Applied Mathematics 205

Unit 1. Data Fitting

Lecturer: Petr Karnakov

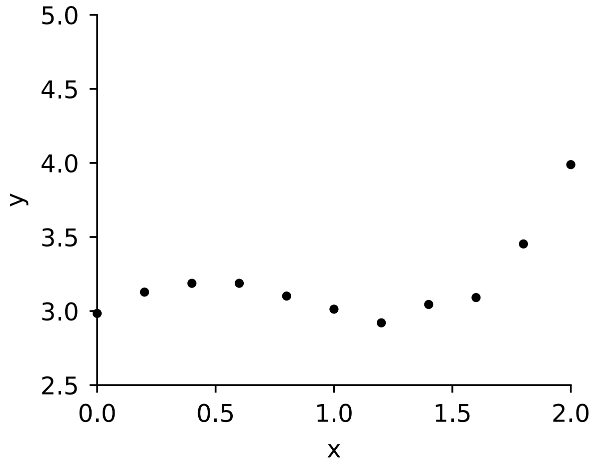
September 7, 2022

Motivation

- Data fitting: Construct a continuous function that represents discrete data. Fundamental topic in Scientific Computing
- We will study two types of data fitting
  - interpolation: fit the data points exactly
  - least-squares: minimize error in the fit (e.g. useful when there is experimental error)
- Data fitting helps us to
  - interpret data: deduce hidden parameters, understand trends
  - process data: reconstructed function can be differentiated, integrated, etc

Motivation

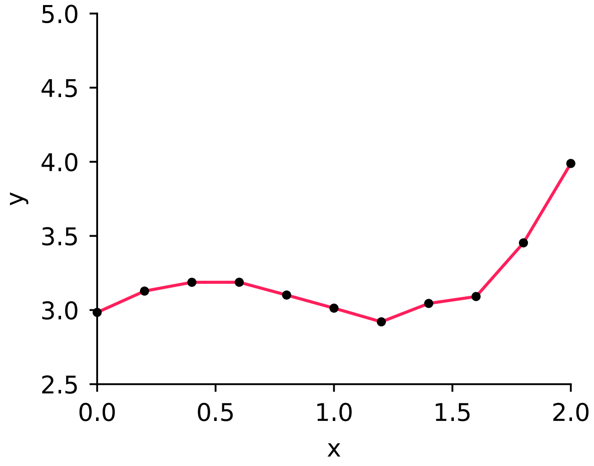
- Suppose we are given the following data points



- Such data could represent
  - time series data (stock price, sales figures)
  - laboratory measurements (pressure, temperature)
  - astronomical observations (star light intensity)

Motivation

- We often need values between the data points
- Easiest thing to do: “connect the dots” (piecewise linear interpolation)

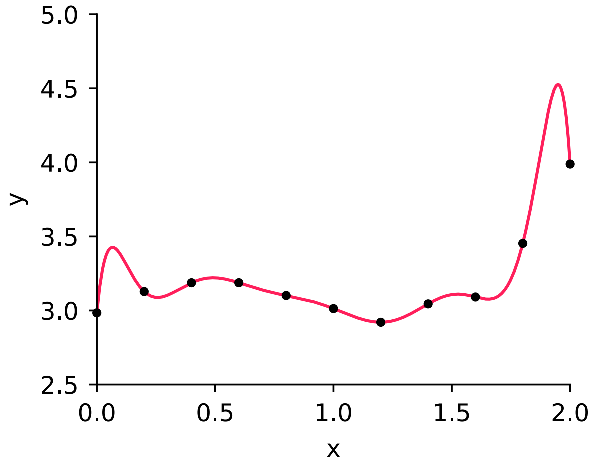


Question: What if we want a smoother approximation?

Motivation

- We have 11 data points, we can use a degree 10 polynomial

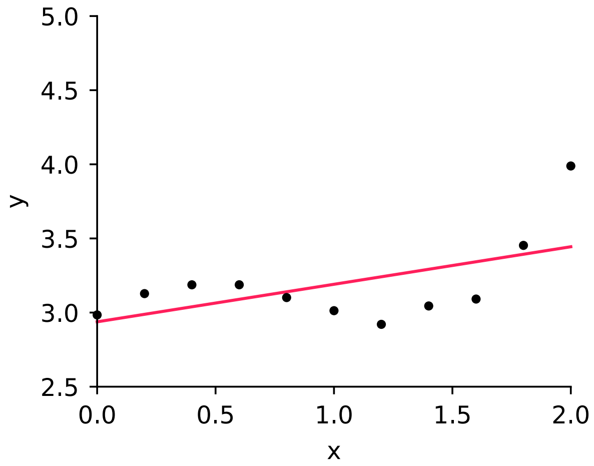
$$y = 2.98 + 16.90x - 219.77x^2 + 1198.07x^3 - 3518.54x^4 + 6194.09x^5 - 6846.49x^6 + 4787.40x^7 - 2053.91x^8 + 492.90x^9 - 50.61x^{10}$$



- However, a degree 10 interpolant doesn't seem to capture the underlying pattern, has bumps and changes rapidly

Motivation

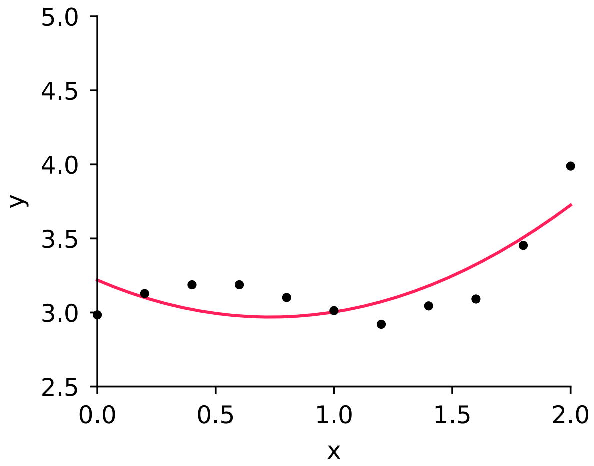
- Let's try linear regression:
  minimize the error in a linear approximation of the data
- Best linear fit: $y = 2.94 + 0.25x$



- Clearly not a good fit!

Motivation

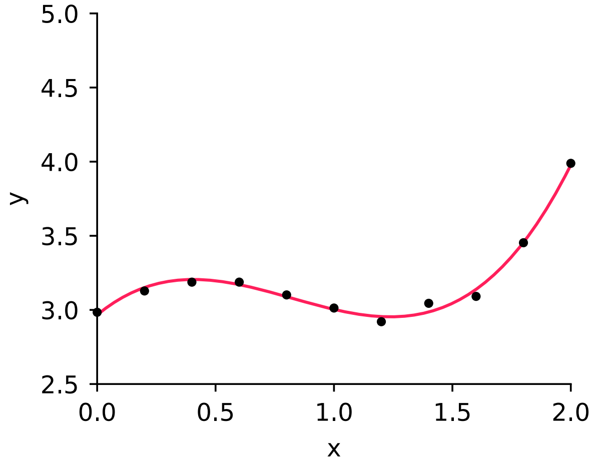
- We can use least-squares fitting to generalize linear regression to higher-order polynomials
- Best quadratic fit: $y = 3.22 - 0.68x + 0.47x^2$



- Still not so good ...

Motivation

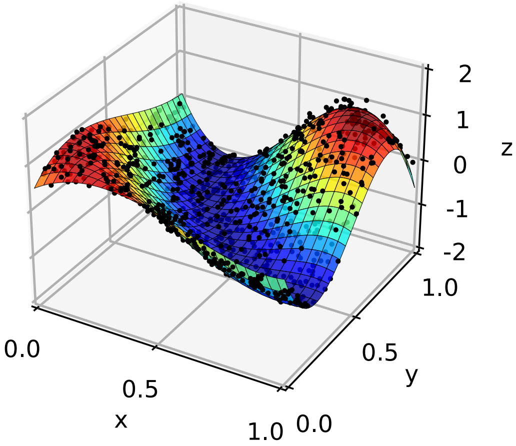
- Best cubic fit: $y = 2.97 + 1.32x - 2.16x^2 + 0.88x^3$



- Looks good! A "cubic model" captures this data well
- In real-world problems it can be challenging to find the "right" model for experimental data

Motivation

- Data fitting is often performed with multi-dimensional data
- 2D example: points (x, y) with feature z



- See [examples/unit1/fit_2d.py]

Motivation: Summary

- Interpolation is a fundamental tool in Scientific Computing, provides simple representation of discrete data
  - Common to differentiate, integrate, optimize an interpolant
- Least squares fitting is typically more useful for experimental data
  - Removes noise using a lower-order model
- Data-fitting calculations are often performed with big datasets
  - Efficient and stable algorithms are very important

Polynomial Interpolation

- Let $\mathbb{P}_n$ denote the set of all polynomials of degree n on $\mathbb{R}$
- Polynomial $p(\cdot; b) \in \mathbb{P}_n$ has the form

$$p(x; b) = b_0 + b_1 x + b_2 x^2 + \ldots + b_n x^n$$

with coefficients $b = [b_0, b_1, \ldots, b_n]^T \in \mathbb{R}^{n+1}$

Polynomial Interpolation

- Suppose we have data

$$\mathcal{S} = \{(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)\}$$

  where $x_0, x_1, \ldots, x_n$ are called interpolation points
- Goal: Find a polynomial that passes through every data point in $\mathcal{S}$
- Therefore, we must have $p(x_i; b) = y_i$ for each $i = 0, \ldots, n$
  $\implies n + 1$ equations
- For uniqueness, we should look for a polynomial with $n + 1$ parameters
  $\implies$ look for $p \in \mathbb{P}_n$

Polynomial Interpolation

- This leads to the following system of $n+1$ equations with $n+1$ unknowns

$$\begin{aligned} b_0 + b_1 x_0 + b_2 x_0^2 + \ldots + b_n x_0^n &= y_0 \\ b_0 + b_1 x_1 + b_2 x_1^2 + \ldots + b_n x_1^n &= y_1 \\ &\vdots \\ b_0 + b_1 x_n + b_2 x_n^2 + \ldots + b_n x_n^n &= y_n \end{aligned}$$

- The system is linear with respect to unknown coefficients $b_0, \ldots, b_n$

Vandermonde Matrix

- The same system in matrix form

$$Vb = y$$

with

  - unknown coefficients $b = [b_0, b_1, \ldots, b_n]^T \in \mathbb{R}^{n+1}$
  - given values $y = [y_0, y_1, \ldots, y_n]^T \in \mathbb{R}^{n+1}$
  - matrix $V \in \mathbb{R}^{(n+1)\times(n+1)}$ called the Vandermonde matrix

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{bmatrix}$$

Existence and Uniqueness

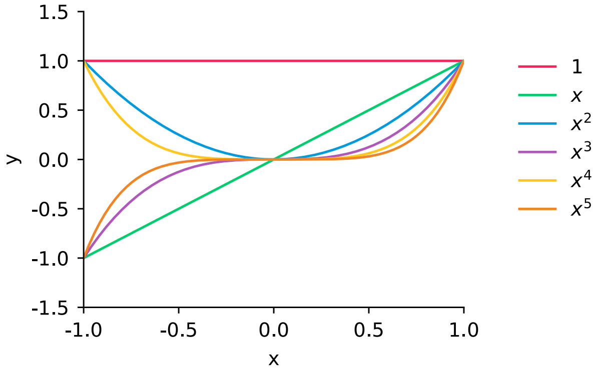
- Let's prove that if the $n+1$ interpolation points are distinct, then $Vb = y$ has a unique solution
- We know from linear algebra that for a square matrix A: if $Az = 0 \implies z = 0$, then $Ab = y$ has a unique solution
- If $Vb = 0$, then $p(\cdot; b) \in \mathbb{P}_n$ has $n+1$ distinct roots
- Therefore we must have $p(\cdot; b) = 0$, or equivalently $b = 0$
- Hence $Vb = 0 \implies b = 0$
  so $Vb = y$ has a unique solution for any $y \in \mathbb{R}^{n+1}$

