**Forward Time Integration of**

**First-Order Initial-Value Problems**

**Euler’s Method**

given first-order initial-value differential equations of the form 

must know  at time ti for time step 

determine the slope at the beginning of the interval 



**Second Order Runge-Kutta**

given first-order initial-value differential equations of the form 

must know  at time ti for time step 

ti

y(t)

t

ti+1

kA

kB

determine the slope at the beginning of the interval 

estimate the slope at the end of the interval using Euler’s method 

use the mean of these two slopes 



**Fourth Order Runge-Kutta (RK4)**

ti+½h

ti

y(t)

t

ti+1

k1

k4

k2

k3

determine the slope at the beginning of the interval 

estimate the slope at the midpoint of the interval using Euler’s method 

estimate the slope at the midpoint of the interval using the new value 

estimate the slope at the end of the interval using the newest value 

use a weighted mean of slopes 



# Second-degree (Quadratic) Predictor and Third-degree (Cubic) Corrector

given first-order initial-value differential equations of the form 

must know  respectively at times ti ti-1 and ti-2

may determine 

ti

ti+1

ti-1

ti-2

= f(y,t)

fi-1

fi

fi-2

t

Note that the vertical axis

is  NOT y

for fixed time step h  = ( t - ti ) / h d = dt / h dt = h d

0

+1

-1

-2

fi-1

fi

fi-2



use second-degree polynomial predictor f = a0 + a1  + a2 2 = 

know coefficients a0, a1 and a2 for interpolant f() = a0 + a1  + a2 2 = ()

yi+1 =  + yi = h + yi = h ( a0 + a12 /2 + a23 /3 ) + yi

yi+1 = yi + h ( a0 + a1 /2 + a2 /3 )

**PREDICTOR (3-step Adams-Bashforth)**

****

now use predicted value of yi+1 to compute f(yi+1) = fi+1\*

0

+1

-1

-2

fi-1

fi

fi-2



fi+1\*

use third-degree polynomial corrector f = b0 + b1  + b2 2 + b3 3  = [ 1  2 3 ]





know coefficients b0, b1, b2 and b3 for new interpolant f() = b0 + b1  + b2 2 + b3 3 = ()

yi+1 =  + yi = h + yi = h ( b0 + b12 /2 + b23 /3 + b34 /4 ) + yi

yi+1 = yi + h ( b0 + b1 /2 + b2 /3 + b3 /4)

**CORRECTOR (4-step Adams-Moulton)**

****

**variable time step second-degree polynomial predictor for f f = a0 + a1 (t-ti) + a2 (t-ti)2**



yi+1 =  + yi

# PREDICTOR

yi+1 = yi + a0 (ti+1-ti) + a1 (ti+1-ti)2 /2 + a2 (ti+1-ti)3 /3

use predicted value of yi+1 to compute f(yi+1,t) = fi+1\*

use third-degree polynomial corrector for f f = b0 + b1 (t-ti) + b2 (t-ti)2 + b3 (t-ti)3



# CORRECTOR

yi+1 = yi + b0 (ti+1-ti) + b1 (ti+1-ti)2 /2 + b2 (ti+1-ti)3 /3 + b3 (ti+1-ti)4 /4

## FIXED TIME STEP

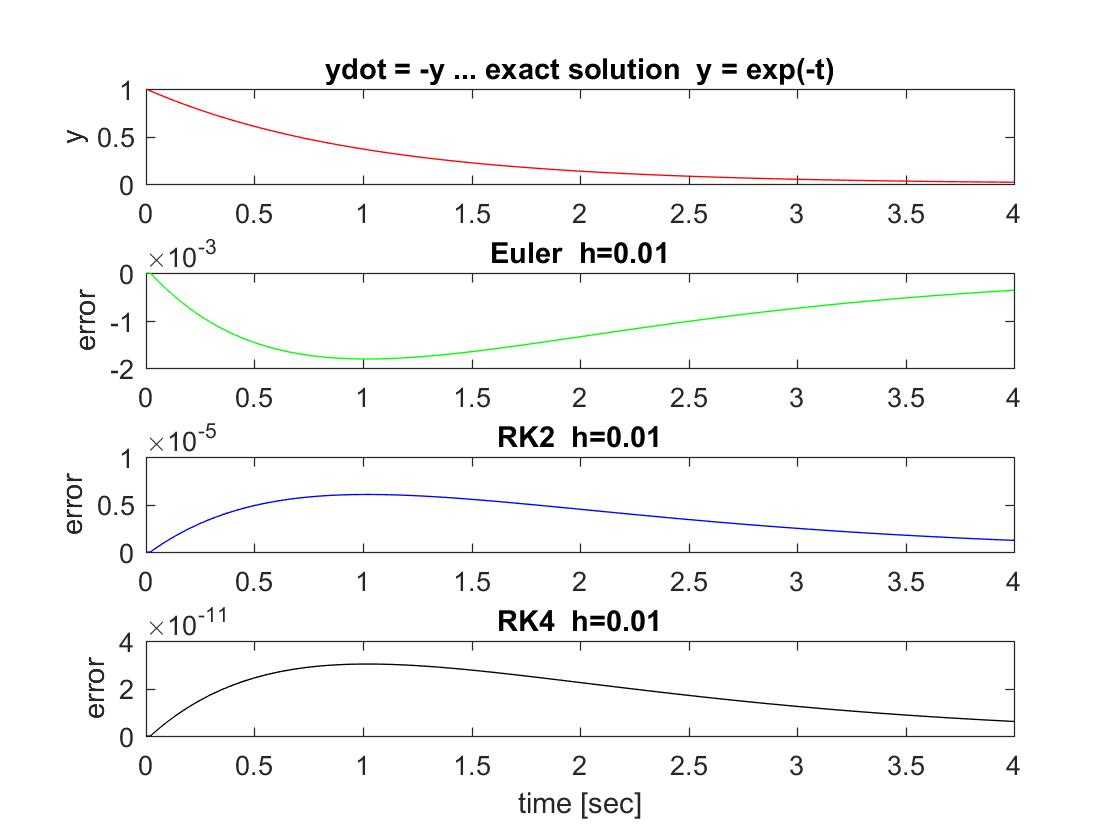
1) Predictor and corrector functions are constant coefficient, weighted sums of fi.

