)$ is employed to estimate the function y_{k+1} at the next time step, t_{k+1}, thus the value y_{k+1} is determined from difference equations of the form,

$$(y_{k+1} - y_k)/h_k = F(f, t_k, t_{k+1}, y_{k+1}, y_k),$$

where $h_k = t_{k+1} - t_k$ and F represents an algebraic relationship among these variables. If y_{k+1} is absent on the right hand side, such algorithms are termed *explicit*. When k_{i+1} is present on the right hand side, such algorithms are called *implicit* and some form of root-finding must be used to derive y_{k+1} if F is a *nonlinear* function of y_{k+1} ([p. 537] Bradie 2006).

In contrast to one-step algorithms, classical multistep algorithms possess two components: first, a starting procedure, which provides approximate solutions y_1, y_1, …, y_{k-1}, at points a, a + h, a + 2h, …, a + (k -1)h and, second, a multistep procedure to approximate the solution, y_k, at a + kh ([p. 356] Hairer, Norsett, & Wanner 1987). In such linear multistep algorithms information from n equispaced past steps are employed to predict y_{k+1} –

$$y_{k+1} = \sum_{m=1}^{n} \alpha_m y_{k+1-m} + h\sum_{k=1}^{n} \beta_m f(t_{k+1-m}, y_{k+1-m}),$$

where $h = t_{k+1} - t_k$ ([p. 407] Heath 2002). Note that the coefficients α_k and β_k are determined by polynomial interpolations ([p. 287] Heath 1997). When $\beta_o \neq 0$, algorithms are termed *implicit* or *closed*. When $\beta_o = 0$, such algorithms are termed *explicit* or *open*.

Observe that an explicit multistep method requires only one new function evaluation, $f(t_k, y_k)$, to ascertain the solution, y_{k+1}, at t_{k+1} independent of the number of steps involved. Similarly, an implicit multistep method requires only a single new function evaluation per iteration. Consequently, compared to one-step algorithms multistep algorithms can be more efficient for the same requested accuracy, or, more accurate for the same amount of work ([p. 303] Kahaner, Moler, & Nash 1989).

Errors in the determination of ivode solutions primarily come in two varieties. The first type of error relates to the accuracy of the difference function employed to approximate the ordinary differential equation. This form of error is termed, *local truncation error*, and it is the error associated with say a single step of the algorithm. Such local errors can be estimated and then promptly controlled by adaptive algorithms ([p. 395] Heath 2002). The second form of error, called *global discretization error*, is the cumulative error related to the difference between the actual solution and the approximation, y_{k+1}. This form of error is difficult to control. Due to stability, the global error is not necessarily an increasing function of number of steps, but can oscillate as well as decrease with number of steps taken ([p. 540] Bradie 2006; [p. 394] Heath 2002). In *most practical situations* the primary source of error is truncation error in numerical ivode solutions ([p. 394] Heath 2002).

10.3.1 Forward Euler Method

The forward Euler algorithm is a simple single-step method, which is rarely employed in practice to integrate ordinary differential equations (ODEs). However, the knowledge of its basic properties is key to comprehending more sophisticated algorithms ([p.391], Heath 2002). This method can be derived most simply from a Taylor series expansion of the function, y(t + h), and, thus

$$y(t + h) = y(t) + hy'(t, y(t)) + 0.5h^2 y''(t, y(t)) + \ldots,$$

where $h = h_k$, $t = kh = t_k$, $y(t + h) = y_{k+1}$, $y'(t_k, y_k) = f(t_K, y_k)$, and $y''(t_k, y_k) \equiv \partial f(t_K, y_k)/\partial t$. Terms of higher than second order are discarded ([p. 391] Heath 2002), which in turn leads to

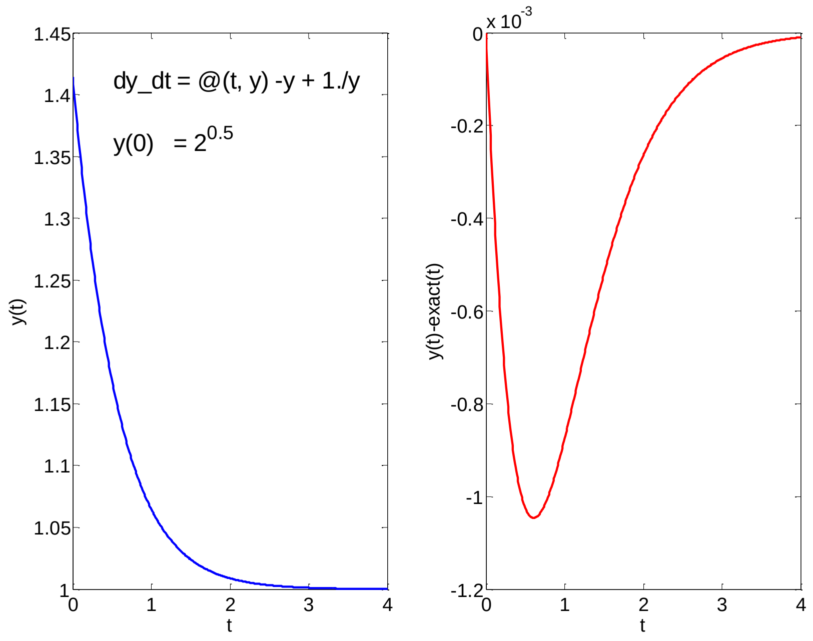
$$y_{k+1} = y_k + hf(t_k, y_k), \; k = 0 \ldots N.$$

This method is implemented in `euler.m`

```
[t, y] = euler(f, tspan, y0, N)
```

whose input arguments are: `f`, the function handle for the ODE, `f(t, y)`, `tspan = [a, b]`, y0, the initial value of y(t) at t = a, and `N` the number of y values sought. For the sake of simplicity, the function `euler.m` outputs its solution at equispaced values of t. Observe that the output arguments are: `t`, a column vector of equispaced independent variables of length, N, and `y` is the corresponding vector of dependent variable. Below the algorithm `euler.m` is employed to solve two ODEs possessing analytic solutions. The solutions from the Euler method are compared to the analytic solutions to gauge solution accuracy.

```
>> dy_dt  = @(t, y) -y + 1./y ;
>> [t, y] = euler(dy_dt, [0; 4], sqrt(2.0), 500);
>> exact  = @(t) sqrt(1.0 + exp(-2*t) );
>> subplot(1,2,1); plot(t, y); xlabel('t'); ylabel('y(t)');
>> subplot(1,2,2); plot(t, y-exact(t));xlabel('t');ylabel('y(t)-exact(t)');
% Observe that with N = 500 steps the global error here ~ 0.5%
```

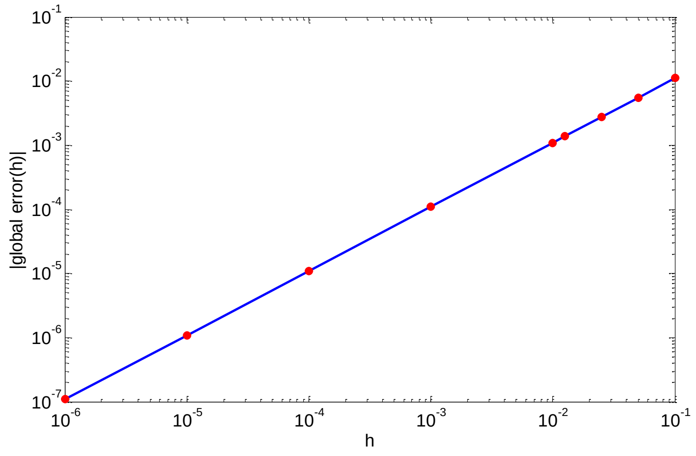


Note the accuracy of the Euler algorithm is modest. This representative solution possesses ≈ 0.1% global errors although N = 500. This is not surprising. This single step approximation possesses a second-order truncation error, but the *global* error is *first*-order accurate in h. This occurs because the number of steps required to approximate, y_N, equals (b - a)/h.

Below is another example of the Euler method. The purpose of this calculation is to quantify the accuracy of the Euler method.

```
>> dy_dt  = @(t, y) -y + 1./y; exact  = @(t) sqrt(1.0 + exp(-2*t) );
>> y0     = sqrt(2.0);    tspan = [0; 1.0];
>> h      =[0.1 0.05 0.025 0.0125 1.0e-02 1.0e-03 1.0e-04 1.0e-05 1.0e-06]';
>> Ns     = round(1.0./h);    tic
>> for k = 1:length(Ns)
     [t, y] = euler(dy_dt, tspan, y0, Ns(k));
     err(k) = y(end) - exact(t(end));
     fprintf('h = %9.2e N = %9.2e er = %16.5e\n', h(k),Ns(k), err(k) );
end; toc
h = 1.00e-001 N = 1.00e+001 er =    -1.13333e-002
h = 5.00e-002 N = 2.00e+001 er =    -5.55776e-003
h = 2.50e-002 N = 4.00e+001 er =    -2.75218e-003
h = 1.25e-002 N = 8.00e+001 er =    -1.36948e-003
h = 1.00e-002 N = 1.00e+002 er =    -1.09453e-003
h = 1.00e-003 N = 1.00e+003 er =    -1.09076e-004
h = 1.00e-004 N = 1.00e+004 er =    -1.09038e-005
h = 1.00e-005 N = 1.00e+005 er =    -1.09034e-006
h = 1.00e-006 N = 1.00e+006 er =    -1.09034e-007
Elapsed time is 31.949461 seconds.
>> loglog(h, abs(err))   % see below
```

Activating the Basic Fitting tab on the `plot(log10(h),log10(abs(err))` finds that the curve of the |global error| below is linear in h, proporrtional to $h^{1.0021}$.



Moreover, Euler's method has a *limited* stability region. It is important to recognize that, although an ODE is absolutely stable, the solver algorithm can be unstable ([p. 289] Kahaner, Moler, & Nash 1989). The stability of Euler's method can be investigated ([p.396] Heath. 2002) by solving $dy/dt = \lambda y$, where λ is a real constant. For an initial solution of y_o, the solution to this ivode is $y_o e^{\lambda t}$. For $\lambda > 0$ all solutions grow exponentially, and any one solution relative to another solution diverges with time and *all* such solutions are termed *unstable*. For $\lambda < 0$, all solutions decay exponentially, and any two solutions converge towards each other. Such an ODE system is *stable*.

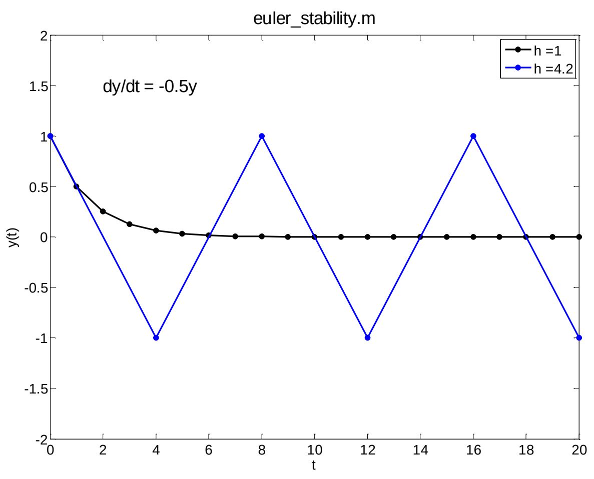
Here the stability of the Euler algorithm (applied to this stable ODE) is investigated. Thus for $\lambda < 0$,

$$
\begin{aligned}
dy/dt &= -|\lambda| y, \\
y(0) &= y_o, \\
y_{k+1} &= y_k - h|\lambda| y_k = (1 - h|\lambda|) y_k, \\
y_{k+1} &= (1 - h|\lambda|)^k y_o = \sigma^k y_o,
\end{aligned}
$$

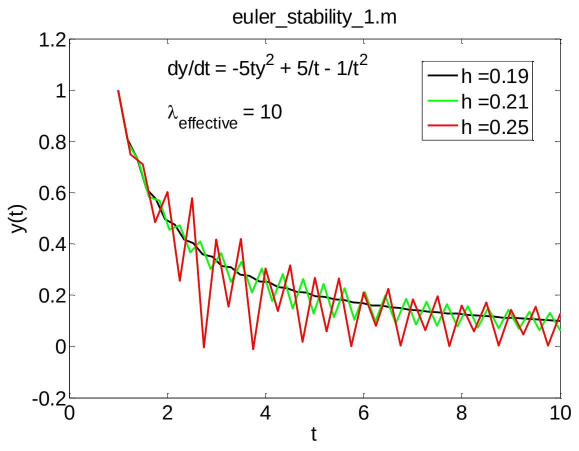
where σ is termed amplification factor. The function, y_k, remains bounded as k increases only if

$$|\sigma| \le 1.$$

For negative real values of λ the computed values of y_k are *computationally stable* if $|(1 - h|\lambda|)| < 1$, or $h \le 2/|\lambda|$. When$|\ (1 - h|\lambda|)\ | > 1$, y_k grows without limit (is unstable), although the actual solutions are stable. Such systems are termed *conditionally stable*. This stability limit is illustrated in the figure below, where the ivode, $dy/dt = -0.5y$, is solved (`euler_stability.m`) via the Euler algorithm. For this ODE the maximum stable step size, h = 4.0. Observe below that oscillatory behavior of a numerical solution, for h = 4.2, is a good indicator of numerical instability ([p. 54] Moin 2010).



Below is another illustration of numerical instability of Euler's algorithm. Here the maximum *stable* step size is 0.2.



