AM 205: lecture 11

- Last time: Numerical integration
- Today: Numerical differentiation, numerical solution of ordinary differential equations

Finite Difference Approximations

Given a function $f : \mathbb{R} \to \mathbb{R}$

We want to approximate derivatives of f via simple expressions involving samples of f

As we saw in Unit 0, convenient starting point is Taylor's theorem

$$f(x+h) = f(x) + f'(x)h + \frac{f''(x)}{2}h^2 + \frac{f'''(x)}{6}h^3 + \cdots$$

Finite Difference Approximations

Solving for $f'(x)$ we get the forward difference formula

$$\begin{aligned} f'(x) &= \frac{f(x+h) - f(x)}{h} - \frac{f''(x)}{2}h + \cdots \\ &\approx \frac{f(x+h) - f(x)}{h} \end{aligned}$$

Here we neglected an $O(h)$ term

Finite Difference Approximations

Similarly, we have the Taylor series

$$f(x-h) = f(x) - f'(x)h + \frac{f''(x)}{2}h^2 - \frac{f'''(x)}{6}h^3 + \cdots$$

which yields the backward difference formula

$$f'(x) \approx \frac{f(x) - f(x-h)}{h}$$

Again we neglected an $O(h)$ term

Finite Difference Approximations

Subtracting Taylor expansion for $f(x-h)$ from expansion for $f(x+h)$ gives the centered difference formula

$$\begin{aligned} f'(x) &= \frac{f(x+h)-f(x-h)}{2h} - \frac{f'''(x)}{6}h^2 + \cdots \\ &\approx \frac{f(x+h)-f(x-h)}{2h} \end{aligned}$$

In this case we neglected an $O(h^2)$ term

Finite Difference Approximations

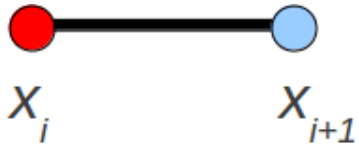
Adding Taylor expansion for $f(x-h)$ and expansion for $f(x+h)$ gives the centered difference formula for the second derivative

$$\begin{aligned} f''(x) &= \frac{f(x+h)-2f(x)+f(x-h)}{h^2} - \frac{f^{(4)}(x)}{12}h^2 + \cdots \\ &\approx \frac{f(x+h)-2f(x)+f(x-h)}{h^2} \end{aligned}$$

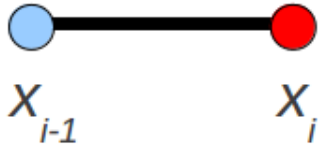
Again we neglected an $O(h^2)$ term

Finite Difference Stencils

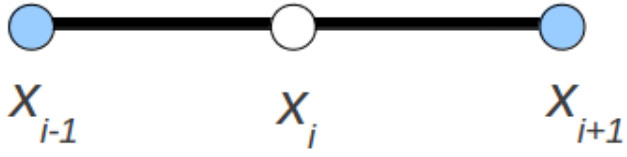
Forward diff.



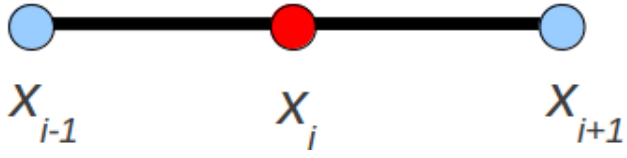
Backward diff.



Centered diff.
1^{st} derivative



Centered diff.
2^{nd} derivative



Finite Difference Approximations

We can use Taylor expansion to derive approximations with higher order accuracy, or for higher derivatives

This involves developing F.D. formulae with "wider stencils," *i.e.* based on samples at $x \pm 2h, x \pm 3h, \ldots$

But there is an alternative that generalizes more easily to higher order formulae:

Differentiate the interpolant!

Finite Difference Approximations

Linear interpolant at $\{(x_0, f(x_0)), (x_0 + h, f(x_0 + h))\}$ is

$$p_1(x) = f(x_0)\frac{x_0 + h - x}{h} + f(x_0 + h)\frac{x - x_0}{h}$$

Differentiating p_1 gives

$$p_1'(x) = \frac{f(x_0 + h) - f(x_0)}{h},$$

which is the forward difference formula

Question: How would we derive the backward difference formula based on interpolation?

