13 Numerical Solution of ODE's

In simulating dynamical systems, we frequently solve ordinary differential equations. These are of the form

$$\frac{dx}{dt} = f(t, x),$$

where the function f has as arguments both time and the state x. The time argument is used if the system is time-dependent (time-varying), but unnecessary if the system is time-invariant. Solving the differential equation means propagating x forward in time from some initial condition, say $x(t = 0)$, and typically the solution will be given as a vector of $[x(0), x(\Delta t), x(2\Delta t), \cdots]$, where Δt is a fixed time step.

The Taylor series is used to derive most of the simple formulas for solving ODE's. The general Taylor series expansion of the generic function g in two variables is

$$\begin{aligned} g(t+\Delta t, x+\Delta x) &= g(t,x) + \frac{\partial g}{\partial t}\Delta t + \frac{\partial g}{\partial x}\Delta x + \\ &\quad \frac{1}{2!}\frac{\partial^2 g}{\partial t^2}\Delta t^2 + \frac{1}{2!}\frac{\partial^2 g}{\partial t \partial x}\Delta t \Delta x + \frac{1}{2!}\frac{\partial^2 g}{\partial x \partial t}\Delta x \Delta t + \frac{1}{2!}\frac{\partial^2 g}{\partial x^2}\Delta x^2 + \cdots . \end{aligned}$$

In the above formula, when the arguments of g and its derivatives are not shown, we mean that it is to be evaluated at (t, x), that is g alone means $g(t, x)$ and so on. We use this shorthand below in several places. The simplest of all the ODE methods is *forward Euler*, created by setting $g = x$ and looking only at the first two terms on the right-hand side of the Taylor series:

$$x(t + \Delta t) = x(t) + \frac{dx}{dt}\Delta t = x(t) + \Delta t f$$

This formula says that x at the next time instant is the current x plus the time step times the slope *evaluated at the current* x. Referring to the Taylor series, we see that the forward Euler does not do anything with the second-order terms (Δt^2 and beyond), and so we say the method is second-order accurate in the step, which will turn out to be first-order accurate when you solve a real problem with many steps. First-order means that if you halve the time step, you can expect about half the error in the overall simulation.

An alternative, the Runge-Kutta methods are popular workhorses, and implemented in the MATLAB commands `ode23()` and `ode45()`. Let's take a look at the first of these: The rule is

$$\begin{aligned} k_1 &= \Delta t f \\ k_2 &= \Delta t f(t + \Delta t/2, x(t) + k_1/2) \\ x(t + \Delta t) &= x(t) + k_2. \end{aligned}$$

We see that k_1 is the same change in x as given by the forward Euler rule. k_2 is another guess for the change in x, but with the slope calculated at the approximate midpoint $[t +$

$\Delta t/2, x(t) + k_1/2]$. You can guess that this will be a somewhat more accurate result. Note, however, that we have to evaluate the function twice in each time step, as opposed to once each time step for the forward Euler method.

Here are your two problems:

1. The listed Runge-Kutta algorithm is third-order accurate in the step and second-order accurate in the whole: in each step, we get a cancelation of all the second-order terms in the Taylor series! Can you figure out how to show this?

   Solution: We have to show that the Runge-Kutta formula captures up to second-order terms in the Taylor series. First, the formula for k_2 is itself approximated as a Taylor series:

$$\begin{aligned} k_2 &= \Delta t f(t + \Delta t/2, x + k_1/2) \\ &= \Delta t \left[f + \frac{\Delta t}{2}\frac{\partial f}{\partial t} + \frac{k_1}{2}\frac{\partial f}{\partial x} + h.o.t. \right] \\ &= \Delta t \left[f + \frac{\Delta t}{2}\frac{\partial f}{\partial t} + \frac{\Delta t}{2}\frac{\partial f}{\partial x} f + h.o.t. \right], \end{aligned}$$

   where we again assume evaluation of functions at $x(t)$ and t when the argument is omitted. The first relation comes from the bivariate Taylor series, and the second by substituting in k_1. The acronym h.o.t. stands for "higher-order terms." Next, write out $x(t + \Delta t)$ according to the last line of the RK2 rule:

$$\begin{aligned} x(t + \Delta t) &= x + k_2 \\ &= x + \Delta t f + \frac{\Delta t^2}{2}\left[\frac{\partial f}{\partial t} + \frac{\partial f}{\partial x} f + h.o.t. \right] \end{aligned}$$