Vandermonde Matrix

- This tells us that we can find the polynomial interpolant by solving the Vandermonde system $Vb = y$
- However, this may be a bad idea since V is ill-conditioned

Monomial Interpolation

- The problem is that Vandermonde matrix corresponds to interpolation using the monomial basis
- Monomial basis for $\mathbb{P}_n$ is $\{1, x, x^2, \ldots, x^n\}$
- As n increases, basis functions become increasingly indistinguishable, columns are more "linearly dependent", the matrix is ill-conditioned
- See [examples/unit1/vander_cond.py], condition number of Vandermonde matrix



Monomial Basis

- Question: What is the practical consequence of this ill-conditioning?
- Answer:
  - We want to solve $Vb = y$
  - Finite precision arithmetic gives an approximation $\hat{b}$
  - Residual $\|V\hat{b} - y\|$ will be small but $\|b - \hat{b}\|$ can still be large! (will be discussed in Unit 2)
  - Similarly, small perturbation in b can give large perturbation in Vb
  - Large perturbations in Vb can yield large $\|Vb - y\|$, hence a "perturbed interpolant" becomes a poor fit to the data

Monomial Basis

- These sensitivities are directly analogous to what happens with an ill-conditioned basis in $\mathbb{R}^n$
- Consider a basis v_1, v_2 of $\mathbb{R}^2$

$$v_1 = [1, 0]^T, \qquad v_2 = [1, 0.0001]^T$$

- Let's express two close vectors

$$y = [1, 0]^T, \qquad \tilde{y} = [1, 0.0005]^T$$

in terms of this basis i.e. $y = b_1 v_1 + b_2 v_2$ and $\tilde{y} = \tilde{b}_1 v_1 + \tilde{b}_2 v_2$
- By solving a 2×2 linear system in each case, we get

$$b = [1, 0]^T, \qquad \tilde{b} = [-4, 5]^T$$

- The answer b is highly sensitive to perturbations in y

Monomial Basis

- The same happens with interpolation using a monomial basis
- The answer (coefficients of polynomial) is highly sensitive to perturbations in the data
- If we perturb b slightly, we can get a large perturbation in Vb so the resulting polynomial no longer fits the data well
- Example of interpolation using Vandermonde matrix [examples/unit1/vander_interp.py]

Interpolation

- We would like to avoid these kinds of sensitivities to perturbations …
  How can we do better?
- Try to construct a basis such that
  the interpolation matrix is the identity matrix
- This gives a condition number of 1, and we also
  avoid solving a linear system with a dense $(n+1) \times (n+1)$ matrix

Lagrange Interpolation

- Key idea: Construct basis $\{L_k \in \mathbb{P}_n, k = 0, \ldots, n\}$ such that

$$L_k(x_i) = \begin{cases} 0, & i \neq k \\ 1, & i = k \end{cases}$$

- The polynomials that achieve this are called Lagrange polynomials
- Lagrange polynomials are given by:

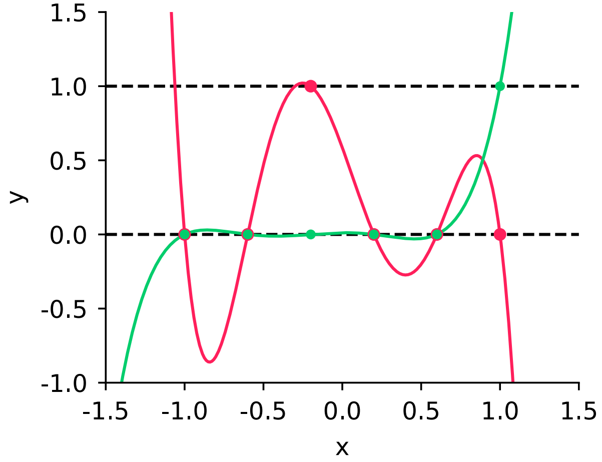
$$L_k(x) = \prod_{j=0, j\neq k}^{n} \frac{x - x_j}{x_k - x_j}$$

- Then the interpolant can be expressed as

$$p(x) = \sum_{k=0}^{n} y_k L_k(x)$$

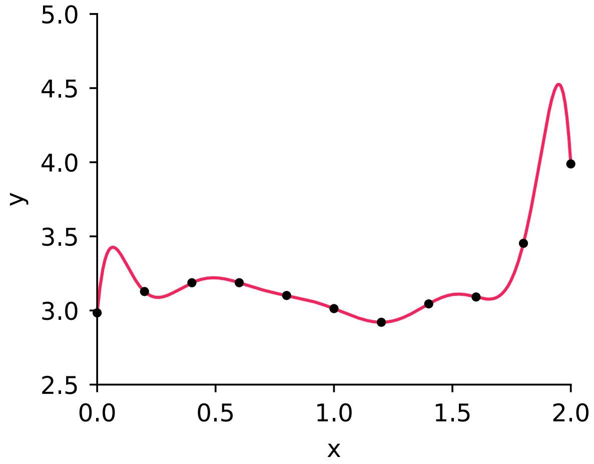
Lagrange Interpolation

- Example: two Lagrange polynomials of degree 5 constructed on points $x \in \{-1, -0.6, -0.2, 0.2, 0.6, 1\}$



Lagrange Interpolation

- Now we can use Lagrange polynomials to interpolate discrete data



- We have solved the problem of interpolating discrete data!

Algorithmic Complexity

- Exercise 1: How does the cost of evaluating a polynomial at one point x scale with n?

$$p(x) = b_0 + b_1 x + b_2 x^2 + \ldots + b_n x^n$$

- Exercise 2: How does the cost of evaluating a Lagrange interpolant at one point x scale with n?

$$p(x) = \sum_{k=0}^{n} y_k \prod_{j=0, j\neq k}^{n} \frac{x - x_j}{x_k - x_j}$$

Interpolation for Function Approximation

- We now turn to a different question:
Can we use interpolation to accurately approximate continuous functions?
- Suppose the interpolation data come from samples of a continuous function f on $[a, b] \subset \mathbb{R}$
- Then we'd like the interpolant to be "close to" f on $[a, b]$
- The error in this type of approximation can be quantified from the following theorem due to Cauchy

$$f(x) - p_n(x) = \frac{f^{(n+1)}(\theta)}{(n+1)!}(x - x_0) \ldots (x - x_n)$$

for some $\theta(x) \in (a, b)$

Polynomial Interpolation Error

- Here we prove this result in the case $n = 1$
- Let $p_1 \in \mathbb{P}_1$ interpolate $f \in C^2[a, b]$ at $\{x_0, x_1\}$
- For some $\lambda \in \mathbb{R}$, let

$$q(x) = p_1(x) + \lambda(x - x_0)(x - x_1),$$

  here q is quadratic and interpolates f at $\{x_0, x_1\}$
- Fix an arbitrary point $\hat{x} \in (x_0, x_1)$ and require $q(\hat{x}) = f(\hat{x})$ to get

$$\lambda = \frac{f(\hat{x}) - p_1(\hat{x})}{(\hat{x} - x_0)(\hat{x} - x_1)}$$

- Goal: Get an expression for λ, and eventually for $f(\hat{x}) - p_1(\hat{x})$

Polynomial Interpolation Error

- Denote the error $e(x) = f(x) - q(x)$
  - $e(x)$ has 3 roots in $[x_0, x_1]$, i.e. $e(x_0) = e(\hat{x}) = e(x_1) = 0$
  - Therefore, $e'(x)$ has 2 roots in (x_0, x_1) (by Rolle's theorem)
  - Therefore, $e''(x)$ has 1 root in (x_0, x_1) (by Rolle's theorem)
- Let $\theta(\hat{x}) \in (x_0, x_1)$ be such that $e''(\theta) = 0$
- Then

$$\begin{aligned} 0 = e''(\theta) &= f''(\theta) - q''(\theta) \\ &= f''(\theta) - p_1''(\theta) - \lambda \frac{\mathrm{d}^2}{\mathrm{d}\theta^2}(\theta - x_0)(\theta - x_1) \\ &= f''(\theta) - 2\lambda \end{aligned}$$

- Hence $\lambda = \frac{1}{2} f''(\theta)$

Polynomial Interpolation Error

- Finally, we get

$$f(\hat{x}) - p_1(\hat{x}) = \lambda(\hat{x} - x_0)(\hat{x} - x_1) = \frac{1}{2} f''(\theta)(\hat{x} - x_0)(\hat{x} - x_1)$$

for any $\hat{x} \in (x_0, x_1)$

- This argument can be generalized to $n > 1$ to give

$$f(x) - p_n(x) = \frac{f^{(n+1)}(\theta)}{(n+1)!}(x - x_0)\ldots(x - x_n)$$

for some $\theta(x) \in (a, b)$

Polynomial Interpolation Error

- For any $x \in [a, b]$, this theorem gives us the error bound

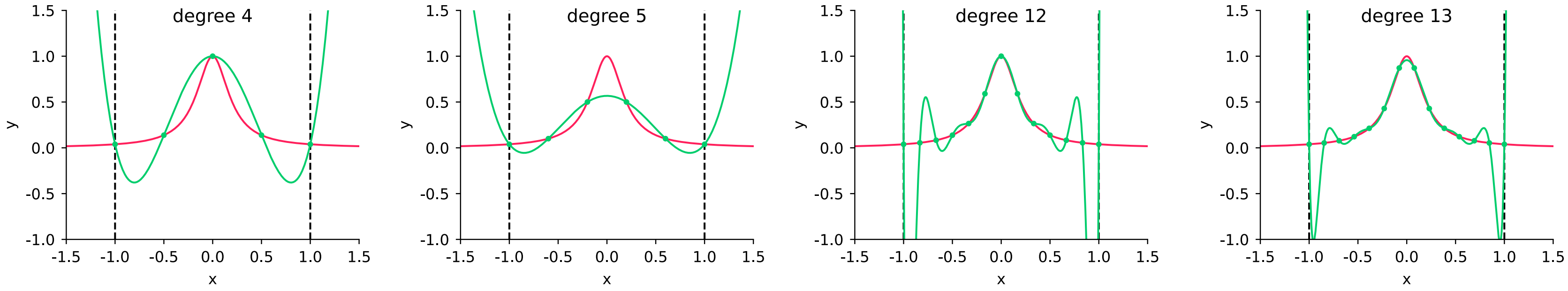
$$|f(x) - p_n(x)| \leq \frac{M_{n+1}}{(n+1)!} \max_{x \in [a,b]} |(x - x_0) \dots (x - x_n)|$$

where $M_{n+1} = \max_{\theta \in [a,b]} |f^{n+1}(\theta)|$

- As n increases,
if $(n+1)!$ grows faster than $M_{n+1} \max_{x \in [a,b]} |(x - x_0) \dots (x - x_n)|$
then p_n converges to f
- Unfortunately, this is not always the case!