10.3.2 Backward Euler Method

This *implicit* method can also be derived from a Taylor series of y(t) around y(t + h) with step of –h

$$y(t) = y(t + h) - hy'(t + h, y(t + h)) + 0.5h^2 y''(t + h, y(t + h)) + \ldots,$$

where $t = t_k$, $y'(t_{k+1}) = f(t_{k+1}, y_{k+1})$, and discarding terms of second or higher order,

$$y_k \;\; = y_{k+1} - hf(t_{k+1}, y_{k+1}),$$
$$y_{k+1} = y_k \; + hf(t_{k+1}, y_{k+1}), \; k = 0 : N.$$

This system is solved in the function

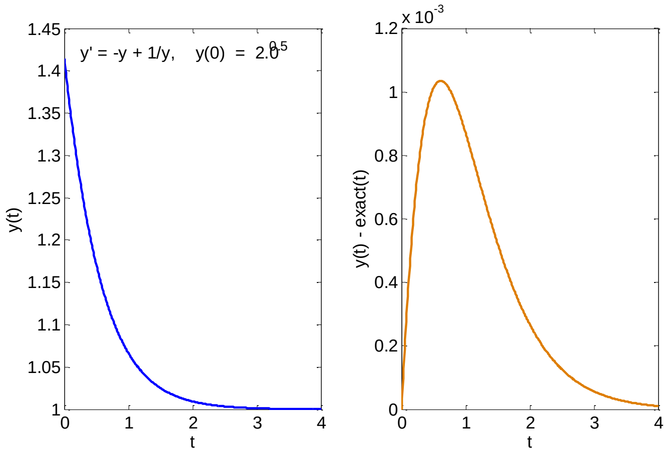
```
function [t, y] = backwards_euler(f, tspan, y0, n)
```

where the function, `fzero`, (`fsolve` if f is a system of equations), is employed to solve g(yy) = 0

$$g(yy) = yy - y_k - f(t_{k+1}, yy)h,$$

for $yy = y_{k+1}$, Thus

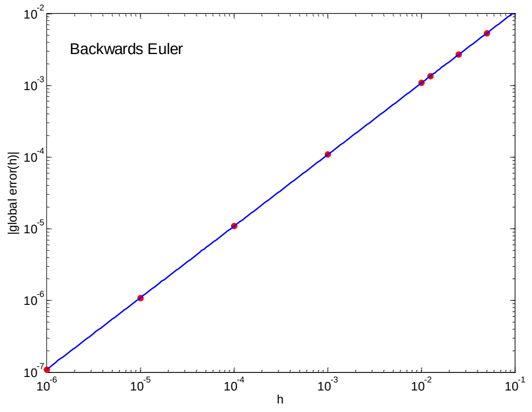
```
>> dy_dt  = @(t, y) -y + 1./y ;
>> [t, y] = backwards_euler(dy_dt, [0; 4], sqrt(2.0), 500);
>> subplot(1,2,1); plot(t, y); xlabel('t'); ylabel('y(t)');
>> text(0.25,1.425, 'y'' = -y + 1/y,    y(0)  =  2.0^{0.5}');
>> subplot(1,2,2);plot(t,y -exact(t));xlabel('t');ylabel('y(t)-exact(t)');
% Observe that with N = 500 steps the global error here ~ 0.2%
```



The purpose of the following calculation is to quantify the accuracy of the backwards Euler method:

```
>> dy_dt  = @(t, y) -y + 1./y; exact  = @(t) sqrt(1.0 + exp(-2*t) );
y0     = sqrt(2.0);    tspan = [0; 1.0];
h      =[0.1 0.05 0.025 0.0125 1.0e-02 1.0e-03 1.0e-04 1.0e-05 1.0e-06]';
Ns     = round(1.0./h);    tic
for k = 1:length(Ns)
    [t, y] = backwards_euler(dy_dt, tspan, y0, Ns(k));
    err(k) = y(end) - exact(t(end));
    fprintf('h = %9.2e N = %9.2e er = %16.5e\n', h(k),Ns(k), err(k) );
end; toc
h = 1.00e-001 N = 1.00e+001 er =     1.04981e-002
h = 5.00e-002 N = 2.00e+001 er =     5.34874e-003
h = 2.50e-002 N = 4.00e+001 er =     2.69991e-003
h = 1.25e-002 N = 8.00e+001 er =     1.35641e-003
h = 1.00e-002 N = 1.00e+002 er =     1.08617e-003
h = 1.00e-003 N = 1.00e+003 er =     1.08992e-004
h = 1.00e-004 N = 1.00e+004 er =     1.09030e-005
h = 1.00e-005 N = 1.00e+005 er =     1.09033e-006
h = 1.00e-006 N = 1.00e+006 er =     1.09034e-007
Elapsed time is 987.212883 seconds.
```

Activating the Basic Fitting tab on the `plot(log10(h),log10(abs(err))` finds that the curve of the |global error| below is linear in h, proporrtional to $h^{0.9979}$.



The global accuracy of this algorithm is only first order in h. Moreover, this implicit method requires that a root finder be employed to find the solution, y_{k+1}, consequently, the backwards Euler algorithm is significantly slower, by a factor of ~ 30, than corresponding forward Euler algorithm. Why use it? Backwards Euler algorithm has a larger stability region than forward Euler method ([p.398] Heath 2002).

The stability of Euler's method can be investigated by integrating dy/dt = -λy,

$$\frac{dy}{dt} = -\lambda y,\ y(0) = y_o,$$

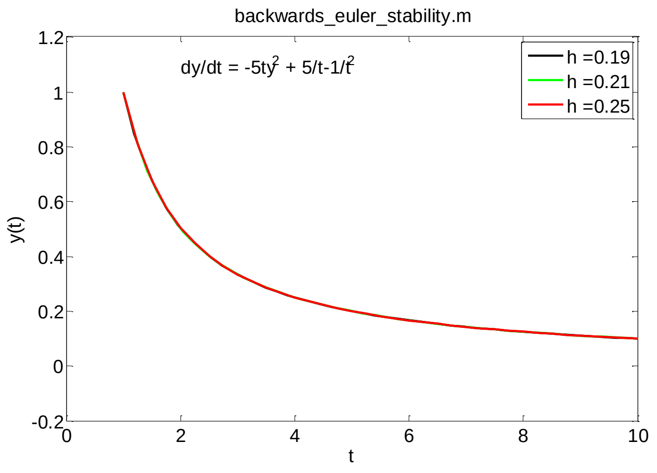
$$y_k = y_{k+1} - h\lambda f(t_{k+1}, y_{k+1}) = (1 + h|\lambda|)y_{k+1},$$

$$y_k = (1 + h|\lambda|)y_{k+1}, \quad \Rightarrow y_k = \frac{y_o}{(1 + h|\lambda|)^k}.$$

For the backwards Euler method to be stable, we only require that

$$\left|\frac{1}{1 + h|\lambda|}\right| \le 1.$$

For *any* h > 0, the backwards Euler method is stable, whenever $\lambda < 0$ ([p.398] Heath 2002). Consrquently, backwards Euler method is termed *unconditionally stable*. Compare the following illustration to the corresponding calculation using the forward Euler algorithm.



10.3.3 Euler-Trapezoid Method

The use of the above Euler methods is severely limited by inefficiency - both are first order in h. However, higher order method methods can be derived from these Euler routines. This simplest approach is to average both routines, or employ the trapezoid rule to integrate f(t, y(t)),

$$y_{k+1} = y_k + 0.5h\left[f(t_k,y_k) + f(t_{k+1},y_{k+1})\right],\ k = 0{:}N,$$

creating the *implicit* Euler trapezoid rule ([p. 400] Heath. 2002). Such systems are solved in

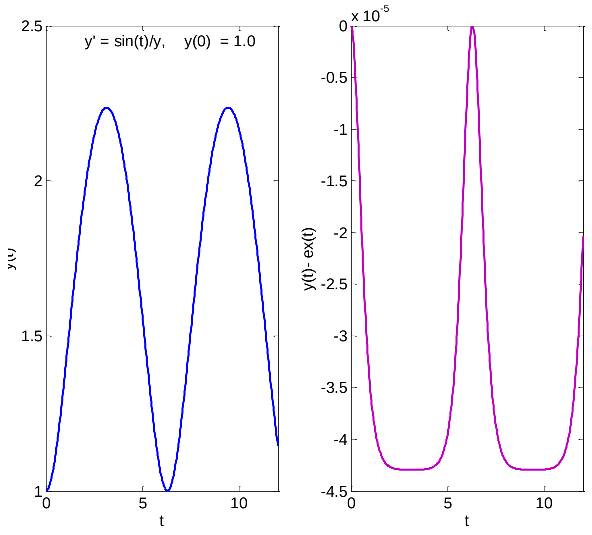
```
function [t, y] = euler_trapezoid(f, tspan, y0, N)
```

where either the intrinsic function **`fzero`** or **`fsolve`** is employed to find yy where g(yy) = 0,

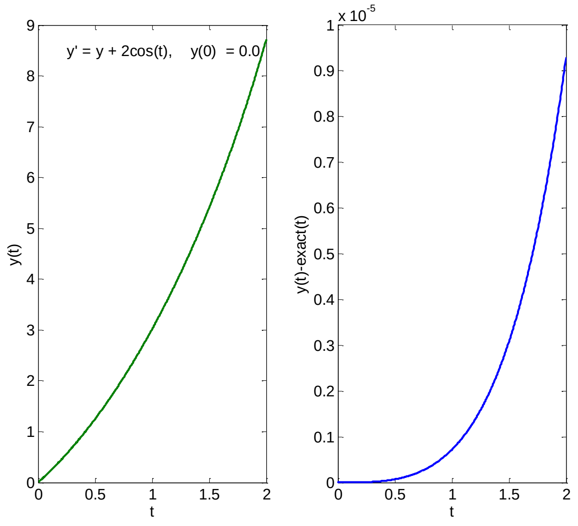
$$g(yy) = (yy - y_k - 0.5*(f(t_k,\ y_k) + f(t_{k+1},\ yy))*h);$$

where yy = y_{k+1}. Thus

```
>> dy_dt = @(t, y) sin(t)./y ;
>> [t, y] = euler_trapezoid(dy_dt, [0 12], 1.0, 500);
>> ex = @(t) sqrt(3 - 2*cos(t));
>> subplot(1,2,1); plot(t, y); xlabel('t'); ylabel('y(t)');
>> text(2.0, 2.45, 'y'' = sin(t)/y,    y(0)  = 1.0');
>> subplot(1,2,2); plot(t, y-ex(t));xlabel('t');ylabel('y(t)- ex(t)');
% Observe with N = 500 steps the global error now ~10^-4
```



```
>> dy_dt = @(t, y) y + 2*cos(t);
[t, y] = euler_trapezoid(dy_dt, [0 2.0], 0.0, 500);
exact = @(t) exp(t) + sin(t) - cos(t) ;
subplot(1,2,1); plot(t, y); xlabel('t'); ylabel('y(t)');
subplot(1,2,2); plot(t,y-exact(t));xlabel('t'); ylabel('y(t)-exact(t)');
>> subplot(1,2,1); plot(t, y); xlabel('t'); ylabel('y(t)');
>> text(0.25, 8.5, 'y'' = y + 2cos(t),    y(0)  = 0.0');
>>subplot(1,2,2); plot(t,y-exact(t));xlabel('t'); ylabel('y(t)-exact(t)');
% Observe that with N = 500 steps the global error here ~ 10^-5!
```



The stability of the trapezoid method can be investigated by integrating the stable ODE,

$$\frac{dy}{dt} = -|\lambda| y,\ y(0) = y_o,$$

$$y_{k+1} = y_k - h(|\lambda| y_{k+1} + |\lambda| y_k)/2, \Rightarrow y_k = \left(\frac{(1 - 0.5h|\lambda|)}{(1 + 0.5h|\lambda|)}\right)^k y_o.$$

Thus the trapezoid method is stable when -

$$\left|\frac{(1 - 0.5h|\lambda|)}{(1 + 0.5h|\lambda|)}\right| \leq 1.$$

which applies for any h > 0, whenever $\lambda < 0$. Thus the trapezoid method is termed *unconditionally stable* ([p.400] Heath 2002).

10.3.4 Taylor Series Method

In section 10.3.1 the Euler method was derived from a truncated Taylor expansion. Why not create higher-order single step methods by keeping additional terms (e.g., p. 404] Heath 2002)? Thus

$$y(t + h) = y(t) + hy'(t) + \frac{1}{2}h^2 y''(t) + \frac{1}{6}h^3 y'''(t) + \cdots$$

which results in a second order function -

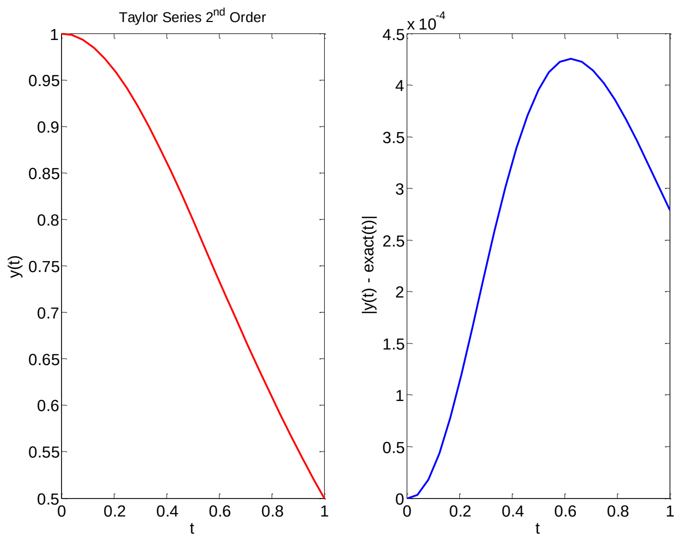
$$y_{k+1} = y_k + hy_k' + 0.5h^2 y_k''.$$

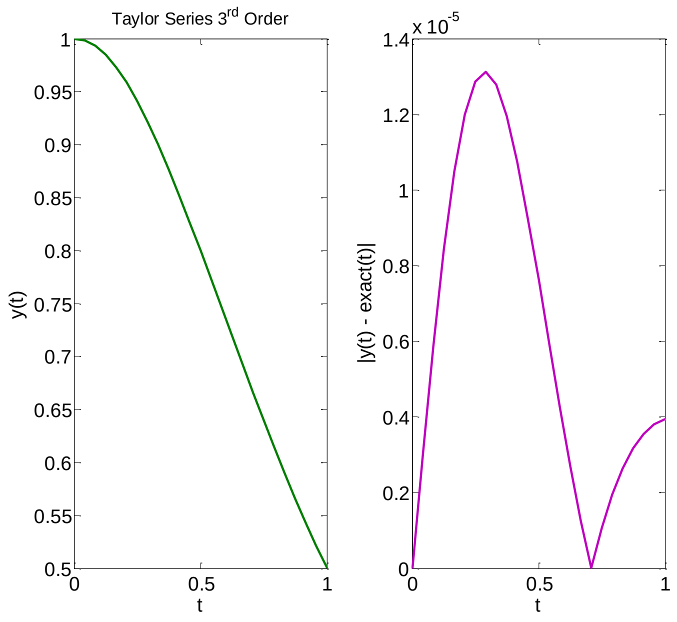
Higher derivatives can be derived from y' = f(t, y), via the chain rule. Differentiating y' = f(t,y) produces

$$\begin{aligned}
y'(t) &= f,\\
y''(t) &= f_t + f_y y' = f_t + f_y f,\\
y'''(t) &= f_{tt} + 2f_{ty} y' + f_y y'' + f_{yy}(y')^2,\\
&= f_{tt} + 2f_{ty} f + f_{yy} f^2 + f_y(f_t + f_y f),\\
y''''(t) &= f_{ttt} + 3f_{tty} y' + 3f_{tyy}(y')^2 + 3f_{ty} y'' + f_y y''' + 3f_{yy} y'y'' + f_{yyy}(y')^3\\
&= f_{ttt} + 3f_{tty} f + 3f_{tyy} f^2 + f_{yyy} f^3 + f_y(f_{tt} + 2f_{ty} f + f_{yy} f^2) + 3(f_t + f_y f)(f_{ty} + f_{yy} f) + f_y^2(f_t + f_y f)
\end{aligned}$$

where the subscripts indicate partial derivatives ([p. 451] Matthews & Fink1999). However, as can be seen, higher-order Taylor methods require complicated differentials which are difficult to calculate by hand ([p. 404] Heath 2002).

This method is implemented in **`example_taylor_2nd.m`** and **`example_taylor_3rd.m`**. The simple ivode, $y' = -2ty^2$ is integrated for $0 \le t \le 1.0$. These solutions and their error are shown below for N = 25.





Inspection of theses m-files illustrate the complicated nature of such algorithms. Other simpler high-order algorithms exist, eg., RK methods discussed in section 10.3.6.

10.3.5 Solving High-Order Differential Equations

"Almost all library software for solving IVP solves systems of first-order equations. If a user has a second or higher order differential equation must be transformed to a system of first-order equations (by hand). Being able to perform this transform is a must for anyone who wants to use current software" ([p. 279] Kahaner, Moler, & Nash 1989).

