

Lecture: Value Function Approximation

Programming and Numerical Methods for Economics—ECNM10115

Juan Zurita & Zhuotong Liu

School of Economics, The University of Edinburgh

Autumn 2025

Where we're going

The Neoclassical Growth Model

- Suppose we want to solve the problem:

$$\begin{aligned} V(k, z) = \max_{c, k', n} \quad & u(c, n) + \beta \mathbb{E}[V(k', z') \mid z] \\ \text{s.t.} \quad & c + k' = zF(k, n) && \text{Resource Constraint} \\ & \log(z') = \rho \log(z) + \epsilon && \log(z) \text{ is an } AR(1) \\ & \epsilon \sim N(0, \sigma) && \text{Shocks to } z \text{ are log-normal} \end{aligned} \tag{1}$$

where

- c is consumption
 - k is capital, and r is the rental price of capital
 - n is labor supply, and w is the wage
 - F is a constant returns to scale production function
 - β , ρ and σ are parameters
- If we can't get a solution by hand, then what does "solve this problem" even mean?

Where we're going

The Neoclassical Growth Model

- Suppose we want to solve the problem:

$$\begin{aligned} V(k, z) = \max_{c, k', n} \quad & u(c, n) + \beta \mathbb{E}[V(k', z') \mid z] \\ \text{s.t.} \quad & c + k' = zF(k, n) && \text{Resource Constraint} \\ & \log(z') = \rho \log(z) + \epsilon && \log(z) \text{ is an } AR(1) \\ & \epsilon \sim N(0, \sigma) && \text{Shocks to } z \text{ are log-normal} \end{aligned} \tag{1}$$

where

- c is consumption
 - k is capital, and r is the rental price of capital
 - n is labor supply, and w is the wage
 - F is a constant returns to scale production function
 - β , ρ and σ are parameters
- If we can't get a solution by hand, then what does "solve this problem" even mean?

What is a “solution” in quantitative economics?

$$\begin{aligned} V(k, z) = \max_{c, k', n} \quad & u(c, n) + \beta \mathbb{E} [V(k', z') \mid z] \\ \text{s.t.} \quad & c + k' = zF(k, n) \quad \text{Resource Constraint} \\ & \log(z') = \rho \log(z) + \epsilon \quad \log(z) \text{ is an } AR(1) \\ & \epsilon \sim N(0, \sigma) \quad \text{Shocks to } z \text{ are log-normal} \end{aligned} \tag{1}$$

- ▶ We will see next week that there is a unique function $V : \mathbb{R}^2 \rightarrow \mathbb{R}$ that satisfies eq. (1)
- ▶ In general, however, we cannot get an exact formula for V (a “closed-form solution”)
- ▶ We have to settle for finding an approximation of V : call it $\hat{V}$
 - ▶ As long as the solution to eq. (1) is unique, if we find an approximation $\hat{V}$ that also satisfies it, then we can call it a day
 - ▶ It turns out that if we repeatedly solve the maximization problem above, starting from an initial guess and updating $\hat{V}$ each iteration, we can be sure that we will converge to the true solution
 - ▶ This process, called **value function iteration** is what we will be learning about next week

This week: Function Approximation

- ▶ This week, we will be focusing on different methods to approximate V with some other function $\hat{V}$
- ▶ The key questions we'll be answering:
 1. What does it mean to say that $\hat{V}$ is “close” to V (i.e, that it approximates it well)
 2. What kinds of approximations work well in practice?
 3. How do we represent these approximations on a computer?
 4. How can we calculate them efficiently?

Section 1

Distance, Functions, and the Generalized Dot Product

How do we measure distance in $\mathbb{R}^n$?

- ▶ Suppose I have two points in $\mathbb{R}^n$: x and y
- ▶ How do I measure how far apart they are?

- ▶ Pythagorean theorem says: draw the corresponding right triangle, and use

$$a^2 + b^2 = c^2$$

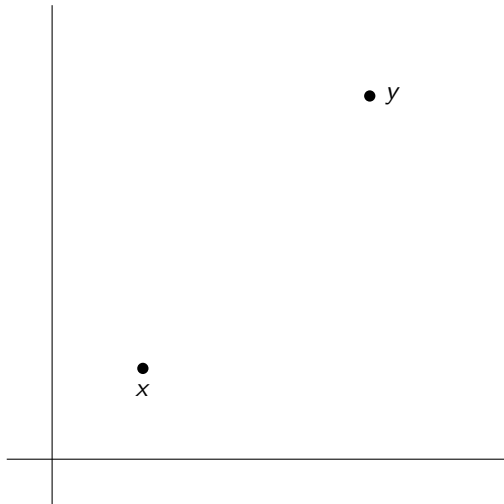
- ▶ In this case

$$c^2 = (y_1 - x_1)^2 + (y_2 - x_2)^2$$

- ▶ This happens to correspond nicely with the norm of the difference between these two points:

$$c^2 = \|y - x\|^2 = (y - x) \cdot (y - x)$$

Recall that $\|x\|^2 := x \cdot x = \sum_{i=1}^n x_i^2$



How do we measure distance in $\mathbb{R}^n$?

- ▶ Suppose I have two points in $\mathbb{R}^n$: x and y
- ▶ How do I measure how far apart they are?
- ▶ Pythagorean theorem says: draw the corresponding right triangle, and use

$$a^2 + b^2 = c^2$$

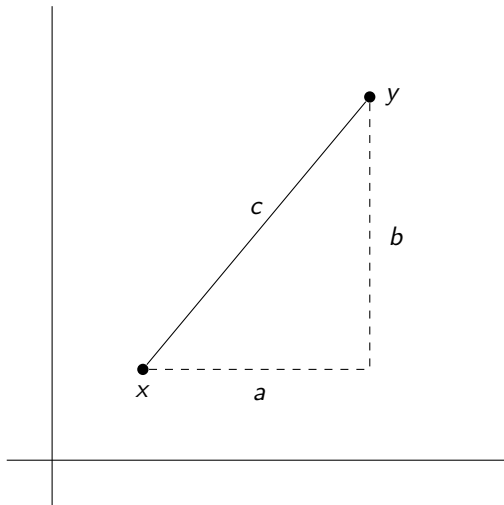
- ▶ In this case

$$c^2 = (y_1 - x_1)^2 + (y_2 - x_2)^2$$

- ▶ This happens to correspond nicely with the norm of the difference between these two points:

$$c^2 = \|y - x\|^2 = (y - x) \cdot (y - x)$$

Recall that $\|x\|^2 := x \cdot x = \sum_{i=1}^n x_i^2$



How do we measure distance in $\mathbb{R}^n$?

- ▶ Suppose I have two points in $\mathbb{R}^n$: x and y
- ▶ How do I measure how far apart they are?

- ▶ Pythagorean theorem says: draw the corresponding right triangle, and use

$$a^2 + b^2 = c^2$$

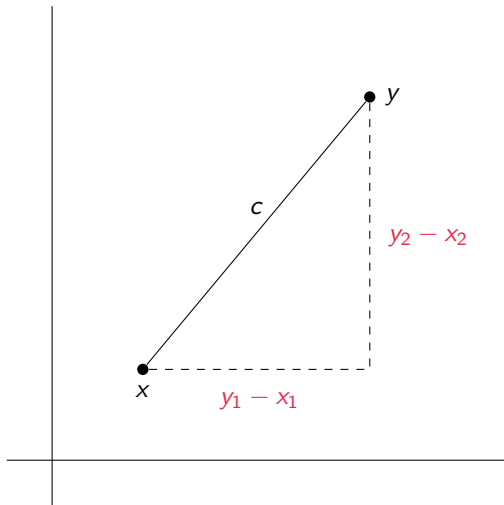
- ▶ In this case

$$c^2 = (y_1 - x_1)^2 + (y_2 - x_2)^2$$

- ▶ This happens to correspond nicely with the norm of the difference between these two points:

$$c^2 = \|y - x\|^2 = (y - x) \cdot (y - x)$$

Recall that $\|x\|^2 := x \cdot x = \sum_{i=1}^n x_i^2$



The dot product

- ▶ We've already seen the dot product show up:

$$x \cdot y := \sum_{i=1}^n x_i y_i$$

- ▶ It has this natural connection to our notion of distance:

$$\|y - x\|^2 = \sum_{i=1}^n (y_i - x_i)^2 = (y - x) \cdot (y - x)$$

- ▶ It is integral to what matrix multiplication looks like:

$$\begin{bmatrix} \text{---} & a_1 & \text{---} \\ \text{---} & a_2 & \text{---} \\ & \vdots & \\ \text{---} & a_n & \text{---} \end{bmatrix} \begin{bmatrix} | & | & & | \\ b_1 & b_2 & \dots & b_k \\ | & | & & | \end{bmatrix} = \begin{bmatrix} a_1 \cdot b_1 & a_1 \cdot b_2 & \dots & a_1 \cdot b_k \\ a_2 \cdot b_1 & a_2 \cdot b_2 & \dots & a_2 \cdot b_k \\ \vdots & \vdots & \ddots & \vdots \\ a_n \cdot b_1 & a_n \cdot b_2 & \dots & a_n \cdot b_k \end{bmatrix}$$

- ▶ But it turns out that the dot product is more important than just that: it also encodes whether or not two vectors are **orthogonal** (perpendicular) to each other

The dot product

- ▶ We've already seen the dot product show up:

$$x \cdot y := \sum_{i=1}^n x_i y_i$$

- ▶ It has this natural connection to our notion of distance:

$$\|y - x\|^2 = \sum_{i=1}^n (y_i - x_i)^2 = (y - x) \cdot (y - x)$$

- ▶ It is integral to what matrix multiplication looks like:

$$\begin{bmatrix} \text{---} & a_1 & \text{---} \\ \text{---} & a_2 & \text{---} \\ & \vdots & \\ \text{---} & a_n & \text{---} \end{bmatrix} \begin{bmatrix} | & | & & | \\ b_1 & b_2 & \dots & b_k \\ | & | & & | \end{bmatrix} = \begin{bmatrix} a_1 \cdot b_1 & a_1 \cdot b_2 & \dots & a_1 \cdot b_k \\ a_2 \cdot b_1 & a_2 \cdot b_2 & \dots & a_2 \cdot b_k \\ \vdots & \vdots & \ddots & \vdots \\ a_n \cdot b_1 & a_n \cdot b_2 & \dots & a_n \cdot b_k \end{bmatrix}$$

- ▶ But it turns out that the dot product is more important than just that: it also encodes whether or not two vectors are **orthogonal** (perpendicular) to each other

The dot product

- ▶ We've already seen the dot product show up:

$$x \cdot y := \sum_{i=1}^n x_i y_i$$

- ▶ It has this natural connection to our notion of distance:

$$\|y - x\|^2 = \sum_{i=1}^n (y_i - x_i)^2 = (y - x) \cdot (y - x)$$

- ▶ It is integral to what matrix multiplication looks like:

$$\begin{bmatrix} \text{---} & a_1 & \text{---} \\ \text{---} & a_2 & \text{---} \\ & \vdots & \\ \text{---} & a_n & \text{---} \end{bmatrix} \begin{bmatrix} | & | & & | \\ b_1 & b_2 & \dots & b_k \\ | & | & & | \end{bmatrix} = \begin{bmatrix} a_1 \cdot b_1 & a_1 \cdot b_2 & \dots & a_1 \cdot b_k \\ a_2 \cdot b_1 & a_2 \cdot b_2 & \dots & a_2 \cdot b_k \\ \vdots & \vdots & \ddots & \vdots \\ a_n \cdot b_1 & a_n \cdot b_2 & \dots & a_n \cdot b_k \end{bmatrix}$$

