Week 14: Monday, Apr 30

Introduction

So far, our discussion of ODE solvers has been rather abstract. We've talked some about how to evaluate ODE solvers, how ODE solvers choose time steps in order to control error, and the different classes of ODE solvers that are available in MATLAB. We have not, however, tackled any concrete example problems other than trivial linear test problems. In part, this is because I have a hard time writing solutions and drawing believable pictures at the board for anything but trivial test problems. So today, let's try a MATLAB-oriented lecture.

1 A ballistics problem

The next problem is a classic of both scientific computing and certain classes of computer games: ballistics calculations. We will solve this problem via a *shooting* method: that is, we will base our solution method on initial value problem solvers, and we try to choose initial conditions in order to satisfy the problem constraints.

1.1 Model without air drag

Let's start with a simple model, one that many of you have probably seen in an introductory physics class. A projectile is fired from a launcher with fixed speed; as a function of the launch angle, where will it hit the ground? In the simplest version of this model, the only force acting on the ball after launch is gravity, so Newton's law tells us

$$m\mathbf{a} = -mg\mathbf{e}_y$$

where $\mathbf{e}_y$ is a unit vector in the vertical direction, m is the particle mass, and $\mathbf{a} = \mathbf{x}'' = (x'', y'')$ is the acceleration vector. If our launcher is positioned at the origin, then the initial conditions for a launch with speed s at angle θ are

$$\begin{aligned} x(0) &= 0, & x'(0) &= s\cos(\theta) = v_{0,x}, \\ y(0) &= 0, & y'(0) &= s\sin(\theta) = v_{0,y}. \end{aligned}$$

Subject to these initial conditions, we can compute the solution analytically:

$$x(t) = v_{0,x} t$$
$$y(t) = v_{0,y} t - gt^2/2.$$

The trajectory returns to the ground at time $t_{\mathrm{final}} = 2v_{0,y}/g$ and at position

$$x_{\mathrm{final}} = x(t_{\mathrm{final}}) = \frac{2}{g} v_{0,y} v_{0,x} = \frac{s^2}{g} \sin(2\theta).$$

Therefore, we can reach a target at distances $d \leq s^2/g$ with launch angles θ that satisfy $\sin(2\theta) = gd/s^2$. In general, if we can hit the target at all there will be two trajectories that work. One will have angle between 0 and $\pi/4$, while the other has an angle between $\pi/4$ and $\pi/2$.

1.2 Model with air drag

In practice, projectiles are affected not only by gravity, but also by air resistance. For a reasonable range of projectiles, and assuming that the projectile does not go so high that changes in atmospheric pressure are an issue, the drag force due to air resistance acts in the direction opposite the velocity, with a magnitude proportional to the square of the velocity. That is,

$$m\mathbf{a} = -mg\mathbf{e}_y - mc\|\mathbf{v}\|\mathbf{v}.$$

If we also consider a constant horizontal wind velocity $w\mathbf{e}_x$, we arrive at

$$m\mathbf{a} = -mg\mathbf{e}_y - mc\|\mathbf{v} - w\mathbf{e}_x\|(\mathbf{v} - w\mathbf{e}_x).$$

The coefficient c is a complicated function of the size, shape, and mass of the projectile, along with the temperature and pressure of the air. For the moment, we will suppose it is simply given.

The differential equation without air drag was simple enough for us to analyze by hand. This model is harder, so we turn to numerical methods. To use MATLAB's ODE solvers, we need to put the model into first-order form:

$$\begin{bmatrix} \mathbf{x} \\ \mathbf{v} \end{bmatrix}' = \begin{bmatrix} \mathbf{v} \\ -g\mathbf{e}_y - c\|\mathbf{v} - w\mathbf{e}_x\|(\mathbf{v} - w\mathbf{e}_x) \end{bmatrix} = f_{\mathrm{ballistics}}(\mathbf{x}, \mathbf{v}, g, c, w).$$

We code this model in a MATLAB script `fball` (Figure 1).

When setting up an ODE model, there are usually ample opportunities to make nonphysical assumptions or errors in mathematics or programming. For this reason, it is good practice to sanity-check our computations. In our case, there are two natural checks:

1. If the coefficient c is zero, this problem simplifies to the model we discussed in the last section. Our numerical ODE solver should therefore recover the same solution we found in our hand analysis.