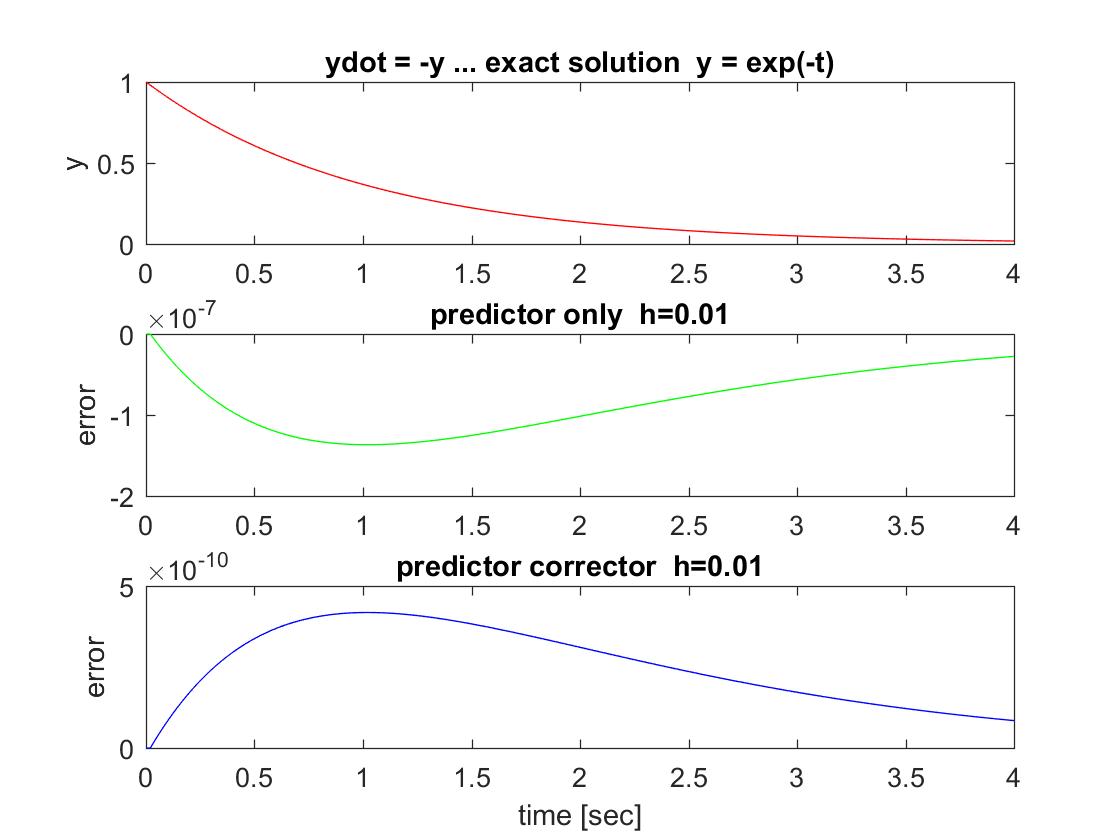
2) Coefficients are the same for all functions .