Most high-order order differential equations of technical interest can be solved as a coupled system of first-order differential equations. As such an example consider a linear harmonic oscillator whose behavior is usually modeled as a single second-order differential equation –

$$\frac{d^2x}{dt^2} = -(k/m)x \quad \text{with initial conditions} \quad x(0) = x_0 \ \& \ v(0) = v_0.$$

As you know, it has the following simple analytic solution –

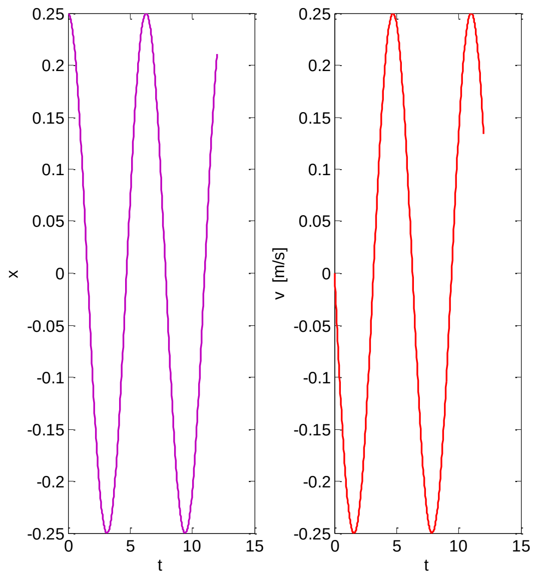
$$x = A\cos(\omega t + \varphi), \quad \omega = \sqrt{\frac{k}{m}},$$
$$A = \sqrt{x_0^2 + \frac{v_0^2}{\omega^2}}, \quad \varphi = \tan^{-1}(-\frac{v_0}{\omega x_0}).$$

Computationally this dynamical system is solved as *two coupled first-order* differential equations –

$$u \equiv \begin{bmatrix} x \\ \frac{dx}{dt} \end{bmatrix}, \quad \frac{du}{dt} \equiv \begin{bmatrix} \frac{dx}{dt} \\ \frac{d^2x}{dt^2} \end{bmatrix} \equiv \begin{bmatrix} v \\ a \end{bmatrix} \quad \Rightarrow u = \begin{bmatrix} u(1) \\ u(2) \end{bmatrix}, \ du\_dt = \begin{bmatrix} u(2) \\ \frac{-ku(1)}{m} \end{bmatrix}.$$

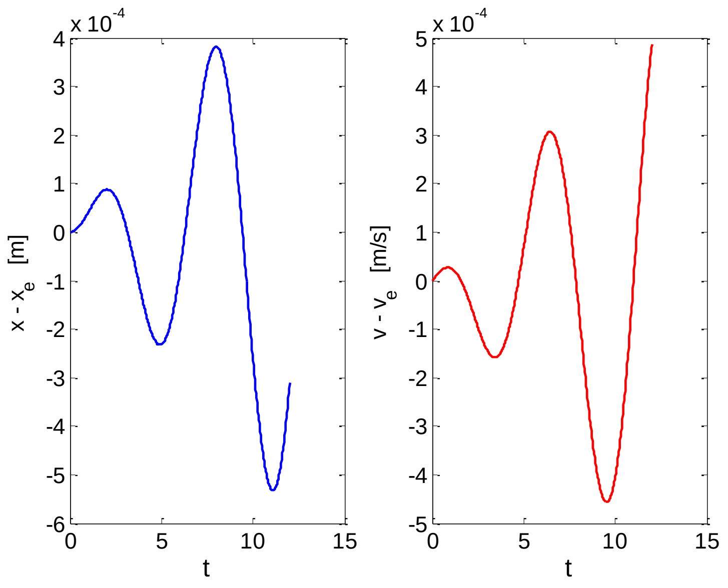
Thus for a oscillator for k = 1.0, m = 1.0, x_0 = 0.25 m, and v_0 = 0.0 m/s,

```
>> k = 1.0; m = 1.0; du_dt  = @(t, u) [u(2); -(k/m)*u(1)];u0 = [0.25; 0.0];
>> [t, u] = euler_trapezoid(du_dt, [0 12], u0, 500);
>> subplot(1,2,1); plot(t, u(:,1)); xlabel('t'); ylabel('x');
>> subplot(1,2,2); plot(t, u(:,2)); xlabel('t'); ylabel('v  [m/s]');
>> tic; [t, u] = euler_trapezoid(du_dt, [0 12], u0, 500); toc
Elapsed time is 1.587 seconds.
```



```
>> w      = sqrt(k/m);         A     = sqrt(u0(1)^2 + (u0(2)/w)^2);
>> tanphi = -u0(2)/(w*u0(1)); phi   = atan(tanphi);
>> x      = @(t) A*cos(w*t + phi);

>> v      = @(t) -w*A*sin(w*t + phi);
>> subplot(1,2,1);plot(t, u(:,1) - x(t)); xlabel('t'); ylabel('x - xe');
>> subplot(1,2,2);plot(t, u(:,2) - v(t)); xlabel('t'); ylabel('v - ve');
```



10.3.6 Runge-Kutta Methods

10.3.6a) Runge-Kutta Second Order

Runge-Kutta methods are single-step approaches which do not require the explicit calculation of high-order derivatives as does the previous Taylor method. To derive this second-order method, you employ the chain rule,

$$y' = f(t,y),$$

$$y'' = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial y} f,$$

where the functions are evaluated at (t, y). Now expand the terms on the right in a Taylor series of *two* variables,

$$f(t + h, y + hf) = f(t,y) + h\frac{\partial f}{\partial t} + h\frac{\partial f}{\partial y} f(t,y) + O(h^2),$$

$$y'' \approx \frac{\partial f}{\partial t} + \frac{\partial f}{\partial y} f \approx \frac{f(t + h, y + hf) - f(t,y)}{h} + O(h).$$

Thus this second-order Taylor series becomes

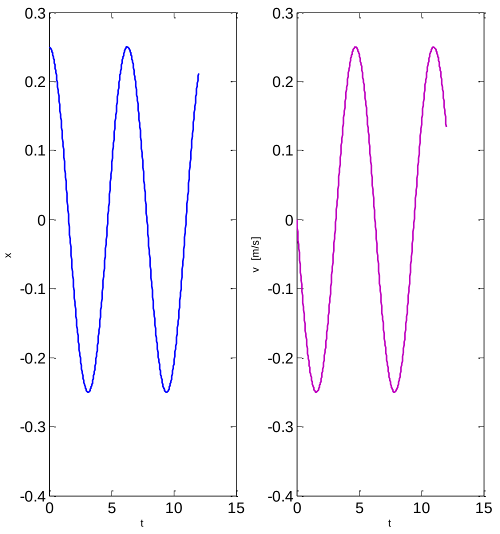
$$y(t+h) = y(t) + hy'(t) + \frac{h^2}{2} y''(t) + \cdots \Rightarrow y_{k+1} = y_k + hy_k' + \frac{h^2}{2} y_k'' + \cdots$$

$$y_{k+1} = y_k + hf(t_k, y_k) + \frac{h^2}{2}\left[\frac{f(t + h, y + hf) - f(t,y)}{h}\right] + \cdots \approx y_k + \frac{h}{2}[\, f(t,y) + f(t + h, y + hf(t_k, y_k))\,],$$

$$y_{k+1} \approx y_k + \frac{h}{2}[k_1 + k_2], \quad k_1 = f(t, y), k_2 = f(t + h, y + hf(t_k, y_k)).$$

This method is implemented in `rk2.m` and a representative ivode is solved below. Note that the global error is second order accurate in h. This *explicit* method is as accurate as `euler_trapezoid.m`, but its solution does not require the use of root finders. This improved accuracy, and *execution speed*, is illustrated in the figure below.

```
k          = 1.0; m = 1.0;
>> du_dt  = @(t, u) [u(2); -(k/m)*u(1)];
>> u0     = [0.25; 0.0];
>> [t, u] = rk2(du_dt, [0 12.0], u0, 500);
>> subplot(1,2,1); plot(t, u(:,1)); xlabel('t'); ylabel('x');
>> subplot(1,2,2); plot(t, u(:,2)); xlabel('t'); ylabel('v  [m/s]');
>> tic; [t, u] = rk2(du_dt, [0 12], u0, 500); toc
Elapsed time is 0.058 seconds.
% 0.06 sec   ~25 shorter duration than euler_trapezoid.m
```



10.3.6b) Runge-Kutta Fourth Order

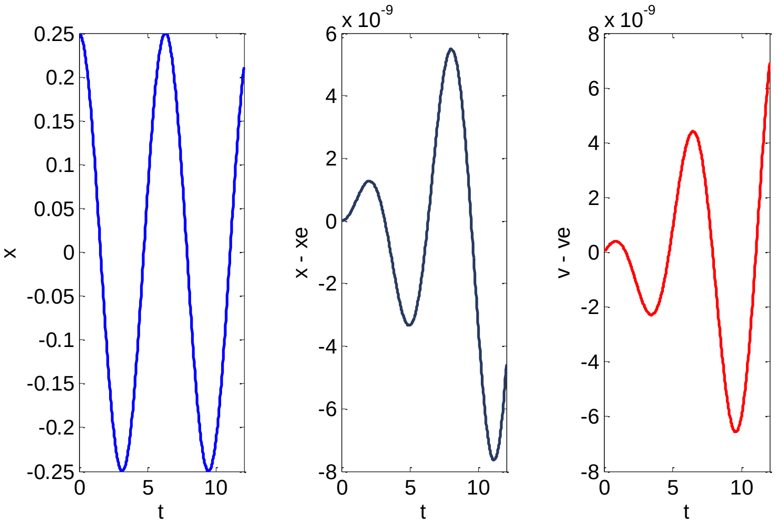
Expanding the Taylor series of f(t, y) but including more terms, and solving rather involved algebra, one can determine the most commonly employed Runge-Kutta method, the fourth-order rule –

$$\begin{aligned}
y_{k+1} &= y_k + (h/6)(k_1 + 2k_2 + 2k_3 + k_4),\\
k_1 &= f(t_k, y_k),\\
k_2 &= f(t_k + h/2, y_k + hk_1/2),\\
k_3 &= f(t_k + h/2, y_k + hk_2/2),\\
k_4 &= f(t_k + h, \quad y_k + hk_3).
\end{aligned}$$

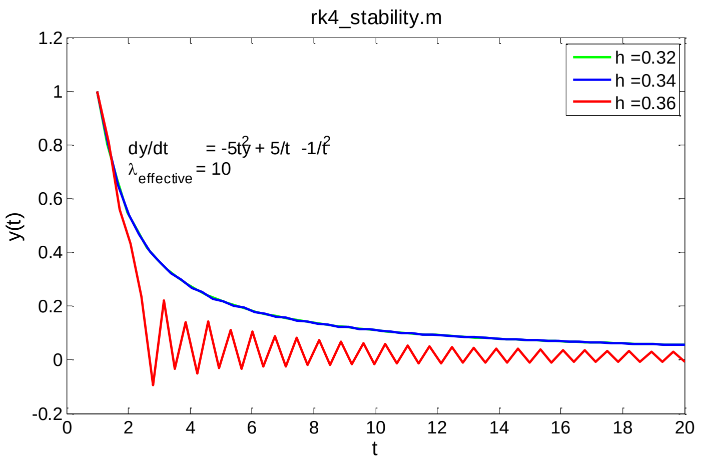
This method is analogous to Simpson's quadrature rule, and identical to Simpson's quadrature rule when f(t, y) is independent of y ([p. 406] Heath 2002). This method is implemented in `rk4.m` and a representative ivode is solved below. Note that the truncation error is *fourth* order accurate in h.

```
>> k      = 1.0; m = 1.0;
>> du_dt  = @(t, u) [u(2); -(k/m)*u(1)];
>> u0     = [0.25; 0.0];
>> tic; [t, u] = rk4(du_dt, [0 12], u0, 500); toc
Elapsed time is 0.090138 seconds.
```

```
>> w      = sqrt(k/m);            A     = sqrt(u0(1)^2 + (u0(2)/w)^2);
>> tanphi = -u0(2)/(w*u0(1));     phi   = atan(tanphi);
>> x      = @(t) A*cos(w*t + phi);v     = @(t) -w*A*sin(w*t + phi);
>> subplot(1,3,1); plot(t, u(:,1)); xlabel('t'); ylabel('x');
>> subplot(1,3,2);plot(t, u(:,1) - x(t)); xlabel('t'); ylabel('x - xe');
>> subplot(1,3,3);plot(t, u(:,2) - v(t)); xlabel('t'); ylabel('v - ve');
>> subplot(1,2,2);plot(t, u(:,2) - v(t)); xlabel('t'); ylabel('v - ve');
% rk4 50% longer duration than rk2 BUT ~10^4 more accurate!
```



The stability of the RK4 algorithm has been addressed by Dormand (1996) and Ascher & Petzold (1998). They show that the stability region is somewhat greater than that associated with the forward Euler's method. Figure 4.4 of Ascher & Petzold and Figure 3 of Dormand suggest that h ≤ 2.8/(|λ|).



10.3.6c) Runge-Kutta-Fehlburg Method

Fehlburg (1970) created a Runge-Kutta algorithm from *six* function calls per step, derived *two* solution estimates, one fourth order in step size and the other fifth order, and estimated solver accuracy from their difference ([p. 406] Heath 2002). Such quantitative determinations of errors lead to the development of *efficient* adaptive error control algorithms through the step size, h, which up to now has been treated as a constant. Since thar time other effective adaptive Runge-Kutta pairs have been developed, e.g. Dormand & Prince (1980).

