Unit 1: Data Fitting

Motivation

Data fitting: Construct a continuous function that represents discrete data, fundamental topic in Scientific Computing

We will study two types of data fitting

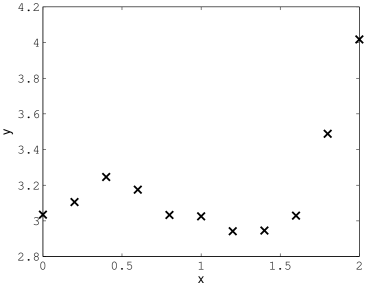
- interpolation: Fit the data points exactly
- least-squares: Minimize error in the fit (useful when there is experimental error, for example)

Data fitting helps us to

- interpret data: deduce hidden parameters, understand trends
- process data: reconstructed function can be differentiated, integrated, etc

Motivation

For example, suppose we are given the following data points



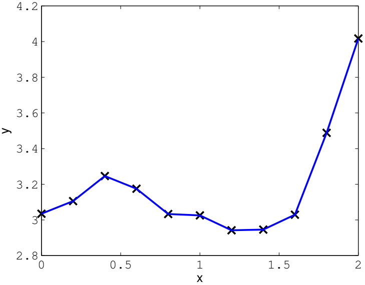
This data could represent

- 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

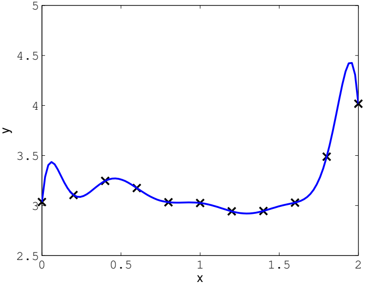
Easiest thing to do: "connect the dots" (piecewise linear interpolation)



Question: What if we want a smoother approximation?

Motivation

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



We will discuss how to construct this type of polynomial interpolant in this unit

Motivation

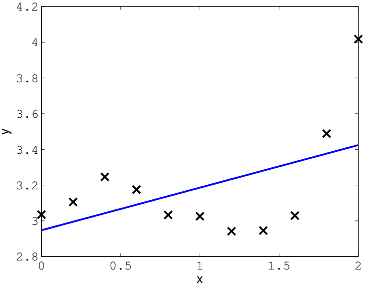
However, a degree 10 interpolant is not aesthetically pleasing: it is too bumpy, and doesn't seem to capture the underlying pattern

Maybe we can capture the data better with a lower order polynomial . . .

Motivation

Let's try linear regression (familiar from elementary statistics): minimize the error in a linear approximation of the data

Best linear fit: $y = 2.94 + 0.24x$

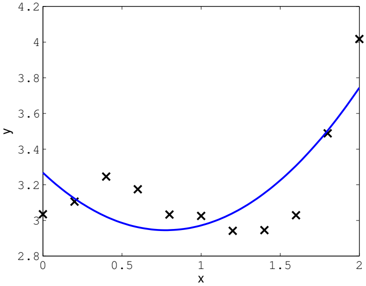


Clearly not a good fit!

Motivation

We can use least-squares fitting to generalize linear regression to higher order polynomials

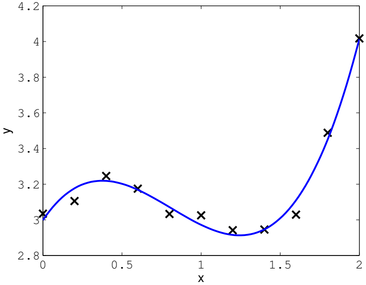
Best quadratic fit: $y = 3.27 - 0.83x + 0.53x^2$



Still not so good . . .

Motivation

Best cubic fit: $y = 3.00 + 1.31x - 2.27x^2 + 0.93x^3$



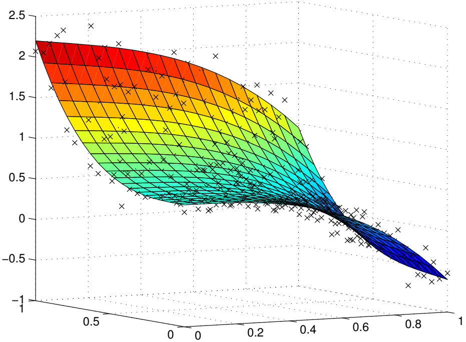
Looks good! A “cubic model” captures this data well

(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 (find the best hypersurface in $\mathbb{R}^N$)

2D example:



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

- Smooths out noise using a lower-dimensional model

These kinds of data-fitting calculations are often performed with huge datasets in practice

- Efficient and stable algorithms are very important

Polynomial Fitting: The Problem Formulation

Let $\mathbb{P}_n$ denote the set of all polynomials of degree n on $\mathbb{R}$

i.e. if $p(\cdot; b) \in \mathbb{P}_n$, then

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

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

The Problem Formulation

Suppose we have the data $\mathcal{S} \equiv \{(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)\}$, where the $\{x_0, x_1, \dots, x_n\}$ 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 $(x_i, y_i) \in \mathcal{S}$, *i.e.* $n + 1$ equations

For uniqueness, we should look for a polynomial with $n + 1$ parameters, *i.e.* look for $p \in \mathbb{P}_n$

Vandermonde Matrix

Then we obtain the following system of $n+1$ equations in $n+1$ unknowns

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

Vandermonde Matrix

This can be written in matrix form $Vb = y$, where

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

$$y = [y_0, y_1, \ldots, y_n]^T \in \mathbb{R}^{n+1}$$

and $V \in \mathbb{R}^{(n+1)\times(n+1)}$ is 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 $p(\cdot; b) \in \mathbb{P}_n$ vanishes at $n + 1$ distinct points

