AM205: brief announcements

Temporary website (until IACS sysadmin returns):
http://seas.harvard.edu/~chr/am205/

Currently posted:

- Syllabus
- Lecture 1 slides
- Assignment 0
- Link to Piazza sign-up

Office hours today from 1:30 PM to 3 PM in Pierce Hall 305

Last time

- Introduced two sources of error: discretization error and truncation error
- Talked about measures of absolute error and relative error
- Talked about algebraic and exponential convergence
- Discussed the condition number as the measure of stability

Finite-precision arithmetic

Key point: we can only represent a finite and discrete subset of the real numbers on a computer.

The standard approach in modern hardware is to use binary floating point numbers (basically "scientific notation" in base 2),

$$\begin{aligned} x &= \pm(1 + d_1 2^{-1} + d_2 2^{-2} + \ldots + d_p 2^{-p}) \times 2^E \\ &= \pm(1.d_1 d_2 \ldots d_p)_2 \times 2^E \end{aligned}$$

Finite-precision arithmetic

We store

$$\underbrace{\pm}_{\text{1 sign bit}} \qquad \underbrace{d_1, d_2, \ldots, d_p}_{p \text{ mantissa bits}} \qquad \underbrace{E}_{\text{exponent bits}}$$

Note that the term bit is a contraction of "binary digit"[1].

This format assumes that $d_0 = 1$ to save a mantissa bit, but sometimes $d_0 = 0$ is required, such as to represent zero.

The exponent resides in an interval $L \leq E \leq U$.

[1] This terminology was first used in Claude Shannon's seminal 1948 paper, *A Mathematical Theory of Communication*.

IEEE floating point arithmetic

Universal standard on modern hardware is IEEE floating point arithmetic (IEEE 754), adopted in 1985.

Development led by Prof. William Kahan (UC Berkeley)[2], who received the 1989 Turing Award for his work.

	total bits	p	L	U
IEEE single precision	32	23	-126	127
IEEE double precision	64	52	-1022	1023

Note that single precision has 8 exponent bits but only 254 different values of E, since some exponent bits are reserved to represent special numbers.

[2] It's interesting to search for `paranoia.c`.

Exceptional values

These exponents are reserved to indicate special behavior, including values such as Inf and NaN:

- Inf = "infinity", *e.g.* 1/0 (also -1/0 = -Inf)
- NaN = "Not a Number", *e.g.* 0/0, Inf/Inf

IEEE floating point arithmetic

Let $\mathbb{F}$ denote the floating point numbers. Then $\mathbb{F} \subset \mathbb{R}$ and $|\mathbb{F}| < \infty$.

Question: how should we represent a real number x, which is not in $\mathbb{F}$?

Answer: There are two cases to consider:

- Case 1: x is outside the range of $\mathbb{F}$ (too small or too large)
- Case 2: The mantissa of x requires more than p bits.

IEEE floating point arithmetic

Case 1: x is outside the range of $\mathbb{F}$ (too small or too large)

Too small:

- Smallest positive value that can be represented in double precision is $\approx 10^{-323}$.
- For a value smaller than this we get underflow, and the value typically set to 0.

Too large:

- Largest $x \in \mathbb{F}$ ($E = U$ and all mantissa bits are 1) is approximately $2^{1024} \approx 10^{308}$.
- For values larger than this we get overflow, and the value typically gets set to Inf.

IEEE floating point arithmetic

Case 2: The mantissa of x requires more than p bits

Need to round x to a nearby floating point number

Let $\texttt{round} : \mathbb{R} \to \mathbb{F}$ denote our rounding operator. There are several different options: round up, round down, round to nearest, *etc.*

This introduces a rounding error:

- absolute rounding error $x - \texttt{round}(x)$
- relative rounding error $(x - \texttt{round}(x))/x$

Machine precision

It is important to be able to quantify this rounding error—it's related to machine precision, often denoted as ϵ or ϵ_{mach}.

ϵ is the difference between 1 and the next floating point number after 1, *i.e.* $\epsilon = 2^{-p}$.

In IEEE double precision, $\epsilon = 2^{-52} \approx 2.22 \times 10^{-16}$.