- ▶ But it turns out that the dot product is more important than just that: it also encodes whether or not two vectors are **orthogonal** (perpendicular) to each other

The dot product

- ▶ We've already seen the dot product show up:

$$x \cdot y := \sum_{i=1}^n x_i y_i$$

- ▶ It has this natural connection to our notion of distance:

$$\|y - x\|^2 = \sum_{i=1}^n (y_i - x_i)^2 = (y - x) \cdot (y - x)$$

- ▶ It is integral to what matrix multiplication looks like:

$$\begin{bmatrix} \text{---} & a_1 & \text{---} \\ \text{---} & a_2 & \text{---} \\ & \vdots & \\ \text{---} & a_n & \text{---} \end{bmatrix} \begin{bmatrix} | & | & & | \\ b_1 & b_2 & \dots & b_k \\ | & | & & | \end{bmatrix} = \begin{bmatrix} a_1 \cdot b_1 & a_1 \cdot b_2 & \dots & a_1 \cdot b_k \\ a_2 \cdot b_1 & a_2 \cdot b_2 & \dots & a_2 \cdot b_k \\ \vdots & \vdots & \ddots & \vdots \\ a_n \cdot b_1 & a_n \cdot b_2 & \dots & a_n \cdot b_k \end{bmatrix}$$

- ▶ But it turns out that the dot product is more important than just that: it also encodes whether or not two vectors are **orthogonal** (perpendicular) to each other

The dot product encodes the angle between two vectors

- ▶ Suppose we have two vectors x and y that lie on the unit circle ($\|x\| = \|y\| = 1$)
- ▶ We know that both vectors are defined (in polar coordinates) by their angles:

$$x = (\cos \theta_x, \sin \theta_x) \quad y = (\cos \theta_y, \sin \theta_y)$$

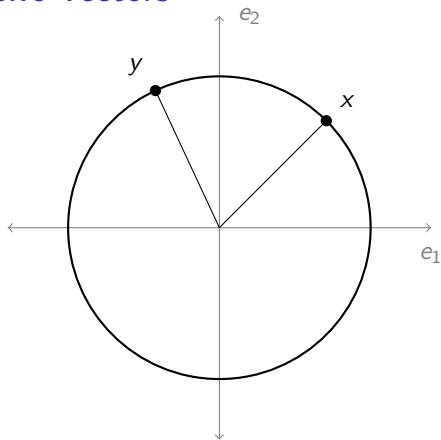
- ▶ Recall the cosine subtraction formula:

$$\cos(\alpha - \beta) = \cos \alpha \cos \beta + \sin \alpha \sin \beta \quad (2)$$

- ▶ That means that the dot product is just:

$$\begin{aligned} x \cdot y &= \cos \theta_x \cos \theta_y + \sin \theta_x \sin \theta_y && \text{Def of dot product} \\ &= \cos(\theta_y - \theta_x) && \text{By eq. (2)} \end{aligned}$$

- ▶ So we know that $x \cdot y = 0$ if and only if the **cosine of the angle between them is zero**. (I.e, the vectors are orthogonal)



This generalizes to when x and y are not on the unit circle, as well as to $\mathbb{R}^n$. In the general case:

$$x \cdot y = \|x\| \times \|y\| \times \cos \theta$$

The dot product encodes the angle between two vectors

- ▶ Suppose we have two vectors x and y that lie on the unit circle ($\|x\| = \|y\| = 1$)
- ▶ We know that both vectors are defined (in polar coordinates) by their angles:

$$x = (\cos \theta_x, \sin \theta_x) \quad y = (\cos \theta_y, \sin \theta_y)$$

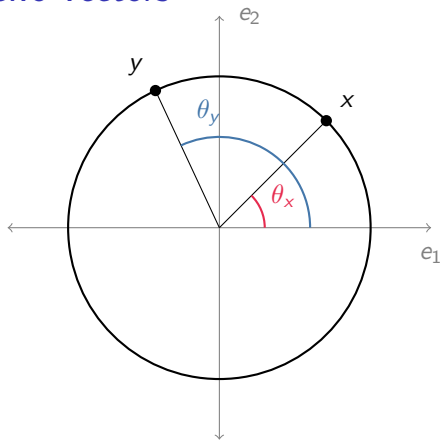
- ▶ Recall the cosine subtraction formula:

$$\cos(\alpha - \beta) = \cos \alpha \cos \beta + \sin \alpha \sin \beta \quad (2)$$

- ▶ That means that the dot product is just:

$$\begin{aligned} x \cdot y &= \cos \theta_x \cos \theta_y + \sin \theta_x \sin \theta_y && \text{Def of dot product} \\ &= \cos(\theta_y - \theta_x) && \text{By eq. (2)} \end{aligned}$$

- ▶ So we know that $x \cdot y = 0$ if and only if the **cosine of the angle between them is zero**. (I.e, the vectors are orthogonal)



This generalizes to when x and y are not on the unit circle, as well as to $\mathbb{R}^n$. In the general case:

$$x \cdot y = \|x\| \times \|y\| \times \cos \theta$$

The dot product encodes the angle between two vectors

- ▶ Suppose we have two vectors x and y that lie on the unit circle ($\|x\| = \|y\| = 1$)
- ▶ We know that both vectors are defined (in polar coordinates) by their angles:

$$x = (\cos \theta_x, \sin \theta_x) \quad y = (\cos \theta_y, \sin \theta_y)$$

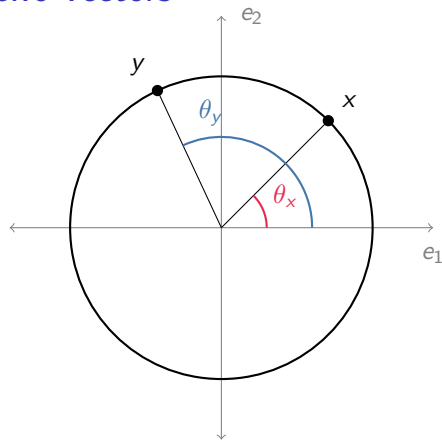
- ▶ Recall the cosine subtraction formula:

$$\cos(\alpha - \beta) = \cos \alpha \cos \beta + \sin \alpha \sin \beta \quad (2)$$

- ▶ That means that the dot product is just:

$$\begin{aligned} x \cdot y &= \cos \theta_x \cos \theta_y + \sin \theta_x \sin \theta_y && \text{Def of dot product} \\ &= \cos(\theta_y - \theta_x) && \text{By eq. (2)} \end{aligned}$$

- ▶ So we know that $x \cdot y = 0$ if and only if the **cosine of the angle between them is zero**. (I.e, the vectors are orthogonal)



This generalizes to when x and y are not on the unit circle, as well as to $\mathbb{R}^n$. In the general case:

$$x \cdot y = \|x\| \times \|y\| \times \cos \theta$$

The dot product encodes the angle between two vectors

- ▶ Suppose we have two vectors x and y that lie on the unit circle ($\|x\| = \|y\| = 1$)
- ▶ We know that both vectors are defined (in polar coordinates) by their angles:

$$x = (\cos \theta_x, \sin \theta_x) \quad y = (\cos \theta_y, \sin \theta_y)$$

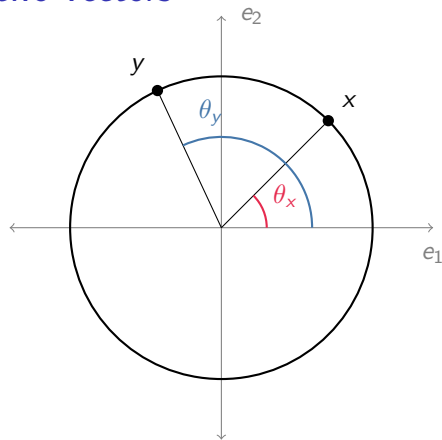
- ▶ Recall the cosine subtraction formula:

$$\cos(\alpha - \beta) = \cos \alpha \cos \beta + \sin \alpha \sin \beta \quad (2)$$

- ▶ That means that the dot product is just:

$$\begin{aligned} x \cdot y &= \cos \theta_x \cos \theta_y + \sin \theta_x \sin \theta_y && \text{Def of dot product} \\ &= \cos(\theta_y - \theta_x) && \text{By eq. (2)} \end{aligned}$$

- ▶ So we know that $x \cdot y = 0$ if and only if the **cosine of the angle between them is zero**. (I.e, the vectors are orthogonal)



This generalizes to when x and y are not on the unit circle, as well as to $\mathbb{R}^n$. In the general case:

$$x \cdot y = \|x\| \times \|y\| \times \cos \theta$$

The dot product encodes the angle between two vectors

- ▶ Suppose we have two vectors x and y that lie on the unit circle ($\|x\| = \|y\| = 1$)
- ▶ We know that both vectors are defined (in polar coordinates) by their angles:

$$x = (\cos \theta_x, \sin \theta_x) \quad y = (\cos \theta_y, \sin \theta_y)$$

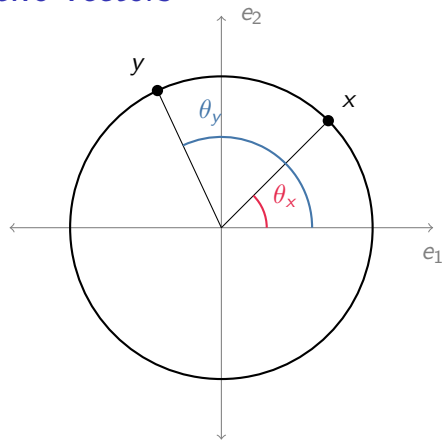
- ▶ Recall the cosine subtraction formula:

$$\cos(\alpha - \beta) = \cos \alpha \cos \beta + \sin \alpha \sin \beta \quad (2)$$

- ▶ That means that the dot product is just:

$$\begin{aligned} x \cdot y &= \cos \theta_x \cos \theta_y + \sin \theta_x \sin \theta_y && \text{Def of dot product} \\ &= \cos(\theta_y - \theta_x) && \text{By eq. (2)} \end{aligned}$$

- ▶ So we know that $x \cdot y = 0$ if and only if the **cosine of the angle between them is zero**. (I.e, the vectors are orthogonal)



This generalizes to when x and y are not on the unit circle, as well as to $\mathbb{R}^n$. In the general case:

$$x \cdot y = \|x\| \times \|y\| \times \cos \theta$$

The dot product encodes the angle between two vectors

- ▶ Suppose we have two vectors x and y that lie on the unit circle ($\|x\| = \|y\| = 1$)
- ▶ We know that both vectors are defined (in polar coordinates) by their angles:

$$x = (\cos \theta_x, \sin \theta_x) \quad y = (\cos \theta_y, \sin \theta_y)$$

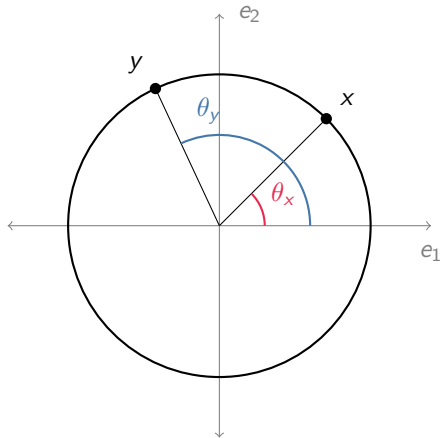
- ▶ Recall the cosine subtraction formula:

$$\cos(\alpha - \beta) = \cos \alpha \cos \beta + \sin \alpha \sin \beta \quad (2)$$

- ▶ That means that the dot product is just:

$$\begin{aligned} x \cdot y &= \cos \theta_x \cos \theta_y + \sin \theta_x \sin \theta_y && \text{Def of dot product} \\ &= \cos(\theta_y - \theta_x) && \text{By eq. (2)} \end{aligned}$$

- ▶ So we know that $x \cdot y = 0$ if and only if the **cosine of the angle between them is zero**. (I.e, the vectors are orthogonal)



This generalizes to when x and y are not on the unit circle, as well as to $\mathbb{R}^n$. In the general case:

$$x \cdot y = \|x\| \times \|y\| \times \cos \theta$$

How can we represent functions on a computer?

