Week 12: Monday, Apr 18

Logistics

- HW 6 is due at 11:59 tonight.
- HW 7 is posted, and will be due in class on 4/25.
- The prelim is graded. An analysis and rubric are on CMS.

Problem du jour

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?

Answer: The Newton iteration is

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

We could also do something less expensive by computing and factoring an approximation to the Jacobian once and re-using it until the convergence becomes too slow. If you use MATLAB's implicit ODE solvers, you can specify the Jacobian (or, if specifying the Jacobian directly is expensive, you can specify the sparsity pattern of the Jacobian).

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

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

In order to do Newton, fixed point, or any other iteration, though, we need a good initial guess. But we can get a good initial guess by polynomial interpolation through previous points. In the context of ODE solvers, this is called a *predictor*, which we pair together with a *corrector* to get the implicit method.

An aside: higher-order differential equations

So far, we have been talking about first-order equations. It is worth spending a moment reminding ourselves how to convert higher-order systems to first-order form. For example,

$$my'' + by' + ky = f(t)$$

becomes

$$\begin{bmatrix} y \\ v \end{bmatrix}' = \begin{bmatrix} v \\ f(t) - \frac{b}{m}v - \frac{k}{m}y \end{bmatrix}.$$

More generally, we can always convert high-order equations to first-order form by introducing auxiliary variables. However, it is sometimes better to recognize that higher-order differential equations are special and treat them directly rather than converting them to the more general form. This is done, for example, in the Newmark family of methods commonly used to time-step finite element simulations (which are second-order in time).

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

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

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 methods) 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.