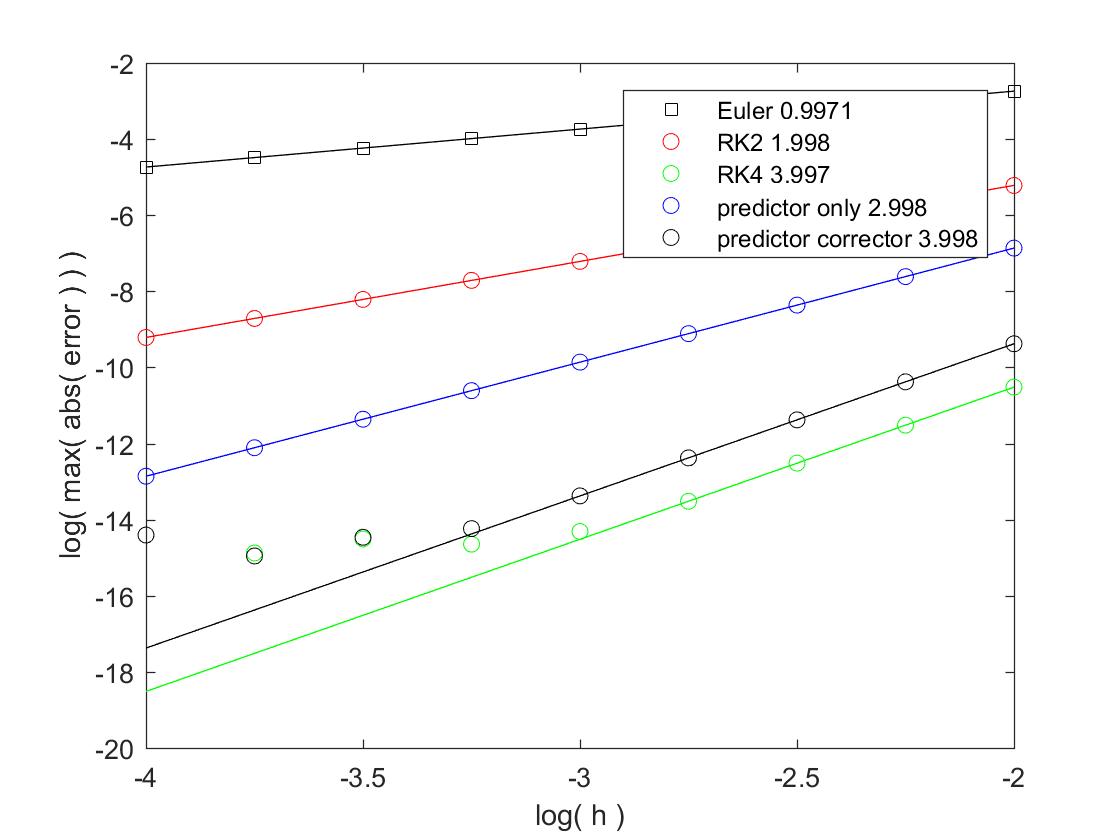
## VARIABLE TIME STEP

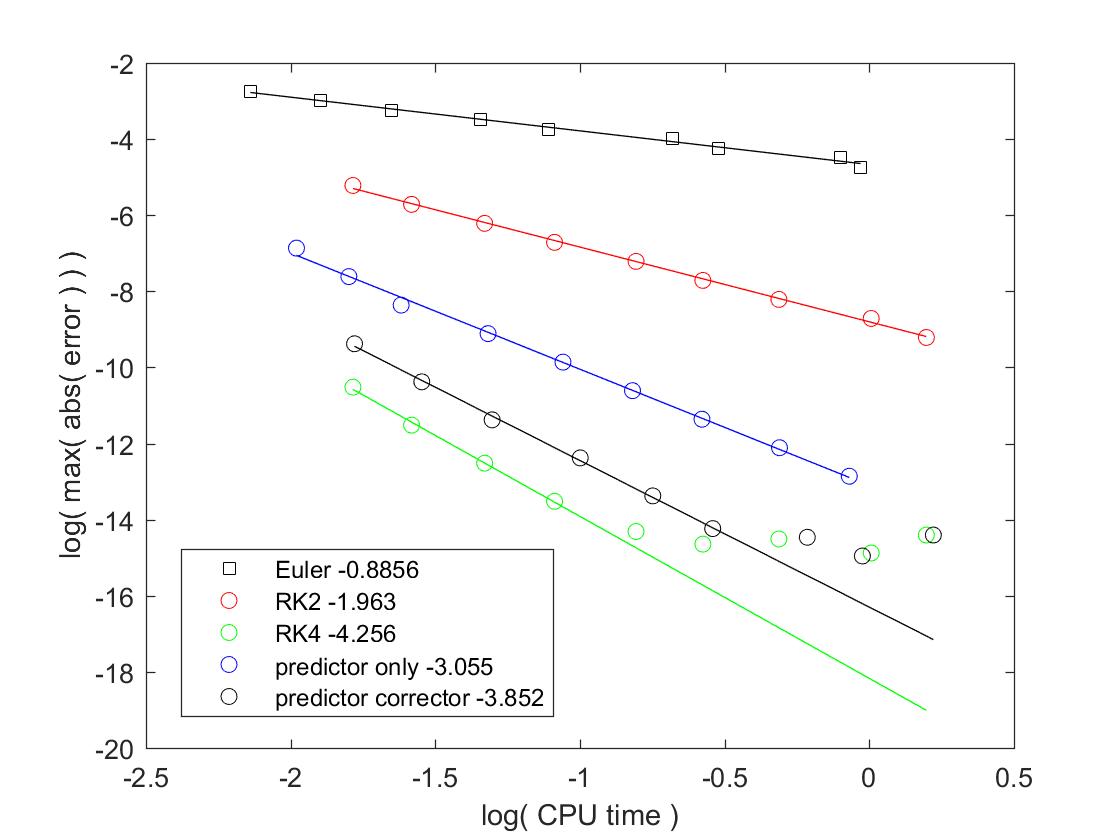
1) Matrices [A] and [B] must be inverted at each time step.

2) Matrices [A] and [B] are the same for all functions .