Rounding Error

Let $x = (1.d_1 d_2 \dots d_p d_{p+1})_2 \times 2^E \in \mathbb{R}_{>0}$.

Then $x \in [x_-, x_+]$ for $x_-, x_+ \in \mathbb{F}$, where
$x_- = (1.d_1 d_2 \dots d_p)_2 \times 2^E$ and $x_+ = x_- + \epsilon \times 2^E$.

$\texttt{round}(x) = x_-$ or x_+ depending on the rounding rule, and hence
$|\texttt{round}(x) - x| < \epsilon \times 2^E$ (why not "$\leq$")[3]

Also, $|x| \geq 2^E$.

[3] With "round to nearest" we have $|\texttt{round}(x) - x| \leq 0.5 \times \epsilon \times 2^E$, but here we prefer the above inequality because it is true for any rounding rule.

Rounding Error

Hence we have a relative error of less than ϵ, *i.e.*,

$$\left| \frac{\texttt{round}(x) - x}{x} \right| < \epsilon.$$

Another standard way to write this is

$$\texttt{round}(x) = x \left(1 + \frac{\texttt{round}(x) - x}{x} \right) = x(1 + \delta)$$

where $\delta = \frac{\texttt{round}(x) - x}{x}$ and $|\delta| < \epsilon$.

Hence rounding give the correct answer to within a factor of $1 + \delta$.

Floating Point Operations

An arithmetic operation on floating point numbers is called a "floating point operation": $\oplus$, $\ominus$, $\otimes$, $\oslash$ versus $+$, $-$, $\times$, $/$.

Computer performance is often measured in "flops": number of floating point operations per second.

Supercomputers are ranked based on number of flops achieved in the "linpack test," which solves dense linear algebra problems.

Currently, the fastest computers are in the petaflop range:
1 petaflop $= 10^{15}$ floating point operations per second

Floating Point Operations

See `http://www.top500.org` for an up-to-date list of the fastest supercomputers.[4]

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
2	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
3	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660
5	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786,432	8,586.6	10,066.3	3,945
6	Swiss National Supercomputing Centre (CSCS) Switzerland	**Piz Daint** - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Cray Inc.	115,984	6,271.0	7,788.9	2,325

[4] Rmax: flops from linpack test. Rpeak: theoretical maximum flops.

Floating Point Operations

Modern supercomputers are very large, link many processors together with fast interconnect to minimize communication time

Floating Point Operation Error

IEEE standard guarantees that for $x, y \in \mathbb{F}$, $x \circledast y = \text{round}(x * y)$ (here $*$ and $\circledast$ represent any one of the 4 arithmetic operations)

Hence from our discussion of rounding error it follows that for $x, y \in \mathbb{F}$, $x \circledast y = (x * y)(1 + \delta)$, for some $|\delta| < \epsilon$

Numerical Stability of an Algorithm

We have discussed rounding for a single operation, but in AM205 we will study numerical algorithms which require many operations

For an algorithm to be useful, it must be stable in the sense that rounding errors do not accumulate and result in "garbage" output

More precisely, numerical analysts aim to prove backward stability: The method gives the exact answer to a slightly perturbed problem

For example, a numerical method for solving $Ax = b$ should give the exact answer for $Ax = (b + \Delta b)$ for small Δb

Numerical Stability of an Algorithm

Hence the importance of conditioning is clear: Backward stability doesn't help us if the mathematical problem is ill-conditioned!

For example, if A is ill-conditioned then a backward stable algorithm for solving $Ax = b$ can still give large error for x

Backward stability analysis is a deep subject which we don't have time to cover in detail in AM205

We will, however, compare algorithms with different stability properties and observe the importance of stability in practice