2. If the coefficient c is positive, then air drag slows down the projectile, so it should not go as far as it would go in the case of $c = 0$.

We check both of these behaviors with the script `testball1` (Figure 3). A visual comparison of trajectories computed with and without air drag is showin in Figure 2.

1.3 Computing points of impact

Now that we have a code that we believe gives plausible trajectories, we need to figure out where those trajectories hit the ground as a function of the launch angle. In order to do this, we want not to stop the simulation at a specific *time*, but when a specific *event* occurs: namely, when the vertical component of the projectile position tries to go from positive to negative. MATLAB provides *event detection* as part of the ODE suite: we define an interesting event in terms of a zero crossing of some test function of the state vector (position and velocity), and we do something special when at points when the test function goes from positive to negative or vice versa. In our case, we are interested in when the y position component of the solution goes from positive to negative values, and we would like to terminate the computation when that occurs. Then we want to extract the final x position. This computation is done in `ftarget` using a test function `hitground` to detect the impact event (Figure 4).

1.4 Computing targeting solutions

At this point, we are interested in the function $f_{\text{target}}(\theta)$ that computes the impact distance as a function of the launch angle. For the case when $c = 0$, we know that this function is proportional to $\sin(2\theta)$. The case $c > 0$ is a little more complicated, but even in this case, $f_{\text{target}}(\theta)$ is very smooth. We

```
% yp = fball(t,y,opt)
%   Compute the right hand side of an ODE for projectile motion
%   in the presence of wind and air drag.  The opt structure should
%   describe a scaled air drag coefficient (c), the gravitational field (g),
%   and the horizontal wind speed (w).
%
function yp = fball(t,y,opt)

  g = opt.g;   % Gravitational field
  c = opt.c;   % Scaled drag coefficient
  w = opt.w;   % Horizontal wind speed

  % Unpack position and velocity
  x = y(1:2);
  v = y(3:4);

  % Velocity relative to wind
  vv    = v;
  vv(1) = vv(1)-w;

  % Compute acceleration
  a    = -c*norm(vv)*vv;
  a(2) = a(2)-g;

  % Return
  yp = [v; a];
```

Figure 1: Right-hand side for ballistics model.