% comp\_int.m - compare integrators

% time integration using Euler, RK2, RK4, predictor, corrector

% simulate ydot = -y exact solution y = exp(-t)

% HJSIII, 21.03.12

clear

% predictor-corrector coefficients

pred\_coeff = [ 23 -16 5 ] / 12;

corr\_coeff = [ 9 19 -5 1 ] / 24;

% time span [sec]

tend = 4;

% time step [sec]

keep\_err = [];

keep\_cpu = [];

for log10\_h = -4 : 0.25 : -2,

% time values

h = 10 ^ log10\_h;

n = floor( tend / h ) + 1;

nm1 = n - 1;

t = h \* (0:nm1)';

% exact solution

y\_exact = exp( -t );

% allocate space for solutions

y\_euler = zeros(n,1);

y\_rk2 = zeros(n,1);

y\_rk4 = zeros(n,1);

y\_pred\_only = zeros(n,1);

f\_pred\_only = zeros(n,1);

y\_pred\_corr = zeros(n,1);

f\_pred\_corr = zeros(n,1);

% use exact solution for first three samples to get started

y\_euler = y\_exact(1:3);

y\_rk2 = y\_exact(1:3);

y\_rk4 = y\_exact(1:3);

y\_pred\_only(1:3) = y\_exact(1:3);

f\_pred\_only(1:3) = -y\_pred\_only(1:3);

y\_pred\_corr(1:3) = y\_exact(1:3);

f\_pred\_corr(1:3) = -y\_pred\_corr(1:3);

% integrate forward in time

% Euler

tic;

for i = 3 : nm1,

k = -y\_euler(i);

q=inv( rand(18,18) ); % function evaluation

y\_euler(i+1) = y\_euler(i) + k \* h;

end

cpu\_euler = toc;

% RK2

tic;

for i = 3 : nm1,

kA = -y\_rk2(i);

q=inv( rand(18,18) ); % function evaluation

yB = y\_rk2(i) + kA \* h;

kB = -yB;

q=inv( rand(18,18) ); % function evaluation

kM = (kA + kB) / 2;

y\_rk2(i+1) = y\_rk2(i) + kM \* h;

end

cpu\_rk2 = toc;

% RK4

tic;

for i = 3 : nm1,

k1 = -y\_rk4(i);

q=inv( rand(18,18) ); % function evaluation

y2 = y\_rk4(i) + k1 \* h/2;

k2 = -y2;

q=inv( rand(18,18) ); % function evaluation

y3 = y\_rk4(i) + k2 \* h/2;

k3 = -y3;

q=inv( rand(18,18) ); % function evaluation

y4 = y\_rk4(i) + k3 \* h;

k4 = -y4;

q=inv( rand(18,18) ); % function evaluation

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

y\_rk4(i+1) = y\_rk4(i) + kM \* h;

end

cpu\_rk4 = toc;

% predictor only

tic;

for i = 3 : nm1,

y\_pred\_only(i+1) = y\_pred\_only(i) ...

+ h \* pred\_coeff \* [ f\_pred\_only(i) f\_pred\_only(i-1) f\_pred\_only(i-2) ]';

f\_pred\_only(i+1) = -y\_pred\_only(i+1);

q=inv( rand(18,18) ); % function evaluation

end

cpu\_pred\_only = toc;

% predictor-corrector

tic;

for i = 3 : nm1,

y\_star = y\_pred\_corr(i) ...

+ h \* pred\_coeff \* [ f\_pred\_corr(i) f\_pred\_corr(i-1) f\_pred\_corr(i-2) ]';

f\_star = -y\_star;

q=inv( rand(18,18) ); % function evaluation

y\_pred\_corr(i+1) = y\_pred\_corr(i) ...

+ h \* corr\_coeff \* [ f\_star f\_pred\_corr(i) f\_pred\_corr(i-1) f\_pred\_corr(i-2) ]';

f\_pred\_corr(i+1) = -y\_pred\_corr(i+1);

q=inv( rand(18,18) ); % function evaluation

end

cpu\_pred\_corr = toc;

% errors

err\_euler = y\_euler - y\_exact;

err\_rk2 = y\_rk2 - y\_exact;

err\_rk4 = y\_rk4 - y\_exact;

err\_pred\_only = y\_pred\_only - y\_exact;

err\_pred\_corr = y\_pred\_corr - y\_exact;

% plot exact solution and errors

figure( 1 )

clf

subplot( 4, 1, 1 )

plot( t,y\_exact,'r' )

ylabel( 'y' )

title( 'ydot = -y ... exact solution y = exp(-t)' )

subplot( 4, 1, 2 )

plot( t,err\_euler,'g')

ylabel( 'error' )

title( [ 'Euler h=' num2str(h,4) ] )

subplot( 4, 1, 3 )

plot( t,err\_rk2,'b')

ylabel( 'error' )

title( [ 'RK2 h=' num2str(h,4) ] )

subplot( 4, 1, 4 )

plot( t,err\_rk4,'k')

ylabel( 'error' )

title( [ 'RK4 h=' num2str(h,4) ] )

xlabel( 'time [sec]' )

figure( 2 )

clf

subplot( 3, 1, 1 )

plot( t,y\_exact,'r' )

ylabel( 'y' )

title( 'ydot = -y ... exact solution y = exp(-t)' )

subplot( 3, 1, 2 )

plot( t,err\_pred\_only,'g')

ylabel( 'error' )

title( [ 'predictor only h=' num2str(h,4) ] )

subplot( 3, 1, 3 )

plot( t,err\_pred\_corr,'b' )

xlabel( 'time [sec]' )

ylabel( 'error' )

title( [ 'predictor corrector h=' num2str(h,4) ] )

keep\_err = [ keep\_err ; [ log10\_h log10(max(abs(err\_euler))) ...

log10(max(abs(err\_rk2))) log10(max(abs(err\_rk4))) ...

log10(max(abs(err\_pred\_only))) log10(max(abs(err\_pred\_corr))) ] ];

keep\_cpu = [ keep\_cpu ; [ log10\_h log10(cpu\_euler) log10(cpu\_rk2) log10(cpu\_rk2) ...