Unit I: Data Fitting

Chapter I.1: Motivation

Motivation

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

We will study two types of data fitting

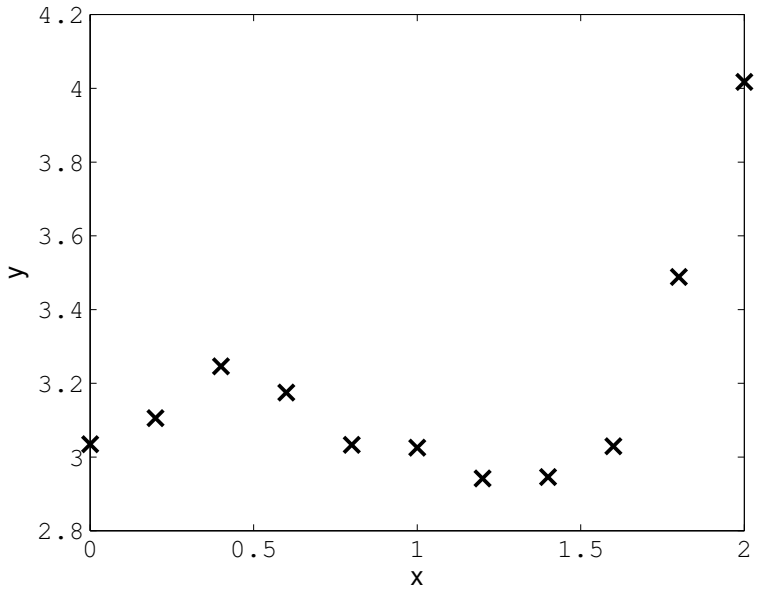
- interpolation: Fit the data points exactly
- least-squares: Minimize error in the fit (useful when there is experimental error, for example)

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



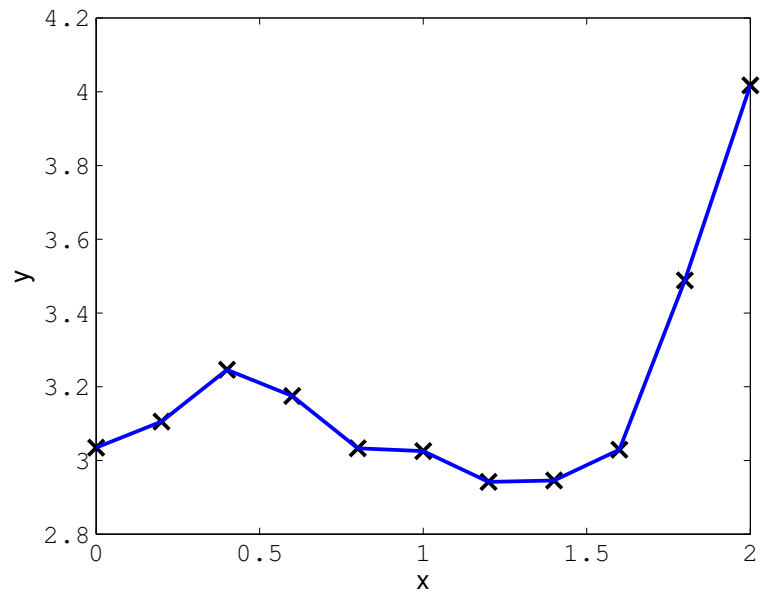
This data could represent

- 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

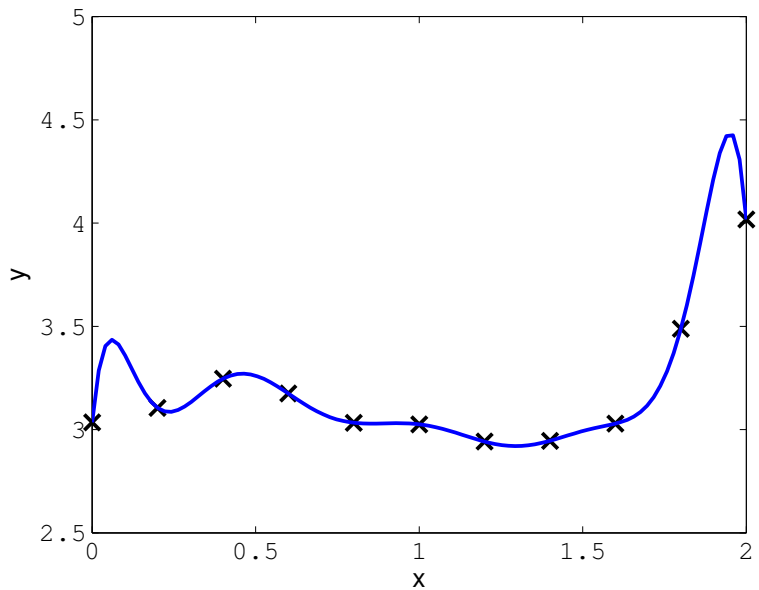
Easiest thing to do: "connect the dots" (piecewise linear interpolation)