Finite Difference Approximations

Similarly, quadratic interpolant, p_2, from interpolation points $\{x_0, x_1, x_2\}$ yields the centered difference formula for f' at x_1:

- Differentiate $p_2(x)$ to get a linear polynomial, $p_2'(x)$
- Evaluate $p_2'(x_1)$ to get centered difference formula for f'

Also, $p_2''(x)$ gives the centered difference formula for f''

Note: Can apply this approach to higher degree interpolants, and interp. pts. need not be evenly spaced

Finite Difference Approximations

So far we have talked about finite difference formulae to approximate $f'(x_i)$ at some specific point x_i

Question: What if we want to approximate $f'(x)$ on an interval $x \in [a, b]$?

Answer: We need to simultaneously approximate $f'(x_i)$ for x_i, $i = 1, \ldots, n$

Differentiation Matrices

We need a map from the vector $F \equiv [f(x_1), f(x_2), \ldots, f(x_n)] \in \mathbb{R}^n$ to the vector of derivatives $F' \equiv [f'(x_1), f'(x_2), \ldots, f'(x_n)] \in \mathbb{R}^n$

Let $\widetilde{F}'$ denote our finite difference approximation to the vector of derivatives, *i.e.* $\widetilde{F}' \approx F'$

Differentiation is a linear operator[1], hence we expect the map from F to $\widetilde{F}'$ to be an $n \times n$ matrix

This is indeed the case, and this map is a differentiation matrix, D

[1] Since $(\alpha f + \beta g)' = \alpha f' + \beta g'$

Differentiation Matrices

Row i of D corresponds to the finite difference formula for $f'(x_i)$, since then $D_{(i,:)}F \approx f'(x_i)$

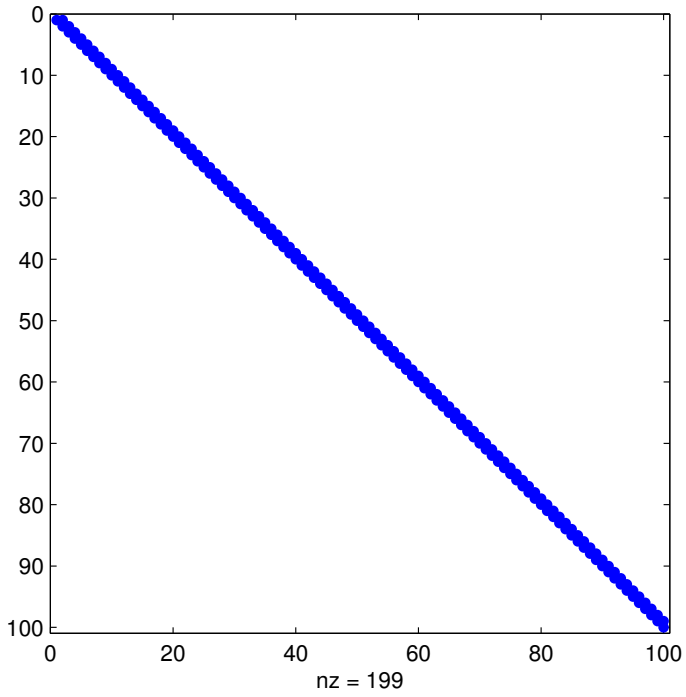
e.g. for forward difference approx. of f', non-zero entries of row i are