log10(cpu\_pred\_only) log10(cpu\_pred\_corr) ] ];

end % bottom - for log10\_h

h = keep\_err(:,1);

eeu = keep\_err(:,2);

er2 = keep\_err(:,3);

er4 = keep\_err(:,4);

epo = keep\_err(:,5);

epc = keep\_err(:,6);

ceu = keep\_cpu(:,2);

cr2 = keep\_cpu(:,3);

cr4 = keep\_cpu(:,4);

cpo = keep\_cpu(:,5);

cpc = keep\_cpu(:,6);

% slopes for error

poly\_eeu = polyfit( h, eeu, 1 );

fit\_eeu = polyval( poly\_eeu, h );

poly\_er2 = polyfit( h, er2, 1 );

fit\_er2 = polyval( poly\_er2, h );

poly\_er4 = polyfit( h(6:end), er4(6:end), 1 );

fit\_er4 = polyval( poly\_er4, h );

poly\_epo = polyfit( h, epo, 1 );

fit\_epo = polyval( poly\_epo, h );

poly\_epc = polyfit( h(6:end), epc(6:end), 1 ); % ignore noise floor

fit\_epc = polyval( poly\_epc, h );

% plot error

figure( 3 )

clf

plot( h,eeu,'ks', h,er2,'ro', h,er4,'go', h,epo,'bo', h,epc,'ko', ...

h,fit\_eeu,'k', h,fit\_er2,'r', h,fit\_er4,'g', h,fit\_epo,'b', h,fit\_epc,'k' )

xlabel( 'log( h )' )

ylabel( 'log( max( abs( error ) ) )' )

legend( [ 'Euler ' num2str(poly\_eeu(1),4) ], ...

[ 'RK2 ' num2str(poly\_er2(1),4) ], ...

[ 'RK4 ' num2str(poly\_er4(1),4) ], ...

[ 'predictor only ' num2str(poly\_epo(1),4) ], ...

[ 'predictor corrector ' num2str(poly\_epc(1),4) ] )

% slopes for CPU time

poly\_ceu = polyfit( ceu, eeu, 1 );

fit\_ceu = polyval( poly\_ceu, ceu );

poly\_cr2 = polyfit( cr2, er2, 1 );

fit\_cr2 = polyval( poly\_cr2, cr2 );

poly\_cr4 = polyfit( cr4(6:end), er4(6:end), 1 );

fit\_cr4 = polyval( poly\_cr4, cr4 );

poly\_cpo = polyfit( cpo, epo, 1 );

fit\_cpo = polyval( poly\_cpo, cpo );

poly\_cpc = polyfit( cpc(6:end), epc(6:end), 1 ); % ignore noise floor

fit\_cpc = polyval( poly\_cpc, cpc );

% plot CPU time

figure( 4 )

clf

plot( ceu,eeu,'ks', cr2,er2,'ro', cr4,er4,'go', cpo,epo,'bo', cpc,epc,'ko', ...

ceu,fit\_ceu,'k', cr2,fit\_cr2,'r', cr4,fit\_cr4,'g', cpo,fit\_cpo,'b', cpc,fit\_cpc,'k' )

xlabel( 'log( CPU time )' )

ylabel( 'log( max( abs( error ) ) )' )

legend( [ 'Euler ' num2str(poly\_ceu(1),4) ], ...

[ 'RK2 ' num2str(poly\_cr2(1),4) ], ...

[ 'RK4 ' num2str(poly\_cr4(1),4) ], ...

[ 'predictor only ' num2str(poly\_cpo(1),4) ], ...

[ 'predictor corrector ' num2str(poly\_cpc(1),4) ] )

% bottom of comp\_int

**Forward Time Integration in MATLAB**

[T,Y] = *solver*(odefun,tspan,y0)

[T,Y] = *solver*(odefun,tspan,y0,options)

|  |  |
| --- | --- |
| odefun | A function handle that evaluates the right side of the differential equations. All solvers solve systems of equations in the form *y*′ = *f*(*t*,*y*) or problems that involve a mass matrix, *M*(*t*,*y*)*y*′ = *f*(*t*,*y*). The ode23s solver can solve only equations with constant mass matrices. ode15s and ode23t can solve problems with a mass matrix that is singular, i.e., differential-algebraic equations (DAEs). |
| tspan | A vector specifying the interval of integration, [t0,tf]. The solver imposes the initial conditions at tspan(1), and integrates from tspan(1) to tspan(end). To obtain solutions at specific times (all increasing or all decreasing), use tspan = [t0,t1,...,tf].  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). All solvers in the ODE suite obtain output values by means of continuous extensions of the basic formulas. Although a solver does not necessarily step precisely to a time point specified in tspan, the solutions produced at the specified time points are of the same order of accuracy as the solutions computed at the internal time points.  Specifying tspan with more than two elements has little effect on the efficiency of computation, but for large systems, affects memory management. |
| y0 | A vector of initial conditions. |
| options | Structure of optional parameters that change the default integration properties. This is the fourth input argument. |

| **Solver** | **Problem Type** | **Order of Accuracy** | **When to Use** |
| --- | --- | --- | --- |
| ode45 | Nonstiff | Medium | Most of the time. This should be the first solver you try. |
| ode23 | Nonstiff | Low | For problems with crude error tolerances or for solving moderately stiff problems. |
| ode113 | Nonstiff | Low to high | For problems with stringent error tolerances or for solving computationally intensive problems. |
| ode15s | Stiff | Low to medium | If ode45 is slow because the problem is stiff. |
| ode23s | Stiff | Low | If using crude error tolerances to solve stiff systems and the mass matrix is constant. |
| ode23t | Moderately Stiff | Low | For moderately stiff problems if you need a solution without numerical damping. |
| ode23tb | Stiff | Low | If using crude error tolerances to solve stiff systems. |