   You recognize the first two terms in the square brackets here as the total derivative of (bivariate) f with t, which is the second derivative of x with time. How does this result look compared to the direct Taylor series for x? We have for the univariate Taylor series applied to x:

$$x(t + \Delta t) = x + \Delta t \frac{dx}{dt} + \frac{\Delta t^2}{2}\frac{dx^2}{dt^2} + h.o.t.$$

   So the RK2 formula does indeed capture the first- and second-order terms of the Taylor series. Note that the h.o.t. that appears when we worked with k_2 is <u>not</u> the same as the h.o.t. that appears in the direct expansion of x - so we cannot claim that the third-order terms are captured in the RK2 rule. RK4 (shown below) captures both the third- and fourth-order terms!

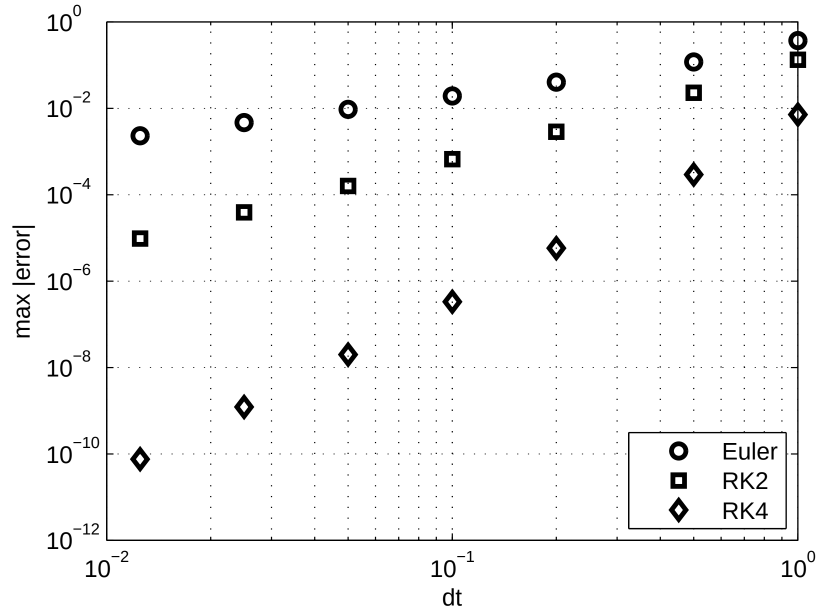
2. Compare the error convergence rates, for the forward Euler and the second-order Runge-Kutta on the simple problem $dx/dt = -x$, $x(t = 0) = 1$. To do this, you will make several simulations with each method, over the time scale $[0, 5]$, and with

Δt = [0.0125 0.025 0.05 0.10 0.20 0.50 1.00]. For each, record the maximum absolute error with respect to the exact solution (a decaying exponential), and then make a summary plot of Δt versus error, including both methods. You should be able to see the first-order convergence of forward Euler and the second-order convergence of RK2.

As an aside, forward Euler has *stability problems* that are more severe than for the Runge-Kutta family, and this is another reason why forward Euler should usually not be your first choice.

Please do not use the canned MATLAB ODE functions to solve the second problem. I am asking you to make the explicit calculations for programming practice, to see the error rates first-hand, and because we sometimes don't have MATLAB available where needed, e.g., on an embedded processor.