Therefore we must have $p(\cdot; b) = 0$, or equivalently $b = 0 \in \mathbb{R}^{n+1}$

Hence $Vb = 0 \implies b = 0$, so that $Vb = y$ 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 $Vb = y$

In general, however, this is a bad idea since V is ill-conditioned

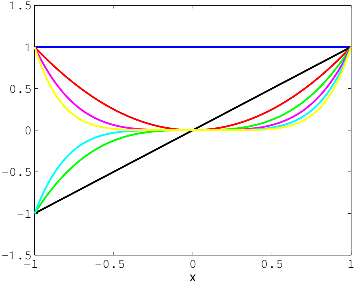
Monomial Interpolation

The problem here is that Vandermonde matrix corresponds to interpolation using the monomial basis

Monomial basis for $\mathbb{P}_n$ is $\{1, x, x^2, \ldots, x^n\}$

Monomial basis functions become increasingly indistinguishable

Vandermonde columns become nearly linearly-dependent [*vander_cond.txt*] $\Longrightarrow$ ill-conditioned matrix!



Monomial Basis

Question: What is the practical consequence of this ill-conditioning?

Answer:

- We want to solve $Vb = y$, but due to finite precision arithmetic we get an approximation $\hat{b}$
- $\hat{b}$ will ensure $\|V\hat{b} - y\|$ is small (in a rel. sense),[1] but $\|b - \hat{b}\|$ can still be large! (see Unit 2 for details)
- Similarly, small perturbation in $\hat{b}$ can give large perturbation in $V\hat{b}$
- Large perturbations in $V\hat{b}$ can yield large $\|V\hat{b} - y\|$, hence a "perturbed interpolant" becomes a poor fit to the data

[1] This "small residual" property is because we use a stable numerical algorithm for solving the linear system

Monomial Basis

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

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

Then, let's express $y = [1, 0]^T$ and $\tilde{y} = [1, 0.0005]^T$ in terms of this basis

We can do this by solving a 2 × 2 linear system in each case (see Unit 2), and hence we get

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

Hence the answer is highly sensitive to perturbations in y!

Monomial Basis

The same effect happens with interpolation with a monomial basis

The answer (the polynomial coefficient vector) is highly sensitive to perturbations in the data

If we perturb b slightly, we can get a large perturbation in Vb and then no longer solve the Vandermonde equation accurately. Hence the resulting polynomial no longer fits the data well.

See code examples:

- [*v_inter.py*] Vandermonde interpolation using text output, for plotting with Gnuplot
- [*v_inter2.py*] Vandermonde interpolation with native Python plotting using Matplotlib

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 as an added bonus we also avoid inverting 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[2]

See Lecture: These polynomials are given by:

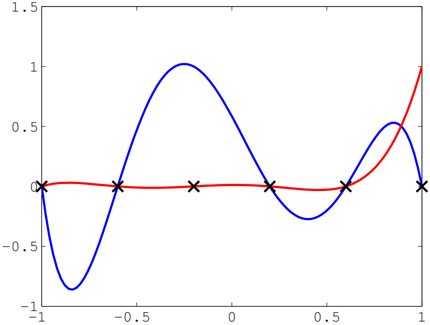
$$L_k(x) = \prod_{j=0, j\neq k}^{n} \frac{x - x_j}{x_k - x_j}$$

and then the interpolant can be expressed as
$p_n(x) = \sum_{k=0}^{n} y_k L_k(x)$

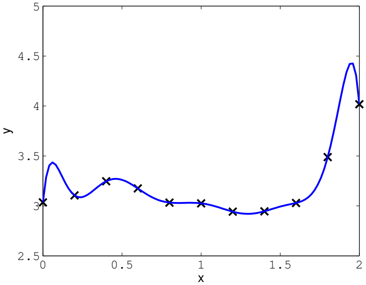
[2] Joseph-Louis Lagrange, 1736–1813

Lagrange Interpolation

Two Lagrange polynomials of degree 5



Hence we can use Lagrange polynomials to interpolate discrete data



We have essentially solved the problem of interpolating discrete data perfectly!

With Lagrange polynomials we can construct an interpolant of discrete data with condition number of 1

Interpolation for Function Approximation

We now turn to a different (and much deeper) 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[3]:

$$\boxed{f(x) - p_n(x) = \frac{f^{(n+1)}(\theta)}{(n+1)!}(x - x_0)\ldots(x - x_n) \text{ for some } \theta \in (a, b)}$$

[3] Augustin-Louis Cauchy, 1789–1857

Polynomial Interpolation Error

We prove this result in the case $n = 1$

Let $p_1 \in \mathbb{P}_1[x_0, x_1]$ interpolate $f \in C^2[a, b]$ at $\{x_0, x_1\}$

For some $\lambda \in \mathbb{R}$, let

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

here q is quadratic and interpolates f at $\{x_0, x_1\}$

Fix an arbitrary point $\hat{x} \in (x_0, x_1)$ and set $q(\hat{x}) = f(\hat{x})$ 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, since then we obtain an expression for $f(\hat{x}) - p_1(\hat{x})$

Polynomial Interpolation Error

Now, let $e(x) \equiv f(x) - q(x)$

- e has 3 roots in $[x_0, x_1]$, i.e. at $x = x_0, \hat{x}, x_1$
- Therefore e' has 2 roots in (x_0, x_1) (by Rolle's theorem)
- Therefore e'' has 1 root in (x_0, x_1) (by Rolle's theorem)