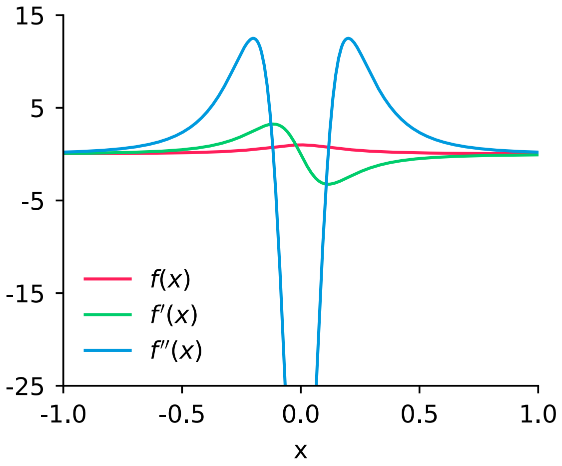
Runge's Phenomenon

- A famous pathological example of the difficulty of interpolation at equally spaced points is Runge's Phenomenon
- Consider Runge's function $f(x) = 1/(1+25x^2)$ for $x \in [-1, 1]$



Runge's Phenomenon

- Reason: derivatives grow fast
- $f(x) = 1/(1 + 25x^2)$
- $f'(x) = -50x/(1 + 25x^2)^2$
- $f''(x) = (3750x^2 - 50)/(((15625x^2 + 1875)x^2 + 75)x^2 + 1)$



Runge's Phenomenon

- Note that p_n is an interpolant, so it fits the evenly spaced samples exactly
- But we are now also interested in the maximum error between f and its polynomial interpolant p_n
- That is, we want $\max_{x \in [-1,1]} |f(x) - p_n(x)|$ to be small!
- This is called the "infinity norm" or the "max norm"

$$\|f - p_n\|_\infty = \max_{x \in [-1,1]} |f(x) - p_n(x)|$$

Runge's Phenomenon

- Note that Runge's function $f(x) = 1/(1 + 25x^2)$ is smooth but interpolating Runge's function at evenly spaced points leads to exponential growth of the infinity norm error!
- We would like to construct an interpolant of f that avoids this kind of pathological behavior

Minimizing Interpolation Error

- To do this, we recall our error equation

$$f(x) - p_n(x) = \frac{f^{n+1}(\theta)}{(n+1)!}(x - x_0)\ldots(x - x_n)$$

- We focus our attention on the polynomial $(x - x_0)\ldots(x - x_n)$, since we can choose the interpolation points
- Intuitively, we should choose $x_0, \ldots, x_n$ such that $\|(x - x_0)\ldots(x - x_n)\|_\infty$ is as small as possible

Chebyshev Polynomials

- Chebyshev polynomials are defined for $x \in [-1, 1]$ by

$$T_n(x) = \cos(n \arccos x), n = 0, 1, 2, \ldots$$

- Or, equivalently, through the recurrence relation

$$\begin{aligned} T_0(x) &= 1, \\ T_1(x) &= x, \\ T_{n+1}(x) &= 2xT_n(x) - T_{n-1}(x), \quad n = 1, 2, 3, \ldots \end{aligned}$$

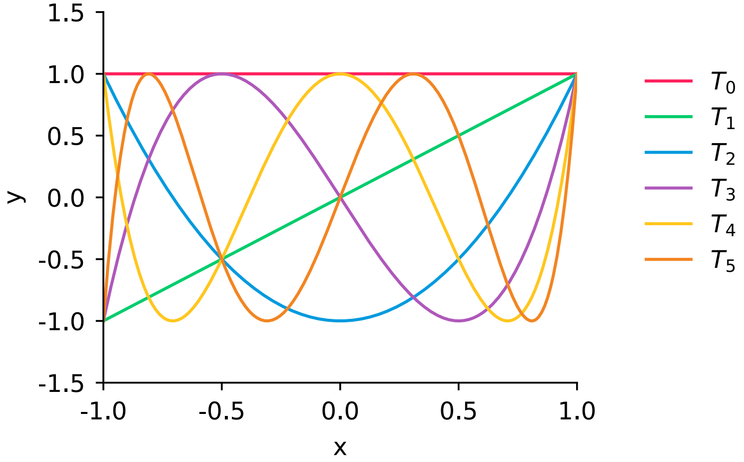
- Result from Approximation Theory:
  The minimal value

$$\min_{x_0, \ldots, x_n} \|(x - x_0) \ldots (x - x_n)\|_\infty = \frac{1}{2^n}$$

  is achieved by the polynomial $T_{n+1}(x)/2^n$

Chebyshev Polynomials

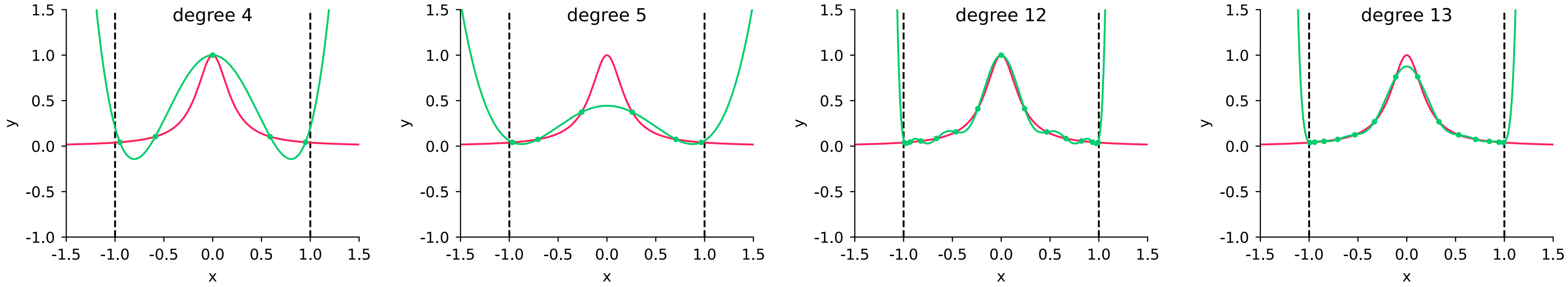
- To set $(x - x_0) \dots (x - x_n) = T_{n+1}(x)/2^n$, we choose interpolation points to be the roots of T_{n+1}
- Chebyshev polynomials "equi-oscillate" (alternate) between -1 and 1, so they minimize the infinity norm



- Exercise: Show that the roots of T_n are given by $x_j = \cos((2j - 1)\pi/2n)$, $j = 1, \dots, n$

Interpolation at Chebyshev Points

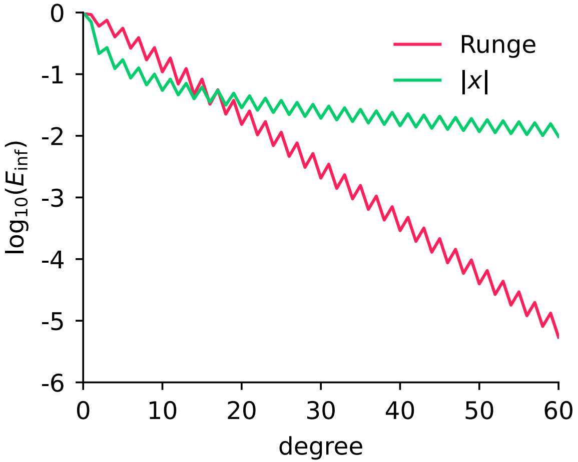
- Revisit Runge's function. Chebyshev interpolation is more accurate



- To interpolate on an arbitrary interval $[a, b]$,
  linearly map Chebyshev points from $[-1, 1]$ to $[a, b]$

Interpolation at Chebyshev Points

- Note that convergence rates depend on smoothness of f
- In general, smoother $f \implies$ faster convergence
- Convergence of Chebyshev interpolation of Runge's function (smooth) and $|x|$ (not smooth)



- Example of interpolation at Chebyshev points [examples/unit1/cheb_interp.py]

Another View on Interpolation Accuracy

- We have seen that the interpolation points we choose have an enormous effect on how well our interpolant approximates f
- The choice of Chebyshev interpolation points was motivated by our interpolation error formula for $f(x) - p_n(x)$
- But this formula depends on f — we would prefer to have a measure of interpolation accuracy that is independent of f
- This would provide a more general way to compare the quality of interpolation points . . . This is provided by the Lebesgue constant

Lebesgue Constant

- Let $\mathcal{X}$ denote a set of interpolation points, $\mathcal{X} = \{x_0, x_1, \ldots, x_n\} \subset [a, b]$
- A fundamental property of $\mathcal{X}$ is its Lebesgue constant, $\Lambda_n(\mathcal{X})$,

$$\Lambda_n(\mathcal{X}) = \max_{x \in [a,b]} \sum_{k=0}^{n} |L_k(x)|$$

- The $L_k \in \mathbb{P}_n$ are the Lagrange basis polynomials associated with $\mathcal{X}$, hence Λ_n is also a function of $\mathcal{X}$
- $\Lambda_n(\mathcal{X}) \geq 1$

Lebesgue Constant

- Think of polynomial interpolation as a map, $\mathcal{I}_n$, where $\mathcal{I}_n : C[a,b] \to \mathbb{P}_n[a,b]$
- $\mathcal{I}_n(f)$ is the degree n polynomial interpolant of $f \in C[a,b]$ at the interpolation points $\mathcal{X}$
- Exercise: Convince yourself that $\mathcal{I}_n$ is linear (e.g. use the Lagrange interpolation formula)
- The reason that the Lebesgue constant is interesting is because it bounds the "operator norm" of $\mathcal{I}_n$:

$$\sup_{f \in C[a,b]} \frac{\|\mathcal{I}_n(f)\|_\infty}{\|f\|_\infty} \leq \Lambda_n(\mathcal{X})$$

Lebesgue Constant

- Proof

$$\begin{aligned}
\|\mathcal{I}_n(f)\|_\infty &= \|\sum_{k=0}^{n} f(x_k)L_k\|_\infty = \max_{x\in[a,b]}\left|\sum_{k=0}^{n} f(x_k)L_k(x)\right| \\
&\le \max_{x\in[a,b]}\sum_{k=0}^{n}|f(x_k)||L_k(x)| \\
&\le \left(\max_{k=0,1,\ldots,n}|f(x_k)|\right)\max_{x\in[a,b]}\sum_{k=0}^{n}|L_k(x)| \\
&\le \|f\|_\infty \max_{x\in[a,b]}\sum_{k=0}^{n}|L_k(x)| \\
&= \|f\|_\infty \Lambda_n(\mathcal{X})
\end{aligned}$$

- Hence $\frac{\|\mathcal{I}_n(f)\|_\infty}{\|f\|_\infty} \le \Lambda_n(\mathcal{X})$, so $\sup_{f\in C[a,b]} \frac{\|\mathcal{I}_n(f)\|_\infty}{\|f\|_\infty} \le \Lambda_n(\mathcal{X})$

Lebesgue Constant

- The Lebesgue constant allows us to bound interpolation error in terms of the smallest possible error from $\mathbb{P}_n$
- Let $p_n^* \in \mathbb{P}_n$ denote the best infinity-norm approximation to f

$$\|f - p_n^*\|_\infty \leq \|f - w\|_\infty$$

  for all $w \in \mathbb{P}_n$
- Some facts about p_n^*
  - $\|p_n^* - f\|_\infty \to 0$ as $n \to \infty$ for any continuous f!
    (Weierstrass approximation theorem)
  - $p_n^* \in \mathbb{P}_n$ is unique
    (follows from the equi-oscillation theorem)
  - In general, p_n^* is unknown

Lebesgue Constant

- Then, we can relate interpolation error to $\|f - p_n^*\|_\infty$

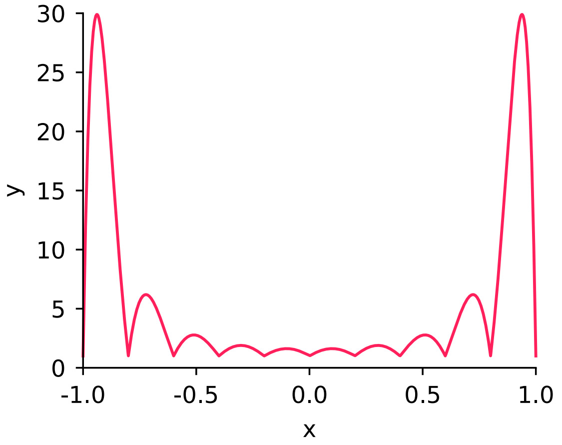
$$
\begin{aligned}
\|f - \mathcal{I}_n(f)\|_\infty &\le \|f - p_n^*\|_\infty + \|p_n^* - \mathcal{I}_n(f)\|_\infty \\
&= \|f - p_n^*\|_\infty + \|\mathcal{I}_n(p_n^*) - \mathcal{I}_n(f)\|_\infty \\
&= \|f - p_n^*\|_\infty + \|\mathcal{I}_n(p_n^* - f)\|_\infty \\
&= \|f - p_n^*\|_\infty + \frac{\|\mathcal{I}_n(p_n^* - f)\|_\infty}{\|p_n^* - f\|_\infty}\|f - p_n^*\|_\infty \\
&\le \|f - p_n^*\|_\infty + \Lambda_n(\mathcal{X})\|f - p_n^*\|_\infty \\
&= (1 + \Lambda_n(\mathcal{X}))\|f - p_n^*\|_\infty
\end{aligned}
$$