**MATLAB Algortihms**

ode45 is based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair. It is a one-step solver – in computing y(tn), it needs only the solution at the immediately preceding time point, y(tn-1). In general, ode45 is the best function to apply as a first try for most problems. [[3]](http://www.mathworks.com/help/matlab/ref/ode23t.html#bti6r9n-45)

ode23 is an implementation of an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine. It may be more efficient than ode45 at crude tolerances and in the presence of moderate stiffness. Like ode45, ode23 is a one-step solver. [[2]](http://www.mathworks.com/help/matlab/ref/ode23t.html#bti6r9n-44)

ode113 is a variable order Adams-Bashforth-Moulton PECE solver. It may be more efficient than ode45 at stringent tolerances and when the ODE file function is particularly expensive to evaluate. ode113 is a multistep solver — it normally needs the solutions at several preceding time points to compute the current solution. [[7]](http://www.mathworks.com/help/matlab/ref/ode23t.html#bti6r9n-49)

The above algorithms are intended to solve nonstiff systems. If they appear to be unduly slow, try using one of the stiff solvers below.

ode15s is a variable order solver based on the numerical differentiation formulas (NDFs). Optionally, it uses the backward differentiation formulas (BDFs, also known as Gear's method) that are usually less efficient. Like ode113, ode15s is a multistep solver. Try ode15s when ode45 fails, or is very inefficient, and you suspect that the problem is stiff, or when solving a differential-algebraic problem. [[9]](http://www.mathworks.com/help/matlab/ref/ode23t.html#bti6r9n-51), [[10]](http://www.mathworks.com/help/matlab/ref/ode23t.html#bti6r9n-52)

ode23s is based on a modified Rosenbrock formula of order 2. Because it is a one-step solver, it may be more efficient than ode15s at crude tolerances. It can solve some kinds of stiff problems for which ode15s is not effective. [[9]](http://www.mathworks.com/help/matlab/ref/ode23t.html#bti6r9n-51)

ode23t is 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. ode23t can solve DAEs. [[10]](http://www.mathworks.com/help/matlab/ref/ode23t.html#bti6r9n-52)

ode23tb is 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 two. By construction, the same iteration matrix is used in evaluating both stages. Like ode23s, this solver may be more efficient than ode15s at crude tolerances. [[8]](http://www.mathworks.com/help/matlab/ref/ode23t.html#bti6r9n-50), [[1]](http://www.mathworks.com/help/matlab/ref/ode23t.html#bti6r9n-43)

## References

[1] Bank, R. E., W. C. Coughran, Jr., W. Fichtner, E. Grosse, D. Rose, and R.Smith, "Transient Simulation of Silicon Devices and Circuits," *IEEE Trans. CAD*, 4 (1985), pp. 436–451.

[2] Bogacki, P. and L. F. Shampine, "A 3(2) pair of Runge-Kutta formulas," *Appl. Math. Letters*, Vol. 2, 1989, pp. 321–325.

[3] Dormand, J. R. and P. J. Prince, "A family of embedded Runge-Kutta formulae," *J. Comp. Appl. Math.*, Vol. 6, 1980, pp. 19–26.

[4] Forsythe, G. , M. Malcolm, and C. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, New Jersey, 1977.

[5] Kahaner, D. , C. Moler, and S. Nash, *Numerical Methods and Software*, Prentice-Hall, New Jersey, 1989.

[6] Shampine, L. F. , *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall, New York, 1994.

[7] Shampine, L. F. and M. K. Gordon, *Computer Solution of Ordinary Differential Equations: the Initial Value Problem*, W. H. Freeman, SanFrancisco, 1975.

[8] Shampine, L. F. and M. E. Hosea, "Analysis and Implementation of TR-BDF2," *Applied Numerical Mathematics 20*, 1996.

[9] Shampine, L. F. and M. W. Reichelt, "The MATLAB ODE Suite," *SIAM Journal on Scientific Computing,* Vol. 18, 1997, pp. 1–22.

[10] Shampine, L. F., M. W. Reichelt, and J.A. Kierzenka, "Solving Index-1 DAEs in MATLAB and Simulink," *SIAM Review*, Vol. 41, 1999, pp. 538–552.

**Simple pendulum from Notes\_09\_02**



% ode\_pendulum\_main.m - main program for example use of ODE solver

% simple pendulum Notes\_09\_02

% HJSIII, 20.04.09

%

% ODE coded in file ode\_pendulum\_yd.m - (JG+m\*a\*a)\*thdd = T - m\*g\*a\*cos(th)

%

% {y} = { th } {yd} = { thd }

% { thd } { thdd }

clear

% global constants

global m JG a g

% constants

d2r = pi / 180;

% physical constants

% th [rad]

% thd [rad/sec]

% thdd [rad/sec^2]

m = 0.46; % mass [lbm]

JG = 1.5; % mass moment [lbm.in.in]

a = 3.7; % centroid offset [in]

g = 386; % acceleration of gravity [ipss]