Let $\theta \in (x_0, x_1)$ be such that[4] $e''(\theta) = 0$

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

[4] Note that θ is a function of $\hat{x}$

Polynomial Interpolation Error

Hence, 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 $\hat{x} \in (x_0, x_1)$ (recall that $\hat{x}$ was chosen arbitrarily)

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

$$\boxed{f(x)-p_n(x) = \frac{f^{(n+1)}(\theta)}{(n+1)!}(x-x_0)\dots(x-x_n) \text{ for some } \theta \in (a, b)}$$

Polynomial Interpolation Error

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

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

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

If $1/(n+1)! \to 0$ faster than

$$M_{n+1} \max_{x \in [a,b]} |(x - x_0) \ldots (x - x_n)| \to \infty$$

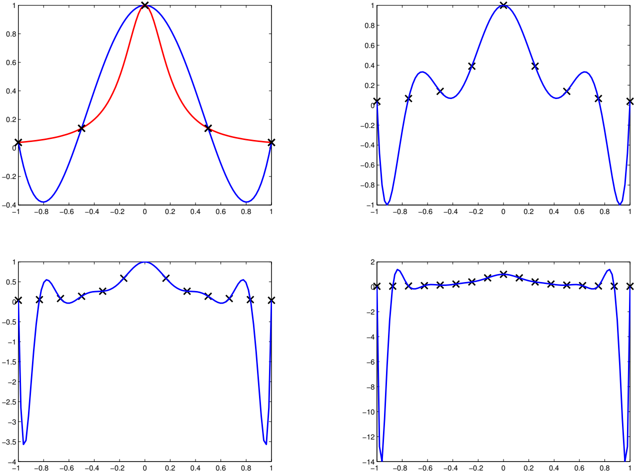
then $p_n \to f$

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 $f(x) = 1/(1 + 25x^2)$ for $x \in [-1, 1]$



Runge's Phenomenon

Note that of course p_n 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 generally referred to as the "infinity norm" or the "max norm":

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

Runge's Phenomenon

Interpolating Runge's function at evenly spaced points leads to exponential growth of infinity norm error!

We would like to construct an interpolant of f such that this kind of pathological behavior is impossible

Minimizing Interpolation Error

To do this, we recall our error equation

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

We focus our attention on the polynomial $(x - x_0)\dots(x - x_n)$, since we can choose the interpolation points

Intuitively, we should choose $x_0, x_1, \dots, x_n$ such that $\|(x - x_0)\dots(x - x_n)\|_\infty$ is as small as possible

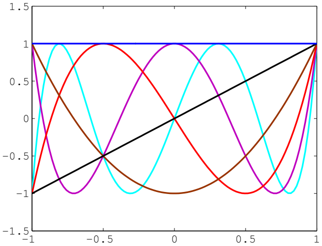
Interpolation at Chebyshev Points

Result from Approximation Theory:

For $x \in [-1, 1]$, the minimum value of $\|(x - x_0) \dots (x - x_n)\|_\infty$ is $1/2^n$, achieved by the polynomial $T_{n+1}(x)/2^n$

$T_{n+1}(x)$ is the Chebyshev poly. (of the first kind) of order $n + 1$ (T_{n+1} has leading coefficient of 2^n, hence $T_{n+1}(x)/2^n$ is monic)

Chebyshev polys "equi-oscillate" between -1 and 1, hence it's not surprising that they are related to the minimum infinity norm



Interpolation at Chebyshev Points

Chebyshev polynomials are defined for $x \in [-1, 1]$ by
$T_n(x) = \cos(n \cos^{-1} x), n = 0, 1, 2, \ldots$

Or equivalently[5], the recurrence relation,
$T_0(x) = 1,$
$T_1(x) = x,$
$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x),\ n = 1, 2, 3, \ldots$

To set $(x - x_0) \ldots (x - x_n) = T_{n+1}(x)/2^n$, we choose interpolation points to be the roots of T_{n+1}

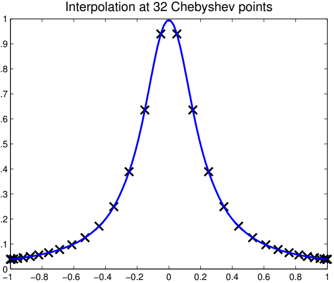
Exercise: Show that the roots of T_n are given by
$x_j = \cos((2j - 1)\pi/2n),\ j = 1, \ldots, n$

[5] Equivalence can be shown using trig. identities for T_{n+1} and T_{n-1}

Interpolation at Chebyshev Points

We can combine these results to derive an error bound for interpolation at "Chebyshev points"

Generally speaking, with Chebyshev interpolation, p_n converges to any smooth f very rapidly! *e.g.* Runge function:



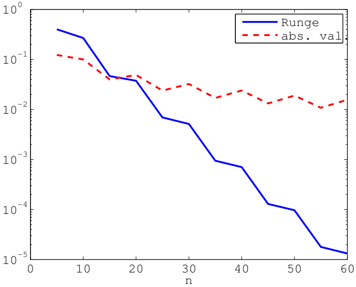
If we want to interpolate on an arbitrary interval, we can map Chebyshev points from $[-1,1]$ to $[a,b]$

Interpolation at Chebyshev Points

Note that convergence rates depend on smoothness of f—precise statements about this can be made, outside the scope of AM205

In general, smoother $f \implies$ faster convergence[6]

e.g. [*ch_inter.py*] compare convergence of Chebyshev interpolation of Runge's function (smooth) and $f(x) = |x|$ (not smooth)



[6] For example, if f is analytic, we get exponential convergence!