$$D_{ii} = -\frac{1}{h}, \quad D_{i,i+1} = \frac{1}{h}$$

This is a sparse matrix with two non-zero diagonals

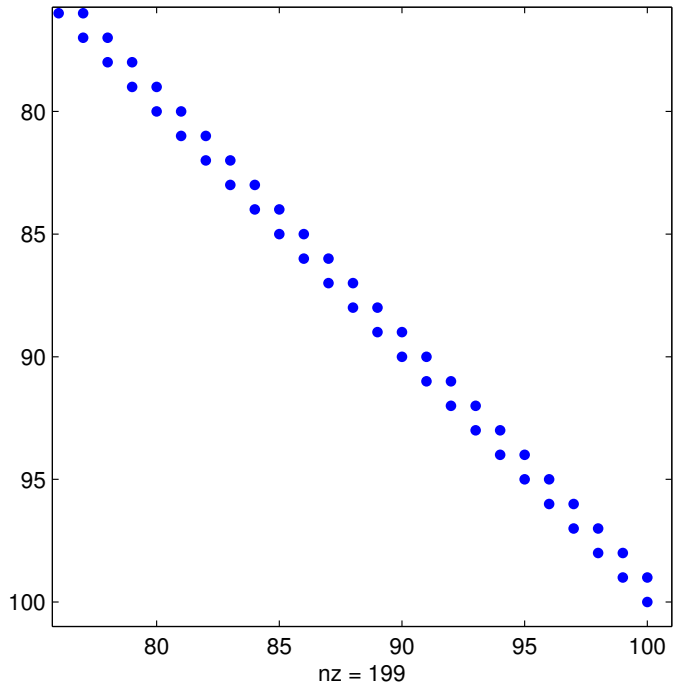
Differentiation Matrices

```
n=100
h=1/(n-1)
D=np.diag(-np.ones(n)/h)+np.diag(np.ones(n-1)/h,1)
plt.spy(D)
plt.show()
```



Differentiation Matrices

But what about the last row?

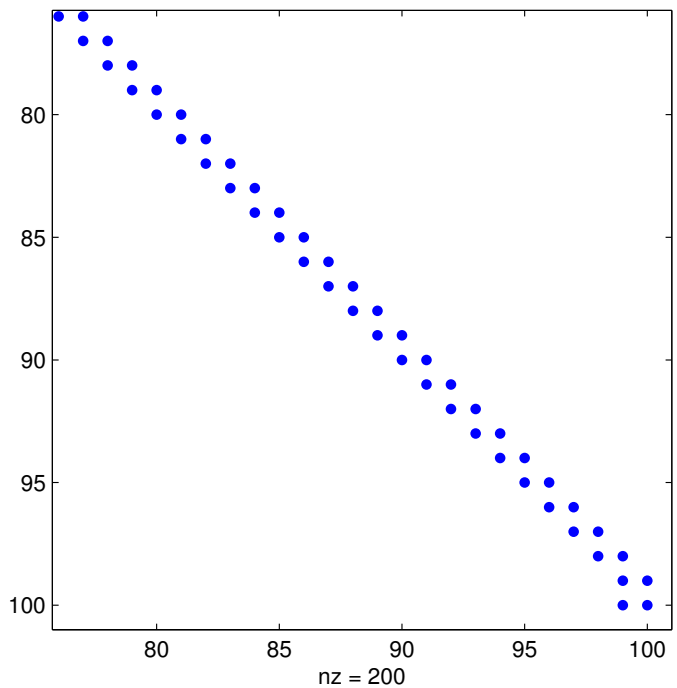


$D_{n,n+1} = \frac{1}{h}$ is ignored!

Differentiation Matrices

We can use the backward difference formula (which has the same order of accuracy) for row n instead

$$D_{n,n-1} = -\frac{1}{h}, \quad D_{nn} = \frac{1}{h}$$



Python demo: Differentiation matrices

Integration of ODE Initial Value Problems

In this chapter we consider problems of the form

$$y'(t) = f(t, y), \quad y(0) = y_0$$

Here $y(t) \in \mathbb{R}^n$ and $f : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$

Writing this system out in full, we have:

$$y'(t) = \begin{bmatrix} y_1'(t) \\ y_2'(t) \\ \vdots \\ y_n'(t) \end{bmatrix} = \begin{bmatrix} f_1(t, y) \\ f_2(t, y) \\ \vdots \\ f_n(t, y) \end{bmatrix} = f(t, y(t))$$

This is a system of n coupled ODEs for the variables $y_1, y_2, \ldots, y_n$

ODE IVPs

Initial Value Problem implies that we know $y(0)$, *i.e.*
$y(0) = y_0 \in \mathbb{R}^n$ is the initial condition