The Runge-Kutta-Fehlburg method, usually called RKF45, approximates a solution to an IVODE using *two* different approximations ([p. 466] Matthews & Fink 1999). At every step the two approximations are compared. If the two approximations agree, the fifth order estimate is used. If the two approximations disagree, the step size is reduced. However, if the two approximations agree too closely, the step size is increased. The following approach is employed. The fourth order approximation is

$$y_{k+1} = y_k \frac{+25k_1}{216} \frac{+1408k_3}{2565} \frac{+2197k_4}{4104} \frac{-k_5}{5},$$

and the corresponding fifth order approximation is

$$z_{k+1} = y_k \frac{+16k_1}{135} \frac{+6656k_3}{12825} \frac{+28561k_4}{56430} \frac{-9k_5}{50} \frac{+2}{55} k_6.$$

The coefficients are:

$$k_1 = hf(t, y_k),$$

$$k_2 = hf(t + \frac{h}{4}, y_k + \frac{k_1}{4}),$$

$$k_3 = hf(t + \frac{3h}{8}, y_k + \frac{3k_1 + 9k_2}{32}),$$

$$k_4 = hf(t + \frac{12}{13}h,\ y_k + \frac{1932k_1 - 7200k_2 + 7296k_3}{2197}),$$

$$k_5 = hf(t + h, y_k + \frac{439k_1}{216} - 8k_2 + \frac{3680k_3}{513} \frac{-845k_4}{4104}),$$

$$k_6 = hf(t + h/2, y_k \frac{-8k_1}{27} + 2k_2 \frac{-3544k_3}{2565} \frac{+1859k_4}{4104} \frac{-11k_5}{40}).$$

The optimal step size is determined from the current step size, h,

$$h_{new} \approx 0.84 \left(\frac{tol \bullet h^{5/4}}{|z_{k+1} - y_{k+1}|} \right)^{1/4}.$$

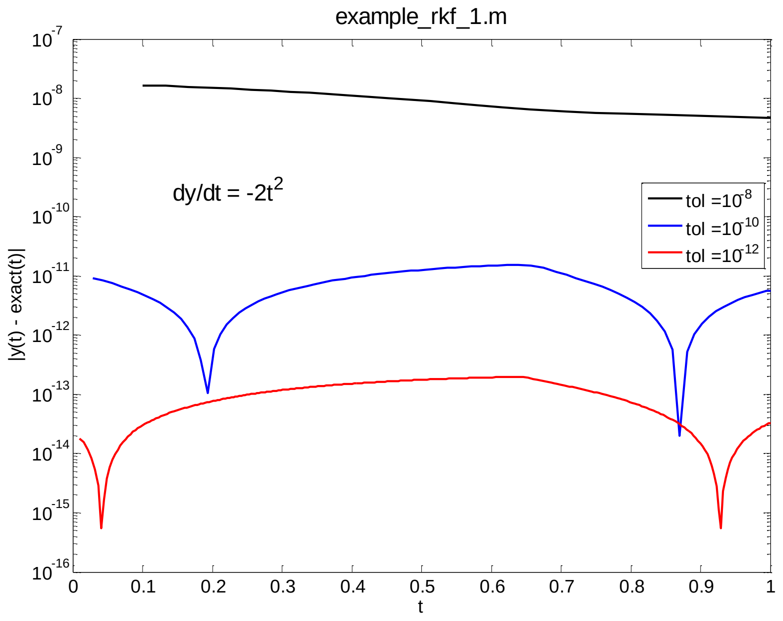
The stability of the RKF regime is similar to that of the RK4 method.

The RKF method is implemented here in `rkf.m`, my MATLAB translation of algorithm 5.3 of Burden & Faires (2001) and a representative ivode is solved in `example_rkf_1.m`. Note that the input parameters are different for rkf.m compared to those of previous methods; here the relative tolerance is supplied in lieu of n the number of steps. Compare the accuracy of the RKF45 method to those of the previous methods. Most IVODE solvers employ similar forms of error tolerance.

MATLAB's built-in function, `ode45`, is a variant of the rkf algorithm. I sugguest you employ this algorithm in lieu the above rkf algorithm.

```
[t, y] = rkf(du_dt, [0,1], 1, tol);

>> example_rkf_1
10^-08   No. of calls = 30
10^-10   No. of calls = 93
10^-12   No. of calls = 292
```



10.3.7 Multi-Step Methods

Up to this point all the algorithms discussed have been single-step methods. Since the approximate solutions exist already at t_o, t_1, t_2, … t_{k-1}, it is useful to investigate multi-step algorithms to calculate $y_k(t_k)$. As mentioned in section 10.1, multi-step methods occur in two flavors, explicit or implicit.

How does one *start* a multi-step method? In the past (prior to 1980s) the obvious tactic was to employ single-step methods, usually RK4 or RKF to generate these values, as we do in the following examples. However, the modern approach is to use variable order algorithms, which employ low-order methods initially and increase the order as more values are calculated ([p. 408], Heath 2002).

Implicit multi-step algorithms are more accurate as well as more stable than explicit multi-step algorithms, however, these invariably nonlinear equations require good initial estimates to determine y_{i+1} ([p. 408] Heath 2002). As will be discussed in a later section (10.3.7e), an excellent initial solution estimate is generated by an explicit method, and then implicit methods are effectively used in form of *predictor-corrector* pairs ([p. 409] Heath 2002). However, to benefit *fully* from the nature of *implicit* multi-step algorithms, they must be solved directly via root-finding techniques when their implicit nature is needed to solve stiff ivodes ([p. 408] Heath 2002).

The basic properties of classical multi-step methods have been summarized by Heath ([p. 410] 2002):

- Not self-starting
- Modifying step size, which is needed in adaptive algorithms, is complicated.
- Good local error estimate from difference between predictor/corrector solutions.
- Complicated to program.
- Although implicit methods more stable than explicit methods, still not unconditionally stable. No multi-step method greater than second order is unconditionally stable.

In the simplest form of classical multistep methods one assumes that the independent variables are equispaced, $t_k = a + kh$, where h is fixed at (b – a)/N for arbitrary integer N. Here N represents the order of the fitting polynomial. Integrating a scalar IVODE produces the *formal* quadrature equation,

$$y_{k+1} = y_k + \int_{t_k}^{t_k+h} f(x, y(x))dx.$$

To solve this integral first the integrand, f(x, y(x)), is approximated as a Lagrange polynomial, using the known values (t_{k-2-n}, f_{k-2-n}) and (t_k, f_k), for n = 1:N, and this polynomial is *extrapolated* to determine the unknown integrand value of f(x, y(x)) at t_k. Second, numerical quadrature is employed to perform the necessary integration in the equation above.

10.3.7a) Explicit Second-Order (N = 2) Adam-Bashforth Method

Here N = 1:2 and the integrand f(x, y(x)), is fitted at two grid points (t_{k-1}, f_{k-1}) & (t_k, f_k), where y(t) is known, using Lagrangian interpolation,

$$P(t) = f_{k-1}L_1(t) + f_kL_2(t).$$

Then this P(t) is evaluated (extrapolated) to $t = t_{k+1}$ to estimate f_{k+1} which now equals $P(t_{k+1})$. The relevant parameters are determined most simply using an equispaced t-grid = $[t_{k-1}; t_k]$ = [1; 2], and extrapolating to t_{k+1} of 3 to approximate f_{k+1} (f_3) -

```
% Determine fk+1 via extrapolation to tk+1 = +3 using Lagrange interpolation
>> td = [1 2]; fd = [1 0]; L1 = lagrange_interpolation(td, fd, 3)
L1 =   -1
>> td = [1 2]; fd = [0 1]; L2 = lagrange_interpolation(td, fd, 3)
L2 =   +2
```

Thus $f_3 = -f_1 + 2f_2$ OR

$$f_{k+1} = -f_{k-1} + 3f_k$$
.

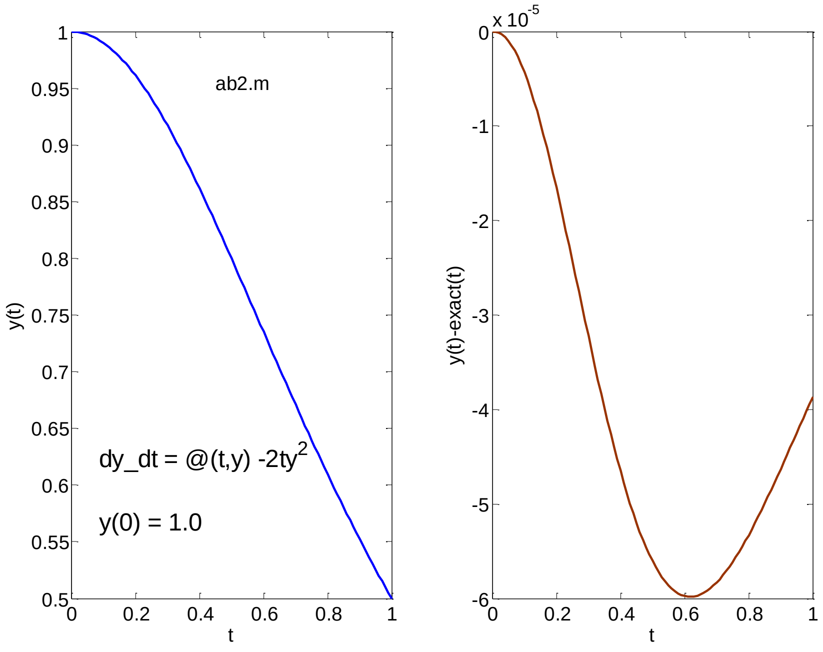
Observe that the values of these L_1 and L_2 parameters are independent of the actual grid values as long the grid values are equispaced. Now one integrates f(x) from $t_k = t_2$ to $t_{k+1} = t_3$ using the trapezoid quadrature rule, which here is $y_3 = y_2 + 0.5(t_3 - t_2)(f_2 + f_3)$. Thus $y_3 = y_2 + 0.5h(3f_2 - f_1)$ and, the explicit 2nd-oder Adams-Bashforth method becomes

$$y_{k+1} = y_k + 0.5h(3f_k - f_{k-1}).$$

The truncation error for this method is $5h^3/12$. Employing $y' = -\lambda y$ and $y(0) = +1$, it can be shown that this representative IVODE possesses a stability range, $-1 \le h\lambda \le 0.0$ for real λ ([p. 178] Dormand 1996; [p. 411] Heath 2002).

This method is implemented in `ab2.m` and a representative ivode is solved below.

```
>> clear
>> dy_dt = @(t,y) -2*t*y.^2;
>> tspan = [0; 1 ];
>> y0    = 1.0;
>> exact = @(t) 1./(1 + t.^2);
>> [t, y] = ab2(dy_dt, tspan, y0, 100);
>> subplot(1,2,1); plot(t, y); xlabel('t'); ylabel('y(t)');
subplot(1,2,2); plot(t,y-exact(t));xlabel('t'); ylabel('y(t)-exact(t)');
```



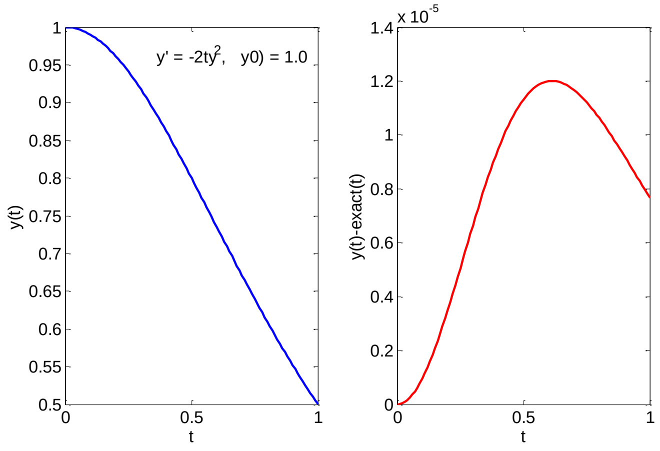
10.3.7b) Implicit Second-Order(N = 2) Adams-Moulton Method

Implicit Adams-Moulton (AM) methods possess greater stability domains compared to AB methods of the same order, however, these AM methods must be iterated to convergence in order to achieve such stability ([p. 410] Heath 2002). Observe that the second order AM method is simply the Euler-Trapezoid method –

$$y_{k+1} = y_k + 0.5h(f(t_k, y_k) + f(t_{k+1}, y_{k+1})),$$

presented earlier ([p. 360] Hairer, Norsett, & Wanner 1993). The truncation error for this method is $-h^3/12$. Employing $y' = -\lambda y$ and $y(0) = +1$, it can be shown that this representative IVODE possesses a stability range, $-\infty \le h\lambda \le 0.0$ for real λ ([p. 411] Heath 2002).

```
dy_dt = @(t,y) -2*t*y.^2;
tspan = [0; 1 ]; y0    = 1.0;
exact = @(t) 1./(1 + t.^2);
[t, y]= euler_trapezoid(dy_dt, tspan, y0, 100);
>> subplot(1,2,1); plot(t, y); xlabel('t'); ylabel('y(t)');
>> text(0.25, 0.9, '-2ty^2,   y0) = 1.0');
>> subplot(1,2,2);plot(t,y-exact(t));xlabel('t');ylabel('y(t)-exact(t)');
```



10.3.7c) Explicit Fourth-Order (N = 4) Adam-Bashforth Method

Here one employs interpolation using (t_1, f_1), (t_2, f_2), (t_3, f_3), and (t_4, f_4), where y and f has been determined, to extrapolate the unknown value f_5 at t_5 -

```
Determine f5 via extrapolation to td(5) = +5 using Lagrange interpolations
>> td = [1 2 3 4]; fd = [1 0 0 0]; lagrange_interpolation(td, fd, 5)
ans = -1
>> td = [1 2 3 4]; fd = [0 1 0 0]; lagrange_interpolation(td, fd, 5)
ans = +4
>> td = [1 2 3 4]; fd = [0 0 1 0]; lagrange_interpolation(td, fd, 5)
ans = -6
>> td = [1 2 3 4]; fd = [0 0 0 1]; lagrange_interpolation(td, fd, 5)
ans = +4
f5 = -f1 +4f2 -6f3 +4f4
```

Now this extrapolated value f_5 is used to integrate f(t, y(t)) from t_4 to t_5 using the Simpson quadrature rule. However, the use of this quadrature rule forces us to employ Lagrangian extrapolation first to derive $f_{4.5}$ -

```
y5 = y4 + (1/6)(t5 – t4)(f4 + 4f4.5 + f5)    → need f4.5 = f(t4.5, y4.5)

>> format rat
>> td = [1 2 3 4]; fd = [1 0 0 0]; lagrange_interpolation(td, fd, 4.5)
ans =  -5/16
>> td = [1 2 3 4]; fd = [0 1 0 0]; lagrange_interpolation(td, fd, 4.5)
ans = +21/16
>> td = [1 2 3 4]; fd = [0 0 1 0]; lagrange_interpolation(td, fd, 4.5)
ans = -35/16
>> td = [1 2 3 4]; fd = [0 0 0 1]; lagrange_interpolation(td, fd, 4.5)
ans = +35/16

f4.5 = (-5f1 +21f2 -35f3 +35f4)/16
y5 = y4 + (1/6)(t5 – t4)(f4 + 4f4.5 + f5) = (h/6)(f4 + 4f4.5 + f5)
y5 = (h/6)(f4 + (-5f1 +21f2 -35f3 +35f4)/4 -f1  +4f2 -6f3  +4f4)
y5 = (h/24)(4f4 + (-5f1 +21f2 -35f3 +35f4)-4f1 +16f2 -24f3 +16f4)
y5 = (h/24)(-9f1 +37f2 -59f3 +55f4)
```

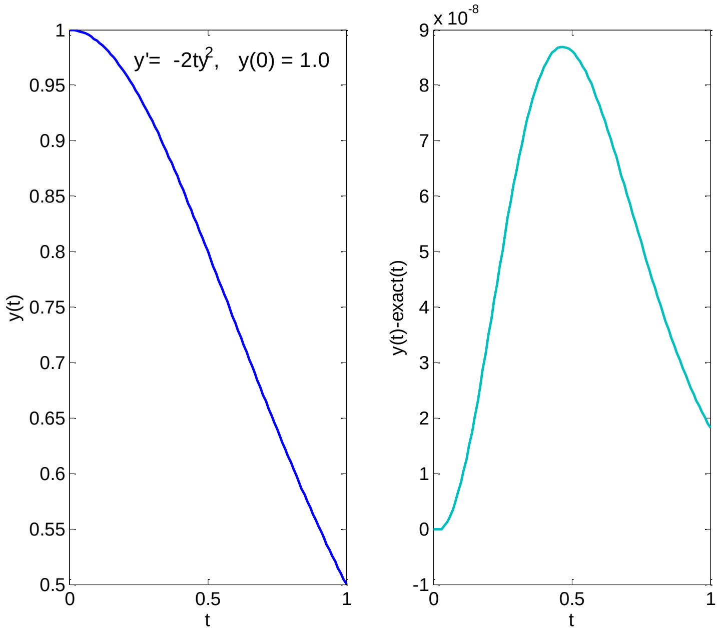
Thus a fourth-order Adam-Bashforth method is

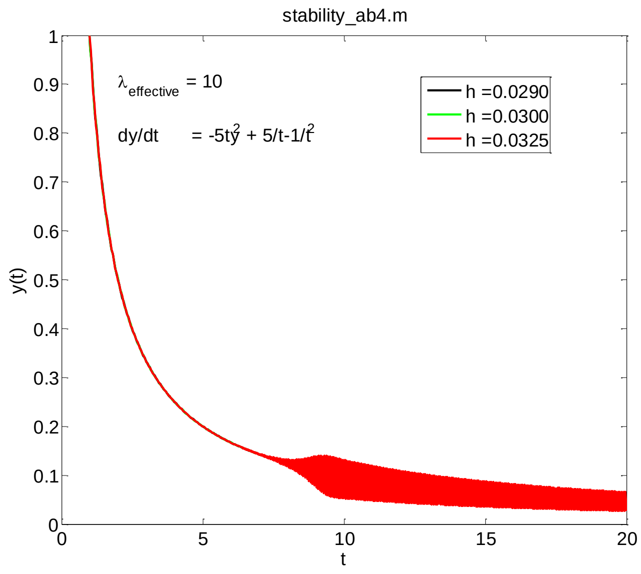
$$y_{k+1} = y_k + \frac{h}{24}\left(55f_k - 59f_{k-1} + 37f_{k-2} - 9f_{k-3}\right)$$

with a truncation error of -(251/720)h^5 ([p. 411] Heath 2002). Employing y' = -λy and y(0) = + 1, it can be shown that this representative IVODE possesses a stability range, $-0.3 \le h\lambda \le 0.0$ for real λ ([p. 411] Heath 2002).