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, $\mathcal{X} \equiv \{x_0, x_1, \dots, x_n\} \subset [a, b]$

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 polynomials associated with $\mathcal{X}$, hence Λ_n is also a function of $\mathcal{X}$

$\Lambda_n(\mathcal{X}) \geq 1$, why?

Lebesgue Constant

Think of polynomial interpolation as a map, $\mathcal{I}_n$, where $\mathcal{I}_n : C[a,b] \rightarrow \mathbb{P}_n[a,b]$

$\mathcal{I}_n(f)$ 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:

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

Hence

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

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, *i.e.* $\|f - p_n^*\|_\infty \leq \|f - w\|_\infty$ 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
- In general, p_n^* is unknown

Bernstein interpolation

The Bernstein polynomials on $[0, 1]$ are

$$b_{m,n}(x) = \left(\begin{array}{c} n \\ m \end{array} \right) x^m (1-x)^{n-m}.$$

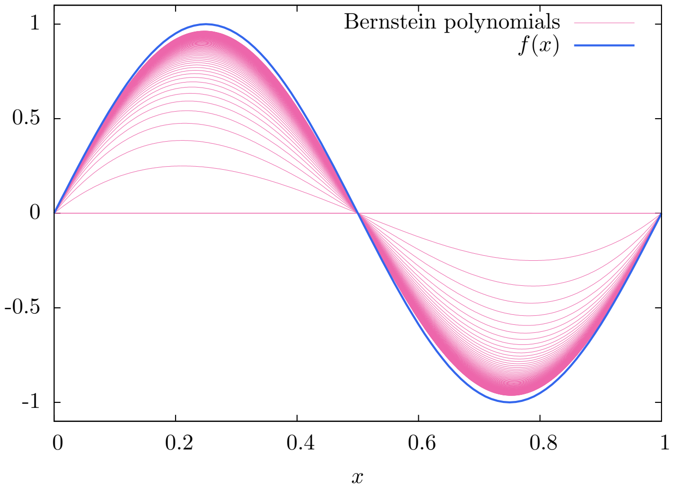
For a function f on the $[0, 1]$, the approximating polynomial is

$$B_n(f)(x) = \sum_{m=0}^{n} f\left(\frac{m}{n}\right) b_{m,n}(x).$$

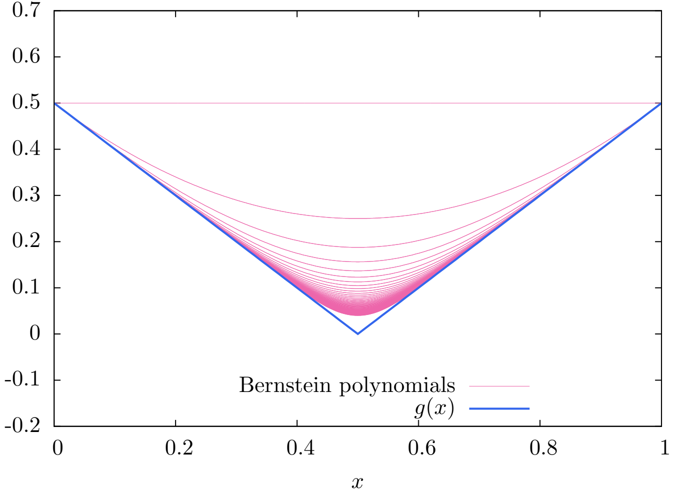
Bernstein interpolation is impractical for normal use, and converges extremely slowly. However, it has robust convergence properties, which can be used to prove the Weierstraß approximation theorem. See a textbook on real analysis for a full discussion.[7]

[7] *e.g. Elementary Analysis* by Kenneth A. Ross

Bernstein interpolation



Bernstein interpolation



Lebesgue Constant

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

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

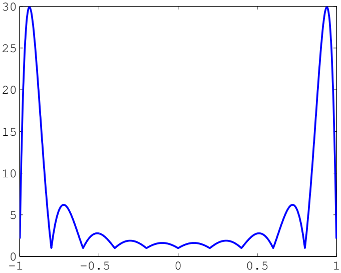
Lebesgue Constant

Small Lebesgue constant means that our interpolation can't be much worse that the best possible polynomial approximation!

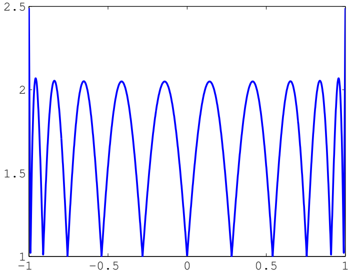
[*lsum.py*] Now let's compare Lebesgue constants for equispaced ($\mathcal{X}_{\mathrm{equi}}$) and Chebyshev points ($\mathcal{X}_{\mathrm{cheb}}$)

Lebesgue Constant

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



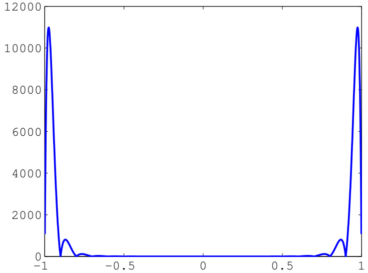
$\Lambda_{10}(\mathcal{X}_{\text{equi}}) \approx 29.9$

