Week 13: Finite Difference Methods
APPM 2460

1 Introduction

In Finite Difference based techniques, we discretize our domain into a set of points, and then at each point we use the values around it to match the derivatives of interpolating polynomials.

2 Approximating the Derivative

Suppose we had an interval $t \in [0,1]$ which we represent with a bunch of points $t_0, t_1, t_2, \ldots, t_N$ that are a distance h apart from each other. If we want to define a function $x(t)$ over that interval, we can find $x(t_0), x(t_1), x(t_2), \ldots, x(t_N)$. We effectively have a bunch of values of x that we know at these specific points.

Now suppose at some point t_n in the middle of that interval we want to approximate the value of $x'(t_n)$. How well can we do?

Consider the Taylor series for $x(t_n)$ evaluated at t_{n+1}. That's

$$\begin{aligned} x(t_{n+1}) &= x(t_n) + h \cdot x'(t_n) + \frac{h^2}{2} \cdot x''(t_n) + \mathcal{O}(h^3) \\ x'(t_n) &= \frac{1}{h}(x(t_{n+1}) - x(t_n)) - \frac{h}{2} \cdot x''(t_n) + \mathcal{O}(h^2) \end{aligned} \tag{1}$$

so we can approximate the first derivative *to order* $\mathcal{O}(h)$ with $x'(t_n) \approx \frac{1}{h}(x(t_{n+1}) - x(t_n))$. That's not surprising; it's just rise over run.

Now, though, we add the Taylor series expanded at $x(t_{n-1})$. Why not?

$$\begin{aligned} x(t_{n-1}) &= x(t_n) - h \cdot x'(t_n) - \frac{h^2}{2} \cdot x''(t_n) + \mathcal{O}(h^3) \\ x'(t_n) &= \frac{1}{h}(x(t_n) - x(t_{n-1})) + \frac{h}{2} \cdot x''(t_n) + \mathcal{O}(h^2) \end{aligned} \tag{2}$$

Again, not too surprising. But by adding the two together (and dividing by two), we get

$$x'(t_n) = \frac{1}{2h}(x(t_{n+1}) - x(t_{n-1})) + \mathcal{O}(h^2) \tag{3}$$

the h term cancels and we are left with an $\mathcal{O}(h^2)$ method!

We call this a *second order centered finite difference stencil*. The approximation is "second order" since the error is dominated by h^2 and it is "centered difference" since the place t_n where we're approximating the derivative x' is centered between the set of time values t_{n+1} and t_{n-1} we will use. The constants multiplying the terms $x(t_{n+1})$ and $x(t_{n-1})$ are called "weights" of the a finite difference "stencil" (arrangement of function values).

It's not too hard to play games like these with other Taylor expansions to get higher order approximations to the first derivative or to approximate other derivatives.

3 Some stencil weights

second order centered stencil for x'		$-1/2$	0	$1/2$	
second order centered stencil for x''		1	-2	1	
second order left-edge stencil for x'	$-3/2$	2	$-1/2$		
second order left-edge stencil for x''	2	-5	4	-1	
second order right-edge stencil for x'			$1/2$	-2	$3/2$
second order right-edge stencil for x''		-1	4	-5	2

(4)

We need one-sided stencils to adequately deal with some boundary geometries.

4 Applying to a simple example

4.1 The First Derivative

Suppose now $t \in [0, 1]$ and $x(t) = t$. We would expect $x'(t) = 1$ and $x''(t) = 0$.
To use the stencil defined in (3) to approximate $x'(t)$ in this interval, we need to define a matrix `Dt` that contains the weights of the stencil.

To create this matrix, we will generate a sparse matrix using the command `spdiags(B, d, m, n)` which (from Matlab's documentation) "creates an m-by-n sparse matrix by taking the columns of B and placing them along the diagonals specified by d."

```
h = 0.01;
t = (0:h:1)' ; N = numel(t);
x = t;

Dt = spdiags([-1/2, 0, 1/2]./h .* ones(N,1), [-1 0 1], N, N);
Dt(1,1:3) =   [-3/2, 2,-1/2]./h; % We need to deal with boundaries somehow.
Dt(N,N-2:N) = [ 1/2,-2, 3/2]./h;

dxdt = Dt * x;

plot(t, dxdt); axis([0 1 -2 2]);
```

Here, the main diagonal is 0, the first superdiagonal is 1, the first subdiagonal is -1.

It may be worthwhile here to use `spy(Dt)` to see what the matrix looks like. This allows us to check that we have defined the desired finite difference stencil.