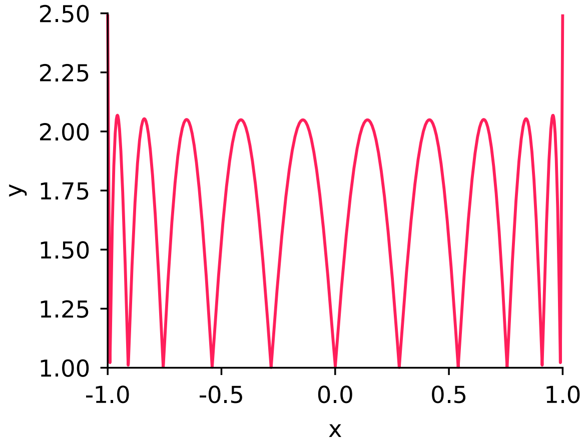
Lebesgue Constant

- Small Lebesgue constant means that our interpolation cannot be much worse than the best possible polynomial approximation!
- See [examples/unit1/lebesgue_const.py]
- Now let's compare Lebesgue constants for equispaced ($\mathcal{X}_{\text{equi}}$) and Chebyshev points ($\mathcal{X}_{\text{cheb}}$)

Lebesgue Constant

- Plot of $\sum_{k=0}^{10} |L_k(x)|$ for $\mathcal{X}_{\text{equi}}$ and $\mathcal{X}_{\text{cheb}}$ (11 pts in $[-1, 1]$)

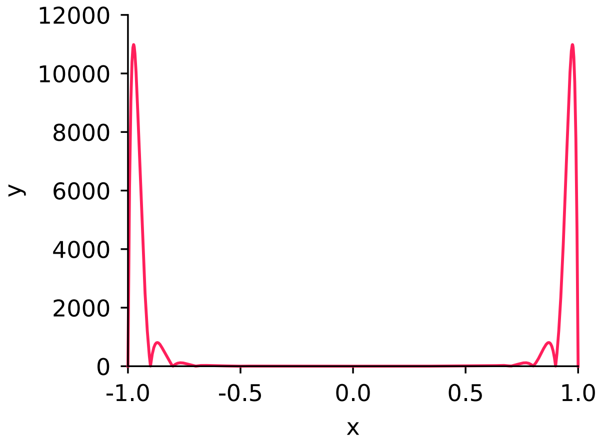


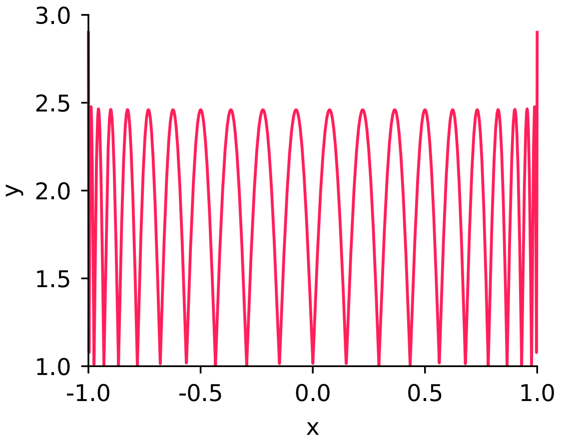


$\Lambda_{10}(\mathcal{X}_{\text{equi}}) \approx 29.9$ $\Lambda_{10}(\mathcal{X}_{\text{cheb}}) \approx 2.49$

Lebesgue Constant

- Plot of $\sum_{k=0}^{20} |L_k(x)|$ for $\mathcal{X}_{\text{equi}}$ and $\mathcal{X}_{\text{cheb}}$ (21 pts in $[-1, 1]$)

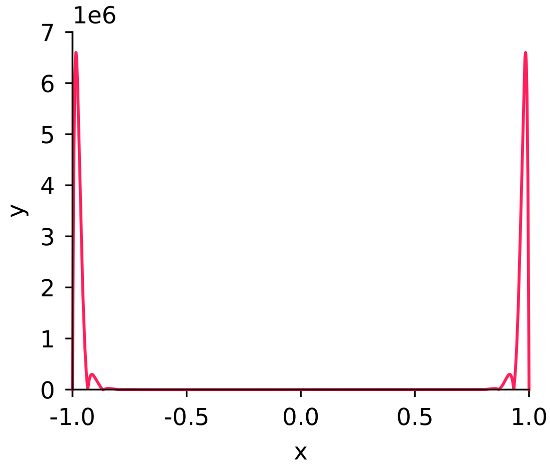


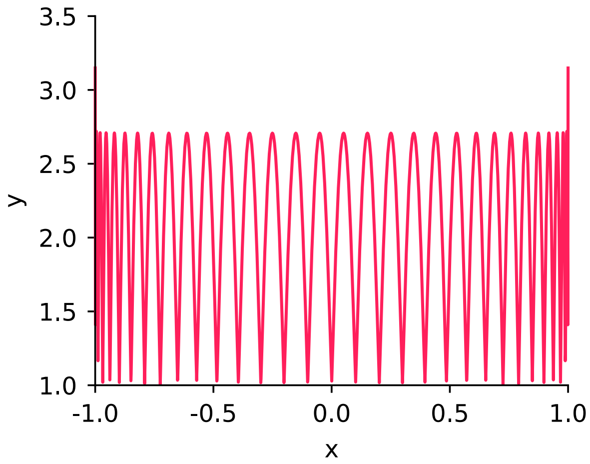


$$\Lambda_{20}(\mathcal{X}_{\text{equi}}) \approx 10\,987 \qquad \Lambda_{20}(\mathcal{X}_{\text{cheb}}) \approx 2.9$$

Lebesgue Constant

- Plot of $\sum_{k=0}^{30} |L_k(x)|$ for $\mathcal{X}_{\text{equi}}$ and $\mathcal{X}_{\text{cheb}}$ (31 pts in $[-1, 1]$)





$\Lambda_{30}(\mathcal{X}_{\text{equi}}) \approx 6\,600\,000$

$\Lambda_{30}(\mathcal{X}_{\text{cheb}}) \approx 3.15$

Lebesgue Constant

- The explosive growth of $\Lambda_n(\mathcal{X}_{\text{equi}})$ is an explanation for Runge's phenomenon
- Asymptotic results as $n \to \infty$

$$\Lambda_n(\mathcal{X}_{\text{equi}}) \sim \frac{2^n}{e\, n \log n} \qquad \text{exponential growth}$$

$$\Lambda_n(\mathcal{X}_{\text{cheb}}) < \frac{2}{\pi} \log(n+1) + 1 \qquad \text{logarithmic growth}$$

- Open mathematical problem: Construct $\mathcal{X}$ that minimizes $\Lambda_n(\mathcal{X})$

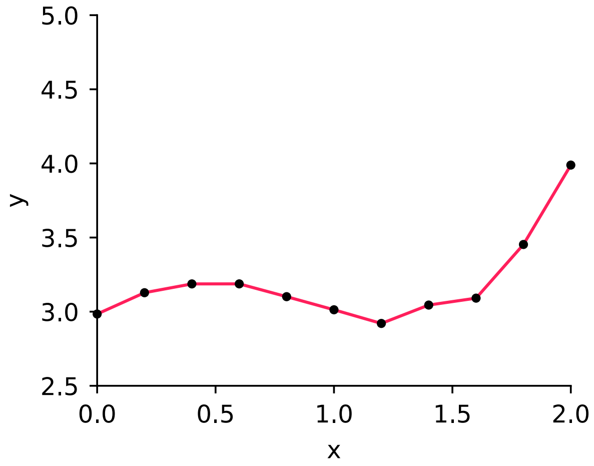
Summary

- Compare and contrast the two key topics considered so far
- Polynomial interpolation for fitting discrete data
  - we get "zero error" regardless of the interpolation points, i.e. we're guaranteed to fit the discrete data
  - Lagrange polynomial basis should be instead of the monomial basis as the number of points increases (diagonal system, well-conditioned)
- Polynomial interpolation for approximating continuous functions
  - for a given set of interpolating points, uses same methodology as for discrete data
  - but now interpolation points play a crucial role in determining the magnitude of the error $\|f - \mathcal{I}_n(f)\|_\infty$

Piecewise Polynomial Interpolation

Piecewise Polynomial Interpolation

- How to avoid explosive growth of error for non-smooth functions?
- Idea: Decompose domain into subdomains and apply polynomial interpolation on each subdomain
- Example: piecewise linear interpolation

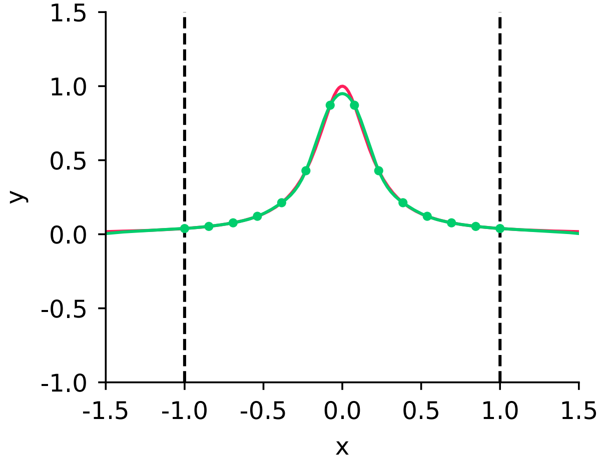


Splines

- Splines are a popular type of piecewise polynomial interpolant
- Interpolation points are now called knots
- Splines have smoothness constraints to "glue" adjacent polynomials
- Commonly used in computer graphics, font rendering, CAD software
  - Bezier splines
  - non-uniform rational basis spline (NURBS)
  - ...
- The name "spline" comes from
  "a flexible piece of wood or metal used in drawing curves"

Splines

- We focus on a popular type of spline: cubic spline
- Piecewise cubic with continuous second derivatives
- Example: cubic spline interpolation of Runge's function



Cubic Splines

- Suppose we have $n+1$ data points: $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$
- A cubic interpolating spline is a function $s(x)$ that
  - is a cubic polynomial on each of n intervals $[x_{i-1}, x_i]$ ($4n$ parameters)
  - passes through the data points ($2n$ conditions)

$$s(x_i) = y_i, \quad i = 0, \dots, n$$

  - has continuous first derivative ($n-1$ conditions)

$$s'_-(x_i) = s'_+(x_i), \quad i = 1, \dots, n-1$$

  - has continuous second derivative ($n-1$ conditions)

$$s''_-(x_i) = s''_+(x_i), \quad i = 1, \dots, n-1$$

- We have $4n-2$ equations for $4n$ unknowns

Cubic Splines

- We are missing two conditions!
- Many options to define them
  - natural cubic spline

$$s''(x_0) = s''(x_n) = 0$$

  - clamped

$$s'(x_0) = s'(x_n) = 0$$

  - “not-a-knot spline”

$$s'''_-(x_1) = s'''_+(x_1) \quad \text{and} \quad s'''_-(x_{n-1}) = s'''_+(x_{n-1})$$

Constructing a Cubic Spline

- Denote $\Delta x_i = x_i - x_{i-1}$ and $\Delta y_i = y_i - y_{i-1}$
- Look for polynomials $p_i \in \mathbb{P}_3, \quad i = 1, \ldots, n$ in the form

$$p_i(x) = t y_i + (1-t)\, y_{i-1} + t\,(1-t)\,(\alpha t + \beta\,(1-t))$$

with unknown α and β, where $t = \frac{x - x_{i-1}}{\Delta x_i}$

- Automatically satisfies interpolation conditions

$$p_i(x_{i-1}) = y_{i-1} \qquad p_i(x_i) = y_i$$

- Conditions on derivatives to make the first derivative continuous

$$p_i'(x_{i-1}) = k_{i-1} \qquad p_i'(x_i) = k_i$$

$$\implies \alpha = y_i - y_{i-1} - \Delta x_i k_i \qquad \beta = y_{i-1} - y_i + \Delta x_i k_{i-1}$$

- New unknown parameters: $k_0, \ldots, k_n$ ($n + 1$ parameters)