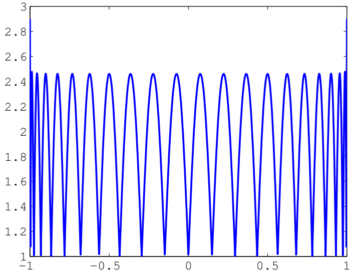


$\Lambda_{10}(\mathcal{X}_{\text{cheb}}) \approx 2.49$

Lebesgue Constant

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



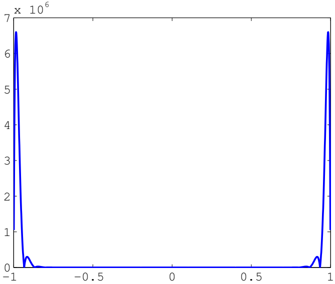


$\Lambda_{20}(\mathcal{X}_{\text{equi}}) \approx 10{,}987$

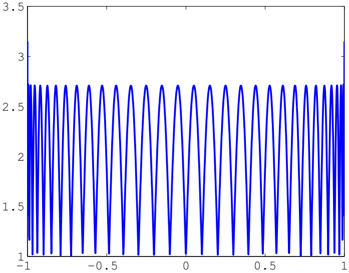
$\Lambda_{20}(\mathcal{X}_{\text{cheb}}) \approx 2.9$

Lebesgue Constant

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



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



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

Lebesgue Constant

The explosive growth of $\Lambda_n(\mathcal{X}_{\mathrm{equi}})$ is an explanation for Runge's phenomenon[8]

It has been shown that as $n \to \infty$,

$$\Lambda_n(\mathcal{X}_{\mathrm{equi}}) \sim \frac{2^n}{e n \log n} \qquad \text{BAD!}$$

whereas

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

Important open mathematical problem: What is the optimal set of interpolation points (*i.e.* what $\mathcal{X}$ minimizes $\Lambda_n(\mathcal{X})$)?

[8] Runge's function $f(x) = 1/(1+25x^2)$ excites the “worst case” behavior allowed by $\Lambda_n(\mathcal{X}_{\mathrm{equi}})$

Summary

It is helpful to compare and contrast the two key topics we've considered so far in this chapter

1. **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
    - Should use Lagrange polynomial basis (diagonal system, well-conditioned)

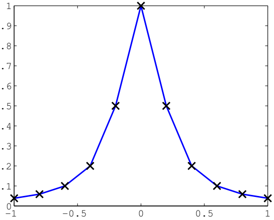
2. **Polynomial interpolation for approximating continuous functions:**
    - For a given set of interpolating points, uses the methodology from 1 above to construct the interpolant
    - But now interpolation points play a crucial role in determining the magnitude of the error $\|f - \mathcal{I}_n(f)\|_\infty$

Piecewise Polynomial Interpolation

We can't always choose our interpolation points to be Chebyshev, so another way to avoid "blow up" is via piecewise polynomials

Idea is simple: Break domain into subdomains, apply polynomial interpolation on each subdomain (interp. pts. now called "knots")

Recall piecewise linear interpolation, also called "linear spline"



Piecewise Polynomial Interpolation

With piecewise polynomials, we avoid high-order polynomials hence we avoid "blow-up"

However, we clearly lose smoothness of the interpolant

Also, can't do better than algebraic convergence[9]

[9] Recall that for smooth functions Chebyshev interpolation gives exponential convergence with n

Splines

Splines are a popular type of piecewise polynomial interpolant

In general, a spline of degree k is a piecewise polynomial that is continuously differentiable $k - 1$ times

Splines solve the "loss of smoothness" issue to some extent since they have continuous derivatives

Splines are the basis of CAD software (AutoCAD, SolidWorks), also used in vector graphics, fonts, *etc.*[10]

(The name "spline" comes from a tool used by ship designers to draw smooth curves by hand)

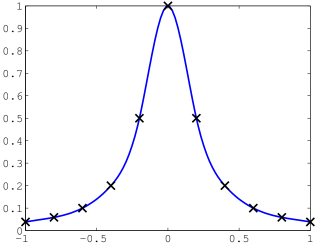
[10] CAD software uses NURB splines, font definitions use Bézier splines

Splines

We focus on a popular type of spline: Cubic spline $\in C^2[a, b]$

Continuous second derivatives $\implies$ looks smooth to the eye

Example: Cubic spline interpolation of Runge function



Cubic Splines: Formulation

Suppose we have knots $x_0, \ldots, x_n$, then cubic on each interval $[x_{i-1}, x_i] \implies$ $4n$ parameters in total

Let s denote our cubic spline, and suppose we want to interpolate the data $\{f_i, i = 0, 1, \ldots, n\}$

We must interpolate at $n + 1$ points, $s(x_i) = f_i$, which provides two equations per interval $\implies$ $2n$ equations for interpolation

Also, $s'_{-}(x_i) = s'_{+}(x_i)$, $i = 1, \ldots, n - 1 \implies$ $n - 1$ equations for continuous first derivative

And, $s''_{-}(x_i) = s''_{+}(x_i)$, $i = 1, \ldots, n - 1 \implies$ $n - 1$ equations for continuous second derivative

Hence $4n - 2$ equations in total

Cubic Splines

We are short by two conditions! There are many ways to make up the last two, *e.g.*