4.2 The Second Derivative

Now we can apply this idea to construct a second order central difference stencil to approximate the second derivative in a similar way:

```
D2t = spdiags([1,-2,1]./h^2 .* ones(N,1), [-1 0 1], N, N);
D2t(1,1:4) = [2 -5 4 -1]./h^2;
D2t(N,N-3:N) = [-1 4 -5 2]./h^2;

dxdt2 = D2t * x;

plot(t,dxdt2); axis([0 1 -2 2]);
```

We see that we get the correct derivative terms from this calculation.

Now here's something interesting. We said that this approximation is a "second order method." But we can also notice from the derivation above that the first derivative term that's unaccounted in our calculation is the $x'''(t)$ term. That is, the calculation computes *second order polynomials* exactly. So what happens if we take the matrices we built above and apply them to a polynomial that is higher than second order? For example, let $x(t) = t^5$.

```
h = 0.1;                    % quite large
t = (0:h:1)' ; N = numel(t);
x = t.^5;                   % quite high

D2t = spdiags([1,-2,1]./h^2 .* ones(N,1), [-1 0 1], N, N);
D2t(1,1:4) = [2 -5 4 -1]./h^2;
D2t(N,N-3:N) = [-1 4 -5 2]./h^2;
dx2d2t = D2t * x;

hold on;
plot(t, dx2d2t);
plot(t,20*t.^3); % this is the truth
hold off;
```

It still seems surprisingly good (though it does deviate some). These methods are pretty robust!

5 A more complicated example: solving differential equations

So, what good is this? Well, suppose we have the forced spring system

$$\begin{aligned} & x''(t) + 4x'(t) + 4x(t) = \cos(t) \\ & \begin{cases} x(0) = 1 \\ x'(0) = 0 \\ 0 \le t \le 10\pi \end{cases} \end{aligned} \tag{5}$$

Once we have the operators built, we can solve this system pretty easily.

```
% first the preliminaries (similar to before but with larger t)
h = 0.01;
t = (0:h:10*pi)' ; N = numel(t);

Dt = spdiags([-1/2, 0, 1/2]./h .* ones(N,1), [-1 0 1], N, N);
Dt(1,1:3) =   [-3/2, 2,-1/2]./h; % We need to deal with boundaries somehow.
Dt(N,N-2:N) = [ 1/2,-2, 3/2]./h;
D2t = spdiags([1,-2,1]./h^2 .* ones(N,1), [-1 0 1], N, N);
D2t(1,1:4) = [2 -5 4 -1]./h^2;
D2t(N,N-3:N) = [-1 4 -5 2]./h^2;

% now we apply them to the problem.

% D2t * x + 4 * Dt * x + 4 * x = cos(t)
% ( D2t + 4 * Dt + 4 * eye(N) ) * x = cos(t)

Op = ( D2t + 4 * Dt + 4*eye(N) );
```

```
f = cos(t);

% Boundary conditions! We want to satisfy the forcing exactly in the interior,
% so we use the first and last rows to specify the initial conditions.
Op(1,:) = zeros(1,N); Op(1,1) = 1;
Op(N,:) = Dt(1,:);
f([1,N]) = [1,0];

x = Op\f;

plot(t,x);
```

If we want to compare this to the truth, we can do so with

```
exact = @(t)((0.88 + 1.6*t).*exp(-2*t) + 0.12*cos(t) + 0.16*sin(t));
hold on;
plot(t,x);
plot(t, exact(t));
legend('FD approximation','exact solution');
hold off;
```

We can make h smaller and do better, but you'll find that if you decrease it too much you'll run out of memory! So a higher order scheme might be even more valuable.

APPM 2460 Homework Week 10

Submit a published pdf of your script and any other supporting code needed to solve the following problem to Canvas by Monday, April 1 at 11:59 p.m. See the 2460 webpage for formatting guidelines.

Consider a mass-spring system with mass $m = 1$, damping $b = 0$, and a spring constant k (this is described mathematically by a forced, undamped harmonic oscillator equation). If the mass is pushed from rest with an initial velocity of 1 the IVP to solve is

$$\ddot{x} + kx = \cos(t), \quad x(0) = 0,\ \dot{x}(0) = 1 \tag{6}$$

(a) Suppose that there are two springs that can be used in this system. The first spring has constant $k_1 = 1$ and the second has constant $k_2 = 2$. Use finite differences over the range $t = [0, 100]$ to solve (6) for both spring constants.

- *Optional*: You can find higher order finite difference stencils online at

  `https://en.wikipedia.org/wiki/Finite_difference_coefficient`

  Try solving the differential equation with a fourth or sixth order method. Note that you'll need more rows to be one-sided stencils!

(d) Plot each of the solutions. What phenomena do you observe?

(c) Which spring is more likely to break in this scenario? Why?