The order of an ODE is the highest-order derivative that appears

Hence $y'(t) = f(t, y)$ is a first order ODE system

ODE IVPs

We only consider first order ODEs since higher order problems can be transformed to first order by introducing extra variables

For example, recall Newton's Second Law:

$$y''(t) = \frac{F(t, y, y')}{m}, \qquad y(0) = y_0, y'(0) = v_0$$

Let $v = y'$, then

$$\begin{aligned} v'(t) &= \frac{F(t, y, v)}{m} \\ y'(t) &= v(t) \end{aligned}$$

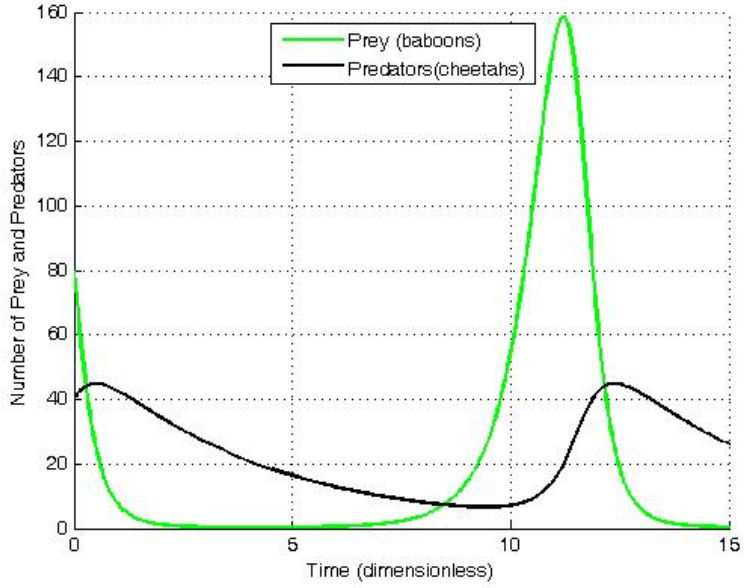
and $y(0) = y_0$, $v(0) = v_0$

ODE IVPs: A Predator–Prey ODE Model

For example, a two-variable nonlinear ODE, the Lotka–Volterra equation, can be used to model populations of two species:

$$y' = \begin{bmatrix} y_1(\alpha_1 - \beta_1 y_2) \\ y_2(-\alpha_2 + \beta_2 y_1) \end{bmatrix} \equiv f(y)$$

The α and β are modeling parameters, describe birth rates, death rates, predator-prey interactions



ODEs in Matlab

Both Python and MATLAB have very good ODE IVP solvers

They employ adaptive time-stepping (h is varied during the calculation) to increase efficiency

Python has functions `odeint` (a general purpose routine) and `ode` (a routine with more options)

Most popular MATLAB function is `ode45`, which uses the classical fourth-order Runge–Kutta method

In the remainder of this chapter we will discuss the properties of methods like the Runge–Kutta method

Approximating an ODE IVP

Given $y' = f(t, y)$, $y(0) = y_0$: suppose we want to approximate y at $t_k = kh$, $k = 1, 2, \ldots$

Notation: Let y_k be our approx. to $y(t_k)$

Euler's method: Use finite difference approx. for y' and sample $f(t, y)$ at t_k:[2]

$$\frac{y_{k+1} - y_k}{h} = f(t_k, y_k)$$

Note that this, and all methods considered in this chapter, are written the same regardless of whether y is a vector or a scalar

[2] Note that we replace $y(t_k)$ by y_k

Euler's Method

Quadrature-based interpretation: integrating the ODE $y' = f(t, y)$ from t_k to t_{k+1} gives

$$y(t_{k+1}) = y(t_k) + \int_{t_k}^{t_{k+1}} f(s, y(s))\mathrm{d}s$$

Apply $n = 0$ Newton–Cotes quadrature to $\int_{t_k}^{t_{k+1}} f(s, y(s))\mathrm{d}s$, based on interpolation point t_k:

$$\int_{t_k}^{t_{k+1}} f(s, y(s))\mathrm{d}s \approx (t_{k+1} - t_k) f(t_k, y_k) = h f(t_k, y_k)$$