- Natural cubic spline: Set $s''(x_0) = s''(x_n) = 0$
- "Not-a-knot spline"[11]: Set $s'''_-(x_1) = s'''_+(x_1)$ and $s'''_-(x_{n-1}) = s'''_+(x_{n-1})$
- Or we can choose any other two equations we like (*e.g.* set two of the spline parameters to zero)[12]

See examples: [*spline.py*] & [*spline2.py*]

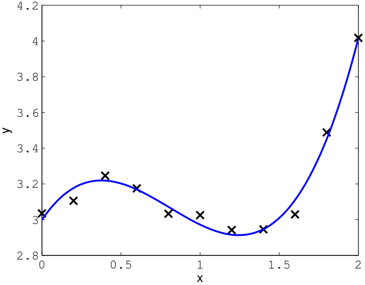
[11] "Not-a-knot" because all derivatives of s are continuous at x_1 and x_{n-1}

[12] As long as they are linearly independent from the first $4n-2$ equations

Linear Least Squares: The Problem Formulation

Recall that it can be advantageous to not fit data points exactly (*e.g.* due to experimental error), we don't want to "overfit"

Suppose we want to fit a cubic polynomial to 11 data points



Question: How do we do this?

The Problem Formulation

Suppose we have m constraints and n parameters with $m > n$ (*e.g.* $m = 11$, $n = 4$ on previous slide)

In terms of linear algebra, this is an overdetermined system $Ab = y$, where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^n$ (parameters), $y \in \mathbb{R}^m$ (data)

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

i.e. we have a "tall, thin" matrix A

The Problem Formulation

In general, cannot be solved exactly (hence we will write $Ab \simeq y$); instead our goal is to minimize the residual, $r(b) \in \mathbb{R}^m$

$$r(b) \equiv y - Ab$$

A very effective approach for this is the method of least squares:[13]
Find parameter vector $b \in \mathbb{R}^n$ that minimizes $\|r(b)\|_2$

As we shall see, we minimize the 2-norm above since it gives us a differentiable function (we can then use calculus)

[13] Developed by Gauss and Legendre for fitting astronomical observations with experimental error

The Normal Equations

Goal is to minimize $\|r(b)\|_2$, recall that $\|r(b)\|_2 = \sqrt{\sum_{i=1}^{n} r_i(b)^2}$

Since minimizing b is the same for $\|r(b)\|_2$ and $\|r(b)\|_2^2$, we consider the differentiable "objective function" $\phi(b) = \|r(b)\|_2^2$

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

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

ϕ is a quadratic function of b, and is non-negative, hence a minimum must exist, (but not nec. unique, *e.g.* $f(b_1, b_2) = b_1^2$)

The Normal Equations

To find minimum of $\phi(b) = y^T y - 2b^T A^T y + b^T A^T A b$, differentiate with respect to b and set to zero[14]

First, let's differentiate $b^T A^T y$ with respect to b

That is, we want $\nabla(b^T c)$ where $c \equiv A^T y \in \mathbb{R}^n$:

$$b^T c = \sum_{i=1}^{n} b_i c_i \implies \frac{\partial}{\partial b_i}(b^T c) = c_i \implies \nabla(b^T c) = c$$

Hence $\nabla(b^T A^T y) = A^T y$

[14] We will discuss numerical optimization of functions of many variables in detail in Unit IV

The Normal Equations

Next consider $\nabla(b^T A^T A b)$ (note $A^T A$ is symmetric)

Consider $b^T M b$ for symmetric matrix $M \in \mathbb{R}^{n \times n}$

$$b^T M b = b^T \left(\sum_{j=1}^{n} m_{(:,j)} b_j \right)$$

From the product rule

$$\begin{aligned}
\frac{\partial}{\partial b_k}(b^T M b) &= e_k^T \sum_{j=1}^{n} m_{(:,j)} b_j + b^T m_{(:,k)} \\
&= \sum_{j=1}^{n} m_{(k,j)} b_j + b^T m_{(:,k)} \\
&= m_{(k,:)} b + b^T m_{(:,k)} \\
&= 2 m_{(k,:)} b,
\end{aligned}$$

where the last line follows from symmetry of M

Therefore, $\nabla(b^T M b) = 2Mb$, so that $\nabla(b^T A^T A b) = 2A^T A b$

The Normal Equations

Putting it all together, we obtain

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

We set $\nabla\phi(b) = 0$ to obtain

$$-2A^T y + 2A^T Ab = 0 \Longrightarrow A^T Ab = A^T y$$

This square $n \times n$ system $A^T Ab = A^T y$ is known as the normal equations

The 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.[15]

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.

[15] Recall $A \in \mathbb{R}^{m \times n}$, $m > n$ is rank-deficient if columns are not L.I., *i.e.* $\exists z \neq 0$ s.t. $Az = 0$

The Normal Equations

Hence if A has full rank (*i.e.* $\text{rank}(A) = n$) 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 it is not as numerically stable as some alternative methods

Question: If we shouldn't use normal equations, how do we actually solve least-squares problems ?