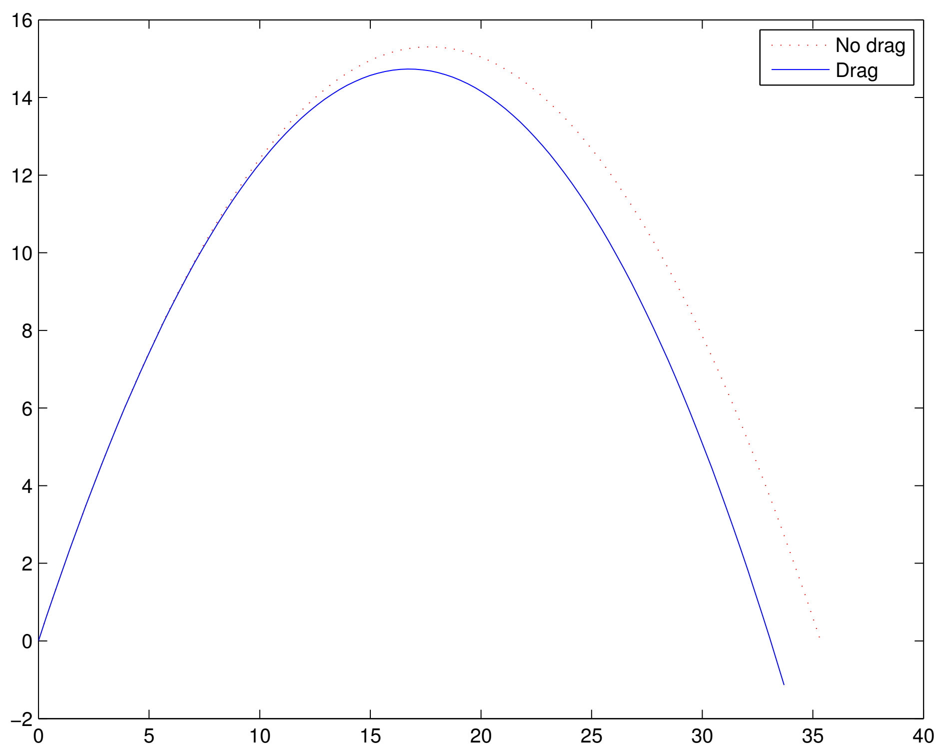

Figure 2: Trajectories with and without drag.

```
% -- Sanity check trajectories computed with and without drag --

theta = pi/3;
v0 = s*[cos(theta); sin(theta)];

% Compute the reference trajectory (absent air resistance)
%   x(y) = s*cos(theta)*t
%   y(t) = -g*t^2/2 + s*sin(theta)*t
% up to time tfinal = 2*s*sin(theta)/g
%
tfinal = 2*v0(2)/g;
tref = linspace(0,tfinal)';
xref = v0(1)*tref;
yref = (v0(2)-g/2*tref).*tref;

% Compute the same reference trajectory with ode45
y0 = [0; 0; v0];
refopt = opt;
refopt.c = 0;
[tout,yout] = ode45(@(t,y) fball(t,y,refopt), tref, y0);

% Compute a similar trajectory with air drag on (no wind)
dopt = opt;
dopt.w = 0;
[toutd,youtd] = ode45(@(t,y) fball(t,y,dopt), tref, y0);

% Do a comparison between analytical and numerical solutions (no drag)
fprintf('Max x error: %g\n', norm( xref-yout(:,1), inf));
fprintf('Max y error: %g\n', norm( yref-yout(:,2), inf));

% Visually compare solutions for drag and no drag
plot(xref, yref, 'r:', youtd(:,1), youtd(:,2), 'b-');
legend('No drag', 'Drag');
```

Figure 3: Test script to check ballistics ODE.

```
% xfinal = ftarget(thetas, opt)
%   Compute impact points as a function of angles for the ballistics
%   ODE with parameters given in opt. In addition to the basic ODE
%   parameters, opt.s should be set to the launch speed.
%
function xfinal = ftarget(thetas, opt)

  xfinal = 0*thetas;
  for j = 1:length(thetas)
    theta = thetas(j);
    v0 = (opt.s)*[cos(theta); sin(theta)];
    y0 = [0; 0; v0];
    tfinal = 2*v0(2)/opt.g;
    odeopt = odeset('Events', @hitground);
    [tout,yout] = ode45(@(t,y) fball(t,y,opt), [0 tfinal], y0, odeopt);
    xfinal(j) = yout(end,1);
  end

function [value,isterminal,direction] = hitground(t,y)

  value      = y(2);  % Check for zero crossings of y position
  isterminal = 1;     % Terminate on a zero crossing
  direction  = -1;    % We only care about crossing y > 0 to y < 0
```

Figure 4: Compute point of impact as a function of θ.

could compute f_{target} and derivatives at arbitrary points based on the ODE [1], but the evaluation costs a little bit; if we included a few more complicating factors in our evaluations, we might reasonably be reluctant to do too many trajectory computations with `ftarget`. So let's use the tools that we build previously, and fit a polynomial approximation f_{target} at a Chebyshev grid on $[0, \pi/2]$.

The function $f_{\text{target}}(\theta)$ in general is a unimodal function: it increases on $[0, \theta_{\max}]$ and then decreases on $[\theta_{\max}, \pi/2]$. If we want to hit a target at distance d, then, there are two things that can happen:

1. If $d > f_{\text{target}}(\theta_{\max})$ then we cannot hit the target at any angle.
2. If $d < f_{\text{target}}(\theta_{\max})$, then there are generally two solutions to the equation $f_{\text{target}}(\theta) - d = 0$: one on the interval $[0, \theta_{\max}]$ and the other on the interval $[\theta_{\max}, \pi/2]$.

We can find the two solutions, if they exist, using MATLAB's `fzero` function (Figure 5). In each step, we work with a polynomial approximation to f_{target} rather than working with f_{target} directly.

1.5 An example trajectory

As an example, let's consider a concrete example with a high drag coefficient ($c = 0.05$ m^{-1}) and some wind ($w = -2.5$ m/s). We want to hit a target at distance $d = 10$ m. The trajectories computed for the two solution angles are shown in Figure 6; the residual error $f_{\text{target}}(\theta)$ is on the order of 10^{-7} for this problem, which is almost certainly smaller than errors due to uncertainty in the model parameters.