Constructing a Cubic Spline

- Expressions for second derivatives

$$p_i''(x_{i-1}) = \frac{-4k_{i-1} - 2k_i}{\Delta x_i} + \frac{6\Delta y_i}{\Delta x_i^2}$$

$$p_i''(x_i) = \frac{2k_{i-1} + 4k_i}{\Delta x_i} - \frac{6\Delta y_i}{\Delta x_i^2}$$

- Conditions on second derivatives: $p_i''(x_i) = p_{i+1}''(x_i) \quad i = 1, \ldots, n-1$

$$\frac{1}{\Delta x_i} k_{i-1} + \left(\frac{2}{\Delta x_i} + \frac{2}{\Delta x_{i+1}} \right) k_i + \frac{1}{\Delta x_{i+1}} k_{i+1} = \left(\frac{3\Delta y_i}{\Delta x_i^2} + \frac{3\Delta y_{i+1}}{\Delta x_{i+1}^2} \right)$$

($n-1$ conditions)

- Two more conditions from boundaries (natural, clamped, etc)
- Tridiagonal linear system of $n+1$ equations for $n+1$ unknowns k_i

Solving a Tridiagonal System

- Tridiagonal matrix algorithm (TDMA),
  also known as the Thomas algorithm
- Simplified form of Gaussian elimination to solve
  a tridiagonal system of $n + 1$ equations for $n + 1$ unknowns u_i

$$
\begin{aligned}
b_0 u_0 + c_0 u_1 &= d_0 \\
a_i u_{i-1} + b_i u_i + c_i u_{i+1} &= d_i, \quad i = 1, \dots, n-1 \\
a_n u_{n-1} + b_n u_n &= d_n
\end{aligned}
$$

- TDMA has complexity $\mathcal{O}(n)$ while Gaussian elimination has $\mathcal{O}(n^3)$

Solving a Tridiagonal System

- Forward pass: for $i = 1, 2, \ldots, n$

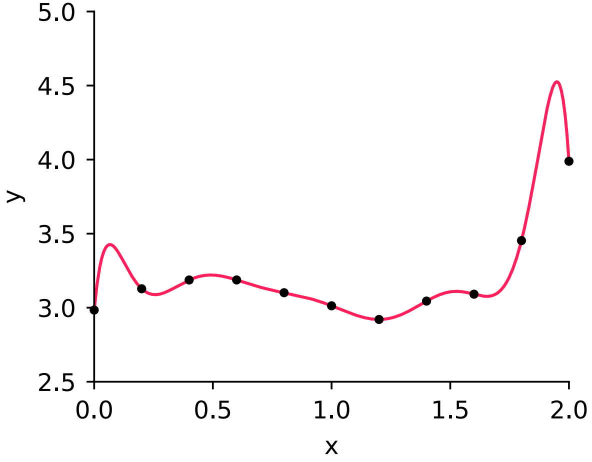
$$
\begin{aligned}
w &= a_i / b_{i-1} \\
b_i &\leftarrow b_i - w c_{i-1} \\
d_i &\leftarrow d_i - w d_{i-1}
\end{aligned}
$$

- Backward pass:

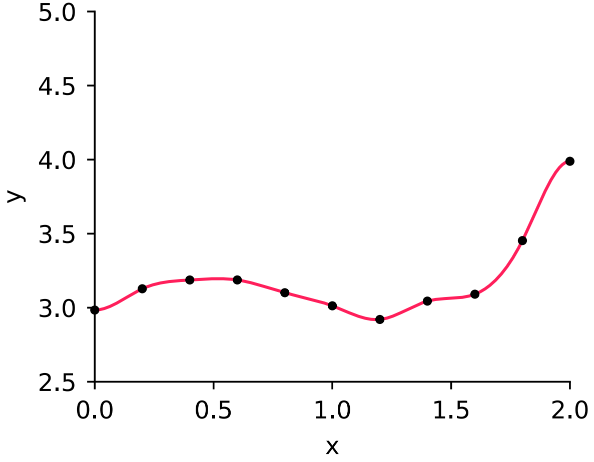
$$
\begin{aligned}
u_n &= d_n / b_n \\
u_i &= (d_i - c_i u_{i+1}) / b_i \quad \text{for } i = n-1, \ldots, 0
\end{aligned}
$$

Example of Spline Interpolation

- See [examples/unit1/spline_tdma.py]
- Spline looks smooth and does not have bumps or rapid changes



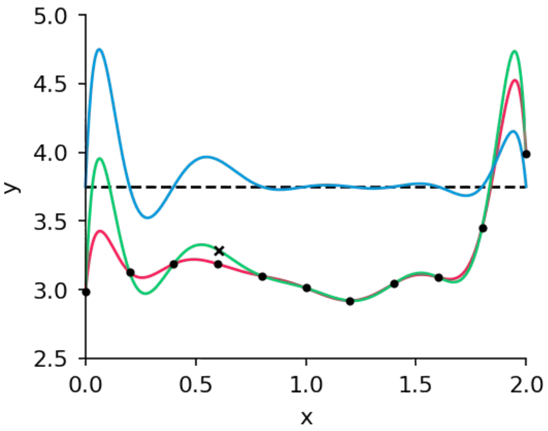
degree 10 polynomial



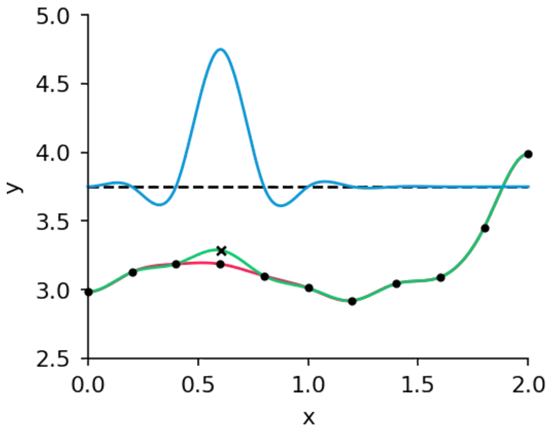
cubic spline

Example: Move One Point

- How does the interpolant change after moving one data point?
- original data, perturbed data, normalized change Δ (a.u.)
- Look at the normalized change $\Delta = (\tilde{f} - f)/\|(\tilde{f} - f)\|_\infty$
  - degree 10 polynomial: Δ remains constant
  - cubic spline: Δ changes in a nonlinear way



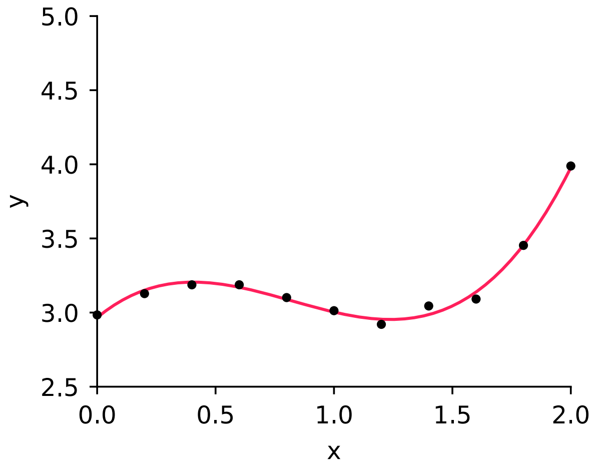
degree 10 polynomial



cubic spline

Linear Least Squares

- Recall that it can be advantageous to not fit data points exactly (e.g. to remove noise), we don't want to "overfit"
- Suppose we want to fit a cubic polynomial to 11 data points



- Question: How do we do this?

Linear Least Squares

- Suppose we have m constraints and n parameters with $m > n$ (on previous slide, $m = 11$ and $n = 4$)
- This is an overdetermined system $Ab = y$, where $A \in \mathbb{R}^{m \times n}$ (basis functions), $b \in \mathbb{R}^n$ (parameters), $y \in \mathbb{R}^m$ (data)

$$\begin{bmatrix} \\ \\ & A & \\ \\ \\ \\ \end{bmatrix} \begin{bmatrix} \\ b \\ \\ \end{bmatrix} = \begin{bmatrix} \\ \\ y \\ \\ \\ \\ \end{bmatrix}$$

Linear Least Squares

- In general, cannot be solved exactly;
  instead our goal is to minimize the residual, $r(b) \in \mathbb{R}^m$

$$r(b) = y - Ab$$

- A very effective approach for this is the method of least squares:
  Find parameter vector $b \in \mathbb{R}^n$ that minimizes $\|r(b)\|_2$
- The 2-norm is convenient since it gives us a differentiable function

Normal Equations

- Our goal is to minimize the objective function

$$\phi(b) := \|r(b)\|_2^2 = \sum_{i=1}^{n} r_i(b)^2$$

- In terms of A, b, and y

$$\begin{aligned}\phi(b) &= \|r\|_2^2 = r^T r = (y - Ab)^T (y - Ab) \\ &= y^T y - y^T Ab - b^T A^T y + b^T A^T Ab \\ &= y^T y - 2b^T A^T y + b^T A^T Ab\end{aligned}$$

where last line follows from $y^T Ab = (y^T Ab)^T$, since $y^T Ab \in \mathbb{R}$

- The minimum must exist since $\phi \geq 0$,
  but may be non-unique (e.g. $f(b_1, b_2) = b_1^2$)

Normal Equations

- To find minimum, set the derivative to zero ($\nabla = \nabla_b$)

$$\nabla \phi(b) = 0$$

- Derivative

$$\nabla \phi(b) = -2\nabla(b^T A^T y) + \nabla(b^T A^T A b)$$

- Rule for the first term

$$\frac{\partial}{\partial b_k} b^T c = \frac{\partial}{\partial b_k} \sum_{i=1}^{n} b_i c_i = c_k$$

$$\implies \nabla(b^T c) = c$$

Normal Equations

- Rule for the second term ($M = (m_{i,j})$)

$$\frac{\partial}{\partial b_k} b^T M b = \frac{\partial}{\partial b_k} \sum_{i,j=1}^{n} m_{i,j} b_i b_j = \sum_{i,j=1}^{n} m_{i,j} \frac{\partial}{\partial b_k} (b_i b_j) =$$

$$= \sum_{i,j=1}^{n} m_{i,j} (\delta_{i,k} b_j + b_i \delta_{j,k}) = \sum_{j=1}^{n} m_{k,j} b_j + \sum_{i=1}^{n} m_{i,k} b_i = (Mb)_k + (M^T b)_k$$

$$\implies \nabla (b^T M b) = Mb + M^T b$$

Normal Equations

- Putting it all together, we obtain

$$\nabla\phi(b) = -2A^T y + 2A^T Ab$$

- We set $\nabla\phi(b) = 0$, which is $-2A^T y + 2A^T Ab = 0$
- Finally, the linear least squares problem is equivalent to

$$A^T Ab = A^T y$$

- This square $n \times n$ system is known as the normal equations

Normal Equations

- For $A \in \mathbb{R}^{m \times n}$ with $m > n$,
  $A^T A$ is singular if and only if
  A is rank-deficient (columns are linearly dependent)
- Proof
  - $(\Rightarrow)$ Suppose $A^T A$ is singular. $\exists z \neq 0$ such that $A^T A z = 0$.
    Hence $z^T A^T A z = \|Az\|_2^2 = 0$, so that $Az = 0$.
    Therefore A is rank-deficient.
  - $(\Leftarrow)$ Suppose A is rank-deficient. $\exists z \neq 0$ such that $Az = 0$.
    Hence $A^T A z = 0$, so that $A^T A$ is singular.