- ▶ If we don't have an explicit formula, we can try encoding the function values on a grid of points
- ▶ Let $f(x) = \sin(x)$, and consider a grid with $n + 1$ points:

$$X_n = \left\{ \frac{2\pi i}{n} \mid i = 0, 1, \dots, n \right\}$$

- ▶ For every point x_i , we save a corresponding

$$\hat{f}_i^n = \sin\left(\frac{2\pi i}{n}\right)$$

- ▶ Maybe we imagine that if we're off grid, we will interpolate linearly (we'll talk about this later)
- ▶ We hope that if n gets large, this is going to be good enough...

We're going to do something more sophisticated later

How can we represent functions on a computer?

- ▶ If we don't have an explicit formula, we can try encoding the function values on a grid of points

- ▶ Let $f(x) = \sin(x)$, and consider a grid with $n + 1$ points:

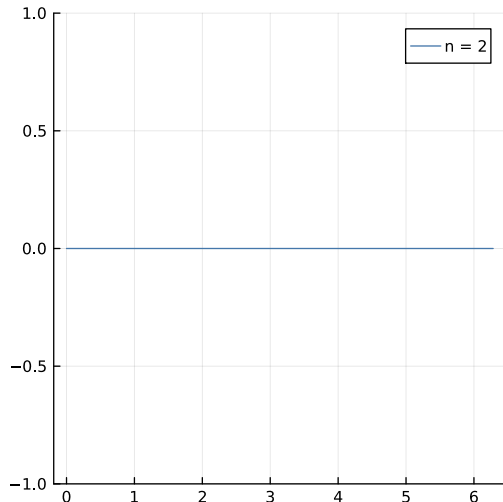
$$X_n = \left\{ \frac{2\pi i}{n} \mid i = 0, 1, \dots, n \right\}$$

- ▶ For every point x_i , we save a corresponding

$$\hat{f}_i^n = \sin\left(\frac{2\pi i}{n}\right)$$

- ▶ Maybe we imagine that if we're off grid, we will interpolate linearly (we'll talk about this later)

- ▶ We hope that if n gets large, this is going to be good enough...



We're going to do something more sophisticated later

How can we represent functions on a computer?

- ▶ If we don't have an explicit formula, we can try encoding the function values on a grid of points

- ▶ Let $f(x) = \sin(x)$, and consider a grid with $n + 1$ points:

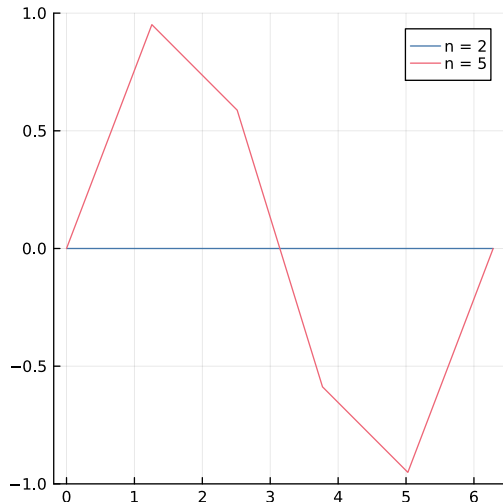
$$X_n = \left\{ \frac{2\pi i}{n} \mid i = 0, 1, \dots, n \right\}$$

- ▶ For every point x_i , we save a corresponding

$$\hat{f}_i^n = \sin\left(\frac{2\pi i}{n}\right)$$

- ▶ Maybe we imagine that if we're off grid, we will interpolate linearly (we'll talk about this later)

- ▶ We hope that if n gets large, this is going to be good enough...



We're going to do something more sophisticated later

How can we represent functions on a computer?

- ▶ If we don't have an explicit formula, we can try encoding the function values on a grid of points

- ▶ Let $f(x) = \sin(x)$, and consider a grid with $n + 1$ points:

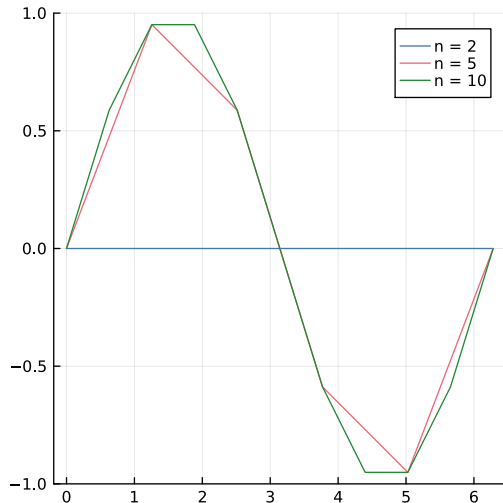
$$X_n = \left\{ \frac{2\pi i}{n} \mid i = 0, 1, \dots, n \right\}$$

- ▶ For every point x_i , we save a corresponding

$$\hat{f}_i^n = \sin\left(\frac{2\pi i}{n}\right)$$

- ▶ Maybe we imagine that if we're off grid, we will interpolate linearly (we'll talk about this later)

- ▶ We hope that if n gets large, this is going to be good enough...



We're going to do something more sophisticated later

How can we represent functions on a computer?

- ▶ If we don't have an explicit formula, we can try encoding the function values on a grid of points

- ▶ Let $f(x) = \sin(x)$, and consider a grid with $n + 1$ points:

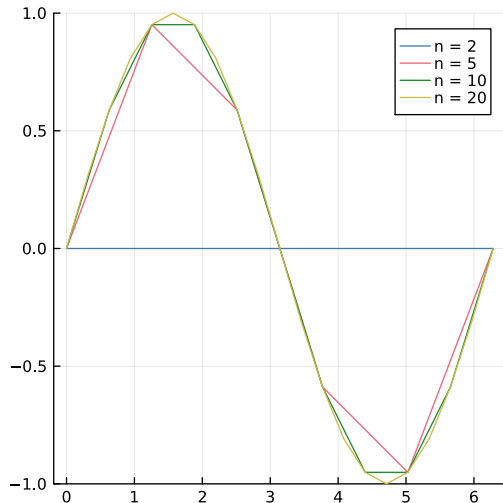
$$X_n = \left\{ \frac{2\pi i}{n} \mid i = 0, 1, \dots, n \right\}$$

- ▶ For every point x_i , we save a corresponding

$$\hat{f}_i^n = \sin\left(\frac{2\pi i}{n}\right)$$

- ▶ Maybe we imagine that if we're off grid, we will interpolate linearly (we'll talk about this later)

- ▶ We hope that if n gets large, this is going to be good enough...



We're going to do something more sophisticated later

How can we represent functions on a computer?

- ▶ If we don't have an explicit formula, we can try encoding the function values on a grid of points

- ▶ Let $f(x) = \sin(x)$, and consider a grid with $n + 1$ points:

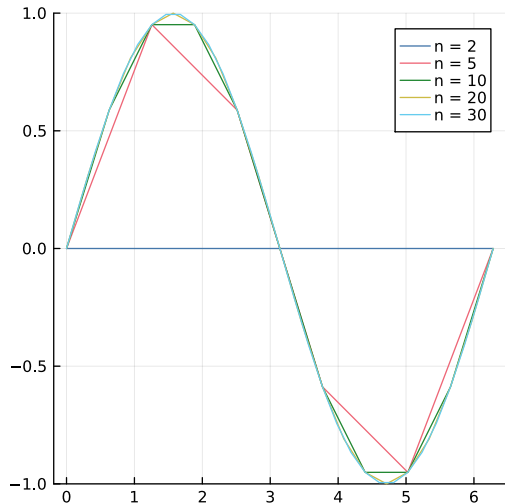
$$X_n = \left\{ \frac{2\pi i}{n} \mid i = 0, 1, \dots, n \right\}$$

- ▶ For every point x_i , we save a corresponding

$$\hat{f}_i^n = \sin\left(\frac{2\pi i}{n}\right)$$

- ▶ Maybe we imagine that if we're off grid, we will interpolate linearly (we'll talk about this later)

- ▶ We hope that if n gets large, this is going to be good enough...



We're going to do something more sophisticated later

How can we represent functions on a computer?

- ▶ If we don't have an explicit formula, we can try encoding the function values on a grid of points

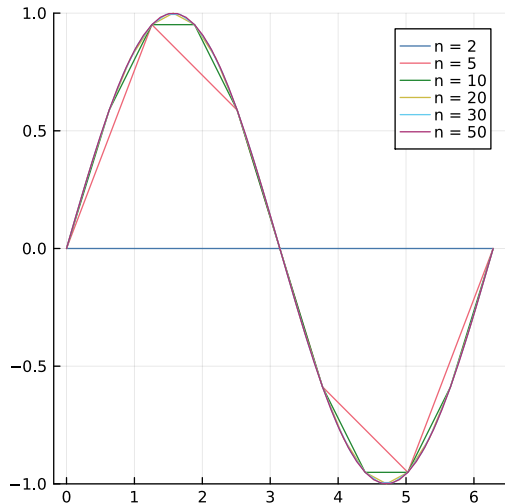
- ▶ Let $f(x) = \sin(x)$, and consider a grid with $n + 1$ points:

$$X_n = \left\{ \frac{2\pi i}{n} \mid i = 0, 1, \dots, n \right\}$$

- ▶ For every point x_i , we save a corresponding

$$\hat{f}_i^n = \sin\left(\frac{2\pi i}{n}\right)$$

- ▶ Maybe we imagine that if we're off grid, we will interpolate linearly (we'll talk about this later)
- ▶ We hope that if n gets large, this is going to be good enough...



We're going to do something more sophisticated later

What is the “length” of a function?

Consider a function f on $[0,1]$, and think about f as the vector $\hat{f}^n \in \mathbb{R}^{n+1}$ with

$$X_n = \left\{ \frac{i}{n} \mid i = 0, \dots, n \right\}$$

► Let's try a definition of $\|f\|$:

$$\|f\|^2 := \lim_{n \rightarrow \infty} \frac{1}{n} \|\hat{f}^n\|^2 \quad (3)$$

We have to divide by n because we're increasing the number of dimensions we're summing over.

► Define $\Delta x_n = \frac{1}{n}$. We can write this as:

$$\|f\|^2 = \lim_{n \rightarrow \infty} \frac{1}{n} \sum_{i=0}^n (\hat{f}_i^n)^2 = \lim_{n \rightarrow \infty} \underbrace{\sum_{i=0}^n f(x_i)^2 \Delta x_n}_{\text{A Riemann Sum}} = \int_0^1 f(x)^2 dx$$

This works on more general domains as well (not just $[0,1]$) as well as for nonuniform grids

What is the “length” of a function?