2 Particle in a box

In the previous example, we solved a two-point boundary value problem by shooting (the two points being the point of launch and of impact). In this

[1] The derivative of the trajectory with respect to the launch angle θ can be computed in terms of an extended ODE system, a so-called *variational equation*. Using this variational equation, we could compute derivatives of the impact location as a function of θ. But using a polynomial interpolant will turn out to be a simpler way of approximating the function and its derivatives.

```
% thetas = find_angle(d, opt)
%   Find angles to hit a target at distance d in the ballistics problem.
%   If the target is unreachable, give an error message.

function [thetas] = find_angle(d, opt)

  % Fit a Chebyshev polynomial to the targeting behavior
  [D,z] = cheb(20);
  thetac = (z+1)*pi/4;
  impactsc = 0*thetac;
  for k = 1:length(thetac)-1
    impactsc(k) = ftarget(thetac(k), opt);
  end

  % Find the farthest-traveling trajectory
  zcrit = chebopt(impactsc);
  topt  = chebeval(thetac,    zcrit);
  xopt  = chebeval(impactsc, zcrit);

  % If we fall below that point, quit
  if d > xopt, error('Target out of range'); end
  g = @(z) chebeval(impactsc, z)-d;
  zs = [ fzero(g, [-1,zcrit]); fzero(g, [zcrit,1]) ];
  thetas = chebeval(thetac, zs);
```

Figure 5: Find targeting angles for a given distance.

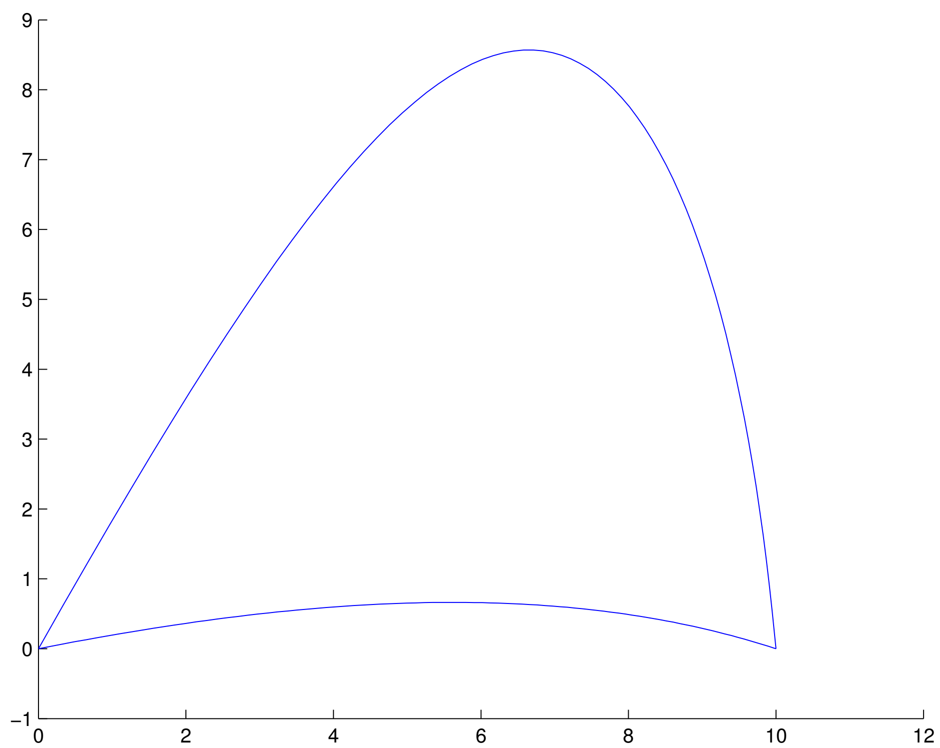

Figure 6: Sample targeting solutions with wind and a high drag coefficient.

example, we will use a finite difference method. The problem we want to solve is a classic example model in quantum mechanics: the particle in a box.