% initial conditions

th0 = -80 \* d2r; % 10 degrees above vertical

y0 = [ th0 0 ]'; % free release

% time range

tspan = [ 0 5 ];

% ode23

[ t, y ] = ode23( 'ode\_pendulum\_yd', tspan, y0 );

th = y(:,1) / d2r; % [deg]

thd = y(:,2); % [rad/s]

% time domain results

figure( 1 )

clf

subplot( 2, 1, 1 )

plot( t,th )

xlabel( 'Time [sec]' )

ylabel( 'Angle [deg]' )

% axis( [ 0 5 -0.1 0.1 ] )

subplot( 2, 1, 2 )

plot( t,thd )

xlabel( 'Time [sec]' )

ylabel( 'Ang vel [rad/s]' )

% axis( [ 0 5 -1.5 1.5 ] )

% bottom - ode\_pendulum\_main

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

function yd = ode\_pendulum\_yd( t, y )

% provides yd for integration

% simple pendulum Notes\_09\_02

% (JG+m\*a\*a)\*thdd = T - m\*g\*a\*cos(th)

% HJSIII, 20.04.09

% global constants

global m JG a g

% free motion

T = 0;

% individual terms

th = y(1);

thd = y(2);

% free swing

thdd = ( T - m\*g\*a\*cos(th) ) / (JG + m\*a\*a);

% simple Coulomb friction in revolute joint with bushing

mu = 0.1; % coefficient of friction

radius = 0.25; % [inch] radius for ID of bushing

W = m\*g; % [lbm.in/s/s] weight of pendulum

Tf = mu \* radius \* W; % [lbf.in] frictional torque

%thdd = ( T -Tf\*sign( thd ) -m\*g\*a\*cos(th) ) / (JG + m\*a\*a);

yd(1,1) = thd;

yd(2,1) = thdd;

return

% bottom - ode\_pendulum\_yd

**Double pendulum from Notes\_09\_02**





% ode\_dp\_main.m - main program for example use of ODE solver

% double pendulum Notes\_09\_02

% HJSIII, 20.04.09

%

% ODE coded in file ode\_dp\_yd.m

%

% {y} = { theta2 } {ydot} = { theta2\_dot }

% { theta3 } = ( theta3\_dot }

% { theta2\_dot } = ( theta2\_dot\_dot }

% ( theta3\_dot } = ( theta3\_dot\_dot }

clear

% constants

d2r = pi /180;

% initial conditions - shoulder 5 deg above vertical, elbow 5 deg up

% th2=0 shoulder horizontal, th3=0 elbow straight

theta2\_start = -85 \* d2r;

theta3\_start = 5 \* d2r;

% free release

y0 = [ theta2\_start theta3\_start 0 0 ]';

% time range

tspan = [ 0 4 ];

% time integration

[ t, y ] = ode23( 'ode\_dp\_yd', tspan, y0 );

% time domain results

th2 = y(:,1);

th3 = y(:,2);

th2d = y(:,3);

th3d = y(:,4);

figure( 1 )

clf

plot( t,th2/d2r,'ro', t,th3/d2r,'go' )

xlabel( 'Time [sec]' )

ylabel( 'Shoulder and elbow angles [deg]' )

legend( 'th2', 'th3' )

% check kinetic and potential energy

% units = m kg sec m/sec m/sec/sec N N.m

gravity = 9.81; % [m/sec/sec]

% approximate human arm

d2 = 0.293; % [m] upper arm

d3 = 0.225; % [m] forearm

m2 = 3.80; % [kg] upper arm

m3 = 2.68; % [kg] forearm

J2 = 33300e-6; % [kg.m.m] upper arm

J3 = 9900e-6; % [kg.m.m] forearm

% symmetric links

a2 = d2 / 2;

a3 = d3 / 2;

% positions and velocities

x2 = a2\*cos(th2);

y2 = a2\*sin(th2);

x2d = -a2\*th2d.\*sin(th2);

y2d = a2\*th2d.\*cos(th2);

x3 = d2\*cos(th2) + a3\*cos(th2+th3);

y3 = d2\*sin(th2) + a3\*sin(th2+th3);

x3d = -d2\*th2d.\*sin(th2) - a3\*(th2d+th3d).\*sin(th2+th3);

y3d = d2\*th2d.\*cos(th2) + a3\*(th2d+th3d).\*cos(th2+th3);

% kinetic

K2 = m2\*( x2d.\*x2d + y2d.\*y2d )/2 + J2\*th2d.\*th2d/2;

K3 = m3\*( x3d.\*x3d + y3d.\*y3d )/2 + J3\*(th2d+th3d).\*(th2d+th3d)/2;

% potential

P2 = m2\*y2\*gravity;

P3 = m3\*y3\*gravity;

% show total energy

TE = K2 + K3 + P2 + P3;

figure( 2 )

clf

subplot( 5,1,1 )

plot( t,K2,'r-' )

ylabel( 'K2' )

subplot( 5,1,2 )

plot( t,K3,'g-' )

ylabel( 'K3' )

subplot( 5,1,3 )

plot( t,P2,'r--' )

ylabel( 'P2' )

subplot( 5,1,4 )

plot( t,P3,'g--' )

ylabel( 'P3' )

subplot( 5,1,5 )

plot( t,TE,'b' )

xlabel( 'Time [sec]' )

ylabel( 'Total' )

% bottom - ode\_dp\_main

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

function ydot = ode\_dp\_yd( t, y )