Normal Equations

- Hence if A has full rank (i.e. $\mathrm{rank}(A) = n$)
  we can solve the normal equations to find the unique minimizer b
- However, in general it is a bad idea to solve the normal equations directly, because of condition-squaring (e.g. $\kappa(A^T A) = \kappa(A)^2$ for square matrices)
- We will consider more efficient methods later
  (e.g. singular value decomposition)

Example: Least-Squares Polynomial Fit

- Find a least-squares fit for degree 11 polynomial to 50 samples of $y = \cos(4x)$ for $x \in [0, 1]$
- Let's express the best-fit polynomial using the monomial basis

$$p(x; b) = \sum_{k=0}^{11} b_k x^k$$

- The ith condition we'd like to satisfy is

$$p(x_i; b) = \cos(4x_i)$$

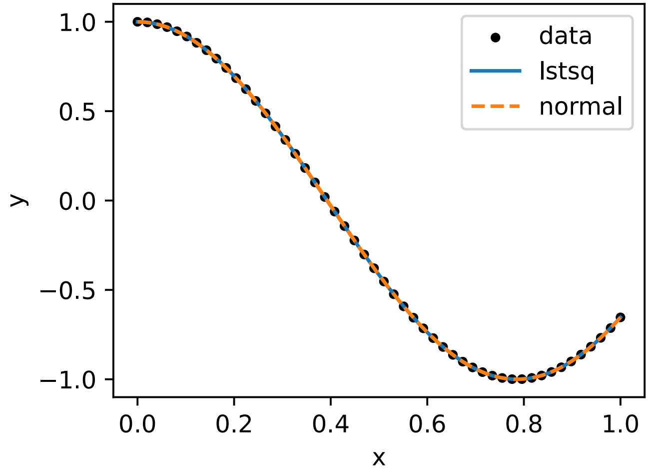
$\implies$ over-determined system with a 50×12 Vandermonde matrix

Example: Least-Squares Polynomial Fit

- See [examples/unit1/lstsq.py]
- Both methods give small residuals

$$\|r(b_{\text{lstsq}})\|_2 = \|y - Ab_{\text{lstsq}}\|_2 = 8.00 \times 10^{-9}$$

$$\|r(b_{\text{normal}})\|_2 = \|y - Ab_{\text{normal}}\|_2 = 1.09 \times 10^{-8}$$



Non-Polynomial Fitting

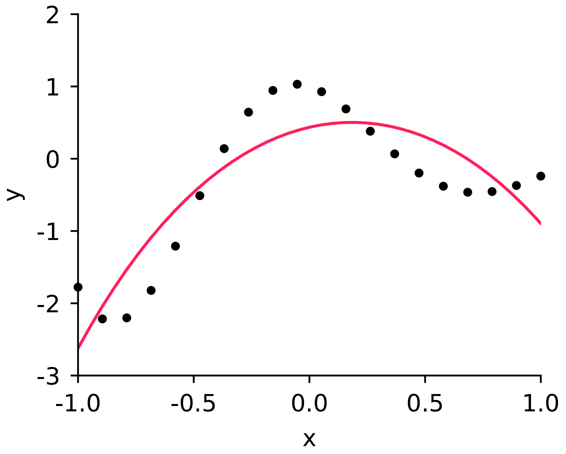
- Least-squares fitting can be used with arbitrary basis functions
- We just need a model that linearly depends on the parameters
- Example: Approximate $f(x) = e^{-x} \cos 4x$ using exponentials

$$f_n(x; b) = \sum_{k=-n}^{n} b_k e^{kx}$$

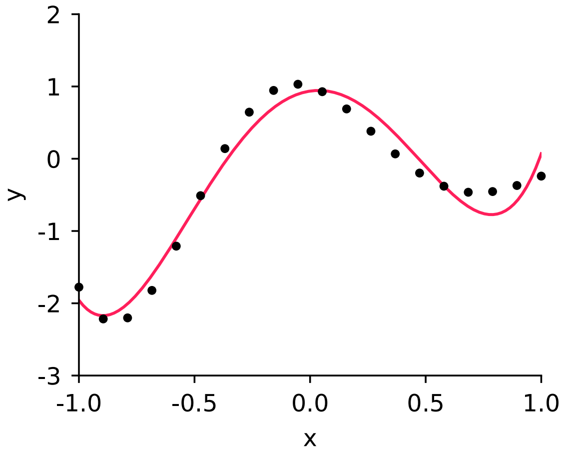
- See [examples/unit1/nonpoly_fit.py]

Non-Polynomial Fitting

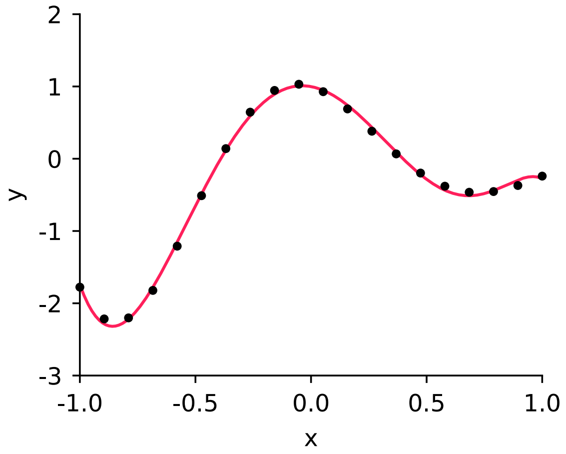
$$f_n(x;b) = b_{-n}e^{-nx} + b_{-n+1}e^{(-n+1)x} + \ldots + b_0 + \ldots + b_n e^{nx}$$



$$n = 1$$
$$\|r(b)\|_2 = 2.22$$



$$n = 2$$
$$\|r(b)\|_2 = 0.89$$



$$n = 3$$
$$\|r(b)\|_2 = 0.2$$

Non-Polynomial Fitting

- Why use non-polynomial basis functions?
  - recover properties of data
    (e.g. sine waves for periodic data)
  - control smoothness
    (e.g. splines correspond to a piecewise-polynomial basis)
  - control asymptotic behavior
    (e.g. require that functions do not grow fast at infinity)

Equivariance

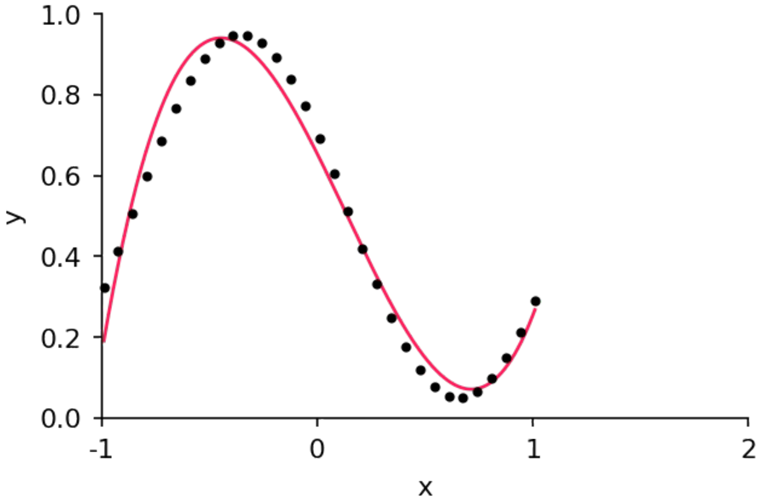
- A procedure is called equivariant to a transformation if applying the transformation to input (e.g. dataset) produces the same result as applying the transformation to output (e.g. fitted model)
- For example, consider a transformation $T(x)$ and find two models
  - $f(\cdot\,; b)$ that fits data (x_i, y_i)
  - $f(\cdot\,; \tilde{b})$ that fits data (Tx_i, y_i)
- The fitting is equivariant to T if

$$f(x; b) = f(Tx; \tilde{b})$$

- Does this hold for linear least squares? Depends on the basis
- (in common speech, used interchangeably with "invariance" but that actually stands for quantities not affected by transformations)

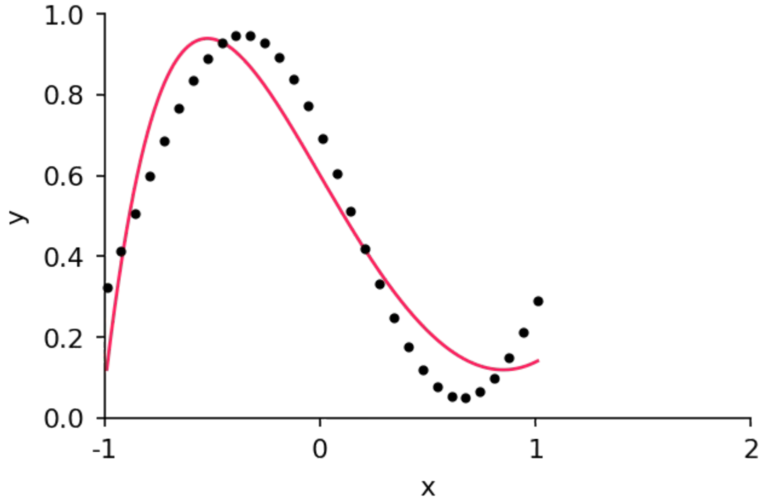
Example: Equivariance to Translation

$$T(x) = x + \lambda$$



$$1, \quad x, \quad x^2, \quad x^3$$

equivariant to translation

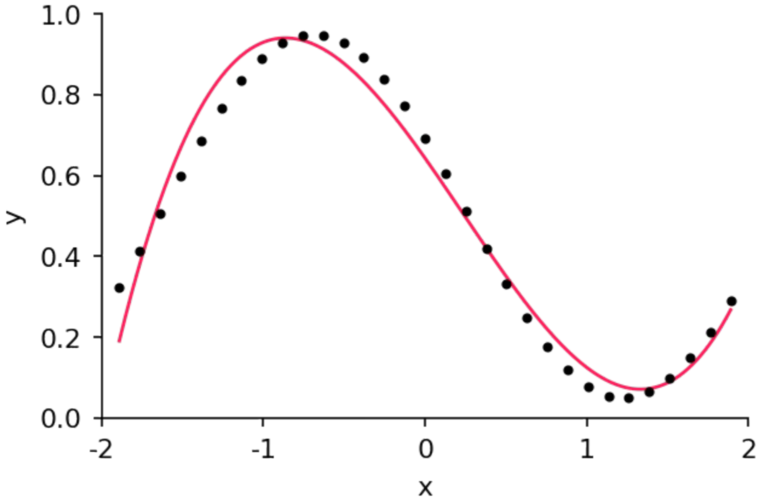


$$e^{-2x}, \quad e^{-x}, \quad 1, \quad e^{x}$$

equivariant to translation

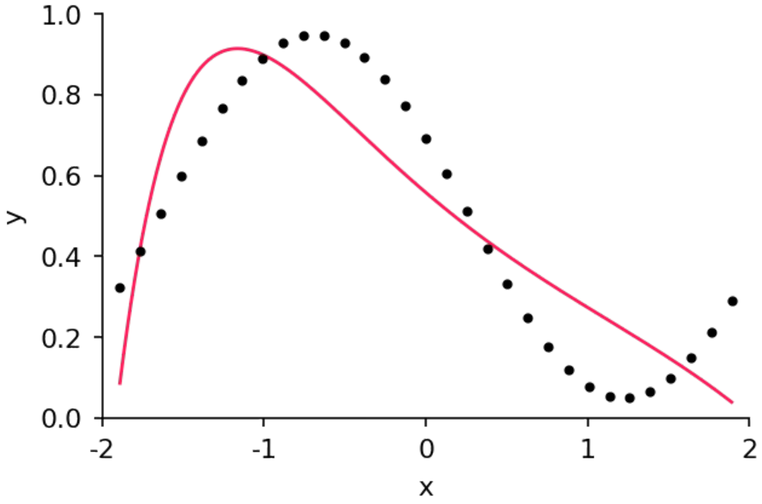
Example: Equivariance to Scaling

$$T(x) = \lambda x$$



$$1, \quad x, \quad x^2, \quad x^3$$

equivariant to scaling



$$e^{-2x}, \quad e^{-x}, \quad 1, \quad e^{x}$$

not equivariant to scaling

Pseudoinverse

- Recall that from the normal equations we have:

$$A^T A b = A^T y$$

- This motivates the idea of the "pseudoinverse" for $A \in \mathbb{R}^{m \times n}$:

$$A^+ = (A^T A)^{-1} A^T \in \mathbb{R}^{n \times m}$$

- Key point: A^+ generalizes A^{-1}, i.e. if $A \in \mathbb{R}^{n \times n}$ is invertible, then $A^+ = A^{-1}$
- Proof: $A^+ = (A^T A)^{-1} A^T = A^{-1}(A^T)^{-1} A^T = A^{-1}$