Consider a function f on $[0,1]$, and think about f as the vector $\hat{f}^n \in \mathbb{R}^{n+1}$ with

$$X_n = \left\{ \frac{i}{n} \mid i = 0, \dots, n \right\}$$

- ▶ Let's try a definition of $\|f\|$:

$$\|f\|^2 := \lim_{n \rightarrow \infty} \frac{1}{n} \|\hat{f}^n\|^2 \quad (3)$$

We have to divide by n because we're increasing the number of dimensions we're summing over.

- ▶ Define $\Delta x_n = \frac{1}{n}$. We can write this as:

$$\|f\|^2 = \lim_{n \rightarrow \infty} \frac{1}{n} \sum_{i=0}^n (\hat{f}_i^n)^2 = \lim_{n \rightarrow \infty} \underbrace{\sum_{i=0}^n f(x_i)^2 \Delta x_n}_{\text{A Riemann Sum}} = \int_0^1 f(x)^2 dx$$

This works on more general domains as well (not just $[0,1]$) as well as for nonuniform grids

What is the “length” of a function?

Consider a function f on $[0,1]$, and think about f as the vector $\hat{f}^n \in \mathbb{R}^{n+1}$ with

$$X_n = \left\{ \frac{i}{n} \mid i = 0, \dots, n \right\}$$

► Let's try a definition of $\|f\|$:

$$\|f\|^2 := \lim_{n \rightarrow \infty} \frac{1}{n} \|\hat{f}^n\|^2 \quad (3)$$

We have to divide by n because we're increasing the number of dimensions we're summing over.

► Define $\Delta x_n = \frac{1}{n}$. We can write this as:

$$\|f\|^2 = \lim_{n \rightarrow \infty} \frac{1}{n} \sum_{i=0}^n (\hat{f}_i^n)^2 = \lim_{n \rightarrow \infty} \underbrace{\sum_{i=0}^n f(x_i)^2 \Delta x_n}_{\text{A Riemann Sum}} = \int_0^1 f(x)^2 dx$$

This works on more general domains as well (not just $[0,1]$) as well as for nonuniform grids

What is the “length” of a function?

Consider a function f on $[0,1]$, and think about f as the vector $\hat{f}^n \in \mathbb{R}^{n+1}$ with

$$X_n = \left\{ \frac{i}{n} \mid i = 0, \dots, n \right\}$$

► Let's try a definition of $\|f\|$:

$$\|f\|^2 := \lim_{n \rightarrow \infty} \frac{1}{n} \|\hat{f}^n\|^2 \quad (3)$$

We have to divide by n because we're increasing the number of dimensions we're summing over.

► Define $\Delta x_n = \frac{1}{n}$. We can write this as:

$$\|f\|^2 = \lim_{n \rightarrow \infty} \frac{1}{n} \sum_{i=0}^n (\hat{f}_i^n)^2 = \lim_{n \rightarrow \infty} \underbrace{\sum_{i=0}^n f(x_i)^2 \Delta x_n}_{\text{A Riemann Sum}} = \int_0^1 f(x)^2 dx$$

This works on more general domains as well (not just $[0,1]$) as well as for nonuniform grids

Distance between functions

- ▶ We can now define the distance between functions:

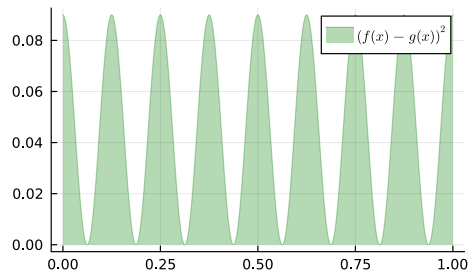
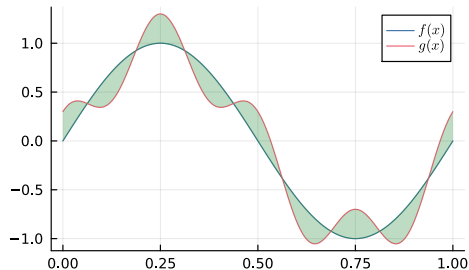
$$\|f - g\|^2 = \int_0^1 (f(x) - g(x))^2 dx$$

- ▶ Suppose we have a function $f : [0, 1] \rightarrow \mathbb{R}$, and a proposed approximation $\hat{f} : [0, 1] \rightarrow \mathbb{R}$.

- ▶ **Question:** How should we judge how “good” and approximation $\hat{f}$ is?

- ▶ **Answer:** Look at

$$\|f - \hat{f}\|^2 = \int_0^1 (f(x) - \hat{f}(x))^2 dx$$



Distance between functions

- ▶ We can now define the distance between functions:

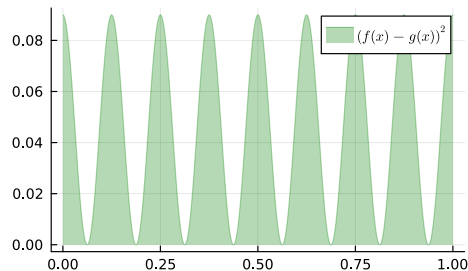
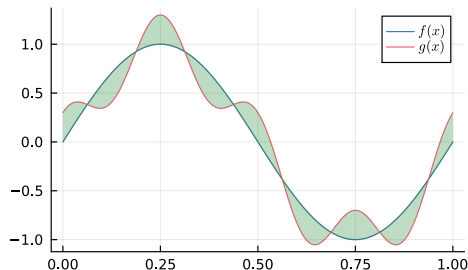
$$\|f - g\|^2 = \int_0^1 (f(x) - g(x))^2 dx$$

- ▶ Suppose we have a function $f : [0, 1] \rightarrow \mathbb{R}$, and a proposed approximation $\hat{f} : [0, 1] \rightarrow \mathbb{R}$.

- ▶ **Question:** How should we judge how “good” and approximation $\hat{f}$ is?

- ▶ **Answer:** Look at

$$\|f - \hat{f}\|^2 = \int_0^1 (f(x) - \hat{f}(x))^2 dx$$



Distance between functions

- ▶ We can now define the distance between functions:

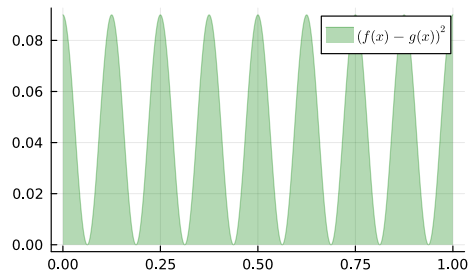
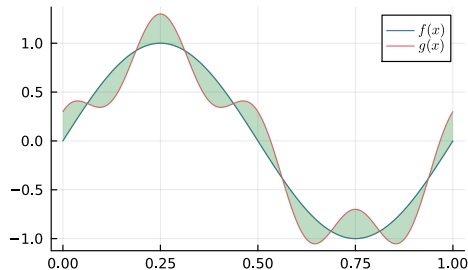
$$\|f - g\|^2 = \int_0^1 (f(x) - g(x))^2 dx$$

- ▶ Suppose we have a function $f : [0, 1] \rightarrow \mathbb{R}$, and a proposed approximation $\hat{f} : [0, 1] \rightarrow \mathbb{R}$.

- ▶ **Question:** How should we judge how “good” and approximation $\hat{f}$ is?

- ▶ **Answer:** Look at

$$\|f - \hat{f}\|^2 = \int_0^1 (f(x) - \hat{f}(x))^2 dx$$



Can functions be orthogonal?

Not examinable

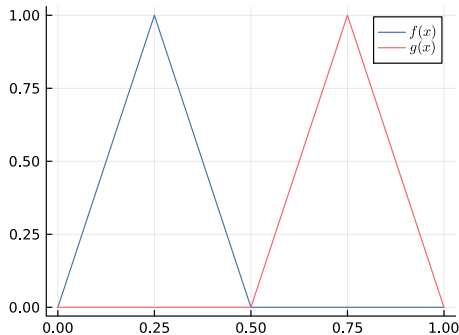
- ▶ We can define something like a “dot product” for functions:

$$\langle f, g \rangle := \int_0^1 f(x)g(x)dx$$

- ▶ This is called an **inner product**
- ▶ Just like with the dot product

$$\langle f, f \rangle = ||f||^2 = \int_0^1 f(x)^2 dx$$

- ▶ Even more importantly, if $\langle f, g \rangle = 0$, that means we can meaningfully say that these function are **orthogonal**



Notice that in this case, $f(x)g(x) = 0$ since one of the two functions is always zero. That means

$$\langle f, g \rangle = 0$$

and so f and g are orthogonal

Section 2

Interpolation with Global Polynomials

Interpolating a function

- ▶ Let $f : [0, 1] \rightarrow \mathbb{R}$ be a continuous function
 - ▶ Suppose you've already been given a grid $X = \{x_i\}_{i=1}^n$ and the evaluated $y = \{y_i\}_{i=1}^n$ where $y_i = f(x_i)$
 - ▶ It's easy to approximate f on grid – we already calculated its values – but we want to be able to approximate f off of the grid without evaluating f any more times
- ▶ Let's look for a polynomial $p(x) = \sum_{s=0}^{n-1} \alpha_s x^s$ that approximates the function well.

Notice that I've chosen a polynomial with as many coefficients as we have data points. If we want to fit our data exactly, we will need as many degrees of freedom as we have observations.

- ▶ It should:
 1. Fit our function exactly on the grid of x_i
 2. Approximate f well off-grid

i.e, $\|f - p\|$ should be small, and ideally should approach zero as n increases
- ▶ This is called an **interpolation problem**

Interpolating a function

- ▶ Let $f : [0, 1] \rightarrow \mathbb{R}$ be a continuous function
 - ▶ Suppose you've already been given a grid $X = \{x_i\}_{i=1}^n$ and the evaluated $y = \{y_i\}_{i=1}^n$ where $y_i = f(x_i)$
 - ▶ It's easy to approximate f on grid – we already calculated its values – but we want to be able to approximate f off of the grid without evaluating f any more times
- ▶ Let's look for a polynomial $p(x) = \sum_{s=0}^{n-1} \alpha_s x^s$ that approximates the function well.

Notice that I've chosen a polynomial with as many coefficients as we have data points. If we want to fit our data exactly, we will need as many degrees of freedom as we have observations.

- ▶ It should:
 1. Fit our function exactly on the grid of x_i
 2. Approximate f well off-grid

i.e., $\|f - p\|$ should be small, and ideally should approach zero as n increases
- ▶ This is called an **interpolation problem**

Interpolating a function

- ▶ Let $f : [0, 1] \rightarrow \mathbb{R}$ be a continuous function
 - ▶ Suppose you've already been given a grid $X = \{x_i\}_{i=1}^n$ and the evaluated $y = \{y_i\}_{i=1}^n$ where $y_i = f(x_i)$
 - ▶ It's easy to approximate f on grid – we already calculated its values – but we want to be able to approximate f off of the grid without evaluating f any more times
- ▶ Let's look for a polynomial $p(x) = \sum_{s=0}^{n-1} \alpha_s x^s$ that approximates the function well.

Notice that I've chosen a polynomial with as many coefficients as we have data points. If we want to fit our data exactly, we will need as many degrees of freedom as we have observations.

