Week 13: Wednesday, Apr 25

The Runge-Kutta concept

Runge-Kutta methods evaluate $f(t, y)$ multiple times in order to get higher order accuracy. For example, the classical Runge-Kutta scheme is

$$\begin{aligned}
K_0 &= f(t_n, y_n) \\
K_1 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}K_0\right) \\
K_2 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}K_1\right) \\
K_3 &= f\left(t_n + h, y_n + hK_2\right) \\
y_{n+1} &= y_n + \frac{h}{6}\left(K_0 + 2K_1 + 2K_2 + K_3\right).
\end{aligned}$$

Note that if f is a function of time alone, this is simply Simpson's rule. This is no accident.

Runge-Kutta methods are frequently used in pairs where a high-order method and a lower-order method can be computed with the same evaluations. Perhaps the most popular such methods are the Fehlberg 4(5) and Dormand-Prince 4(5) pairs — the MATLAB code `ode45` uses the Dormand-Prince pair. The difference between the two methods is then used as an estimate of the *local* error in the lower-order method. If a local error estimate seems too large, it is natural to try again with a shorter step based on an asymptotic expansion of the error. This method of step control works well on many problems in practice, but it is not foolproof (as we will see in HW 7). For example, in some settings the adaptive error control may suggest a time step which is fine for local error, but terrible for stability.

Adaptive time stepping routines generally use tolerances for both absolute and relative errors. A time step is accepted if

$$|e_i| < \max\left(\text{rtol}_i |y_i|, \text{atol}_i\right)$$

where rtol_i and atol_i are the tolerances for the ith component of the solution vector. The error tolerances have default values (10^{-3} relative and 10^{-6} absolute), but in practice it may be a good idea to set the tolerances yourself.

In principle, comparing two methods gives us an error estimate only for the lower-order method. However, one often takes a step with the higher-order method (at least for non-stiff problems). This cheat works well in practice, but we use the dignified-sound name of *local extrapolation* to dodge awkward questions about its mathematical legitimacy.

There are a bewildering variety of Runge-Kutta methods. Some are explicit, others are implicit. Some preserve interesting structural properties. Some are based on equally-spaced interpolation points, others evaluate on Gauss-Legendre points. In some, the stages can be computed one at a time; in others, the stages all depend on each other. But these methods are beyond the scope of the current discussion.

Matlab's ode45

For most non-stiff problems, `ode45` is a good first choice of integrators. The basic calling sequence is

[tout, yout] = **ode45**(f, tspan, y0);

The function `f(t,y)` returns a column vector. On output, `tout` is a column vector of evaluation times and `yout` is a matrix of solution values (one per row). Usually, `tspan` has two entries: `tspan = [t0 tmax]`. However, we can also specify points where we want solution values. In general, the underlying ODE solver does not put time steps at each of these points; instead it fills in the values using polynomial interpolation (this is called *dense output*).

The `ode45` function takes an optional output called `opt` that contains a structure produced by `odeset`. Using `odeset`, we can set error tolerances, put bounds on the step size, indicate to the solver that certain components must be non-negative, look for special events, or request diagnostic output.

The multistep concept

The Runge-Kutta methods proceed from time t_n to time t_{n+1}, then stop looking at t_n. An alternative is to use not only the behavior at t_n, but also the behavior at previous times t_{n-1}, t_{n-2}, etc. Methods that do this are *multistep methods*. Most such methods are based on linear interpolation.

For non-stiff problems, the *Adams family* are the most popular multi-step methods. The k-step explicit Adams methods (or Adams-Bashforth meth-

ods) interpolate $f(t_j, y_j)$ at points $t_{n-k}, \ldots, t_n$ with a degree k polynomial $p(t)$. Then in order to estimate

$$y(t_{n+1}) = y(t_n) + \int_{t_n}^{t_{n+1}} f(s, y(s))\, ds,$$

one computes

$$y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} p(s)\, ds.$$

The *implicit* Adams methods (or Adams-Moulton methods) also interpolate through the unknown point. Though they are not A-stable, Adams-Moulton methods have larger stability regions and smaller error constants than Adams-Bashforth methods. Often, the two are used together to form a predictor-corrector pair: predict with Adams-Bashforth, then correct to Adams-Moulton. Because these methods are typically used for non-stiff problems, fixed point iteration often provides an adequate corrector.