Least-squares polynomial fit

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

(Why not use the Lagrange basis? Lagrange loses its nice properties here since $m > n$, so we may as well use monomials)

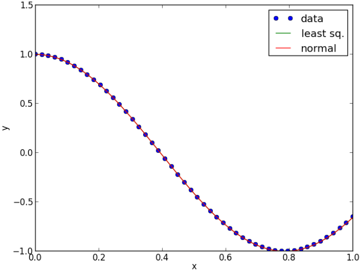
The ith condition we'd like to satisfy is $p(x_i; b) = \cos(4x_i) \implies$ over-determined system with "50×12 Vandermonde matrix"

Least-squares polynomial fit

[*lfit.py*] But solving the normal equations still yields a small residual, hence we obtain a good fit to the data

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

$$\|r(b_{\mathrm{lst.sq.}})\|_2 = \|y - Ab_{\mathrm{lst.sq.}}\|_2 = 8.00 \times 10^{-9}$$



Non-polynomial Least-squares fitting

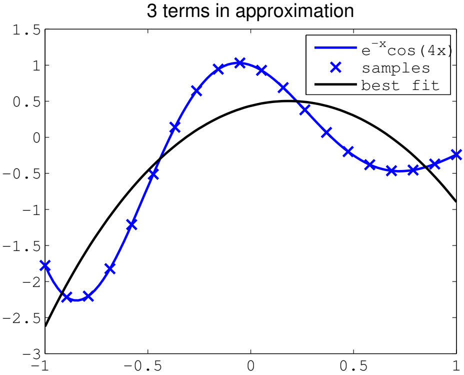
So far we have dealt with approximations based on polynomials, but we can also develop non-polynomial approximations

We just need the model to depend linearly on parameters

Example [*np_lfit.py*]: Approximate $e^{-x}\cos(4x)$ using $f_n(x;b) \equiv \sum_{k=-n}^{n} b_k e^{kx}$

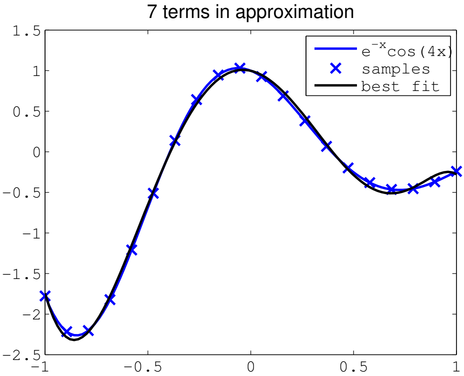
(Note that f_n is linear in b: $f_n(x;\gamma a + \sigma b) = \gamma f_n(x;a) + \sigma f_n(x;b)$)

Non-polynomial Least-squares fitting



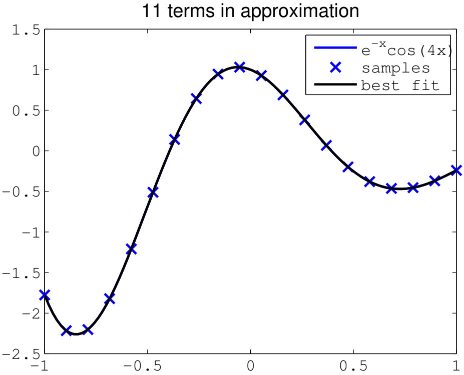
$$n = 1, \quad \frac{\|r(b)\|_2}{\|b\|_2} = 4.16 \times 10^{-1}$$

Non-polynomial Least-squares fitting



$$n = 3, \quad \frac{\|r(b)\|_2}{\|b\|_2} = 1.44 \times 10^{-3}$$

Non-polynomial Least-squares fitting



$$n = 5, \quad \frac{\|r(b)\|_2}{\|b\|_2} = 7.46 \times 10^{-6}$$

Pseudoinverse

Recall that from the normal equations we have:

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

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

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

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 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 overconstrained systems (more constraints than parameters)

But least-squares also applies to underconstrained systems: $Ab = y$ with $A \in \mathbb{R}^{m \times n}$, $m < n$

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

i.e. we have a "short, wide" matrix A

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 vectors b that give $r(b) = 0$, we want to be able to select one of them

Underdetermined Least Squares

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

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

This can be treated using Lagrange multipliers (discussed later in the Optimization section)

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:

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

As a result, 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

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

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: $S = \mu \mathrm{I} \in \mathbb{R}^{n\times n}$ for $\mu \in \mathbb{R}_{>0}$

Underdetermined Least Squares

Example [*under_lfit.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 (can't use Lagrange since $m \neq n$): A is a submatrix of a Vandermonde matrix

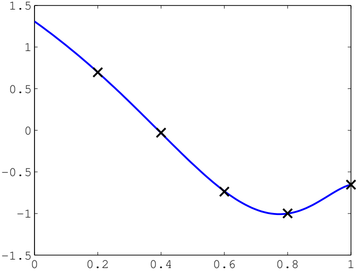
If we naively use the normal equations we see that $\text{cond}(A^T A) = 4.78 \times 10^{17}$, *i.e.* "singular to machine precision"!

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.* $\mu = 0.001$)



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

$$\|b\|_2 = 4.40$$

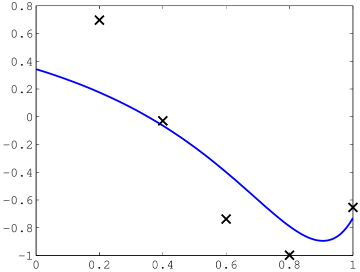
$$\text{cond}(A^T A + S^T S) = 1.54 \times 10^7$$

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.* $\mu = 0.5$)



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

$$\|b\|_2 = 1.15$$

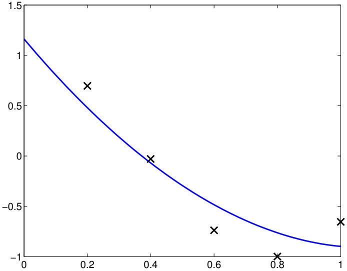
$$\mathrm{cond}(A^T A + S^T S) = 62.3$$

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 $S = \texttt{diag}(0.1, 0.1, 0.1, 10, 10 \ldots, 10)$