This method is implemented in **ab4.m** and a representative ivode is solved below –

```
>> dy_dt = @(t,y) -2*t*y.^2;
>> tspan = [0; 1.0]; y0 = 1.0;
>> exact = @(t) 1./(1 + t.^2);
>> [t, y] = ab4(dy_dt, tspan, y0, 100);
>> subplot(1,2,1); plot(t, y); xlabel('t'); ylabel('y(t)');
>> text(0.25, 0.9,'y''=  -2ty^2,    y(0) = 1.0');
>> subplot(1,2,2); plot(t,y-exact(t));xlabel('t'); ylabel('y(t)-exact(t)');
```





10.3.7d) Implicit Fourth-Order (N = 4) Adams-Moulton Method

Again one employs a four-point interpolation using (t_2, f_2), (t_3, f_3), and (t_4, f_4), where y and f has been determined, as well as the unknown value f_5 at t_5 to extrapolate $f_{4.5}$ -

```
>> format rat;
>> td = [2 3 4 5]; fd = [1 0 0 0]; lagrange_interpolation(td, fd, 4.5)
ans =   +1/16
>> td = [2 3 4 5]; fd = [0 1 0 0]; lagrange_interpolation(td, fd, 4.5)
ans =   -5/16
>> td = [2 3 4 5]; fd = [0 0 1 0]; lagrange_interpolation(td, fd, 4.5)
ans =  +15/16
>> td = [2 3 4 5]; fd = [0 0 0 1]; lagrange_interpolation(td, fd, 4.5)
ans =   +5/16
f4.5 = (+f2 -5f3 +15f4 +5f5)/16
```

Now this extrapolated value $f_{4.5}$ is used to integrate f(t, y(t)) from t_4 to t_5 using the Simpson quadrature rule in terms of the unknown f_5 -

```
y5 = y4 + (1/6)(t5 - t4)(f4 + 4f4.5 + f5)
y5 = y4 + (1/6)(t5 - t4)(f4 + 4f4.5 + f5) = (h/6)(f4 + 4f4.5 + f5)
y5 = (h/6)(f4 + 4(+f2 -5f3 +15f4 +5f5)/16  + f5)
y5 = (h/24)(4f4 + (+f2 -5f3 +15f4 + 5f5) +4f5)
y5 = (h/24)(+f2 -5f3 +19f4 +9f5)
```

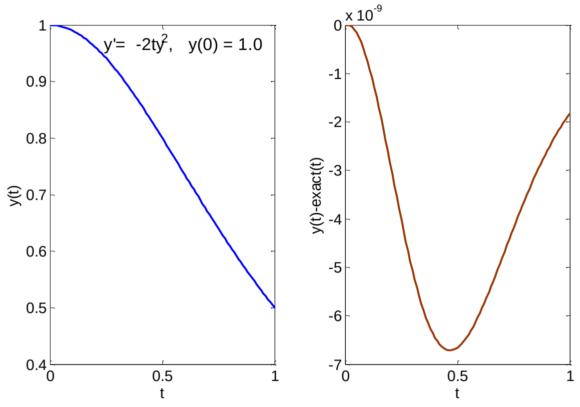
Thus the implicit fourth-order Adams-Moulton method is

$$y_{k+1} = y_k + \frac{h}{24}\left(f_{k-2} - 5f_{k-1} + 19f_k + 9f_{k+1}\right)$$

with a truncation error of -(19/720)h^5 ([p. 411] Heath 2002). Employing y' = -λy and y(0) = + 1, it can be shown that this representative IVODE possesses a stability range, $-3.0 \le h\lambda \le 0.0$ for real λ ([p. 411] Heath 2002).

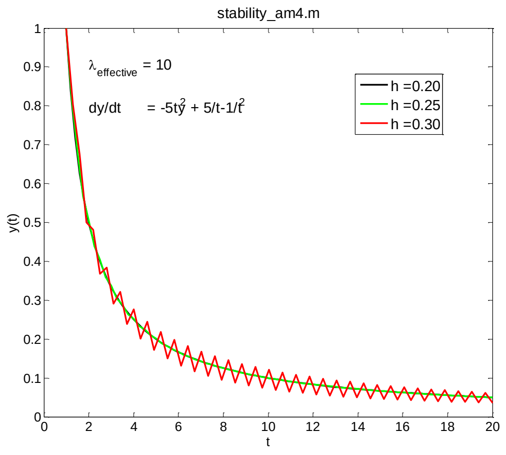
Adams-Moulton methods are more accurate as well as more stable than Adams-Bashforth methods of the same order ([p. 58] Shampine, Gladwell, & Thompson 2003). However, such improvements come at the added cost of root-finding.

```
>> dy_dt = @(t,y) -2*t*y.^2; tspan =[0; 1.0]; y0=1.0; exact =@(t) 1./(1 + t.^2);
>> [t, y] = am4(dy_dt, tspan, y0, 100);
>> subplot(1,2,1); plot(t, y); xlabel('t'); ylabel('y(t)');
>> text(0.25, 0.9,'y''=  -2ty^2,   y(0) = 1.0');
>> subplot(1,2,2); plot(t,y-exact(t));xlabel('t'); ylabel('y(t)-exact(t)');
```



Observe the factor of greater than ten difference in execution times between fourth order AB and AM methods -

```
>> tic; [t, y] = am4(dy_dt, tspan, y0, 250); toc
Elapsed time is 0.138934 seconds.
>> tic; [t, y] = ab4(dy_dt, tspan, y0, 250); toc
Elapsed time is 0.008883 seconds.
```



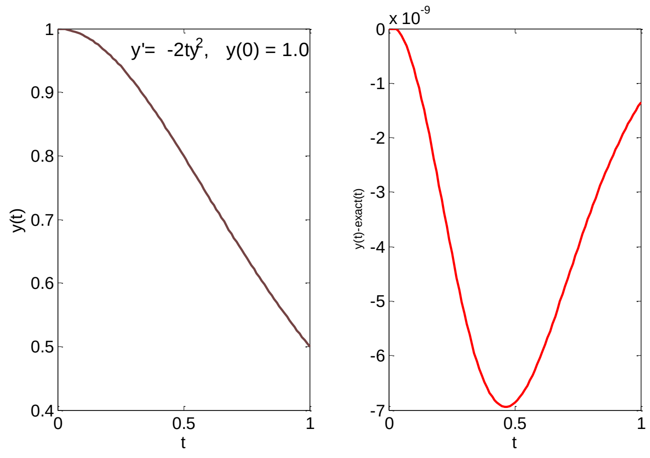
10.3.7e) Predictor-Corrector Methods

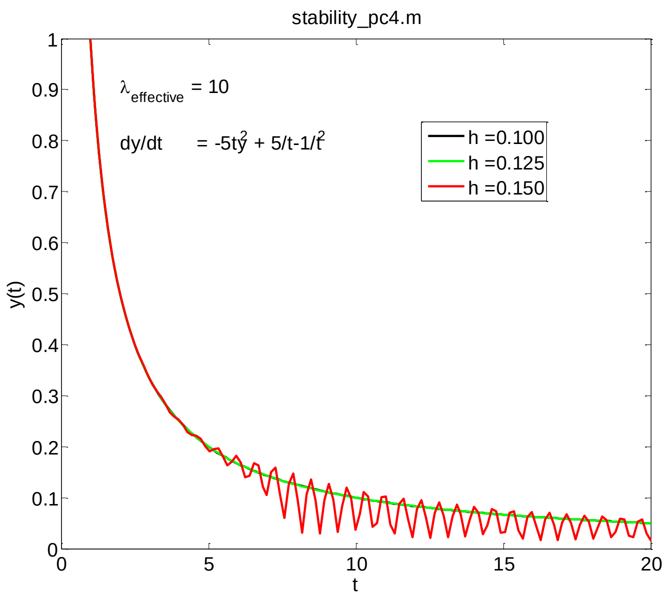
"A disappointing feature of multistep processes is the deterioration of the stability properties with increasing order" ([p. 179] Dormand 1996).

Rather than employ root finding techniques to solve implicit methods, such as Adams-Moulton algorithms, explicit methods, such as Adams-Bashforth algorithm, is used to *predict* approximately y_{i+1} and then these values are corrected by an implicit method, such as the Adams-Moulton. A popular pair of algorithms is fourth-order Adams-Bashforth method as the predictor, and fourth-order implicit Adams-Moulton method as the corrector. with a truncation error of $-(19/720)h^5$ ([p. 411] Heath 2002). Employing $y' = -\lambda y$ and $y(0) = +1$, it can be shown that this representative IVODE possesses a stability range, $-1.5 \leq h\lambda \leq 0.0$ for real λ ([p. 261], Hairer & Wanner 1991).

This method is implemented in **pc4.m** and a representative ivode is solved below -

```
>> dy_dt = @(t,y) -2*t*y.^2;
>> tspan = [0; 1.0]; y0 = 1.0;
>> exact = @(t) 1./(1 + t.^2);
>> [t, y] = pc4(dy_dt, tspan, y0, 100);
>> subplot(1,2,1); plot(t, y); xlabel('t'); ylabel('y(t)');
>> text(0.25, 0.9,'y''=  -2ty^2,   y(0) = 1.0');
>> subplot(1,2,2); plot(t,y-exact(t));xlabel('t'); ylabel('y(t)-exact(t)');
```





10.4 MATHWORK'S IVODE SOLVERS

MATLAB possesses a variety of *sophisticated* intrinsic functions for solving systems of *first order* ivodes. These solvers determine a unique solution, y(t) (y(t) is usually a vector) to the following system of ODEs

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

$$y(t_o) = y_o.$$

where t is a scalar, and y(t) is the unknown solution *vector*, and the user supplied function f(t, y), which itself is a function of t and y(t).

10. 4.1 Non-Stiff IVODE Solvers

Most relevant to this course!

`ode45` - explicit Runge-Kutta (4, 5) formula. Dormand-Prince pair, very similar to RKF formula. It is a one-step solver. In general it is the best function to employ as a first try.

`ode23` - explicit Runge-Kutta (2, 3) formula. It is a one-step solver. It may be more efficient at low tolerances and in the presence of mild stiffness.

`ode113` - variable order Adam-Bashforth-Moulton predictor-corrector method. It may be more efficient than ode45 at stringent tolerances. It is a multi-step solver.

`ode15i` - Solve fully implicit differential equations, variable order method

`dde23` – Solve delay differential equations (DDEs) with constant delays

`ddesd` - Solve delay differential equations (DDEs) with general delays.

4.2 Stiff IVODE Solvers

`ode15s` - variable-order solver based on numerical differentiation formulae. Optionally, it can use the backward difference formulae, BDFs (Gear's method). It is a multi-step solver. If you suspect a problem is stiff, or if `ode45` fails or is very inefficient, try ode15s.

`ode23s` - based on modified Rosenbrock formula of order 2. It is a one-step solver. It may be more efficient at crude tolerances.

`ode23t` - an implementation of the trapezoidal rule using a free interpolant. Use this solver if the problem is only moderately stiff and you need a solution without numerical damping.

`ode23tb` - an implementation of TR-BDF2, an implicit Runge-Kutta formula with a first stage that is a trapezoidal rule step and a second stage that is a backward differentiation formula. of order 2.

10.4.3 Properties of MATHWORK's IVODE Solvers:

Solver	Type	Accuracy	When to use
ode45	nonstiff	medium	Should be 1st try
ode23	nonstiff	low	Using low tolerance
ode113	nonstiff	low/high	Stringent tolerance
ode15s	stiff	low/medium	If ode45 is slow
ode23s	stiff	low	Crude error tolerance.
ode23t	stiff	low	
ode23tb	stiff	low	

10.4.3a) ODE Solver Syntax:

```
>> [t, y] = odeXXXX(@ode_fn, tspan, y0);   OR
>> sol    = odeXXXX(@ode_fn, tspan, y0);
```

NOTE: use `@odeXXX`, if you don't use an anonymous for `ode_fn`. If you an m-file for the ode, then use `@odeXXX` or `'odeXXX'` (string) .

Input arguments:

`ode_fn` -- function which evaluates ODE system of *1st order* ODEs.

```
dy_dt   = ode_fn(t, y)
```

where t is a scalar and dy_dt & y are column vectors.

`tspan` -- Vector specifying the interval of integration. The solver assumes that the independent starts/ends at tspan(1) & tspan(end).

For tspan vectors with *two* elements [t0 tf], the solver returns the solution evaluated at every integration step. For tspan vectors with more than two elements, the solver returns solutions evaluated at the given time points. The time values must be in order, either increasing or decreasing.

Specifying tspan with more than two elements does not affect the internal time steps that the solver uses to traverse the interval from tspan(1) to tspan(end). Specifying tspan with more than two elements has little effect on the efficiency of computation.

`y0` column vector of initial values of y.

Output Arguments:

`t` -column vector of time points

`y` -solution array. Each row in y corresponds to the solution at a time returned in the corresponding row of t.

OR (alternatively)

`sol.x` steps chosen by the ivode solver.

`sol.y` each column, sol.y(:,i) . contains solution corresponding to x(i).

`sol.name` solver name.