With multistep methods, we can adapt not only the time step, but also the order. Very high-order methods may be appropriate when the solution is smooth and we want to either minimize the number of time steps or to meet very strict accuracy requirements. The MATLAB routine `ode113` implements a variable-order Adams-Bashforth-Moulton predictor-corrector solver.

The Adams methods interpolate the function values f; the *backward differentiation formulas* (BDF) instead interpolate y. The next step y_{n+1} is chosen so that the polynomial interpolating (t_{n-k}, y_{n-k}) through (t_{n+1}, y_{n+1}) has derivative at t_{n+1} equal to $f(t_{n+1}, y_{n+1})$. MATLAB's solver `ode15s` uses a variable-order *numerical differentiation formula* (a close relative of BDF). The `ode15s` code would be a typical first choice for stiff problems.

Example: the van der Pol equation

The van der Pol oscillator is a model nonlinear differential equation that shows up rather quickly in most discussions of the topic. The differential equation is:

$$x'' - \mu(1 - x^2)x' + x = 0.$$

When $\mu = 0$, this is a simple harmonic oscillator, and solutions have the form

$$x(t) = x(0)\cos(t) + x'(0)\sin(t)$$

When μ is nonzero, the picture gets slightly more complicated. You may or may not remember from a physics, calculus, or ODE class that the ODE

$$x'' + bx' + x = 0$$

has decaying oscillating solutions for $b > 0$ and exponentially growing solutions for $b < 0$. The coefficient b is interpreted as damping (with $b < 0$ corresponding to "anti-damping" behavior where solutions gain energy over time). In the case of the van der Pol equation, b is replaced by a nonlinear term which is negative when $|x| < 1$ and positive when $|x| > 1$. So the effective behavior, seen in practice, is that there is a balance between growth behavior for small x and decay behavior for larger x. The result is that the solution bounces back and forth between slow motions for $x > 1$ and $x < -1$ with fast transitions in between, and the speed of those transitions is governed by the magnitude of μ.

Now let's write code to actually solve the system. The ODE solvers in MATLAB require that we express our problem as a first-order system in standard form, which we do by introducing the auxiliary variable $y = x'$:

$$\begin{bmatrix} x \\ y \end{bmatrix}' = \begin{bmatrix} y \\ \mu(1-x^2)y - x \end{bmatrix} = f_{vdp}(x, y, \mu).$$

This is a demo system in the MATLAB documentation, so MATLAB already has a function `vanderpoldemo(x,y,mu)` to evaluate the right hand side f_{vdp} of this system. Based on the advice given in lecture, we should choose `ode45` if the problem is non-stiff (μ modest) and `ode15s` if the problem is stiff (μ large). Our script, `runvdp` will compute the solution using both methods and compare both the results and the timings. For $\mu = 1$, we find that both solvers perform adequately; for $\mu = 100$, `ode45` takes 5 seconds and 26368 steps while `ode15s` takes 0.45 seconds and 761 steps. For $\mu = 200$, `ode45` takes 20 seconds and over 104868 steps, while `ode 15s` takes 0.50 seconds. and just over 1010 steps. Both solvers return results that are nearly indistinguishable visually (see Figure 1).

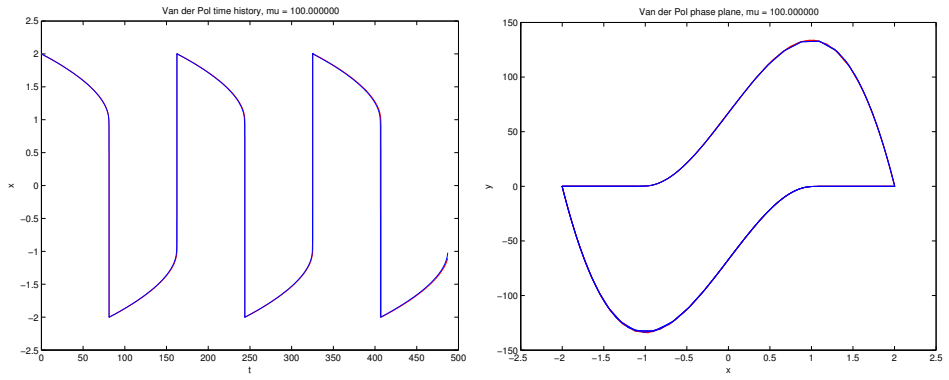

Figure 1: Van der Pol solutions for $\mu = 100$, via `ode45` (red) and `ode15s` (blue). The left plot shows x vs t; right shows $x(t)$ vs $y(t) = x'(t)$.