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

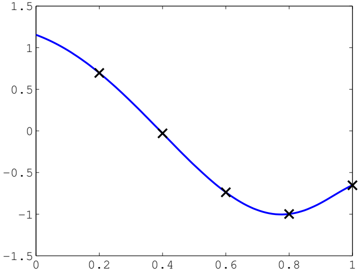
$$\|b\|_2 = 4.27$$

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

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 $x \in [0, 1]$



$$\|r(b)\|_2 = 1.03 \times 10^{-15}$$

$$\|b\|_2 = 7.18$$

Python routine gives Lagrange multiplier based solution, 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: We are referring to linearity in the parameters, not linearity of the model

(*e.g.* polynomial $p_n(x; b) = b_0 + b_1 x + \ldots + b_n x^n$ is nonlinear in x, but linear in b!)

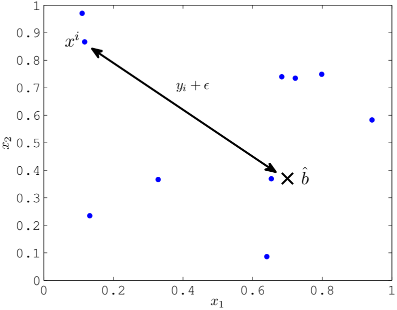
In nonlinear least-squares, we fit functions that are nonlinear in the parameters

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 a measurement for 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

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

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 $1/2$ factor in $\phi(b)$ has no effect on the minimizing b, but leads to slightly cleaner formulae 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^{th} 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

An important recurring theme is that linear systems can be solved in "one shot," whereas nonlinear generally requires iteration

We will briefly introduce Newton's method for solving this system and defer detailed discussion until the optimization section

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 $F(b) \equiv J_r(b)^T r(b)$

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 a pain to deal with the second derivatives in the previous formula, second derivatives get messy!

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: $b_{k+1} = b_k + \Delta b_k$ where

$$J_r(b_k)^T J_r(b_k) \Delta b_k = -J_r(b_k)^T r(b_k), \qquad k = 1, 2, 3, \ldots$$

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 $J_r(b_k)\Delta b_k \simeq -r(b_k)$ at each iteration

This is a common refrain in Scientific Computing: 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" — no closed form equations hence not possible to differentiate the residual!

Computing the Jacobian

So, what is the alternative to “differentiation by hand”?

Finite difference approximation: for $h \ll 1$ we have

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

Avoids tedious, error prone differentiation of r by hand!

Also, can be used for differentiating “black box” mappings since we only need to be able to evaluate $r(b)$

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[16] $S = \mathrm{I}$ or $S = J(b_k)$, and some heuristic is used to choose μ_k

This looks like our "regularized" underdetermined linear least squares formulation!

[16] In this context diag(A) means "zero the off-diagonal part of A"

Levenberg–Marquardt Method

Key point: The regularization term $\mu_k \operatorname{diag}(S^T S)$ improves the reliability of the algorithm in practice

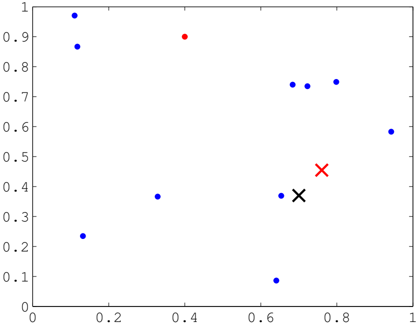
Levenberg–Marquardt is implemented in Python and Matlab's optimization toolbox

We need to pass the residual to the routine, and we can also pass the Jacobian matrix or ask for a finite-differenced Jacobian

Now let's solve our transmitter/receiver problem

Nonlinear Least Squares: Example

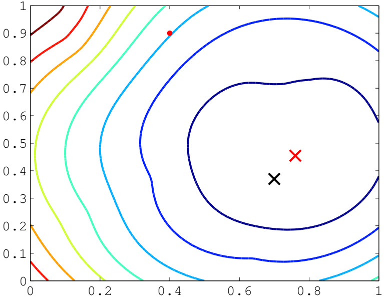
Python example: Using *nonlinlsq.py* we provide an initial guess (●), and converge to the solution (×)



Nonlinear Least Squares: Example

Levenberg–Marquardt minimizes $\phi(b)$, as we see from the contour plot of $\phi(b)$ below

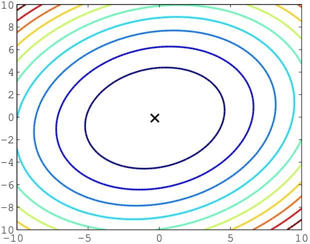
Recall $\times$ is the true transmitter location, $\color{red}{\times}$ is our best-fit to the data; $\phi(\color{red}{\times}) = 0.0248 < 0.0386 = \phi(\times)$.

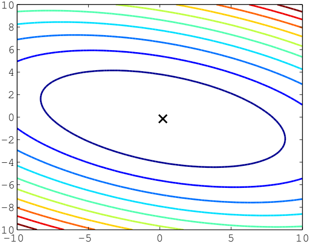


These contours are quite different from what we get in linear problems

Linear Least-Squares Contours

Two examples of linear least squares contours for $\phi(b) = \|y - Ab\|_2^2$, $b \in \mathbb{R}^2$





In linear least squares $\phi(b)$ is quadratic, hence contours are "hyperellipses"