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 I.2

Motivation

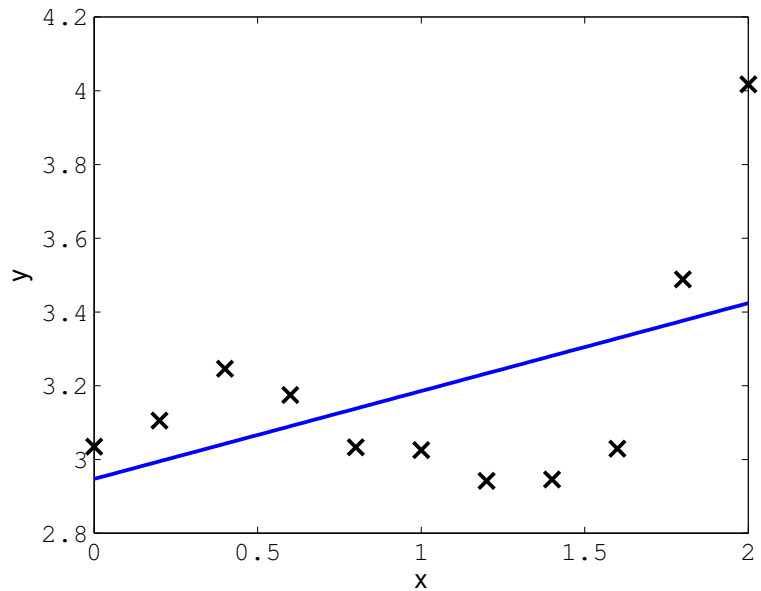
However, degree 10 interpolant is not aesthetically pleasing: too bumpy, doesn't seem to capture the "underlying pattern"

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: $y = 2.94 + 0.24x$

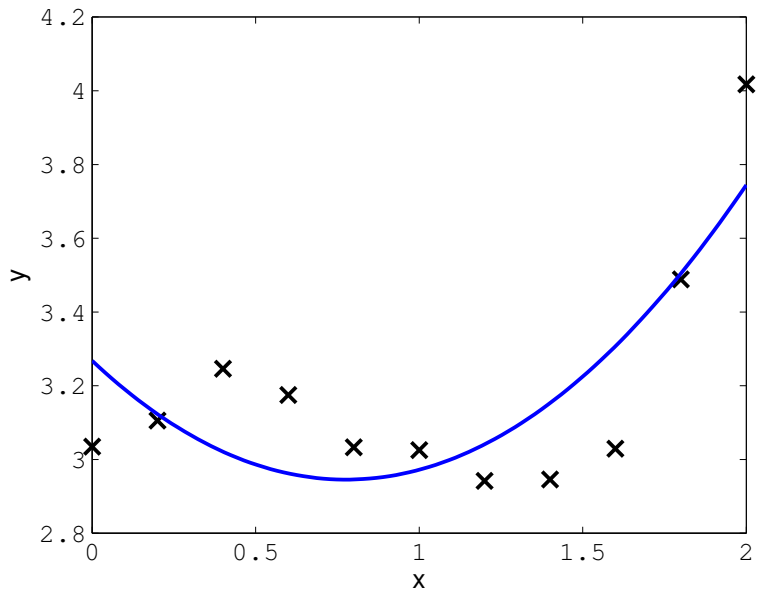


Clearly not a good fit!