10.4.3b) Additional ODE Solver Arguments:

For more advanced applications, you can also specify as input arguments solver options and additional problem parameters.

options structure of optional parameters that change the default integration properties. This parameter must be *fourth* input argument

```
>> sol2 = odeXXX(@ode_fn, tspan, y0, options);
```

p1,p2... .-- parameters that the solver passes to ode_fn. You must insert *after* **`options`**:

```
>> [t,y] =odeXXX(ode_fn,tspan,y0,options,p1,p2);
```

You create options structure with **`odeset`** function -

```
>>options=odeset('name1',value1,'name2',value2, …);
```

To determine what actually are the optional variables & their default values, type

```
>> odeset
```

To query options, use **`odeget`** function -

```
>> odeget(options, 'name1');
```

At each step, the solver estimates the local error e in the ith component of the solution. This error must be less than or equal to the acceptable error, which is a function of the specified relative tolerance, RelTol, and the specified absolute tolerance, AbsTol.

```
|e(i)|< max(RelTol*abs(y(i)),AbsTol(i))
```

Useful option variables:

Default value for '**`RelTol`**' is too large for many coputations!

Name:	Default Value	
`'RelTol'`	1.0e-03	Applies to all components.
`'Abstol'`	1.0e-06	
`'NormControl'`	'off'	Control error relative to norm.
`'Stats'`	'off'	Displays statistics of computation.

OTHERS: 'Vectorize' 'OutputFcn'   'Refine'   'Jacobian'   'OutputSel'

10.4.3c) Solving IVODES

To solve ivodes *computationally* via MATLAB intrinsics:

a) Initial/final values must be supplied.
b) A finite time span must be supplied.
c) High-order ODEs must be translated into systems of 1st order ODEs.

To solve an ODE or system of ODEs, you *MUST* -

a) Create the system of the relevant ODE equations
b) Supply a full complement of the appropriate initial/final values
c) Supply time span over which you will calculate the solutions.

Minimum information required to solve a scalar IVODE:

a) anonymous for f(t, y(t)).   [You can also use m-files]
b) range of t
c) initial condition, y_o.
d) name of MATLAB ODE solver (**`ode45, ode15s, ode113`**, etc)

Say that we want to solve the following scalar first-order ODE

$$y' = \frac{2}{t} y + t^2 e^t, \quad 1 \le t \le 2, \quad y(1) = 0,$$

9) Pick MATLAB solver, say, **`ode45`**

b) Create a MATLAB function f(t, y)-

```
dy_dt = @(t, y) (2./t).*y + t.^2*exp(t);
```

c) Pick duration:

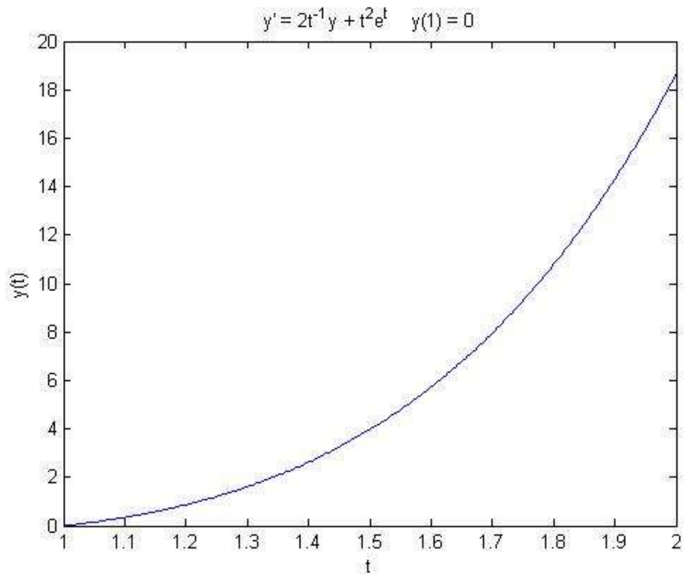
```
tspan = [1 2]
```

d) Pick initial condition:

y0 = 0,

```
>> sol = ode45(dy_dt, [1, 2], 0);
    >> [sol.x' sol.y']
    ans =
    1              0
    1.0001   0.00020099
    1.0004     0.001207
    1.0023    0.0062652
    1.0115     0.032261
    1.0577      0.18076
    1.1577      0.62251
    1.2577       1.2642
    1.3577       2.1552
    1.4577       3.3531
    1.5577       4.9253
    1.6577       6.9503
    1.7577       9.5192
    1.8577       12.738
    1.9577        16.73
    2            18.683
>> plot(x, deval(sol, x));

>> xlabel('t'); ylabel('y(t)');
>> title('y'' = 2t^{-1}y + t^2e^t    y(1) = 0')
```



10.4.3d) High-Order IVODES & Systems of IVODEs

Systems of ODEs: Follow the same approach as above *BUT USING VECTORS*

Example: 1st – Order, Coupled, Nonlinear ODEs,

$$dx/dt = +x + y - x(x^2 + y^2)$$

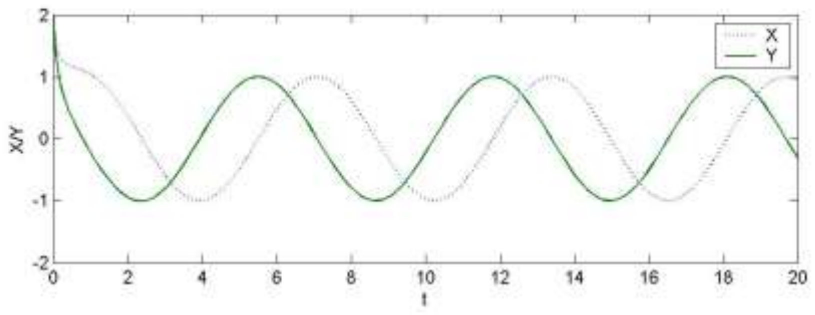
$$dy/dt = -x + y - y(x^2 + y^2)$$

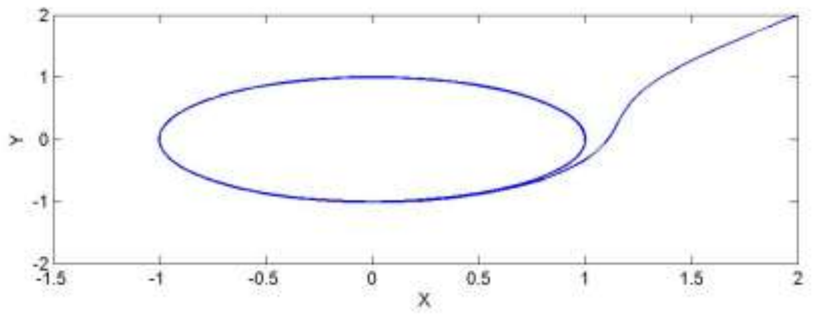
$$x(0) = 2 \quad \& \quad y(0) = 2, \quad 0 \le t \le 20$$

```
% u(1) = x  and u(2) =  y
>> du_dt = @(t, u) [ (+u(1) + u(2) - u(1).*(u(1).^2 + u(2).^2));
                     (-u(1) + u(2) - u(2).*(u(1).^2 + u(2).^2))];
>> [t, u] = ode45(du_dt, [0, 20], [2;2]);
```

NOTE: solution matrix u contains BOTH x & y as columns

```
>> subplot(2,1,1), plot(t, u(:, 1), ':', t, u(:, 2))
>> legend('X', 'Y');
>> xlabel('t'); ylabel('X/Y');
>> subplot(2,1,2), plot(u(:, 1), u(:, 2))
>> xlabel('X'); ylabel('Y')
```





Higher-Order IVODEs $\Rightarrow$ Create a system of coupled first-order ODEs

For example: $$\frac{d^2\theta}{dt^2} + \sin\theta = 0, \; [\text{single second order ODE}]$$

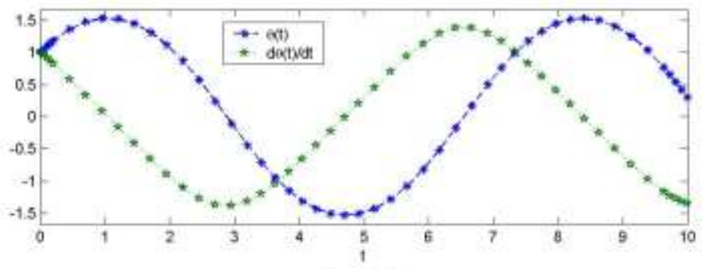
$$\text{Define:}\quad y_1(t) = \theta(t),\ \text{and}\ y_2(t) = \frac{d\theta}{dt}$$

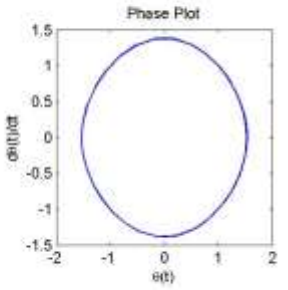
$$\frac{d}{dt}y_1(t) = y_2(t), \left[\text{two first order ODEs}\right]$$

$$\frac{d}{dt}y_2(t) = -\sin(y_1(t))$$

```
>> pend = @(t, y) [y(2); -sin(y(1))];
>> tspan = [0 10];
>> y0    = [1; 1];
>> [t, y] = ode45(@pend, tspan, y0);

>> subplot(2, 1, 1),plot(t, y(:,1),'*--', t, y(:,2),'p:');
>> xlabel('t'); axis equal
>> legend('\theta(t)', 'd\theta(t)/dt');
>> subplot(2, 1, 2),plot(y(:,1),y(:,2));
>> axis square; title('Phase Plot');
>> xlabel('\theta(t)');ylabel('d\theta(t)/dt');
```





Another example -

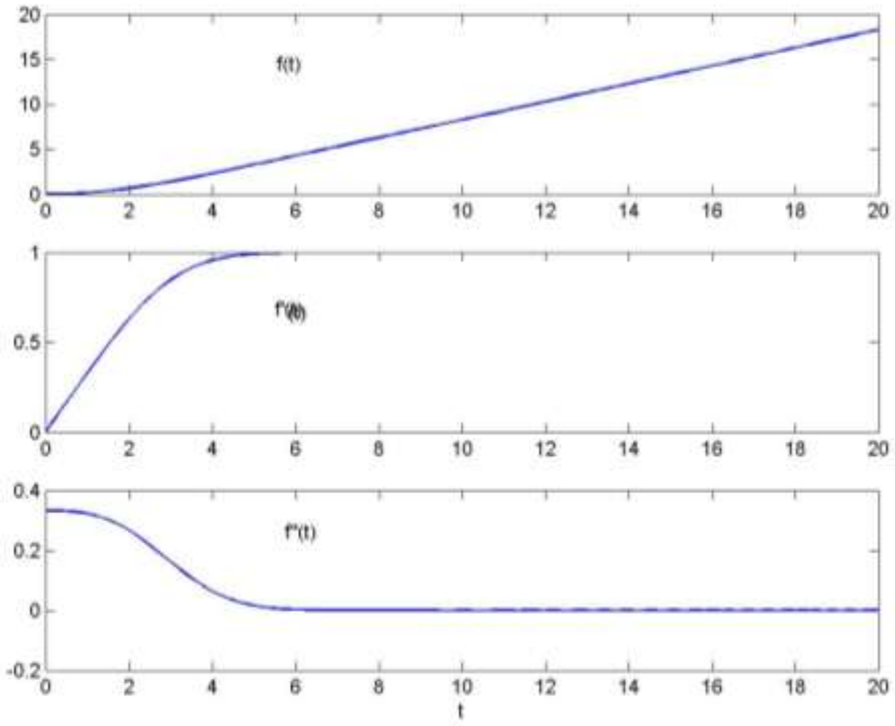
$$
\begin{aligned}
f''' &= -ff''/2,\\
f(0) &= 0,\\
f'(0) &= 0,\\
f''(0) &= 0.332,\\
0 \le t &\le 20
\end{aligned}
$$

```
function df_dt = blasius(t, u) % stored in an m-file

  %f     = [f, f', f''];
  %df_dt = [f', f'', f''']= [f', f'', -f*f''/2]

   f     = u(1);
   fp    = u(2);
   fpp   = u(3);
   df_dt = [fp; fpp; -0.5.*f.*fpp];
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

>> [t, u] = ode45(@blasius, [0, 20], [0;0; 0.332]);
```



10.4.3e) Event Identifications

`cannon_shell_events.m` models the trajectory of a *low-speed* cannon ball subject to quadratic drag forces. The *low-speed* modifier means we can neglect the curvature of the Earth and thus investigate motion in context of a *flat Earth* → a local Cartesian coordinate system, where y is the vertical direction and x is horizontal direction.

It would be useful if the ivode integration stops when a cannon ball hits the ground. This can be done through the use of an *events* function. To use the events function (here an m-file) one must employ the `options` structure -

Components associated with event identification are in red

```
options              = odeset('AbsTol',1.0e-09,'RelTol',1.0e-06,'Events',@ground);
[t  s, te, se, ie] = ode45(@projectile, [0, tmax], initial, options);

te & se are the time and solution, corresponding to an event occurrence
```

You must add an m-file - here called `ground.m`

- Input arguments -- t and y -
- *Output* parameters --
- value
- isterminal
- direction

Action is taken when value == 0.0.

```
 function [value, isterminal, direction] = ground(t, y)
       value       = y(2);
       isterminal  =  [1];
       direction   = [-1];
```

- `value(i)` is the value of the `i`th event function.
- `Here value = y, the height of the cannon ball.`
- Set `isterminal(i) = 1` if the integration is to terminate at a zero of this event function, otherwise, `0,` and continue the integration.
- `Here isterminal = +1.  Yes, crossing the x-axis terminates execution.`
- Set `direction(i) = 0` if *all* zeros are to be located (the default), `+1` if want value = 0 the value is increasing, and `-1` if you want value = 0, when value is decreasing, .
- Here you want integration to stop when *falling* object crosses x axis, thus set direction to (-1).

Thus, integration stops *only*, when y = = 0 *and* direction is downward!