The basic equation of quantum mechanics is Schrödinger's equation; for problems with one space variable, this is

$$H\psi = \left(-\frac{\hbar}{2m} \frac{d^2}{dx^2} + V(x) \right) \psi = E\psi.$$

This is an eigenvalue problem: we want to find E such that the equation has a nontrivial solution, and there are only a discrete set of such E. For the duration of this discussion, let us assume $\hbar/2m = 1$. Now, suppose we have a potential energy $V(x)$ which is zero on $[0, 1]$ and ∞ elsewhere. The squared wave function ψ represents the probability that a particle at energy E is at any given position, and particles can't make it past the infinite energy barriers at 0 and 1, so we actually have a two-point boundary value problem:

$$\begin{aligned} \frac{d^2\psi}{dx^2} &= E\psi, \quad x \in (0, 1) \\ \psi(0) &= 0 \\ \psi(1) &= 0. \end{aligned}$$

A little thought shows the appropriate solutions are $\psi_k = \sin(k\pi x)$ and $E_k = (k\pi)^2$. Let's suppose we did not know that and look at a way to compute the solution numerically.

The standard second-order accurate approximation of a second derivative at a point x is

$$\psi''(x) \approx \frac{\psi(x-h) - \psi(x) + \psi(x+h)}{h^2}.$$

Now suppose we have a regular mesh of points $x_0 = 0$ to $x_{N+1} = 1$, with $x_j = jh$, $h = 1/(N + 1)$. Then we can compute

$$-\psi''(x_i) \approx \frac{-\psi(x_{i-1}) + 2\psi(x_i) - \psi(x_{i+1})}{h^2}$$

Therefore, we can compute an approximation $u_i \approx \psi(x_i)$ by approximating the differential equation at the interior points $x_1, \ldots, x_N$:

$$h^{-2} \left[-u_{i-1} + 2u_i - u_{i+1} \right] - Eu_i = 0.$$

At the end points, we use the boundary conditions $u_0 = u_{N+1} = 0$, which gives us the end conditions

$$
\begin{aligned}
h^{-2}\left[2u_1 - u_2\right] - Eu_1 = 0. \\
h^{-2}\left[-u_{N-1} + 2u_N\right] - Eu_N = 0.
\end{aligned}
$$

Putting everything together, we can write the discrete problem concisely as

$$(h^{-2}T - E)u = 0.$$

where T is the standard tridiagonal stencil

$$
T = \begin{bmatrix}
2 & -1 & & & & \\
-1 & 2 & -1 & & & \\
& -1 & 2 & -1 & & \\
& & \ddots & \ddots & \ddots & \\
& & & -1 & 2 & -1 \\
& & & & -1 & 2
\end{bmatrix}
$$

Therefore, the eigenvalues and eigenvectors for the continuous problem can be approximated by eigenvalues and eigenvectors for a discrete problem. The function `particlebox` (Figure 7) computes the first four eigenvalues / energy levels using this approximation using a finite difference mesh with N interior points. Because we know the exact solutions in this problem, it is easy for us to assess the convergence of our code as a function of h; we do this with the script `particleboxcvg` (Figure 8). Using `particleboxcvg`, we can see that the smallest eigenvalue of the discrete problem estimates the continuous eigenvalue to accuracy $O(h^2)$.

```
% E = particlebox(N)
%   Compute the first four energy levels for the "particle in a box"
%   model using a finite-difference discretization for d^2/dx^2
%   with N interior mesh points.

function E = particlebox(N)

  % Points go 0 to N+1; 0 and N+1 satisfy BCs
  h = 1/(N+1);
  T = -diag(ones(N-1,1),-1) + 2*eye(N) - diag(ones(N-1,1),1);

  % Estimate eigenvalues and eigenvectors
  [V,D] = eig(T/h^2);
  E = diag(D);

  % Compute the first four energy levels
  E = E(1:4);
```

Figure 7: Finite difference computation of the first four energy levels for a particle in a box.

```
% Do a simple convergence study
N = 10;
h = []; E = [];
for j = 1:5
  h(j) = 1/(N+1);
  E(:,j) = particlebox(N);
  N = N*2;
end
loglog(h, pi^2-E(1,:));

% Rate of convergence
est_order = log( (pi^2-E(1,end-1))/(pi^2-E(1,end)) )/log(2);
fprintf('Estimated order of convergence: %f\n', est_order);
```

Figure 8: Convergence analysis of a finite difference computation of the first four energy levels for a particle in a box.