- ▶ It should:
 1. Fit our function exactly on the grid of x_i
 2. Approximate f well off-grid

i.e., $\|f - p\|$ should be small, and ideally should approach zero as n increases

- ▶ This is called an **interpolation problem**

The Vandermonde Matrix

- If we want $p(x_i) = y_i$ for all i , then that implies:

$$a_0 + a_1x_i + a_2x_i^2 + \cdots + a_{n-1}x_i^{n-1} = y_i \quad \text{for } i = 1, \dots, n \quad (4)$$

- Notice that this is a linear system of equations in the coefficients a :

$$\underbrace{\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^{n-1} \end{bmatrix}}_V \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \quad (5)$$

- V is called the **Vandermonde matrix**
- It turns out that the solution to this system is unique, so long as the x_i are distinct
- Interpolating a function this way is called **Lagrange Interpolation**

Proof

The Vandermonde Matrix

- If we want $p(x_i) = y_i$ for all i , then that implies:

$$a_0 + a_1x_i + a_2x_i^2 + \cdots + a_{n-1}x_i^{n-1} = y_i \quad \text{for } i = 1, \dots, n \quad (4)$$

- Notice that this is a linear system of equations in the coefficients a :

$$\underbrace{\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^{n-1} \end{bmatrix}}_V \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \quad (5)$$

- V is called the **Vandermonde matrix**
- It turns out that the solution to this system is unique, so long as the x_i are distinct
- Interpolating a function this way is called **Lagrange Interpolation**

Proof

The Vandermonde Matrix

- If we want $p(x_i) = y_i$ for all i , then that implies:

$$a_0 + a_1x_i + a_2x_i^2 + \cdots + a_{n-1}x_i^{n-1} = y_i \quad \text{for } i = 1, \dots, n \quad (4)$$

- Notice that this is a linear system of equations in the coefficients a :

$$\underbrace{\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^{n-1} \end{bmatrix}}_V \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \quad (5)$$

- V is called the **Vandermonde matrix**
- It turns out that the solution to this system is unique, so long as the x_i are distinct
- Interpolating a function this way is called **Lagrange Interpolation**

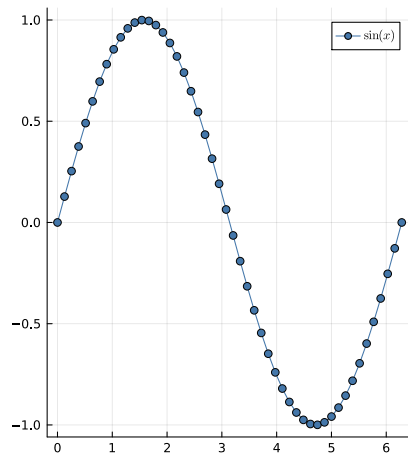
Proof

Lagrange Interpolation in Practice

```
function vandermonde(X)
    n = length(X)
    V = [xi^s for xi in X, s in 0:n-1]
    return V
end

lagrange(X,y)= vandermonde(X)\y

function evaluate(a, x)
    sum(a[s] * x^(s-1) for s in eachindex(a))
end
```

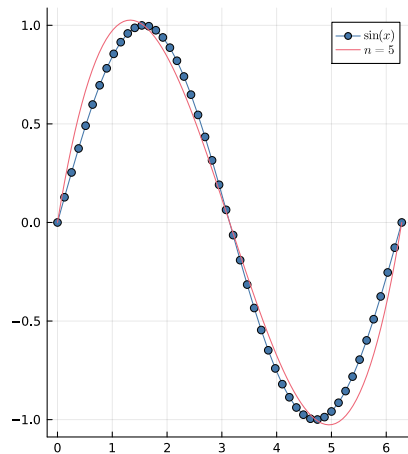


Lagrange Interpolation in Practice

```
function vandermonde(X)
    n = length(X)
    V = [xi^s for xi in X, s in 0:n-1]
    return V
end

lagrange(X,y)= vandermonde(X)\y

function evaluate(a, x)
    sum(a[s] * x^(s-1) for s in eachindex(a))
end
```

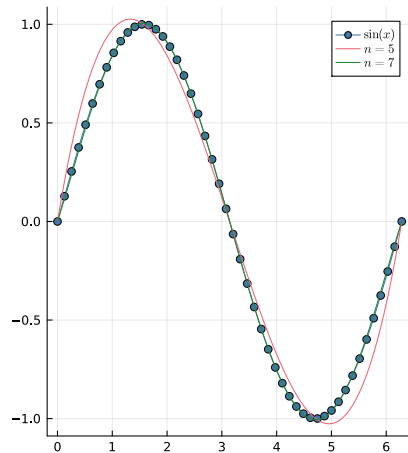


Lagrange Interpolation in Practice

```
function vandermonde(X)
    n = length(X)
    V = [xi^s for xi in X, s in 0:n-1]
    return V
end

lagrange(X,y)= vandermonde(X)\y

function evaluate(a, x)
    sum(a[s] * x^(s-1) for s in eachindex(a))
end
```

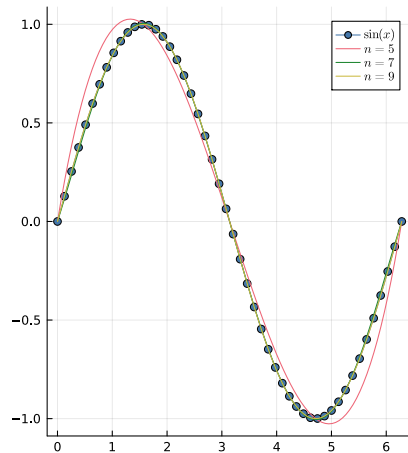


Lagrange Interpolation in Practice

```
function vandermonde(X)
    n = length(X)
    V = [xi^s for xi in X, s in 0:n-1]
    return V
end

lagrange(X,y)= vandermonde(X)\y

function evaluate(a, x)
    sum(a[s] * x^(s-1) for s in eachindex(a))
end
```



Lagrange Interpolation can fail badly

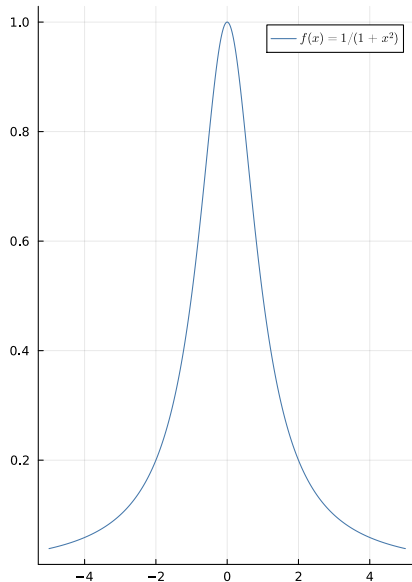
- ▶ So far, it seems Lagrange interpolation works well
- ▶ Unfortunately, there are a number of well known cases where it fails **catastrophically**

- ▶ Consider

$$f(x) = \frac{1}{1+x^2}$$

- ▶ When $n = 4$ the interpolant isn't great, but 4 points isn't that many
- ▶ By the time we're up to $n = 11$, it doesn't look like things are getting better
- ▶ In fact, you can show that this is a case where Lagrange interpolation **will never converge**
- ▶ Adding more data does not fix the problem. This is called the **Runge phenomenon**

Runge Phenomenon



Lagrange Interpolation can fail badly

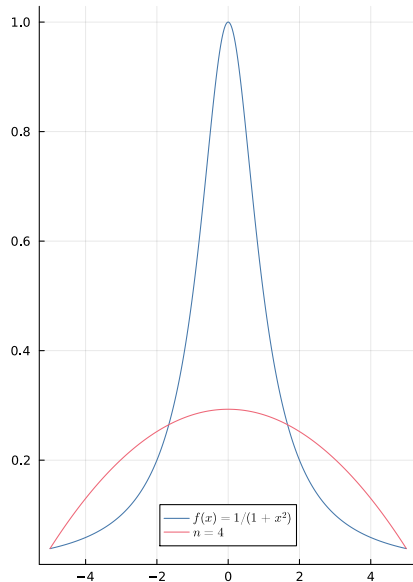
- ▶ So far, it seems Lagrange interpolation works well
- ▶ Unfortunately, there are a number of well known cases where it fails **catastrophically**

- ▶ Consider

$$f(x) = \frac{1}{1+x^2}$$

- ▶ When $n = 4$ the interpolant isn't great, but 4 points isn't that many
- ▶ By the time we're up to $n = 11$, it doesn't look like things are getting better
- ▶ In fact, you can show that this is a case where Lagrange interpolation **will never converge**
- ▶ Adding more data does not fix the problem. This is called the **Runge phenomenon**

Runge Phenomenon



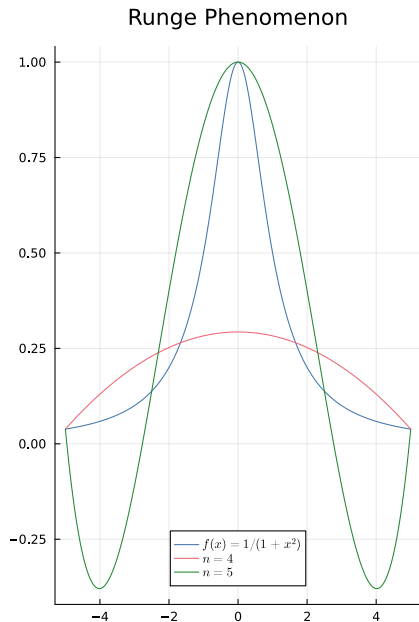
Lagrange Interpolation can fail badly

- ▶ So far, it seems Lagrange interpolation works well
- ▶ Unfortunately, there are a number of well known cases where it fails **catastrophically**

- ▶ Consider

$$f(x) = \frac{1}{1+x^2}$$

- ▶ When $n = 4$ the interpolant isn't great, but 4 points isn't that many
- ▶ By the time we're up to $n = 11$, it doesn't look like things are getting better
- ▶ In fact, you can show that this is a case where Lagrange interpolation **will never converge**
- ▶ Adding more data does not fix the problem. This is called the **Runge phenomenon**



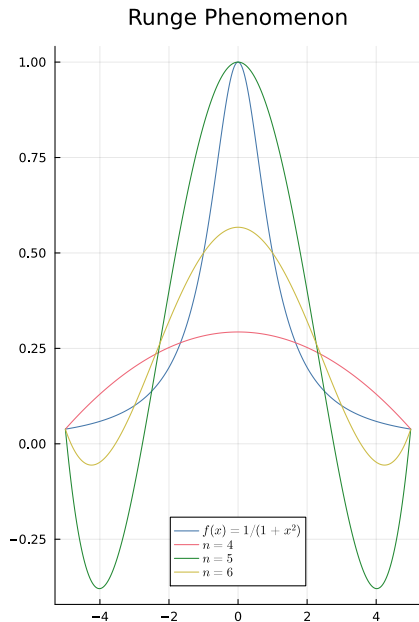
Lagrange Interpolation can fail badly

- ▶ So far, it seems Lagrange interpolation works well
- ▶ Unfortunately, there are a number of well known cases where it fails **catastrophically**

- ▶ Consider

$$f(x) = \frac{1}{1+x^2}$$

- ▶ When $n = 4$ the interpolant isn't great, but 4 points isn't that many
- ▶ By the time we're up to $n = 11$, it doesn't look like things are getting better
- ▶ In fact, you can show that this is a case where Lagrange interpolation **will never converge**
- ▶ Adding more data does not fix the problem. This is called the **Runge phenomenon**