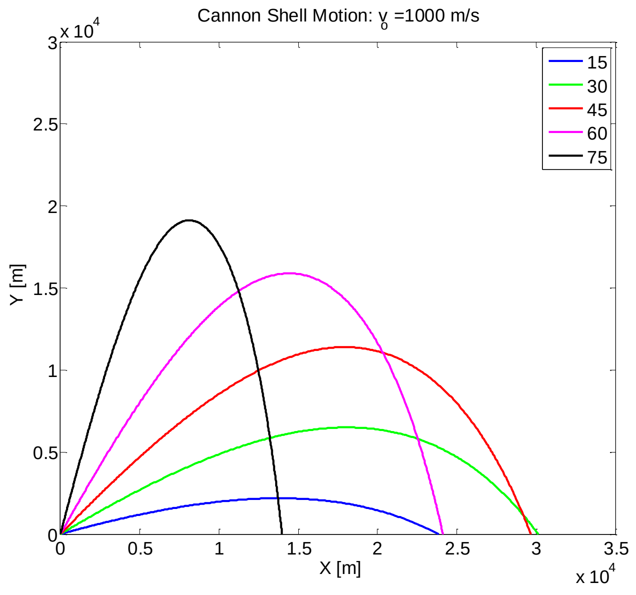
```
  function [ ] = cannon_shell_events
% Initial Value Ordinary Differential Equation
% Supply initial speed, v0, and maximum flight time

   v0      = 1000.0;
   tmax    = 200.0;
   angles  = [15 30 45 60 75];    % in degrees
   color   = [ 'b'; 'g'; 'r'; 'm'; 'k'];
   radian  = 180/pi;
   angles  = angles./radian;

   hold on;
     for k = 1:length(angles)
       initial = [0; 0; v0*cos(angles(1,k)); v0*sin(angles(1,k))];
       options = odeset('RelTol', 1.0e-10, 'AbsTol', 1.0e-10,'Events', @ground);
       [t  s, te, se, ie]  = ode45(@projectile, [0, tmax], initial, options);
       fprintf('time of impact = %5.2f at x = %7.2f and y = %7.2e\n', te, se(1:2));
```

```
        x         = s(:,1); y         = s(:,2);  % s       = [x;y;ux;uy]
        plot(x, y, color(k), 'LineWidth', 2);
        ylim([0.0, 30000.0])
      end
    xlabel(' X [m]');ylabel(' Y [m]');
    title(strcat('Cannon Shell Motion: v_o = ', num2str(v0),' m/s' ));
    legend(num2str((180.0*angles/pi)'));
    hold off;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function du_dt = projectile(t, u)
 %  u     = [x  y  ux     uy]'
 % du_dt = [ux uy dux/dt duy/dt]'
 x = u(1); y = u(2); vx = u(3); vy = u(4);
 % Quadratic velocity dependent drag
 g            = -9.8;
 B2           = 4.0e-05;
 v            = sqrt(vx^2 + vy^2);
 drag         =  B2*v*v;
 % atmospheric density fall off with height
 drag         = drag*exp(-y/1.0e04);
 du_dt(1,1)   = vx;
 du_dt(2,1)   = vy;
 v            = sqrt(vx*vx + vy*vy);
 drag         =  B2*v*v;
 du_dt(3,1)   =   -(vx/v)*drag;
 du_dt(4,1)   = g -(vy/v)*drag;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 function [value, isterminal, direction] = ground(t, y)
        value      = y(2);
        isterminal = [1];
        direction  = [-1];
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
>> cannon_shell_events
time of impact = 41.74  at x = 23865.95 and y = -2.34e-013
time of impact = 71.68  at x = 30133.08 and y = -2.69e-010
time of impact = 95.26  at x = 29675.17 and y = -3.35e-010
time of impact = 113.38 at x = 24124.12 and y = -2.36e-012
time of impact = 125.27 at x = 13975.27 and y = -3.96e-010
```



10.5 STIFF ODES

Stiff ODEs are ODEs that possess very disparate time scales. Many algorithms such as RK4 and RKF, are inefficient at solving such equations since the short-lived components require very small step sizes to maintain stability ([p. 401] Heath 2002).

An example of a stiff ODE is the following (Dormand 1996):

$$\frac{dy_1}{dx} = (994y_1 \quad -1998y_2)/5, \quad y_1(0) = +1,$$

$$\frac{dy_2}{dx} = (2997y_1 -5999y_2)/5, \quad y_2(0) = -2.$$

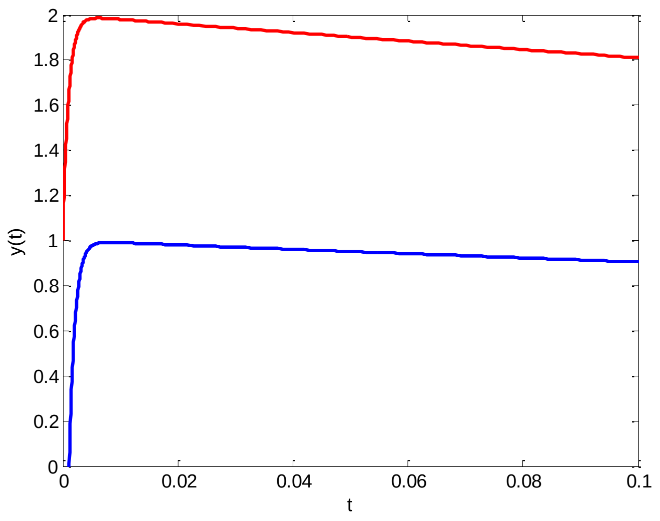
The exact solution is

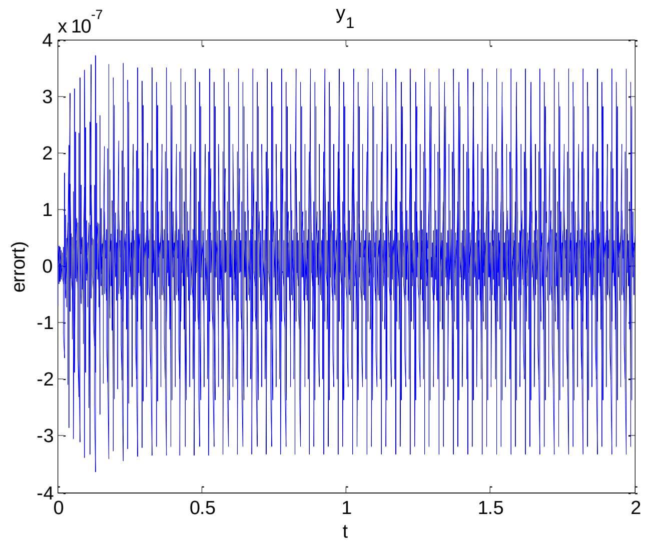
$$y_1(x) = 2e^{-x} - e^{-1000x},$$

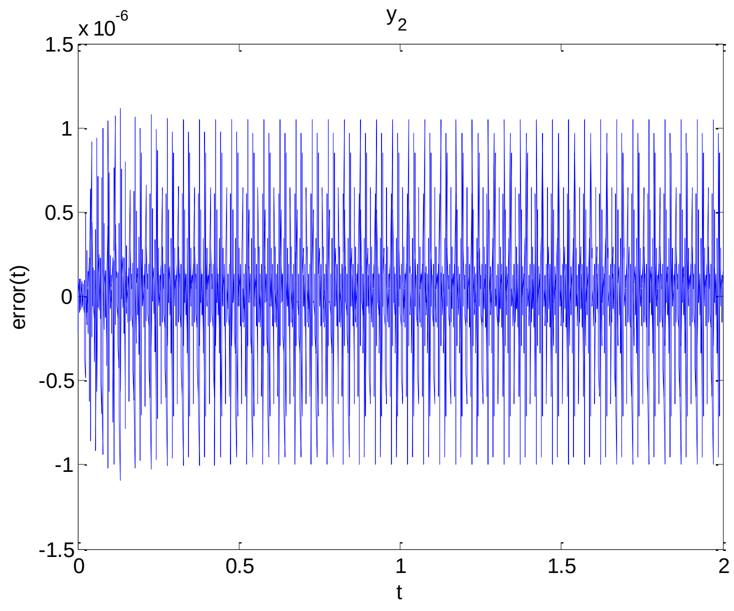
$$y_2(x) = e^{-x} - 3e^{-1000x}.$$

```
>> options = odeset('AbsTol', 1.0e-06, 'RelTol', 1.0e-09, 'Stats', 'on');
>> ex1 =@(x) 2*exp(-x) - exp(-1000*x);
>> ex2 =@(x) exp(-x) - 3* exp(-1000*x);
>> f = @(t, y) [(994*y(1) -1998*y(2))/5; (2997*y(1) -5999*y(2))/5];
>> [t, u] =ode45(f, [0, 2], [1, -2], options);
632 successful steps
42 failed attempts
4045 function evaluations
>> length(t)
ans = 2529
>> max(abs(ex1(t) - u(:,1)) )
ans = 3.7247e-007
>> max(abs(ex2(t) - u(:,2)) )
ans = 1.1174e-006

>> plot(t, u(:,1), 'r', t, u(:,2), 'b')
>> axis([0, 0.1, 0, 2])
>> plot(t, ex1(t) - u(:,1))
>> plot(t, ex2(t) - u(:,2))
```







A significant increase in solver efficiency is found when stiff ode solvers are used!

```
>> [t, u] =ode15s(f, [0, 2], [1, -2], options);
131 successful steps
0 failed attempts
243 function evaluations
1 partial derivatives
23 LU decompositions
239 solutions of linear systems
>> length(t)
ans = 132      %versus 2529 with ode45
>> max(abs(ex1(t) - u(:,1)) )
ans = 2.3574e-006
>> max(abs(ex2(t) - u(:,2)) )
ans = 2.5204e-006
```

```
>> [t, u] =ode23s(f, [0, 2], [1, -2], options);
268 successful steps
1 failed attempts
1344 function evaluations
268 partial derivatives
269 LU decompositions
807 solutions of linear systems
>> length(t)
ans = 269
>> max(abs(ex1(t) - u(:,1)) )
ans = 1.5707e-005
>> max(abs(ex2(t) - u(:,2)) )
ans = 1.8161e-005
```

10.6 REFERENCES

Ascher, U. M., & Petzold, L. R. 1998, *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*, (SIAM:Philadelphia, PA).

Bradie, B. 2006, *A Friendly Introduction to Numerical Analysis*, (Prentice-Hall: Englewood Cliff, NJ).

Borelli, R. L., & Coleman, C. S. 1987, *Differential Equations: A Modeling Approach*, (Prentice-Hall: Englewood Cliff, NJ).

Burden, R. L., & Faires, J. D. 2001, *Numerical Analysis, 7th Ed.*, (Brook-Cole: Pacific Grove, CA).

Dormand, J. R. 1996, *Numerical Methods for Differential Equations: A Computational Approach*, (CRC Press: Boca Raton, FL).

Dormand, J. R, & Prince, P. J. 1980, *J. Comput. Appl. Math.*, **2**7, 19-26.

Dubin, D 2003, *Numerical and Analytic Methods for Scientists and Engineers Using Mathematica* (John Wiley & Sons: NY, NY).

Fehlberg, E., 1970, *Computing*, **6**: 61-71.

Heath M T, 2002, *Scientific Computing, An Introductory Survey*, 2nd Ed. (McGraw-Hill: NY, NY).

Hairer, E., Norsett, S. P., & Wanner, G. 1987, *Solving Ordinary Differential Equations I: Nonstiff Problems* (Springer-Verlag: Berlin).

Hairer, E., & Wanner, G. 1991, *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems* (Springer-Verlag: Berlin).

Kahaner, D., Moler, C., & Nash, S. 1989, *Numerical Methods and Software*, (Prentice-Hall: Englewood Cliff, NJ).

Mathews, J. H. & Fink, K. D. 1999, *Numerical Methods Using MATLAB, 3rd Ed,* (Prentice Hall: Upper Saddle River, NJ).

Moin, P. 2010, *Fundamentals of Engineering Numerical Analysis, 2nd Ed.* (Cambridge University Press: Cambridge UK).

Rice, J. R. 1983, *Numerical Methods, Software and Analysis: IMSL Reference Edition*. (McGraw-Hill: NY, NY).

Shampine, L F, Gladwell, I, & Thompson, S, 2003, *Solving ODEs with MATLAB*, (Cambridge University Press: Cambridge UK).

10.7 QUESTIONS

10.1. True or false: An ODE solution that is unbounded as time increases is necessarily unstable.

10.2. True or false: In approximating a solution of an ODE numerically, the global error grows only if the solution sought is unstable.

10.3. True or false: In solving an ODE numerically, the roundoff error and the truncation error are independent of each other.

10.4. True or false: In solving an IVP for an ODE numerically, the global error is always at least as large as the sum of the local errors.

10.5. True or false: For approximating a stable solution of an ODE numerically, an implicit method is always stable.

10.6. True or false: In numerically approximating a stable solution of an ODE, one can take arbitrarily large time steps using an unconditionally stable method and still achieve any required accuracy.

10.7. True or false: Stiff ODEs are always difficult and expensive to solve.

10.8. (a) In general, does a differential equation, by itself, determine a unique solution?
(b) If so, why, and if not, what additional information must be specified to determine a solution uniquely?

10.9. (a) What is meant by a *first-order* ODE?*(*b) Why are higher-order ODEs usually transformed into equivalent first-order ODEs before solving them numerically?

10.10. (a) Describe in words the distinction between a stable and an unstable solution of an ODE. (b) What is a mathematical criterion for determining the stability of an of ODE $y' = f(t,y)$? ***(c)*** Can the stability or instability of an ODE solution change with time?

10.11. Which of the following types of first-order ODEs have stable solution. *(a)* An ODE whose solutions converge toward each other *(b)* An ODE whose Jacobian matrix has only eigenvalues with negative real parts. *(c)* A stiff ODE.*(d)* An ODE with exponentially decaying solutions

10.12. Classify each of the following ODEs as having unstable, stable, or asymptotically stable solutions.
(a) $y' = y + t$.
(b) $y' = y - t$.
(c) $y' = t - y$.
(d) $y' = l$.

10.13. How does a typical numerical solution of an ODE differ from an analytical solution?

10.14.(a) What is Euler's method for solving an ODE*? (b)* Show at least one way it can be derived.

10.15. In solving an ODE numerically, which is usually more significant, rounding error or truncation error?

10.16. Describe in words the difference between the local error and the global error in solving an IVP for an ODE numerically.

10.17. Under what condition is the global error in solving an IVP for an ODE likely to be smaller than the sum of the local errors at each step?

10.18. (a) Define in words the error amplification factor for one step of a numerical method for solving an IVP for an ODE. (b) Does the amplification factor depend only on 'the equation, only on the method of solution, or on both? (c) What is the value of the amplification factor for one step of Euler's method?
(d) What stability region does this imply for Euler's method?

10.19. (a) What is the basic difference between an explicit method and an implicit method for solving an ODE numerically? *(b)* Comparing these two types of methods, list one relative advantage for each. (c) Name a specific example of a method (or family of methods) of each type.

10.20. The use of an implicit method for solving a nonlinear ODE requires the iterative solution of a nonlinear equation. How can one get a good starting guess for this iteration?

10.21. Is it possible for a numerical solution method to be unstable when applied to a stable ODE?

10.22. What does it mean for the accuracy of a numerical method for solving ODEs to be of order *p?*

10.23. (a) For solving ODEs, what is the highest order accuracy that a linear multistep method can have and still be unconditionally stable? (b) Give an example of a method having these properties (by name or by formula).

10.24. Compare the stability regions (i.e., the stability constraints on the step size) for the Euler and backward Euler methods for solving a scalar ODE.

10.25. For the backward Euler method, which factor places a stronger restriction on the choice of step size: stability or accuracy?

10.26. Which of the following numerical methods for solving a stable ODE numerically are unconditionally stable?
(a) Euler's method
(b) Backward Euler method
(c) Trapezoid rule

10.27. (a) What is meant by a *stiff* ODE? (b) Why may a stiff ODE be difficult to solve numerically? (c) What type of method is appropriate for solving stiff ODEs?

10.28. Suppose one is using the backward Euler method to solve a nonlinear ODE numerically. The resulting nonlinear algebraic equation at each step must be solved iteratively. If a fixed number of iterations are performed at each step, is the resulting method unconditionally stable? Why?

10.29. Explain why implicit methods are better than explicit methods for solving stiff ODEs numerically.

10.30. What is the simplest numerical method that is stable for solving a stiff ODE?

10.31. For solving ODEs numerically, why is it usually impractical to generate methods of very high accuracy by using many terms in a Taylor series expansion?

10.32. In solving an ODE numerically, with which type of method, Runge-Kutta or multistep, is it easier to supply values for the numerical solution at arbitrary output points within each step?

