10. Polynomial interpolation

Last time

- Sensitivity of problems
- Absolute and relative errors
- Condition number

Goals for today

- Interpolation
- Piecewise polynomial interpolation
- Global polynomial interpolation

Representing data

- Suppose have **discrete data points**
- From e.g. measurements of physical / economic problem
- Often **sample** from system with continuous output
- How **reconstruct** function from discrete sample?

Different approaches

- (At least) two different methods to construct functions:
  1. One that **passes through** data points: **interpolation**
  2. One that gives "best approximation": **approximation theory**

Different approaches

- (At least) two different methods to construct functions:
  1. One that **passes through** data points: **interpolation**
  2. One that gives “best approximation”: **approximation theory**
- If start from function, can compare to find error

Interpolation

- Given data points (t_i, y_i) for $i = 0, \dots, n$
- Want: function $f(x)$ that passes through the points:

$$f(t_i) = y_i \quad \forall i$$

Interpolation

- Given data points (t_i, y_i) for $i = 0, \dots, n$
- Want: function $f(x)$ that passes through the points:

$$f(t_i) = y_i \quad \forall i$$

- The t_i are **nodes** or **knots**
- Assume ordered: $a = t_0 < t_1 < \cdots < t_n = b$

Polynomial interpolation

- Most obvious class of functions f to use: **polynomials**

Polynomial interpolation

- Most obvious class of functions f to use: **polynomials**

- Suppose $f(x) = a_0 + a_1 x + \cdots + a_n x^n$
- Then $f(t_i) = a_0 + a_1 t_i + \cdots + a_n t_i^n = y_i \quad \forall i$

Polynomial interpolation

- Most obvious class of functions f to use: **polynomials**

- Suppose $f(x) = a_0 + a_1 x + \cdots + a_n x^n$
- Then $f(t_i) = a_0 + a_1 t_i + \cdots + a_n t_i^n = y_i \quad \forall i$

- Find the a_i that solve this system of equations
- What kind of system is it?

Polynomial interpolation II

- Each equation can be written $\begin{pmatrix} 1 & t_i & t_i^2 & \cdots & t_i^n \end{pmatrix} \cdot \mathbf{a} = y_i$
- $\mathbf{a} = (a_1, a_2, \ldots, a_n)^T$ is (column) vector of unknown a_i

Polynomial interpolation II

- Each equation can be written $\begin{pmatrix} 1 & t_i & t_i^2 & \cdots & t_i^n \end{pmatrix} \cdot \mathbf{a} = y_i$
- $\mathbf{a} = (a_1, a_2, \dots, a_n)^T$ is (column) vector of unknown a_i

- Hence get equation of form V $\cdot\, a = \mathbf{y}$
- V is **Vandermonde matrix** with ith row $\begin{pmatrix} 1 & t_i & t_i^2 & \cdots & t_i^n \end{pmatrix}$

Polynomial interpolation II

- Each equation can be written $\begin{pmatrix} 1 & t_i & t_i^2 & \cdots & t_i^n \end{pmatrix} \cdot \mathbf{a} = y_i$
- $\mathbf{a} = (a_1, a_2, \dots, a_n)^T$ is (column) vector of unknown a_i

- Hence get equation of form V $\cdot\, a = \mathbf{y}$
- V is **Vandermonde matrix** with ith row $\begin{pmatrix} 1 & t_i & t_i^2 & \cdots & t_i^n \end{pmatrix}$

- **In principle** can solve polynomial interpolation like this
- But in fact this algorithm is unstable + expensive

Lagrange interpolation

- Alternative technique for polynomial interpolation
- Build polynomial interpolant as sum

Lagrange interpolation

- Alternative technique for polynomial interpolation
- Build polynomial interpolant as sum
- Look for **cardinal basis** functions:

$$\ell_k(t_i) = [i = k]$$

- **Iverson bracket notation**:

$$[\mathcal{S}] = \begin{cases} 1, & \text{if statement } \mathcal{S} \text{ is correct} \\ 0, & \text{if not} \end{cases}$$

(See Knuth & Patashnik, *Concrete Mathematics*)

Two points

- Simplest case: Line joining (t_0, y_0) and (t_1, y_1)
- Need degree-1 polynomial ℓ_1 such that $\ell_1(t_1) = 1$ and $\ell_1(t_2) = 0$

Two points

- Simplest case: Line joining (t_0, y_0) and (t_1, y_1)
- Need degree-1 polynomial ℓ_1 such that $\ell_1(t_1) = 1$ and $\ell_1(t_2) = 0$

- Could use $\ell_1(x) = ax + b$ and substitute
- Instead, since t_2 is a root, $p(x) = c(x - t_2)$