Again, this gives Euler's method:

$$y_{k+1} = y_k + h f(t_k, y_k)$$

Python example: Euler's method for $y' = \lambda y$

Backward Euler Method

We can derive other methods using the same quadrature-based approach

Apply $n = 0$ Newton–Cotes quadrature based on interpolation point t_{k+1} to

$$y(t_{k+1}) = y(t_k) + \int_{t_k}^{t_{k+1}} f(s, y(s))\mathrm{d}s$$

to get the backward Euler method:

$$y_{k+1} = y_k + hf(t_{k+1}, y_{k+1})$$

Backward Euler Method

(Forward) Euler method is an explicit method: we have an explicit formula for y_{k+1} in terms of y_k

$$y_{k+1} = y_k + hf(t_k, y_k)$$

Backward Euler is an implicit method, we have to solve for y_{k+1} which requires some extra work

$$y_{k+1} = y_k + hf(t_{k+1}, y_{k+1})$$

Backward Euler Method

For example, approximate $y' = 2\sin(ty)$ using backward Euler:

At the first step ($k = 1$), we get

$$y_1 = y_0 + h\sin(t_1 y_1)$$

To compute y_1, let $F(y_1) \equiv y_1 - y_0 - h\sin(t_1 y_1)$ and solve for $F(y_1) = 0$ via, say, Newton's method

Hence implicit methods are more complicated and more computationally expensive at each time step

Why bother with implicit methods? We'll see why shortly...

Trapezoid Method

We can derive methods based on higher-order quadrature

Apply $n = 1$ Newton–Cotes quadrature (Trapezoid rule) at t_k, t_{k+1} to

$$y(t_{k+1}) = y(t_k) + \int_{t_k}^{t_{k+1}} f(s, y(s))\mathrm{d}s$$

to get the Trapezoid Method:

$$y_{k+1} = y_k + \frac{h}{2}\left(f(t_k, y_k) + f(t_{k+1}, y_{k+1})\right)$$

One-Step Methods

The three methods we've considered so far have the form

$$\begin{aligned} y_{k+1} &= y_k + h\Phi(t_k, y_k; h) && \text{(explicit)} \\ y_{k+1} &= y_k + h\Phi(t_{k+1}, y_{k+1}; h) && \text{(implicit)} \\ y_{k+1} &= y_k + h\Phi(t_k, y_k, t_{k+1}, y_{k+1}; h) && \text{(implicit)} \end{aligned}$$

where the choice of the function Φ determines our method

These are called one-step methods: y_{k+1} depends on y_k

(One can also consider multistep methods, where y_{k+1} depends on earlier values $y_{k-1}, y_{k-2}, \ldots$, we'll discuss this briefly later)

Convergence

We now consider whether one-step methods converge to the exact solution as $h \to 0$

Convergence is a crucial property, we want to be able to satisfy an accuracy tolerance by taking h sufficiently small

In general a method that isn't convergent will give misleading results and is useless in practice!

Convergence

We define global error, e_k, as the total accumulated error at $t = t_k$

$$e_k \equiv y(t_k) - y_k$$

We define truncation error, T_k, as the amount "left over" at step k when we apply our method to the exact solution and divide by h

e.g. for an explicit one-step ODE approximation, we have

$$T_k \equiv \frac{y(t_{k+1}) - y(t_k)}{h} - \Phi(t_k, y(t_k); h)$$

Convergence

The truncation error defined above determines the local error introduced by the ODE approximation

For example, suppose $y_k = y(t_k)$, then for the case above we have

$$hT_k \equiv y(t_{k+1}) - y_k - h\Phi(t_k, y_k; h) = y(t_{k+1}) - y_{k+1}$$

Hence hT_k is the error introduced in one step of our ODE approximation[3]

Therefore the global error e_k is determined by the accumulation of the T_j for $j = 0, 1, \ldots, k-1$

Now let's consider the global error of the Euler method in detail

[3] Because of this fact, the truncation error is defined as hT_k in some texts