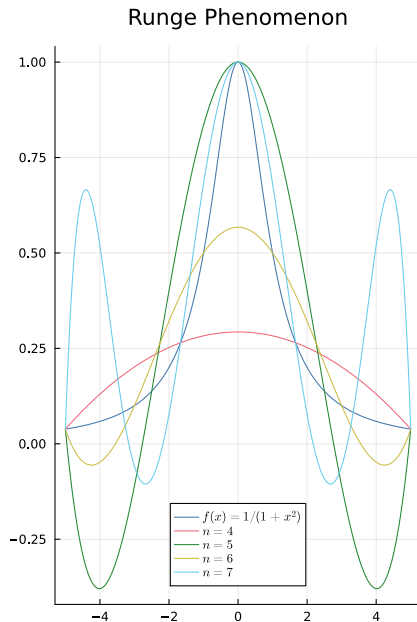
Lagrange Interpolation can fail badly

- ▶ So far, it seems Lagrange interpolation works well
- ▶ Unfortunately, there are a number of well known cases where it fails **catastrophically**

- ▶ Consider

$$f(x) = \frac{1}{1+x^2}$$

- ▶ When $n = 4$ the interpolant isn't great, but 4 points isn't that many
- ▶ By the time we're up to $n = 11$, it doesn't look like things are getting better
- ▶ In fact, you can show that this is a case where Lagrange interpolation **will never converge**
- ▶ Adding more data does not fix the problem. This is called the **Runge phenomenon**



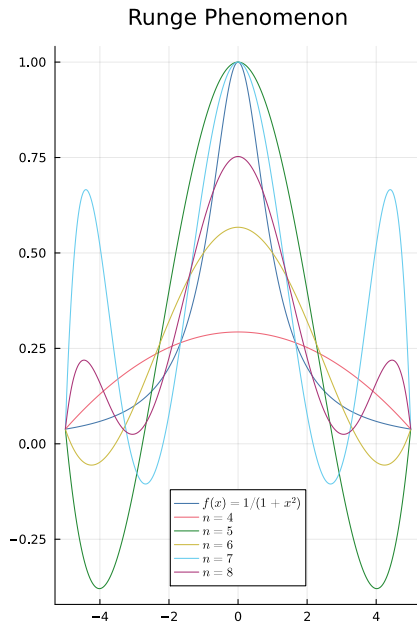
Lagrange Interpolation can fail badly

- ▶ So far, it seems Lagrange interpolation works well
- ▶ Unfortunately, there are a number of well known cases where it fails **catastrophically**

- ▶ Consider

$$f(x) = \frac{1}{1+x^2}$$

- ▶ When $n = 4$ the interpolant isn't great, but 4 points isn't that many
- ▶ By the time we're up to $n = 11$, it doesn't look like things are getting better
- ▶ In fact, you can show that this is a case where Lagrange interpolation **will never converge**
- ▶ Adding more data does not fix the problem. This is called the **Runge phenomenon**



Lagrange Interpolation can fail badly

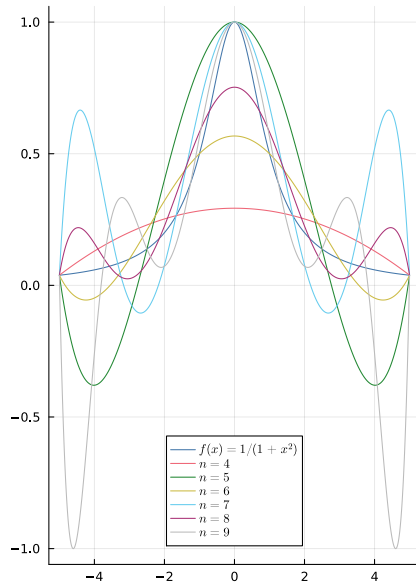
- ▶ So far, it seems Lagrange interpolation works well
- ▶ Unfortunately, there are a number of well known cases where it fails **catastrophically**

- ▶ Consider

$$f(x) = \frac{1}{1+x^2}$$

- ▶ When $n = 4$ the interpolant isn't great, but 4 points isn't that many
- ▶ By the time we're up to $n = 11$, it doesn't look like things are getting better
- ▶ In fact, you can show that this is a case where Lagrange interpolation **will never converge**
- ▶ Adding more data does not fix the problem. This is called the **Runge phenomenon**

Runge Phenomenon



Lagrange Interpolation can fail badly

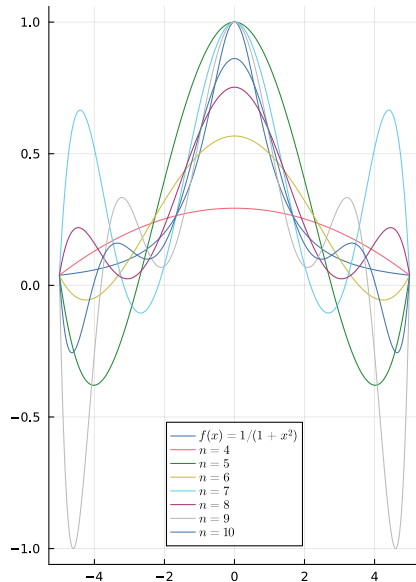
- ▶ So far, it seems Lagrange interpolation works well
- ▶ Unfortunately, there are a number of well known cases where it fails **catastrophically**

- ▶ Consider

$$f(x) = \frac{1}{1+x^2}$$

- ▶ When $n = 4$ the interpolant isn't great, but 4 points isn't that many
- ▶ By the time we're up to $n = 11$, it doesn't look like things are getting better
- ▶ In fact, you can show that this is a case where Lagrange interpolation **will never converge**
- ▶ Adding more data does not fix the problem. This is called the **Runge phenomenon**

Runge Phenomenon



Lagrange Interpolation can fail badly

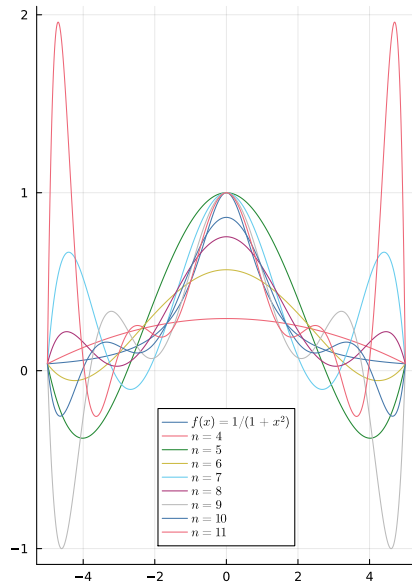
- ▶ So far, it seems Lagrange interpolation works well
- ▶ Unfortunately, there are a number of well known cases where it fails **catastrophically**

- ▶ Consider

$$f(x) = \frac{1}{1+x^2}$$

- ▶ When $n = 4$ the interpolant isn't great, but 4 points isn't that many
- ▶ By the time we're up to $n = 11$, it doesn't look like things are getting better
- ▶ In fact, you can show that this is a case where Lagrange interpolation **will never converge**
- ▶ Adding more data does not fix the problem. This is called the **Runge phenomenon**

Runge Phenomenon



Lagrange Interpolation can fail badly

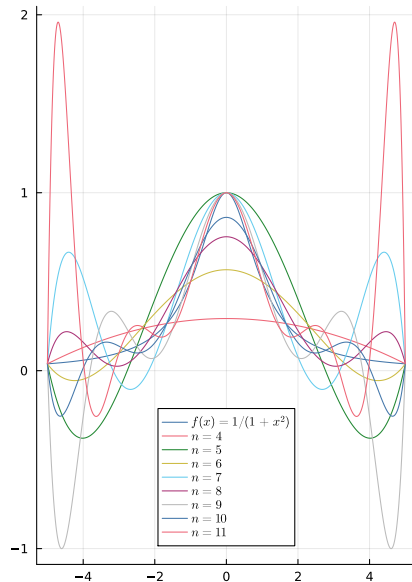
- ▶ So far, it seems Lagrange interpolation works well
- ▶ Unfortunately, there are a number of well known cases where it fails **catastrophically**

- ▶ Consider

$$f(x) = \frac{1}{1+x^2}$$

- ▶ When $n = 4$ the interpolant isn't great, but 4 points isn't that many
- ▶ By the time we're up to $n = 11$, it doesn't look like things are getting better
- ▶ In fact, you can show that this is a case where Lagrange interpolation **will never converge**
- ▶ Adding more data does not fix the problem. This is called the **Runge phenomenon**

Runge Phenomenon



How to avoid the Runge phenomenon

- ▶ The Runge phenomenon (explosive oscillation at the edges) tends to occur in most polynomial interpolation schemes with *evenly spaced grids*
 - ▶ High order polynomial terms tend to grow explosively as x gets larger
 - ▶ When you try to hit the extra data points on the edge of the domain by adding a high order polynomial term like x^{11} , that induces even more oscillations elsewhere in the domain
- ▶ To avoid this, you can:
 1. use another family of smooth polynomials called **Chebyshev polynomials**
 2. use piecewise polynomials (Linear Interpolation, Splines, etc...)

I'll define what all of these mean in just a couple of slides

How to avoid the Runge phenomenon

- ▶ The Runge phenomenon (explosive oscillation at the edges) tends to occur in most polynomial interpolation schemes with *evenly spaced grids*
 - ▶ High order polynomial terms tend to grow explosively as x gets larger
 - ▶ When you try to hit the extra data points on the edge of the domain by adding a high order polynomial term like x^{11} , that induces even more oscillations elsewhere in the domain
- ▶ To avoid this, you can:
 1. use another family of smooth polynomials called **Chebyshev polynomials**
 2. use piecewise polynomials (Linear Interpolation, Splines, etc...)

I'll define what all of these mean in just a couple of slides

Chebyshev Polynomials

- ▶ Define $T_n(x) = \cos(n \cos^{-1} x)$ for $x \in [-1, 1]$
- ▶ The family of polynomials $\{T_n\}_{n=0}^{\infty}$ are called the **Chebyshev polynomials**
- ▶ Why are these actually polynomials?

▶ You can show that these functions satisfy the formula (recurrence relationship):

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x) \quad (6)$$

- ▶ If you start from $T_0 = \cos(0) = 1$ and $T_1(x) = \cos(\cos^{-1} x) = x$ (both clearly polynomials) and you just keep multiplying by x and adding them together, you must end up with a polynomial at the end
- ▶ Let's see this in practice:

$$\begin{aligned} T_2(x) &= 2xT_1(x) - T_0(x) &= 2x(x) - 1 &= 2x^2 - 1 \\ T_3(x) &= 2xT_2(x) - T_1(x) &= 2x(2x^2 - 1) - x &= 4x^3 - 3x \\ T_4(x) &= 2xT_3(x) - T_2(x) &= 2x(4x^3 - 3x) - (2x^2 - 1) &= 8x^4 - 8x^2 + 1 \\ &\vdots \end{aligned}$$

Chebyshev Polynomials

- ▶ Define $T_n(x) = \cos(n \cos^{-1} x)$ for $x \in [-1, 1]$
- ▶ The family of polynomials $\{T_n\}_{n=0}^{\infty}$ are called the **Chebyshev polynomials**
- ▶ Why are these actually polynomials?
 - ▶ You can show that these functions satisfy the formula (recurrence relationship):

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x) \quad (6)$$

- ▶ If you start from $T_0 = \cos(0) = 1$ and $T_1(x) = \cos(\cos^{-1} x) = x$ (both clearly polynomials) and you just keep multiplying by x and adding them together, you must end up with a polynomial at the end
- ▶ Let's see this in practice:

$$\begin{aligned} T_2(x) &= 2xT_1(x) - T_0(x) &= 2x(x) - 1 &= 2x^2 - 1 \\ T_3(x) &= 2xT_2(x) - T_1(x) &= 2x(2x^2 - 1) - x &= 4x^3 - 3x \\ T_4(x) &= 2xT_3(x) - T_2(x) &= 2x(4x^3 - 3x) - (2x^2 - 1) &= 8x^4 - 8x^2 + 1 \\ &\vdots \end{aligned}$$