Pseudoinverse

- Also:
  - Even when A is not invertible we still have $A^+ A = I$
  - In general $AA^+ \neq I$ (hence this is called a “left inverse”)
- And it follows from our definition that $b = A^+ y$,
  i.e. $A^+ \in \mathbb{R}^{n \times m}$ gives the least-squares solution
- Note that we define the pseudoinverse differently in different contexts

Underdetermined Least Squares

- So far we have focused on overdetermined systems (more equations than parameters)
- But least-squares also applies to underdetermined systems: $Ab = y$ with $A \in \mathbb{R}^{m \times n}$, $m < n$

$$\begin{bmatrix} & & A & & \end{bmatrix} \begin{bmatrix} \\ b \\ \\ \end{bmatrix} = \begin{bmatrix} y \end{bmatrix}$$

Underdetermined Least Squares

- For $\phi(b) = \|r(b)\|_2^2 = \|y - Ab\|_2^2$, we can apply the same argument as before (i.e. set $\nabla\phi = 0$) to again obtain

$$A^T Ab = A^T y$$

- But in this case $A^T A \in \mathbb{R}^{n\times n}$ has rank at most m (where $m < n$), why?
- Therefore $A^T A$ must be singular!
- Typical case: There are infinitely many vectors b that give $r(b) = 0$, we want to be able to select one of them

Underdetermined Least Squares

- First idea, pose a constrained optimization problem to find the feasible b with minimum 2-norm:

$$\begin{array}{ll} \text{minimize} & b^T b \\ \hline \text{subject to} & Ab = y \end{array}$$

- This can be treated using Lagrange multipliers (discussed later in Unit 4)
- Idea is that the constraint restricts us to an $(n - m)$-dimensional hyperplane of $\mathbb{R}^n$ on which $b^T b$ has a unique minimum

Underdetermined Least Squares

- We will show later that the Lagrange multiplier approach for the above problem gives:

$$b = A^T(AA^T)^{-1}y$$

- Therefore, in the underdetermined case the pseudoinverse is defined as

$$A^+ = A^T(AA^T)^{-1} \in \mathbb{R}^{n \times m}$$

- Note that now $AA^+ = I$, but $A^+A \neq I$ in general (i.e. this is a “right inverse”)

Underdetermined Least Squares

- Here we consider an alternative approach for solving the underconstrained case
- Let's modify ϕ so that there is a unique minimum!
- For example, let

$$\phi(b) = \|r(b)\|_2^2 + \|Sb\|_2^2$$

  where $S \in \mathbb{R}^{n \times n}$ is a scaling matrix
- This is called regularization: we make the problem well-posed ("more regular") by modifying the objective function

Underdetermined Least Squares

- Calculating $\nabla\phi = 0$ in the same way as before leads to the system

$$(A^T A + S^T S)b = A^T y$$

- We need to choose S in some way to ensure $(A^T A + S^T S)$ is invertible
- Can be proved that if $S^T S$ is positive definite
  then $(A^T A + S^T S)$ is invertible
- Simplest positive definite regularizer:

$$S = \mu \mathbf{I} \in \mathbb{R}^{n \times n}$$

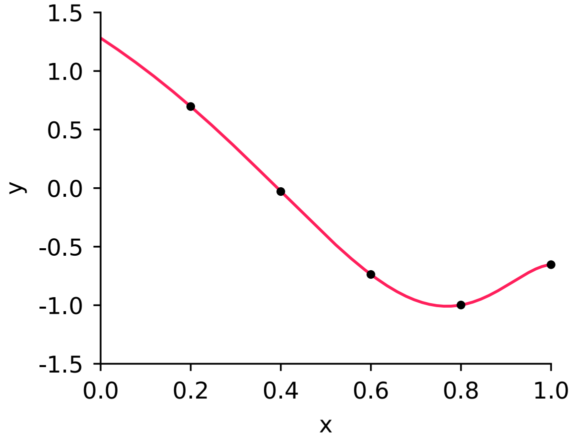
  for $\mu > 0$, $\mu \in \mathbb{R}$

Underdetermined Least Squares

- See [examples/unit1/under_lstsq.py]
- Find least-squares fit for degree 11 polynomial to 5 samples of $y = \cos(4x)$ for $x \in [0, 1]$
- 12 parameters, 5 constraints $\implies A \in \mathbb{R}^{5\times 12}$
- We express the polynomial using the monomial basis: A is a submatrix of a Vandermonde matrix
- Let's see what happens when we regularize the problem with some different choices of S

Underdetermined Least Squares

- Find least-squares fit for degree 11 polynomial to 5 samples of $y = \cos(4x)$ for $x \in [0, 1]$
- Try $S = 0.001\mathrm{I}$ (i.e. $\mu = 0.001$)



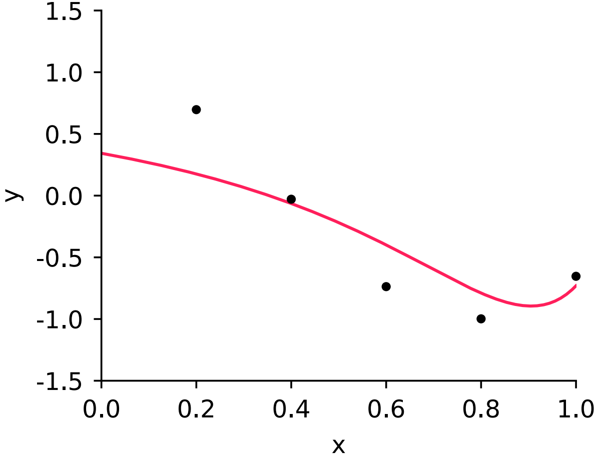
$$\|r(b)\|_2 = 1.07 \times 10^{-4}$$

$$\mathrm{cond}(A^T A + S^T S) = 1.54 \times 10^7$$

- Fit is good since regularization term is small but condition number is still large

Underdetermined Least Squares

- Find least-squares fit for degree 11 polynomial to 5 samples of $y = \cos(4x)$ for $x \in [0, 1]$
- Try $S = 0.5\mathrm{I}$ (i.e. $\mu = 0.5$)



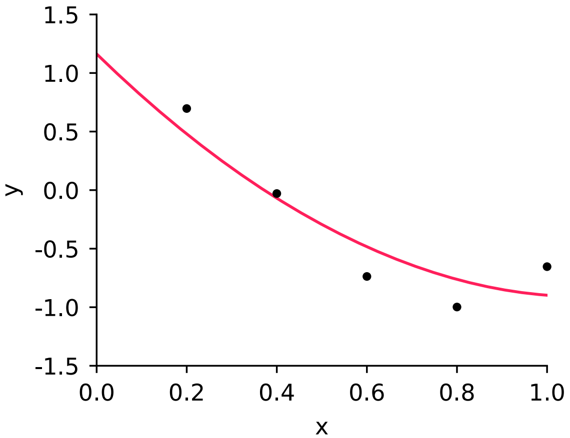
$$\|r(b)\|_2 = 6.60 \times 10^{-1}$$

$$\mathrm{cond}(A^T A + S^T S) = 62.3$$

- Regularization term now dominates: small condition number and small $\|b\|_2$, but poor fit to the data!

Underdetermined Least Squares

- Find least-squares fit for degree 11 polynomial to 5 samples of $y = \cos(4x)$ for $x \in [0, 1]$
- Try $S = \texttt{diag}(0.1, 0.1, 0.1, 10, 10 \ldots, 10)$



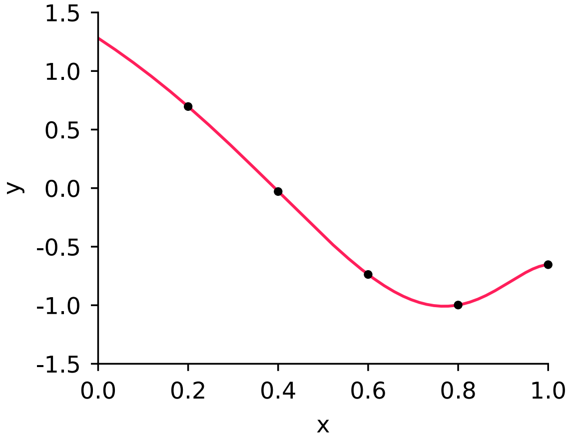
$$\|r(b)\|_2 = 4.78 \times 10^{-1}$$

$$\mathrm{cond}(A^T A + S^T S) = 5.90 \times 10^3$$

- We strongly penalize $b_3, b_4, \ldots, b_{11}$, hence the fit is close to parabolic

Underdetermined Least Squares

- Find least-squares fit for degree 11 polynomial to 5 samples of $y = \cos(4x)$ for $x \in [0, 1]$
- Use **`numpy.lstsq`**



$$\|r(b)\|_2 = 4.56 \times 10^{-15}$$

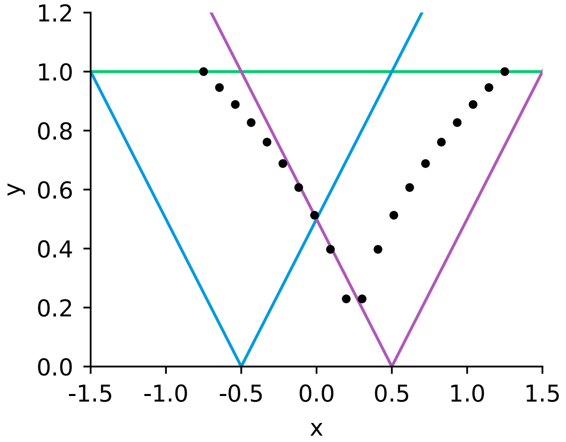
- Python routine uses Lagrange multipliers, hence satisfies the constraints to machine precision

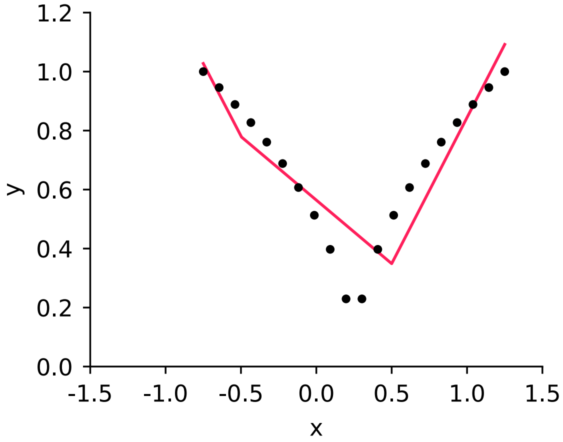
Nonlinear Least Squares

- So far we have looked at finding a "best fit" solution to a linear system (linear least-squares)
- A more difficult situation is when we consider least-squares for nonlinear systems
- Key point: Linear least-squares fitting of model $f(x; b)$ refers to linearity in the parameters b, while the model can be a nonlinear function of x (e.g. a polynomial $f(x; b) = b_0 + \ldots + b_n x^n$ is linear in b but nonlinear in x)
- In nonlinear least squares, we fit models that are nonlinear in the parameters

Nonlinear Least Squares: Motivation

- Consider a linear least-squares fit of $f(x) = \sqrt{|x - 0.25|}$



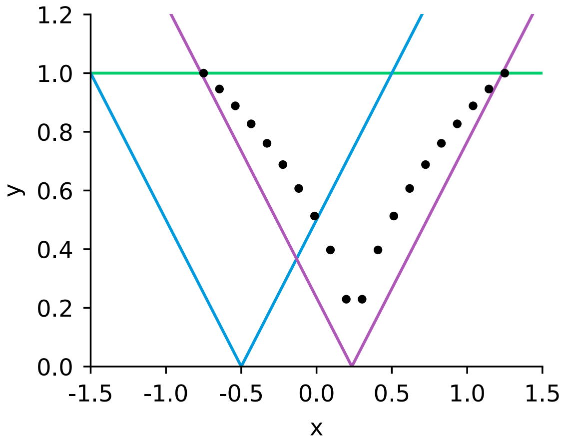


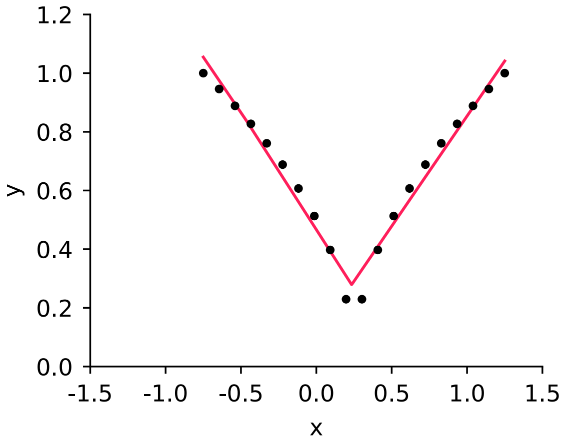
basis: 1, $|x + 0.5|$, $|x - 0.5|$

$0.07 + 0.28\,|x + 0.5| + 0.71\,|x - 0.5|$

Nonlinear Least Squares: Motivation

- We can improve the accuracy using "adaptive" basis functions, but now the model is nonlinear in λ