Motivation

We can use least-squares fitting to generalize linear regression to higher order polynomials (see I.3)

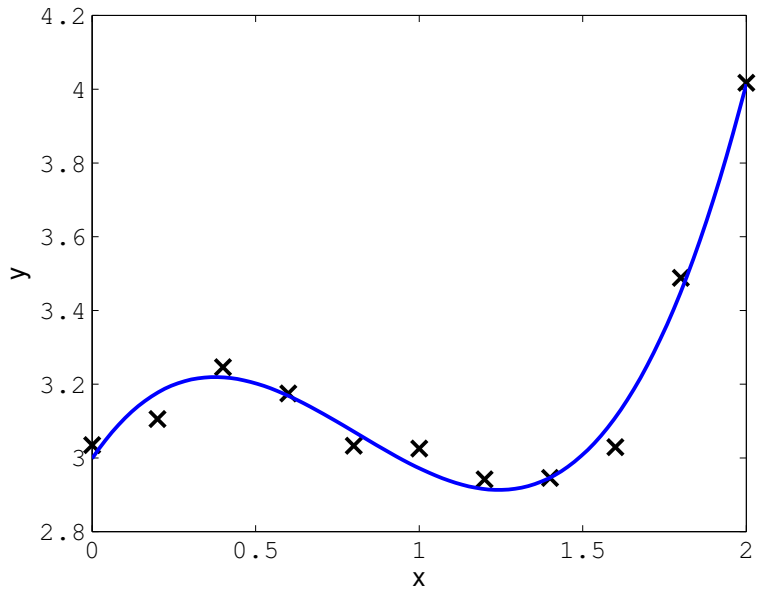
Best quadratic fit: $y = 3.27 - 0.83x + 0.53x^2$



Still not so good...

Motivation

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



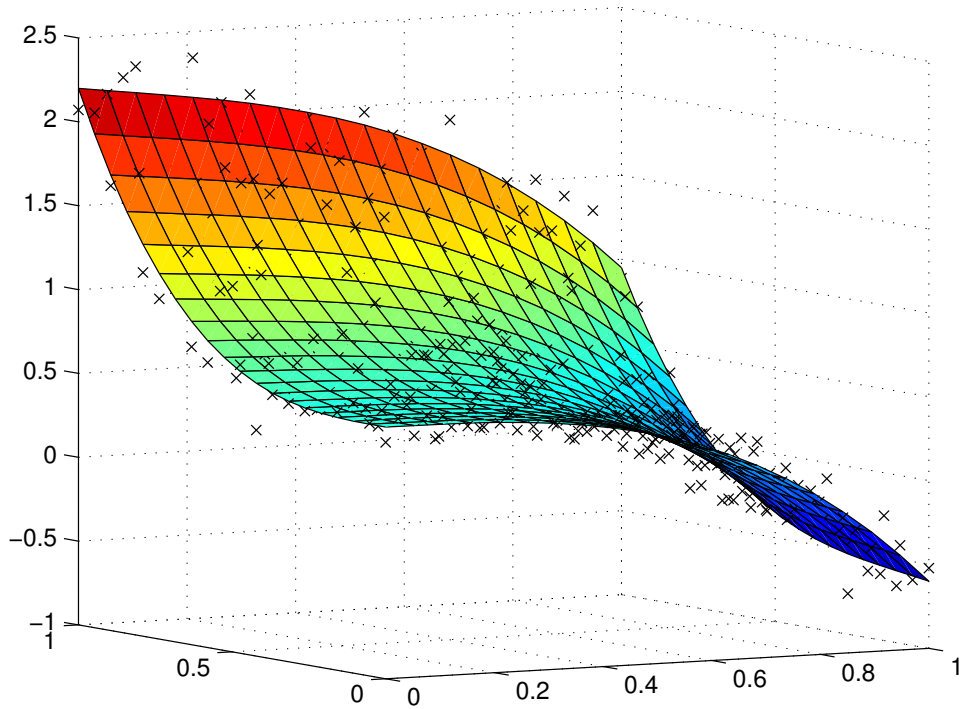
Looks good! A "cubic model" captures this data well

(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

Unit I: Data Fitting

Chapter I.2: Polynomial Interpolation

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 $\mathcal{S} \equiv \{(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)\}$, where the $\{x_0, x_1, \ldots, x_n\}$ 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 $(x_i, y_i) \in \mathcal{S}$, 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

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

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 $y \in \mathbb{R}^{n+1}$