Chebyshev Polynomials

- ▶ Define $T_n(x) = \cos(n \cos^{-1} x)$ for $x \in [-1, 1]$
- ▶ The family of polynomials $\{T_n\}_{n=0}^{\infty}$ are called the **Chebyshev polynomials**
- ▶ Why are these actually polynomials?
 - ▶ You can show that these functions satisfy the formula (recurrence relationship):

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x) \quad (6)$$

- ▶ If you start from $T_0 = \cos(0) = 1$ and $T_1(x) = \cos(\cos^{-1} x) = x$ (both clearly polynomials) and you just keep multiplying by x and adding them together, you must end up with a polynomial at the end
- ▶ Let's see this in practice:

$$\begin{aligned} T_2(x) &= 2xT_1(x) - T_0(x) &= 2x(x) - 1 &= 2x^2 - 1 \\ T_3(x) &= 2xT_2(x) - T_1(x) &= 2x(2x^2 - 1) - x &= 4x^3 - 3x \\ T_4(x) &= 2xT_3(x) - T_2(x) &= 2x(4x^3 - 3x) - (2x^2 - 1) &= 8x^4 - 8x^2 + 1 \\ &\vdots \end{aligned}$$

Chebyshev Polynomials

- ▶ Define $T_n(x) = \cos(n \cos^{-1} x)$ for $x \in [-1, 1]$
- ▶ The family of polynomials $\{T_n\}_{n=0}^{\infty}$ are called the **Chebyshev polynomials**
- ▶ Why are these actually polynomials?
 - ▶ You can show that these functions satisfy the formula (recurrence relationship):

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x) \quad (6)$$

- ▶ If you start from $T_0 = \cos(0) = 1$ and $T_1(x) = \cos(\cos^{-1} x) = x$ (both clearly polynomials) and you just keep multiplying by x and adding them together, you must end up with a polynomial at the end
- ▶ Let's see this in practice:

$$\begin{aligned} T_2(x) &= 2xT_1(x) - T_0(x) &= 2x(x) - 1 &= 2x^2 - 1 \\ T_3(x) &= 2xT_2(x) - T_1(x) &= 2x(2x^2 - 1) - x &= 4x^3 - 3x \\ T_4(x) &= 2xT_3(x) - T_2(x) &= 2x(4x^3 - 3x) - (2x^2 - 1) &= 8x^4 - 8x^2 + 1 \\ &\vdots \end{aligned}$$

Chebyshev Polynomials

- ▶ Define $T_n(x) = \cos(n \cos^{-1} x)$ for $x \in [-1, 1]$
- ▶ The family of polynomials $\{T_n\}_{n=0}^{\infty}$ are called the **Chebyshev polynomials**
- ▶ Why are these actually polynomials?
 - ▶ You can show that these functions satisfy the formula (recurrence relationship):

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x) \quad (6)$$

- ▶ If you start from $T_0 = \cos(0) = 1$ and $T_1(x) = \cos(\cos^{-1} x) = x$ (both clearly polynomials) and you just keep multiplying by x and adding them together, you must end up with a polynomial at the end
- ▶ Let's see this in practice:

$$T_2(x) = 2xT_1(x) - T_0(x) = 2x(x) - 1 = 2x^2 - 1$$

$$T_3(x) = 2xT_2(x) - T_1(x) = 2x(2x^2 - 1) - x = 4x^3 - 3x$$

$$T_4(x) = 2xT_3(x) - T_2(x) = 2x(4x^3 - 3x) - (2x^2 - 1) = 8x^4 - 8x^2 + 1$$

$\vdots$

Chebyshev Polynomials

- ▶ Define $T_n(x) = \cos(n \cos^{-1} x)$ for $x \in [-1, 1]$
- ▶ The family of polynomials $\{T_n\}_{n=0}^{\infty}$ are called the **Chebyshev polynomials**
- ▶ Why are these actually polynomials?
 - ▶ You can show that these functions satisfy the formula (recurrence relationship):

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x) \quad (6)$$

- ▶ If you start from $T_0 = \cos(0) = 1$ and $T_1(x) = \cos(\cos^{-1} x) = x$ (both clearly polynomials) and you just keep multiplying by x and adding them together, you must end up with a polynomial at the end
- ▶ Let's see this in practice:

$$\begin{aligned} T_2(x) &= 2xT_1(x) - T_0(x) &= 2x(x) - 1 &= 2x^2 - 1 \\ T_3(x) &= 2xT_2(x) - T_1(x) &= 2x(2x^2 - 1) - x &= 4x^3 - 3x \\ T_4(x) &= 2xT_3(x) - T_2(x) &= 2x(4x^3 - 3x) - (2x^2 - 1) &= 8x^4 - 8x^2 + 1 \\ &\vdots \end{aligned}$$

Properties of Chebyshev Polynomials

- ▶ Bounded between $[-1, 1]$ so long as $x \in [-1, 1]$
- ▶ These polynomials are orthogonal to each other. Specifically (and not examinable), they are orthogonal with respect to an appropriate weighting function. I.e.,

$$\int_{-1}^1 T_n(x) T_k(x) w(x) dx = 0$$

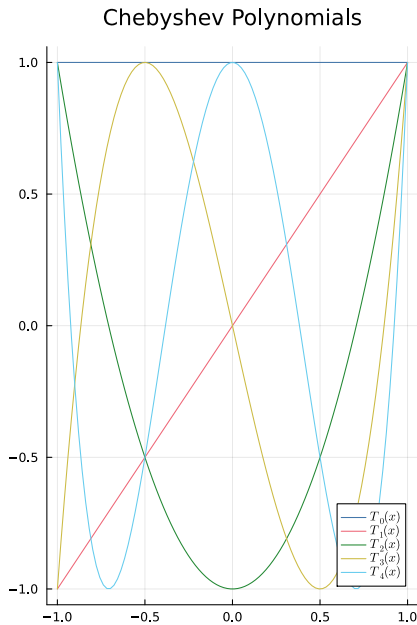
for $n \neq k$ and $w(x) = \frac{1}{\sqrt{1-x^2}}$

- ▶ You want nodes $\{x_k\}$ that are *unevenly spaced*.
- ▶ There are a known set of interpolation points that minimize the approximation error:

$$x_k = -\cos\left(\frac{(2k-1)\pi}{2n}\right) \quad \text{for } k = 1, \dots, n$$

- ▶ **Chebyshev Approximation Theorem:** As long as our function f is smooth (has continuous k th derivatives for some $k \geq 1$) Chebyshev approximation converges “nicely” to f

Theorem



Properties of Chebyshev Polynomials

- ▶ Bounded between $[-1, 1]$ so long as $x \in [-1, 1]$
- ▶ These polynomials are orthogonal to each other
Specifically (and not examinable), they are orthogonal with respect to an appropriate weighting function. I.e.,

$$\int_{-1}^1 T_n(x) T_k(x) w(x) dx = 0$$

for $n \neq k$ and $w(x) = \frac{1}{\sqrt{1-x^2}}$

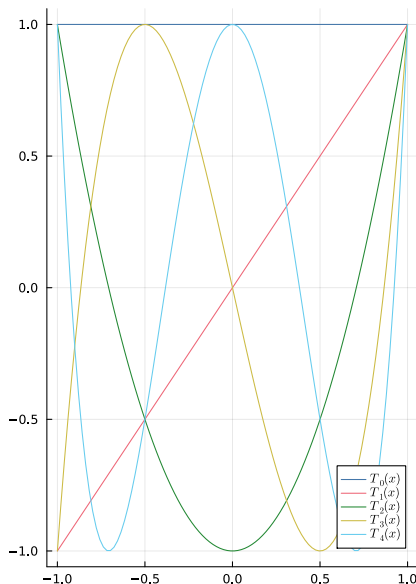
- ▶ You want nodes $\{x_k\}$ that are *unevenly spaced*.
- ▶ There are a known set of interpolation points that minimize the approximation error:

$$x_k = -\cos\left(\frac{(2k-1)\pi}{2n}\right) \quad \text{for } k = 1, \dots, n$$

- ▶ **Chebyshev Approximation Theorem:** As long as our function f is smooth (has continuous k th derivatives for some $k \geq 1$) Chebyshev approximation converges “nicely” to f

Theorem

Chebyshev Polynomials



Properties of Chebyshev Polynomials

- ▶ Bounded between $[-1, 1]$ so long as $x \in [-1, 1]$
- ▶ These polynomials are orthogonal to each other
Specifically (and not examinable), they are orthogonal with respect to an appropriate weighting function. I.e.,

$$\int_{-1}^1 T_n(x) T_k(x) w(x) dx = 0$$

for $n \neq k$ and $w(x) = \frac{1}{\sqrt{1-x^2}}$

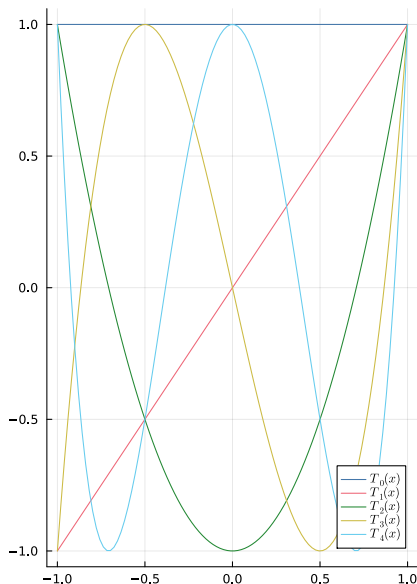
- ▶ You want nodes $\{x_k\}$ that are *unevenly spaced*.
- ▶ There are a known set of interpolation points that minimize the approximation error:

$$x_k = -\cos\left(\frac{2k-1}{2n}\pi\right) \quad \text{for } k = 1, \dots, n$$

- ▶ **Chebyshev Approximation Theorem:** As long as our function f is smooth (has continuous k th derivatives for some $k \geq 1$) Chebyshev approximation converges “nicely” to f

Theorem

Chebyshev Polynomials



Properties of Chebyshev Polynomials

- ▶ Bounded between $[-1, 1]$ so long as $x \in [-1, 1]$
- ▶ These polynomials are orthogonal to each other
Specifically (and not examinable), they are orthogonal with respect to an appropriate weighting function. I.e.,

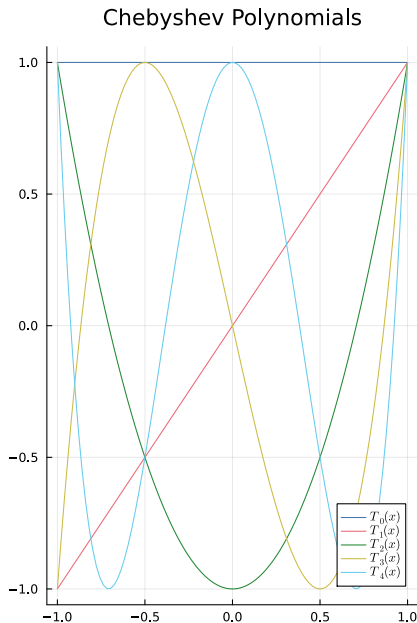
$$\int_{-1}^1 T_n(x) T_k(x) w(x) dx = 0$$

for $n \neq k$ and $w(x) = \frac{1}{\sqrt{1-x^2}}$

- ▶ You want nodes $\{x_k\}$ that are *unevenly spaced*.
- ▶ There are a known set of interpolation points that minimize the approximation error:

$$x_k = -\cos\left(\frac{2k-1}{2n}\pi\right) \quad \text{for } k = 1, \dots, n$$

- ▶ **Chebyshev Approximation Theorem:** As long as our function f is smooth (has continuous k th derivatives for some $k \geq 1$) Chebyshev approximation converges “nicely” to f Theorem