% provides yd for integration

% double pendulum Notes\_09\_02

% HJSIII, 20.04.09

% units = m kg sec m/sec m/sec/sec N N.m

gravity = 9.81; % [m/sec/sec]

% approximate human arm

d2 = 0.293; % [m] upper arm

d3 = 0.225; % [m] forearm

m2 = 3.80; % [kg] upper arm

m3 = 2.68; % [kg] forearm

J2 = 33300e-6; % [kg.m.m] upper arm

J3 = 9900e-6; % [kg.m.m] forearm

% symmetric links

a2 = d2 / 2;

a3 = d3 / 2;

% extract state variables

th2 = y(1);

th3 = y(2);

th2d = y(3);

th3d = y(4);

% coefficients

JB = m3\*a3\*a3 + J3;

JA = JB + m2\*a2\*a2 + m3\*d2\*d2 + J2 + 2\*m3\*d2\*a3\*cos(th3);

C = JB + m3\*d2\*a3\*cos(th3);

D = m3\*d2\*a3\*sin(th3);

G2 = (m2\*a2+m3\*d2) \* gravity \* cos(th2);

G3 = m3\*a3 \* gravity \* cos(th2+th3);

% motor torques

T2 = 0;

T3 = 0;

% test

M = [ JA C ;

C JB ];

rhs = [ T2+D\*th3d\*th3d+2\*D\*th2d\*th3d-G2-G3 ;

T3-D\*th2d\*th2d-G3 ];

thdd = inv(M) \* rhs;

% return derivatives of state derivatives

ydot = [ th2d th3d thdd(1) thdd(2) ]';

return

% bottom - ode\_dp\_yd

**Second-Order Initial-Value Problems**

**Taylor Series Using Second Derivatives**

given second-order initial-value differential equations of the form

 and 

must know  at time ti

may determine  and 

for time step 

# 

**Fourth Order Runge-Kutta-Nystrom Using Second Derivatives (Fehlberg)**

given second-order initial-value differential equations of the form

 and 

must know  at time ti

may determine  and 











# Fifth-degree (Quintic) Predictor and Seventh-degree (Heptic) Corrector

# with Second Derivatives

given second-order initial-value differential equations of the form

 and 

must know  respectively at times ti, ti-1 and ti-2

may determine  and 

ti

ti+1

ti-1

ti-2

= f(y)

fi

gi

t

fi-1

gi-1

fi-2

gi-2

fixed time step h  = ( t - ti ) / h d = dt / h dt = h d d /dt = 1 / h

0

+1

-1

-2



fi

gi

fi-1

gi-1

fi-2

gi-2

use fifth degree polynomial predictor 









know coefficients ai for interpolant 





**PREDICTOR** **(3-step Obreshkov)**



now use predicted value of to compute  and 

0

+1

-1

-2



fi

gi

fi-1

gi-1

fi-2

gi-2

fi+1\*

gi+1\*

use seventh degree polynomial corrector 







know coefficients bi 





**CORRECTOR (4-step Obreshkov)**



# Third-degree (Cubic) Predictor and Third-degree (Cubic) Corrector

# with Second Derivatives

given second-order initial-value differential equations of the form

 and 

must know  respectively at times ti and ti-1

may determine  and 

ti

ti+1

ti-1

= f(y)

fi

gi

t

fi-1

gi-1

fixed time step h  = ( t - ti ) / h d = dt / h dt = h d d /dt = 1 / h

0

+1

-1



fi

gi

fi-1

gi-1

use third degree polynomial predictor 









know coefficients ai for interpolant 





**PREDICTOR** **(2-step Obreshkov per Sehnalova)**



now use predicted value of to compute  and 

fi

gi

fi+1\*

gi+1\*



+1

0

use third degree polynomial corrector 







know coefficients bi 





**CORRECTOR** **(2-step Obreshkov per Sehnalova)**



# Least-Squares Quadratic Predictor and Cubic Corrector

**with Second Derivatives**

given second-order initial-value differential equations of the form

 and 

must know  respectively at times ti, ti-1 and ti-2

may determine  and 

ti

ti+1

ti-1

ti-2

= f(y)

fi-1

fi

fi-2

t

fixed time step h  = ( t - ti ) / h d = dt / h dt = h d d /dt = 1 / h

0

+1

-1

-2

fi-1

fi

fi-2



use quadratic polynomial predictor f = a0 + a1  + a2 2















know coefficients a0, a1 and a2 for interpolant f() = a0 + a1  + a2 2 = ()

yi+1 =  + yi = h + yi = h ( a0 + a12 /2 + a23 /3 ) + yi

yi+1 = yi + h ( a0 + a1 /2 + a2 /3 )

**PREDICTOR** **(least-squares 3-step Adams-Bashforth)**



now use predicted value of yi+1 to compute f(yi+1) = fi+1\* and g(yi+1) = gi+1\*

0

+1

-1

-2

fi-1

fi

fi-2



fi+1\*

use cubic polynomial corrector f = b0 + b1  + b2 2 + b3 3







know coefficients b0, b1, b2 and b3 for new interpolant f() = b0 + b1  + b2 2 + b3 3 = ()

yi+1 =  + yi = h + yi = h ( b0 + b12 /2 + b23 /3 + b34 /4 ) + yi

yi+1 = yi + h ( b0 + b1 /2 + b2 /3 + b3 /4)

**CORRECTOR (least-squares 4-step Adams-Moulton)**



**does not work as well as Obreshkov**