Two points

- Simplest case: Line joining (t_0, y_0) and (t_1, y_1)
- Need degree-1 polynomial ℓ_1 such that $\ell_1(t_1) = 1$ and $\ell_1(t_2) = 0$

- Could use $\ell_1(x) = ax + b$ and substitute
- Instead, since t_2 is a root, $p(x) = c(x - t_2)$

- So $p(t_1) = c(t_1 - t_2) = 1$
- Hence $c = \frac{1}{x - t_1}$, giving $\ell_1(x) = \frac{x - t_2}{t_1 - t_2}$
- Symmetry gives ℓ_2

Lagrange interpolant

- **Lagrange interpolant** or **Lagrange polynomial** satisfies

$$L(t_1) = y_1 \quad \text{and} \quad L(t_2) = y_2$$

- Since cardinal basis functions we have

$$L(x) = y_1 \ell_1(x) + y_2 \ell_2(x)$$

Lagrange interpolant

- **Lagrange interpolant** or **Lagrange polynomial** satisfies

$$L(t_1) = y_1 \quad \text{and} \quad L(t_2) = y_2$$

- Since cardinal basis functions we have

$$L(x) = y_1 \ell_1(x) + y_2 \ell_2(x)$$

- *Any* linear polynomial $ax + b$ can be written in this way
- So ℓ_1, ℓ_2 form new **basis** of linear polynomials

Piecewise linear interpolation

- Possible interpolant: Separate polynomials on each $[t_i, t_{i+1}]$
- E.g. piecewise linear satisfying cardinality conditions:
- Piecewise-linear “hat” function with value 1 at t_k and zero for all other t_i
- Piecewise-linear interpolant (“join the dots”) is linear combination of hat **basis functions**:
- *Any* piecewise-linear function can be so expressed

Piecewise polynomial interpolation

- Piecewise-linear interpolation gives non-smooth result
- Make smoother by replacing linear pieces with higher-degree polynomials s
- **Splines**, e.g. cubic splines

Global Lagrange interpolation in n points

- Find *single* polynomial that interpolates all $n + 1$ points simultaneously
- Know this is possible by Vandermonde argument

Global Lagrange interpolation in n points

- Find *single* polynomial that interpolates all $n+1$ points simultaneously
- Know this is possible by Vandermonde argument
- Generalise from 2 to n points:

$$\ell_k(x) = c_k(x - t_1) \cdots (\widehat{x - t_k}) \cdots (x - t_n)$$

where $\hat{\cdot}$ indicates a *missing* term

Global Lagrange interpolation II

- We want $\ell_k(t_k) = 1$, so

$$c_k = \frac{1}{(t_k - t_1)\cdots\widehat{(t_k - t_k)}\cdots(x - t_n)}$$

Global Lagrange interpolation II

- We want $\ell_k(t_k) = 1$, so

$$c_k = \frac{1}{(t_k - t_1) \cdots \widehat{(t_k - t_k)} \cdots (x - t_n)}$$

- Thus $\ell_k(x) = \prod_{j \neq k} \frac{x - t_i}{t_k - t_i}$
- And $L(x) = \sum_{k=0}^{n} y_k \ell_k(x)$

Global Lagrange interpolation II

- We want $\ell_k(t_k) = 1$, so

$$c_k = \frac{1}{(t_k - t_1)\cdots\widehat{(t_k - t_k)}\cdots(x - t_n)}$$

- Thus $\ell_k(x) = \prod_{j \neq k} \frac{x - t_i}{t_k - t_i}$
- And $L(x) = \sum_{k=0}^{n} y_k \ell_k(x)$

- Uniqueness: Suppose not unique and subtract
- Have *constructed new basis* for (vector) space of degree-n polynomials

What can go wrong?

- Will see in PS 4 that global Lagrange interpolation can go *very badly wrong* if use equally-spaced points
- Much better to use points that *cluster* near endpoints of interval

Array indexing in Julia

- Indexing of standard Julia arrays starts at `x[1]`
- Often convenient (but not always!) to start at 0 instead
- Can be done by overloading the `getindex` function:

```julia
struct MyVector
    v::Vector
end

Base.getindex(w::MyVector, i) = w.v[i+1]

v = MyVector([0, 1, 2, 3])
v[0]
```

Array indexing in Julia II

- `OffsetArrays.jl` package provides arbitrary indices:

```
julia> using OffsetArrays

julia> v = OffsetArray([1:7], -3:3);  # -3:3 specifies indic

julia> v[0]
1
```

Summary

- Constructed Lagrange polynomials that **interpolate** data
- Form a basis