Problems to ponder

1. Describe how to use `ode45` and `plot` to display the solution to an initial value problem

$$mx'' + bx' + kx = \begin{cases} 1, & 0 \le t \le 1 \\ 0, & 1 \le t \le t_{\text{final}} \end{cases}$$

   where $x(0) = x'(0) = 0$.

2. For implicit methods like backward Euler or the trapezoidal rule, we need to solve a nonlinear equation at each update. For backward Euler, for example, we have

$$y_{n+1} - hf(y_{n+1}) - y_n = 0.$$

   What is Newton's iteration for computing y_{n+1} given y_n?

3. For a *non-stiff* problem where f' is not too large, note that we could also use fixed point iteration:

$$y_{n+1}^{\text{new}} = y_n + hf(y_{n+1}^{\text{old}}).$$

   Assuming f is Lipschitz with constant L, show this fixed point iteration converges when $Lh < 1$.

```
% runvdp(mu)
%  Demonstrate relative performance of ode45 and ode15s for
%  the van der Pol oscillator as a function of mu.
%
function runvdp(mu)

  tau = (3-2*log(2))*mu + 4.676*mu^(-1/3); % Estimated period
  tspan = [0, 3*tau];                      % Go about 3 cycles
  y0 = [2; 0];                             % Initial conditions
  opt = odeset('Stats', 'on');             % Print diagnostics
  ode = @(t,y) vanderpoldemo(t,y,mu);      % MATLAB already has f_vdp

  fprintf('\n--- ODE45 solve ---\n');
  tic; [tn,yn] = ode45(ode, tspan, y0, opt); toc

  fprintf('\n--- ODE15s solve ---\n');
  tic; [ts,ys] = ode15s(ode, tspan, y0, opt); toc

  figure(1);
  plot(tn,yn(:,1), 'r-', ts,ys(:,1), 'b-');
  xlabel('t');
  ylabel('x');
  title(sprintf('Van der Pol time history, mu = %f', mu));

  figure(2);
  plot(yn(:,1), yn(:,2), 'r-', ys(:,1), ys(:,2), 'b-');
  xlabel('x');
  ylabel('y');
  title(sprintf('Van der Pol phase plane, mu = %f', mu));
```

Figure 2: Script to compare `ode45` and `ode15s` for the van der Pol oscillator.

4. Suppose we have used a time-stepping algorithm to compute $y_k \approx y(t_k)$. To get from step k to $k+1$, consider using either one or two steps of forward Euler:

$$\begin{aligned} y_{k+1} &= y_k + hf(y_k) \\ z_{k+1/2} &= y_k + \frac{h}{2} f(y_k) \\ z_{k+1} &= z_{k+1/2} + \frac{h}{2} f(z_{k+1}) \end{aligned}$$

   Write the first term in a Taylor expansion for the local error $y_{k+1} - u(t_{k+1})$, where u is the solution to the initial value problem

$$u'(t) = f(u(t)), \quad u(t_k) = y_k$$

   How could you combine z_{k+1} and y_{k+1} to estimate this local error?

5. Consider the test equation

$$y' = \lambda y.$$

   Suppose we approximate the solution at time $t_k = kh$ using forward Euler. Show that if $\hat{y}_k$ is the kth step of the forward Euler method, then $\hat{y}_k = z(t_k)$ where $z(t)$ is the solution to the modified differential equation

$$z' = \hat{\lambda} z.$$

   How is $\hat{\lambda}$ related to λ? Repeat the exercise for backward Euler.

6. Based on the description in these notes, derive the backward differentiation formula based on interpolation of y at points t_n and t_{n+1}.