Solution: See MATLAB results and code below. The Euler method is first-order: a ten-fold reduction in Δt gives a ten-fold reduction in error. RK2 is second-order: the ten-fold reduction in Δt gives a one-hundred-fold reduction in error. Most impressive, the RK4 is fourth-order so we get a ten-thousand-fold improvement for a ten-fold reduction in Δt. Note that RK4 requires four evaluations per time step, and RK2 requires two. But it is usually well worth it! Finally, notice that the RK2 and the forward Euler both give awful errors with $\Delta t = 1$. This is no surprise, since the time constant of the system is one also. The RK4 gives us a much better result here, the result of those extra evaluations.



```
Errors for the Different dt's and Methods:
dt: 0.013     Euler: 2.31e-003   RK2: 9.67e-006   RK4: 7.56e-011
dt: 0.025     Euler: 4.65e-003   RK2: 3.90e-005   RK4: 1.22e-009
dt: 0.050     Euler: 9.39e-003   RK2: 1.59e-004   RK4: 2.00e-008
```

```
dt: 0.100     Euler: 1.92e-002   RK2: 6.62e-004   RK4: 3.33e-007
dt: 0.200     Euler: 4.02e-002   RK2: 2.86e-003   RK4: 5.80e-006
dt: 0.500     Euler: 1.18e-001   RK2: 2.27e-002   RK4: 2.91e-004
dt: 1.000     Euler: 3.68e-001   RK2: 1.32e-001   RK4: 7.12e-003
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Show the error properties of forward Euler vs.
% second-order and fourth-order Runge-Kutta integration schemes.

function simEulerRK2
clear all;

tfinal = 5 ;    % final simulation time
dtvector = [.0125 .025 .05 .1 .2 .5 1] ; % vector of dt's to try
xinitial = 1 ; % initial value of x

figure(1);clf;hold off;
figure(2);clf;hold off;
subplot('Position',[.2 .2 .5 .5]);

disp('Errors for the Different dt''s and Methods:');
fn = fopen('simEulerRK2.dat','w');
fprintf(fn,'Errors for the Different dt''s and Methods:\n');

for dt = dtvector,  % cycle through all the dt's

    n = tfinal/dt ;             % the number if steps to take
    timevector = 0:dt:n*dt ;    % vector of times (length n+1)
    exact = exp(-timevector);   % exact solution to the diff eqn:
                                % x' = f(t,x)

    xEuler(1) = xinitial ; % initial value for the Euler rule
    xRK2(1) = xinitial ; % initial value for the RK2 rule
    xRK4(1) = xinitial ; % initial value for the RK4 rule

    % do the Euler version
    for i = 1:n,
        t = timevector(i) ;
        xEuler(i+1) = xEuler(i) + dt*f(t,xEuler(i)) ;
    end;

    % do the RK2 version
    for i = 1:n,
        t = timevector(i) ;
        k1 = dt*f(t,xRK2(i)) ;
        k2 = dt*f(t+dt/2, xRK2(i)+k1/2) ;
        xRK2(i+1) = xRK2(i) + k2 ;
    end;
    % do the RK4 version!
    for i = 1:n,
        t = timevector(i) ;
        k1 = dt*f(t,xRK4(i)) ;
        k2 = dt*f(t+dt/2, xRK4(i)+k1/2) ;
        k3 = dt*f(t+dt/2, xRK4(i)+k2/2) ;
        k4 = dt*f(t+dt, xRK4(i)+k3) ;
        xRK4(i+1) = xRK4(i) + (k1 + 2*k2 + 2*k3 + k4) / 6 ;
    end;

    figure(1);
    plot(timevector,log10(abs(xEuler-exact)),'b',...
        timevector,log10(abs(xRK2-exact)),'r',...
        timevector,log10(abs(xRK4-exact)),'g','LineWidth',2) ;
    hold on;
    grid;

    disp(sprintf(...
        'dt: %5.3f     Euler: %6.2e   RK2: %6.2e   RK4: %6.2e',...
        dt, max(abs(xEuler-exact)), max(abs(xRK2-exact)), ...
        max(abs(xRK4-exact))));

    fprintf(fn,...
        'dt: %5.3f     Euler: %6.2e   RK2: %6.2e   RK4: %6.2e\n',...
        dt, max(abs(xEuler-exact)), max(abs(xRK2-exact)), ...
        max(abs(xRK4-exact)));

    figure(2);
    loglog(dt, (max(abs(xEuler-exact))),'o',...
        dt,(max(abs(xRK2-exact))),'s', ...
        dt, (max(abs(xRK4-exact))),'d','LineWidth',2);
    hold on;
    if dt == max(dtvector),
        legend('Euler','RK2','RK4',4);
        xlabel('dt');
        ylabel('max |error|');
        grid;
    end;

    clear n timevector exact xEuler xRK2 xRK4 t k1 k2 k3 k4 ;
end;
fclose(fn);
figure(2);
print -deps simEulerRK2.eps
```

```
% here is the function definition
function [dxdt] = f(t,x) ;
    dxdt = -x ;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```