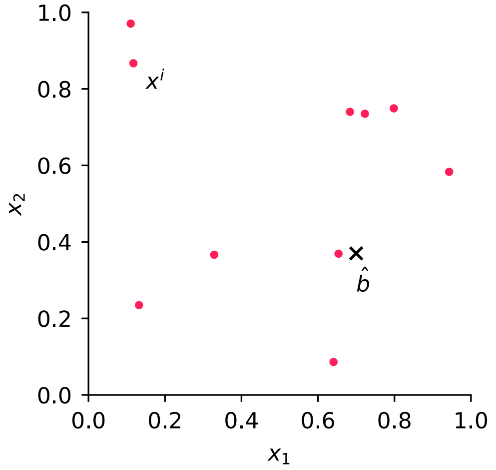
basis: 1, $|x + 0.5|$, $|x - \lambda|$

$$-0.3 - 0.03\,|x + 0.5| + 0.78\,|x - \lambda|$$

$$\lambda = 0.23$$

Nonlinear Least Squares: Example

- Example: Suppose we have a radio transmitter at $\hat{b} = (\hat{b}_1, \hat{b}_2)$ somewhere in $[0, 1]^2$ ($\times$)
- Suppose that we have 10 receivers at locations $(x_1^1, x_2^1), (x_1^2, x_2^2), \ldots, (x_1^{10}, x_2^{10}) \in [0, 1]^2$ (●)
- Receiver i returns the distance y_i to the transmitter, but there is some error/noise (ϵ)



Nonlinear Least Squares: Example

- Let b be a candidate location for the transmitter
- The distance from b to (x_1^i, x_2^i) is

$$d_i(b) = \sqrt{(b_1 - x_1^i)^2 + (b_2 - x_2^i)^2}$$

- We want to choose b to match the data as well as possible, hence minimize the residual $r(b) \in \mathbb{R}^{10}$ where $r_i(b) = y_i - d_i(b)$

Nonlinear Least Squares: Example

- In this case, $r_i(\alpha + \beta) \neq r_i(\alpha) + r_i(\beta)$, hence nonlinear least-squares!
- Define the objective function

$$\phi(b) = \frac{1}{2}\|r(b)\|_2^2$$

where $r(b) \in \mathbb{R}^{10}$ is the residual vector

- The $\frac{1}{2}$ factor has no effect on the minimizing b, but leads to slightly cleaner formulas later on

Nonlinear Least Squares

- As in the linear case, we seek to minimize ϕ by finding b such that $\nabla\phi = 0$
- We have $\phi(b) = \frac{1}{2}\sum_{j=1}^{m}(r_j(b))^2$
- Hence for the i-component of the gradient vector, we have

$$\frac{\partial \phi}{\partial b_i} = \frac{\partial}{\partial b_i}\frac{1}{2}\sum_{j=1}^{m} r_j^2 = \sum_{j=1}^{m} r_j \frac{\partial r_j}{\partial b_i}$$

Nonlinear Least Squares

- This is equivalent to $\nabla\phi = J_r(b)^T r(b)$
  where $J_r(b) \in \mathbb{R}^{m\times n}$ is the Jacobian matrix of the residual

$$\{J_r(b)\}_{ij} = \frac{\partial r_i(b)}{\partial b_j}$$

- Exercise: Show that $J_r(b)^T r(b) = 0$ reduces
  to the normal equations when the residual is linear

Nonlinear Least Squares

- Hence we seek $b \in \mathbb{R}^n$ such that:

$$J_r(b)^T r(b) = 0$$

- This has n equations, n unknowns
- In general, this is a nonlinear system that we have to solve iteratively
- A common situation is that linear systems can be solved in "one shot", while nonlinear generally requires iteration
- We will briefly introduce Newton's method for solving this system and defer detailed discussion until Unit 4

Nonlinear Least Squares

- Recall Newton's method for a function of one variable:
  find $x \in \mathbb{R}$ such that $f(x) = 0$
- Let x_k be our current guess, and $x_k + \Delta x = x$, then Taylor expansion gives

$$0 = f(x_k + \Delta x) = f(x_k) + \Delta x f'(x_k) + O((\Delta x)^2)$$

- It follows that $f'(x_k)\Delta x \approx -f(x_k)$
  (approx. since we neglect the higher order terms)
- This motivates Newton's method:

$$f'(x_k)\Delta x_k = -f(x_k)$$

where $x_{k+1} = x_k + \Delta x_k$

Nonlinear Least Squares

- This argument generalizes directly to functions of several variables
- For example, suppose $F : \mathbb{R}^n \to \mathbb{R}^n$, then find x s.t. $F(x) = 0$ by

$$J_F(x_k)\Delta x_k = -F(x_k)$$

where J_F is the Jacobian of F, $\Delta x_k \in \mathbb{R}^n$, $x_{k+1} = x_k + \Delta x_k$

Nonlinear Least Squares

- In the case of nonlinear least squares,
  to find a stationary point of ϕ we need to find b such that

$$J_r(b)^T r(b) = 0$$

- That is, we want to solve $F(b) = 0$ for $F(b) = J_r(b)^T r(b)$
- We apply Newton's Method, hence need to find the Jacobian J_F of the function $F : \mathbb{R}^n \to \mathbb{R}^n$

Nonlinear Least Squares

- Consider $\frac{\partial F_i}{\partial b_j}$ (this will be the ij entry of J_F):

$$\begin{aligned}
\frac{\partial F_i}{\partial b_j} &= \frac{\partial}{\partial b_j}\left(J_r(b)^T r(b)\right)_i \\
&= \frac{\partial}{\partial b_j}\sum_{k=1}^{m}\frac{\partial r_k}{\partial b_i} r_k \\
&= \sum_{k=1}^{m}\frac{\partial r_k}{\partial b_i}\frac{\partial r_k}{\partial b_j} + \sum_{k=1}^{m}\frac{\partial^2 r_k}{\partial b_i \partial b_j} r_k
\end{aligned}$$

Gauss–Newton Method

- It is generally difficult to deal with the second derivatives in the previous formula (numerical sensitivity, cost, complex derivation)
- Key observation: As we approach a good fit to the data, the residual values $r_k(b)$, $1 \leq k \leq m$, should be small
- Hence we omit the term $\sum_{k=1}^{m} r_k \frac{\partial^2 r_k}{\partial b_i \partial b_j}$.

Gauss–Newton Method

- Note that $\sum_{k=1}^{m} \frac{\partial r_k}{\partial b_j} \frac{\partial r_k}{\partial b_i} = (J_r(b)^T J_r(b))_{ij}$,
  so that when the residual is small $J_F(b) \approx J_r(b)^T J_r(b)$
- Then putting all the pieces together, we obtain the iteration

$$J_r(b_k)^T J_r(b_k) \Delta b_k = -J_r(b_k)^T r(b_k)$$

  where $b_{k+1} = b_k + \Delta b_k$
- This is known as the Gauss–Newton Algorithm
  for nonlinear least squares

Gauss–Newton Method

- This looks similar to Normal Equations at each iteration, except now the matrix $J_r(b_k)$ comes from linearizing the residual
- Gauss–Newton is equivalent to solving the linear least squares problem at each iteration

$$J_r(b_k)\Delta b_k = -r(b_k)$$

- This is a common approach: replace a nonlinear problem with a sequence of linearized problems

Computing the Jacobian

- To use Gauss–Newton in practice, we need to be able to compute the Jacobian matrix $J_r(b_k)$ for any $b_k \in \mathbb{R}^n$
- We can do this "by hand",
  e.g. in our transmitter/receiver problem we would have:

$$[J_r(b)]_{ij} = -\frac{\partial}{\partial b_j}\sqrt{(b_1 - x_1^i)^2 + (b_2 - x_2^i)^2}$$

- Differentiating by hand is feasible in this case,
  but it can become impractical if $r(b)$ is more complicated
- Or perhaps our mapping $b \to y$ is a "black box"

Computing the Jacobian

- Alternative approaches
  - Finite difference approximation

$$[J_r(b_k)]_{ij} \approx \frac{r_i(b_k + e_j h) - r_i(b_k)}{h}$$

    (requires only function evaluations, but prone to rounding errors)
  - Symbolic computations
    Rule-based computation of derivatives (e.g. SymPy in Python)
  - Automatic differentiation
    Carry information about derivatives through every operation
    (e.g. use TensorFlow or PyTorch)

Gauss–Newton Method

- We derived the Gauss–Newton algorithm method in a natural way:
  - apply Newton's method to solve $\nabla\phi = 0$
  - neglect the second derivative terms that arise
- However, Gauss–Newton is not widely used in practice since it doesn't always converge reliably

Levenberg–Marquardt Method

- A more robust variation of Gauss–Newton is the Levenberg–Marquardt Algorithm, which uses the update

$$[J^T(b_k)J(b_k) + \mu_k \operatorname{diag}(S^T S)]\Delta b = -J(b_k)^T r(b_k)$$

where $S = \mathrm{I}$ or $S = J(b_k)$, and some heuristics to choose μ_k

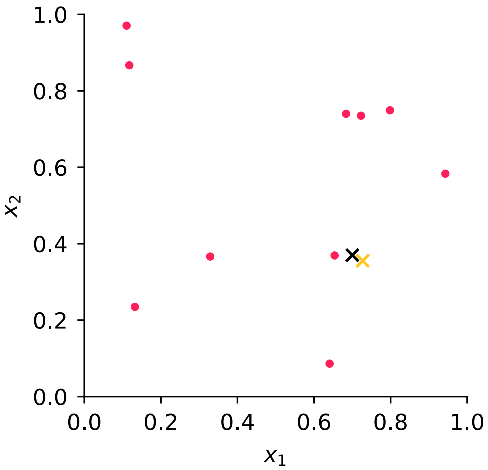
- This looks like our “regularized” underdetermined linear least squares formulation!

Levenberg–Marquardt Method

- Key point: The regularization term $\mu_k \operatorname{diag}(S^T S)$ improves the reliability of the algorithm in practice
- Levenberg–Marquardt is available SciPy
- We need to pass the residual to the routine, and we can also pass the Jacobian matrix or ask to use finite-differences
- Now let's solve our transmitter/receiver problem

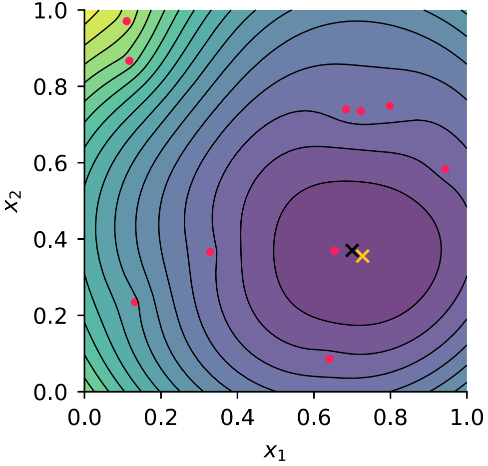
Nonlinear Least Squares: Example

- See [examples/unit1/nonlin_lstsq.py]



Nonlinear Least Squares: Example

- Levenberg–Marquardt minimizes $\phi(b)$



- The minimized objective is even lower than for the true location (because of the noise)

$$\phi(\times) = 0.0044 < 0.0089 = \phi(\times)$$

$\times$ is our best-fit to the data, $\times$ is the true transmitter location