10.33. *(a)* What is the basic difference between a single-step method and a multistep method for solving an ODE numerically?*(b)* Comparing these two types of methods, list one relative advantage for each. (c) Name a specific example of a method (or family of methods) of each type.

10.34. List two advantages and two disadvantages of multistep methods compared with classical Runge-Kutta methods for solving ODEs numerically.

10.35. What is the principal drawback of a Taylor series methods compared with Runge-Kutta methods for solving ODEs numerically?

10.36. (a) What is the principal advantage of extrapolation methods for solving ODEs numerically? (b) What are the disadvantages of such methods?

10.37. In using a multistep method to solve an ODE numerically, why might one still need to have a single-step method available?

10.38. Why are multistep methods for solving ODEs numerically often used in predictor-corrector pairs?

10.39. If a predictor-corrector method for solving an ODE is implemented as a PECE scheme, does the second evaluation affect the value obtained for the solution at the point being computed? If so, what is the effect, and if not, then why is the second evaluation done?

10.40. List two reasons why multivalue methods are easier to implement than multistep methods for solving ODEs adaptively with automatic error control.

10.41. For each of the following properties, state which type of ODE method, multistep or classical Runge-Kutta, more accurately fits the description:

(*a)* Self starting
(*b)* More efficient in attaining high accuracy
(c) Can be efficient for stiff problems
(d) Easier to program
(e) Easier to change step size
(f) Easier to obtain a local error estimate
(g) Easier to produce output at arbitrary intermediate points within each step

10.42. Give two approaches to starting a multistep method initially when past solution history is not yet available.

Exercises

10.1. Write each of the following ODEs as an equivalent first-order system of ODEs:

$$\text{(a) } y'' = t + y + y',$$
$$\text{(b) } y''' = y'' + ty,$$
$$\text{(c) } y''' = y'' - 2y' + y - t + 1.$$

10.2. Write each of the following ODEs as an equivalent first-order system of ODEs:

(a) Van der Pol equation: $y'' = y'(1 - y^2) - y$
(b) Blasius equation: $y''' = -yy''$
(c) Newton's Second Law of Motion for two-body problem:

$$y_1'' = -GMy_1/(y_1^2+y_2^2)^{3/2},$$
$$y_2'' = -GMy_2/(y_1^2+y_2^2)^{3/2},$$

10.3. Are solutions of the following system of ODEs stable?

$$y_1' = -y_1 + y_2,$$
$$y_2' = -2y_2,$$

Explain your answer.

10.4. Consider the ODE $y' = -5y$ with initial condition *$y(0) = 1$.* We will solve this ODE numerically using a step size of $h = 0.5$.
(a) Are solutions to this ODE stable?
(b) Is Euler's method stable for this ODE using this step size?
(c) Compute the numerical value for the approximate solution at t = 0.5 given by Euler's method.
(d) Is the backward Euler method stable for this ODE using this step size?
(e) Compute the numerical value for the approximate solution at t = 0.5 given by the backward Euler method.

10.5. With an initial value of $y_0 = 1$ at $t_0 = 0$ and a time step of $h = 1$, compute the approximate solution value y_1 at time $t_1 = 1$ for the ODE $y' = -y$ using each of the following two numerical methods. (Your answers should be numbers, not formulas.)
(a) Euler's method
(b) Backward Euler method

10.6. For the ODE, initial value, and step size given in Example 10.9, prove that fixed-point iteration for solving the implicit equation for y_1 is in fact convergent. What is the convergence rate?

10.7. Consider the IV P *$y'' = y$* for $t \geq 0$, with initial values *$y(O)$* = 1 and *$y'(O)$* = 2.
(a) Express this second-order ODE as an equivalent system of two first-order ODEs.
(b) What are the corresponding initial conditions for the system of ODEs in part *a?*
(c) Are solutions of this system stable?
(d) Perform one step of Euler's method for this ODE system using a step size of *h* = 0.5.
(e) Is Euler's method stable for this problem using this step size?
(f) Is the backward Euler, method stable for this problem using this step size?

10.8. Consider the IVP for the ODE *$y' = -y^2$* with the initial condition *$y(O)$* = 1. We will use the backward Euler method to compute the approximate value of the solution y_l at time $t_l = 0.1$(i.e., take one step using the backward Euler method with step size *h* = 0.1 starting from *y_o* = 1 at $t_o = 0$). Since the backward Euler method is implicit, and the ODE is nonlinear, we will need to solve a nonlinear algebraic equation for y_1.
(a) Write out that nonlinear algebraic equation for y_l.
(b) Write out the Newton iteration for solving the nonlinear algebraic equation.
(c) Obtain a starting guess for the Newton iteration by using one step of Euler's method for the ODE.
(d) Finally, compute an approximate value for the solution y_l by using one iteration of Newton's method for the nonlinear algebraic equation.

10.9. For each property listed below, state which of the following three ODE methods has or have the given property.

(1) $y_{k+1} = y_k + 0.5h(f(t_k,y_k) + f(t_{k+1},y_k+ hf(t_k,y_k)))$
(2) $y_{k+1} = y_k +0.5h(3f(t_k,y_k) - f(t_{k-1},y_{k-1}))$
(3) $y_{k+1} = y_k +0.5h(f(t_k,y_k) + f(t_{k+1},y_{k+1}))$

(*a)* Second-order accurate
(*b)* Single-step method
(c) Implicit method
(d) Self-starting
(e) Unconditionally stable
(f) Runge-Kutta type method
(g) Good for solving stiff ODEs

10.10. Use the linear ODE *$y' = \lambda y$* to analyze the accuracy and stability of Heun's method (see Section 10.3.6). In particular, verify that this method is second-order accurate, and describe or plot its stability region in the complex plane.

10.11. Applying the midpoint quadrature rule on the interval $[t_k, t_{k+1}]$ leads to the implicit *midpoint method*

$y_{k+1} = y_k + 0.5h_k(f(t_k,y_k) + f(t_{k+1}, 0.5h_k(y_k+ y_{k+1})/2)$

for solving the ODE *$y' = f(t, y)$.* Determine the order of accuracy and the stability region of this method.

10.12. The centered difference approximation

$y' \approx (y_{k+1} - y_{k-1})/(2h)$

leads to the two-step *leapfrog method*

$y_{k+1} = y_k + 2hf(t_k,y_k)$

for solving the ODE *$y' = f(t,y)$.* Determine the order of accuracy and the stability region of this method.

10.13. Let *A* be an *n* x *n* matrix. Compare and contrast the behavior of the linear difference equation

$$x_{k+1} = Ax_k$$

with that of the linear differential equation

$$x' = Ax.$$

What is the general solution in each case? In each case, what property of the matrix *A* would imply that the solution remains bounded for any starting vector *x_o?* You may assume that the matrix *A* is diagonalizable.

10.14. Give a criterion for stability or asymptotic stability of the solutions to the kth order, scalar, homogeneous, constant-coefficient ODE

$$u^{(k)} + c_{k-1} u^{(k-1)} + \ldots + c_1u' + c_0u = 0$$

(Hint: Transform to a first-order system ***$y' = Ay$*** and observe that *A* is a *companion matrix;* see Section 4.2.1.)

Computer Problems

10.1. *(a)* Use a library routine to solve the *Lotka-Volterra model* of predator-prey population dynamics given in Example 10.4, integrating from t = 0 to t = 25. Use the parameter values $\alpha_1 = 1$, $\beta_1 = 0.1$, $\alpha_2 = 0.5$, $\beta_2 = 0.02$, and initial populations $y_1(0) = 100$ and $y_2(0) = 10$. Plot each of the two populations as a function of time, and on a separate graph plot the trajectory of the point $(y_1(t), y_2(t))$ in the plane as a function of time. The latter is sometimes called a "phase portrait." Give a physical interpretation of the behavior you observe. Try other initial populations and observe the results using the same type of graphs. Can you find nonzero initial populations such that either of the populations eventually becomes extinct? Can you find nonzero initial populations that never change? *(Hint:* You can find such a *stationary point* without solving the differential equation.)
(b) Repeat part *a,* but this time use the *Leslie Gower model*

$$y_1' = y_1(\alpha_1 - \beta_1 y_2)$$
$$y_2' = y_2(\alpha_2 - \beta_2 y_2 / y_1)$$

Use the same parameter values except take $\beta_2 = 10$. How does the behavior of the solutions differ between the two models?

10.2. The *Kermack-McKendrick model* for the course of an epidemic in a population is given by the system of ODEs

$$y_1' = -cy_1y_2,$$
$$y_2' = +cy_1y_2 - dy_2,$$
$$y_3' = +dy_2,$$

where y_1 represents susceptibles, y_2 represents infectives in circulation, and y_3 represents infectives removed by isolation, death, or recovery and immunity. The parameters c and d represent the infection rate and removal rate, respectively. Use a library routine to solve this system numerically, with the parameter values c = 1 and $d = 5$, and initial values $y_1(0) = 95$, $y_2(0) = 5$, $y_3(0) = 0$. Integrate from t = 0 to t = 1. Plot each solution component on the same graph as a function of t. As expected with an epidemic, you should see the number of infectives grow at first, then diminish to zero. Experiment with other values for the parameters and initial conditions. Can you find values for which the epidemic does not grow, or for which the entire population is wiped out?

10.3. Experiment with several different library routines having automatic step-size selection to solve the ODE

$$y' = -200ty^2$$

numerically. Consider two different initial conditions, $y(0) = 1$ and $y(-3) = 1/901$, and in each case compute the solution until t = 1. Monitor the step size used by the routines and discuss how and why it changes as the solution progresses. Explain the difference in behavior for the two different initial conditions. Compare the different routines with respect to efficiency for a given accuracy requirement.

10.4. Consider the system of ODEs modeling chemical reaction kinetics given in Example 10.3. *(a)* What is the Jacobian matrix for this ODE system, and what are its eigenvalues? If the rate constants are positive, are the solutions of this system stable? Under what conditions will the system be stiff? *(b)* Solve the ODE system numerically, assuming initial concentrations $y_1(0) = y_2(0) = y_3(0) = 1$. Take $k_1 = 1$ and experiment with values of k_2 of varying magnitude, specifically, $k2 = 10$, 100, and 1000. For each value of k_2, solve the system using a Runge-Kutta method, an Adams method, and a method designed for stiff systems, such as a backward differentiation formula. You may use library routines for this purpose, or you may wish to develop your own routines, perhaps using the classical fourth-order Runge-Kutta method, the fourth-order Adams-Bashforth predictor and Adams Moulton corrector, and the BDF formula given in Sections 10.3.6 and 10.3.8. If you develop your own codes, a fixed step size will suffice for this exercise. If you use library routines, compare the different methods with respect to their efficiency, as measured by function evaluations or execution time, for a given accuracy. If you develop you own codes, compare the different methods with respect to accuracy and stability for a given step size. In each instance, integrate the ODE system from t = 0 until the solution is approximately in steady state, or until the method is clearly unstable or grossly inefficient.

10.5. The following system ODEs models nonlinear chemical reactions

$$y_1' = -\alpha y_1 + \beta y_2 y_3,$$
$$y_2' = \alpha y_1 - \beta y_2 y_3 - \gamma y_2^2,$$
$$y_3' = \gamma y_2^2,$$

where $\alpha = 4 \times 10^{-2}$, $\beta = 10^4$, and $\gamma = 3 \times 10^7$. Starting with initial conditions $y_1(0) = 1$ and $y_2(0) = y_3(0) = 0$, integrate this ODE from t = 0 to t = 3. You may use either a library routine or an ODE solver of your own design. Try both stiff and nonstiff methods, and experiment with various error tolerances. Compare the efficiencies of the stiff and nonstiff methods as a function of the error tolerance.

10.6. The following system of ODEs, formulated by Lorenz, represents a crude model of atmospheric circulation:

$$y_1' = +\sigma(y_2 - y_1),$$
$$y_2' = ry_1 - y_2 - y_1 y_3,$$
$$y_3' = y_1 y_2 - by_3,$$

Taking $\sigma = 10$, $b = 8/3$, $r = 28$, and initial values $y_1(0) = y_3(0) = 0$ and $y_2(0) = 1$, integrate this ODE from t = 0 to t = 100. Plot each of y_1, y_2, and y_3 as a function of t, and also plot each of the trajectories $(y_1(t), y_2(t))$, $(y_1(t), y_3(t))$, and $(y_2(t), y_3(t))$ as a function of t, each on a separate plot. Try perturbing the initial values by a tiny amount and see how much difference this makes in the final value of *y(100).*

10.7. An important problem in classical mechanics is to determine the motion of two bodies under mutual gravitational attraction. Suppose that a body of mass m is orbiting a second body of much larger mass *M,* such as the earth orbiting the sun. From Newton's laws 'of motion and gravitation, the orbital trajectory *(x(t),y(t))* is described by the system of second-order ODEs the two larger bodies appear fixed -

$$x'' = -GMx/r^3,$$
$$y'' = -GMy/r^3,$$

where G is the gravitational constant and $r = (x_2 + y_2)^{1/2}$ is the distance of the orbiting body from the center of mass of the two bodies. For this exercise, we choose units such that G M = 1.

a) Use a library routine to solve this system of ODEs with the initial conditions

$$x(0) = 1 - e, \quad y(0) = 0,$$
$$x'(0) = 0, \quad y'(0) = \left(\frac{1 + e}{1 - e}\right)^{1/2},$$

where e is the eccentricity of the resulting elliptical orbit, which has period2π. Try the values e = 0 (which should give a circular orbit);' e = 0.5, and e = 0.9. For each case, solve the ODE for at least one period and obtain output at enough intermediate points to draw a smooth plot of the orbital trajectory. Make separate plots of *x* versus t, *y* versus t, and *y* versus *x.* Experiment with different error tolerances to see how they affect the cost of the integration and how close the orbit comes to being closed. If you trace the trajectory through several periods, does the orbit tend to wander or remain steady?

(b) Check your numerical solutions in part *a* to see how well they conserve the following quantities, which should remain constant: Conservation of energy:

$$0.5\left((x')^2 + (y')^2\right) - \frac{1}{r}$$

Conservation of angular momentum:

$$xy' - yx'.$$

10.9. A definite integral $\int_a^b f(t)dt$ can be evaluated by solving the equivalent ODE *y'(t)* = *f(t),* *a* ≤ t ≤ *b,* with initial condition *y(a)* = 0. The value of the integral is then simply *y(b).* Use a library ODE solver to evaluate each definite integral in the first several Computer Problems for Chapter 8, and compare its efficiency with that of an adaptive quadrature routine for the same accuracy.

10.10. Homotopy methods for solving systems of nonlinear algebraic equations parameterize the solution space *x(t)* and then follow a trajectory from an initial guess to the final solution. As one example of this approach, for solving a system of nonlinear equations *f(x)* = 0, with initial guess x_o, the following ODE initial value problem is a continuous analogue of Newton's method:

$$x' = -J_f^{-1}(x)f(x), \;\; x(0) = x_o,$$

where *J,* is the Jacobian matrix of *l,* and of course the inverse need not be computed explicitly. Use this method to solve the nonlinear system given in Computer Problem 5.23. Starting from the given initial guess, integrate the resulting system of ODEs from t = 0 until a steady state is reached. Compare the resulting solution with that obtained by a conventional nonlinear system solver. Plot the trajectory of the components of *x(t)* from t = 0 to the final solution. You may also want to try this technique on some of the other.