Chebyshev Regression (Approximation) Algorithm

We want an n th degree Chebyshev approximation:

1. Compute the $m \geq n + 1$ Chebyshev interpolation nodes on $[-1, 1]$:

$$z_k = -\cos\left(\frac{2k-1}{2m}\pi\right) \quad k = 1, \dots, m$$

2. For interpolation on $[a, b]$ instead of $[-1, 1]$, adjust the nodes to the appropriate interval:

$$x_k = (z_k + 1) \left(\frac{b-a}{2}\right) + a \quad k = 1, \dots, m$$

3. Evaluate f at the appropriate points: $y_k = f(x_k)$ for $k = 1, \dots, m$

4. Compute the Chebyshev coefficients:

$$c_i = \left(\frac{\sum_{k=1}^m y_k T_i(z_k)}{\sum_{k=1}^m T_i(z_k)^2} \right)$$

5. Construct the approximation:

$$\hat{f}(x) = \sum_{i=0}^n c_i T_i \left(2 \frac{x-a}{b-a} - 1 \right) \quad (7)$$

Chebyshev Regression (Approximation) Algorithm

We want an n th degree Chebyshev approximation:

1. Compute the $m \geq n + 1$ Chebyshev interpolation nodes on $[-1, 1]$:

$$z_k = -\cos\left(\frac{2k-1}{2m}\pi\right) \quad k = 1, \dots, m$$

2. For interpolation on $[a, b]$ instead of $[-1, 1]$, adjust the nodes to the appropriate interval:

$$x_k = (z_k + 1) \left(\frac{b-a}{2}\right) + a \quad k = 1, \dots, m$$

3. Evaluate f at the appropriate points: $y_k = f(x_k)$ for $k = 1, \dots, m$

4. Compute the Chebyshev coefficients:

$$c_i = \left(\frac{\sum_{k=1}^m y_k T_i(z_k)}{\sum_{k=1}^m T_i(z_k)^2} \right)$$

5. Construct the approximation:

$$\hat{f}(x) = \sum_{i=0}^n c_i T_i \left(2 \frac{x-a}{b-a} - 1 \right) \quad (7)$$

Chebyshev Regression (Approximation) Algorithm

We want an n th degree Chebyshev approximation:

1. Compute the $m \geq n + 1$ Chebyshev interpolation nodes on $[-1, 1]$:

$$z_k = -\cos\left(\frac{2k-1}{2m}\pi\right) \quad k = 1, \dots, m$$

2. For interpolation on $[a, b]$ instead of $[-1, 1]$, adjust the nodes to the appropriate interval:

$$x_k = (z_k + 1) \left(\frac{b-a}{2}\right) + a \quad k = 1, \dots, m$$

3. Evaluate f at the appropriate points: $y_k = f(x_k)$ for $k = 1, \dots, m$

4. Compute the Chebyshev coefficients:

$$c_i = \left(\frac{\sum_{k=1}^m y_k T_i(z_k)}{\sum_{k=1}^m T_i(z_k)^2} \right)$$

5. Construct the approximation:

$$\hat{f}(x) = \sum_{i=0}^n c_i T_i \left(2 \frac{x-a}{b-a} - 1 \right) \quad (7)$$

Chebyshev Regression (Approximation) Algorithm

We want an n th degree Chebyshev approximation:

1. Compute the $m \geq n + 1$ Chebyshev interpolation nodes on $[-1, 1]$:

$$z_k = -\cos\left(\frac{2k-1}{2m}\pi\right) \quad k = 1, \dots, m$$

2. For interpolation on $[a, b]$ instead of $[-1, 1]$, adjust the nodes to the appropriate interval:

$$x_k = (z_k + 1) \left(\frac{b-a}{2}\right) + a \quad k = 1, \dots, m$$

3. Evaluate f at the appropriate points: $y_k = f(x_k)$ for $k = 1, \dots, m$

4. Compute the Chebyshev coefficients:

$$c_i = \left(\frac{\sum_{k=1}^m y_k T_i(z_k)}{\sum_{k=1}^m T_i(z_k)^2} \right)$$

5. Construct the approximation:

$$\hat{f}(x) = \sum_{i=0}^n c_i T_i \left(2 \frac{x-a}{b-a} - 1 \right) \quad (7)$$

Chebyshev Regression (Approximation) Algorithm

We want an n th degree Chebyshev approximation:

1. Compute the $m \geq n + 1$ Chebyshev interpolation nodes on $[-1, 1]$:

$$z_k = -\cos\left(\frac{2k-1}{2m}\pi\right) \quad k = 1, \dots, m$$

2. For interpolation on $[a, b]$ instead of $[-1, 1]$, adjust the nodes to the appropriate interval:

$$x_k = (z_k + 1) \left(\frac{b-a}{2}\right) + a \quad k = 1, \dots, m$$

3. Evaluate f at the appropriate points: $y_k = f(x_k)$ for $k = 1, \dots, m$

4. Compute the Chebyshev coefficients:

$$c_i = \left(\frac{\sum_{k=1}^m y_k T_i(z_k)}{\sum_{k=1}^m T_i(z_k)^2} \right)$$

5. Construct the approximation:

$$\hat{f}(x) = \sum_{i=0}^n c_i T_i \left(2 \frac{x-a}{b-a} - 1 \right) \quad (7)$$

Chebyshev in Practice

```
m = n + 1
```

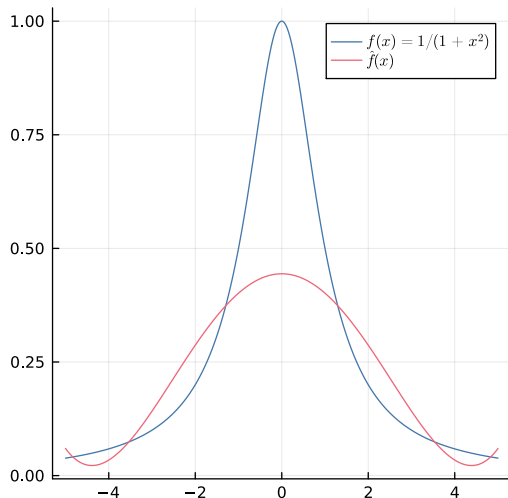
```
T(n, x) = cos(n * acos(x))  
z = [-cos((2k - 1)/(2m) * pi) for k = 1:m]  
x = (z .+ 1) .* (b - a)/2 .+ a  
y = f.(x)
```

```
c = map(0:m) do i # Calculate coeffs  
    num = sum( y[k] * T(i, z[k])  
              for k in 1:m )  
    den = sum( T(i, z[k])^2  
              for k in 1:m )  
    return num/den  
end
```

```
fh(x) = sum(  
    # evaluate approx  
    ci * T(i, 2 * (x-a)/(b-a) - 1)  
    for (ci, i) in zip(c, 0:n)  
)
```

Chebyshev fixes Runge

$n = 5$



Chebyshev in Practice

```
m = n + 1
```

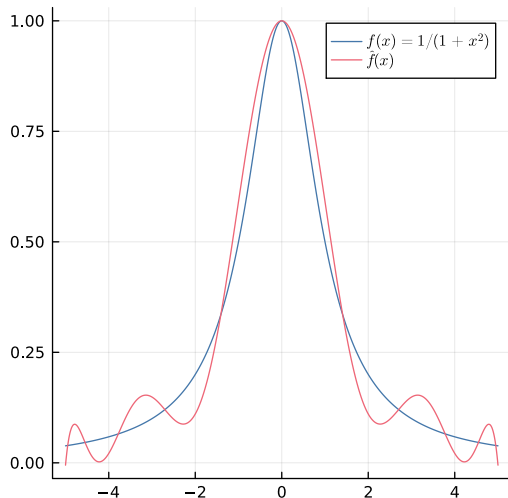
```
T(n, x) = cos(n * acos(x))  
z = [-cos((2k - 1)/(2m) * pi) for k = 1:m]  
x = (z .+ 1) .* (b - a)/2 .+ a  
y = f.(x)
```

```
c = map(0:m) do i # Calculate coeffs  
    num = sum( y[k] * T(i, z[k])  
               for k in 1:m )  
    den = sum( T(i, z[k])^2  
               for k in 1:m )  
    return num/den  
end
```

```
fh(x) = sum(  
    ci * T(i, 2 * (x-a)/(b-a) - 1)  
    for (ci, i) in zip(c, 0:n)  
)
```

Chebyshev fixes Runge

$n = 10$



Chebyshev in Practice

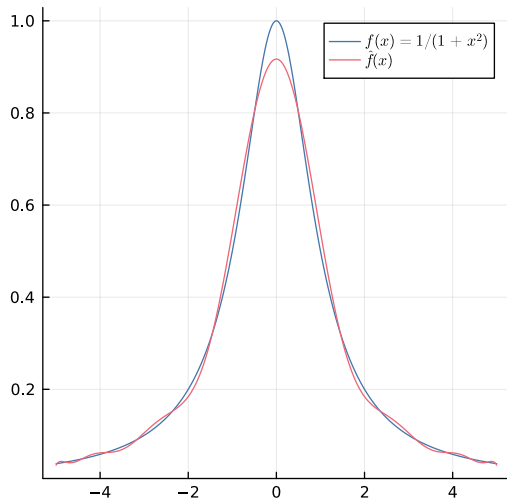
```
m = n + 1
```

```
T(n, x) = cos(n * acos(x))  
z = [-cos((2k - 1)/(2m) * pi) for k = 1:m]  
x = (z .+ 1) .* (b - a)/2 .+ a  
y = f.(x)
```

```
c = map(0:m) do i # Calculate coeffs  
    num = sum( y[k] * T(i, z[k])  
               for k in 1:m )  
    den = sum( T(i, z[k])^2  
               for k in 1:m )  
    return num/den  
end
```

```
fh(x) = sum(  
    # evaluate approx  
    ci * T(i, 2 * (x-a)/(b-a) - 1)  
    for (ci, i) in zip(c, 0:n)  
)
```

Chebyshev fixes Runge
 $n = 15$



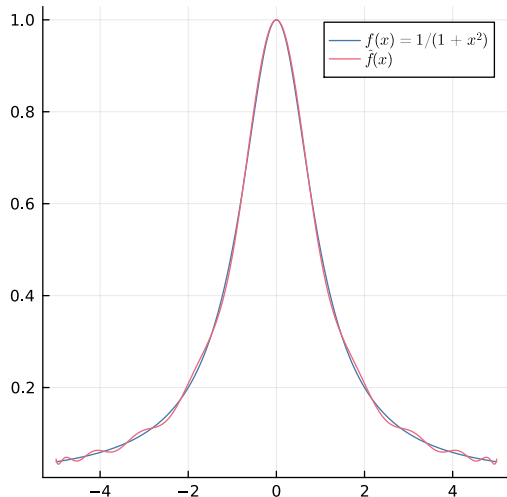
Chebyshev in Practice

```
m = n + 1
```

```
T(n, x) = cos(n * acos(x))  
z = [-cos((2k - 1)/(2m) * pi) for k = 1:m]  
x = (z .+ 1) .* (b - a)/2 .+ a  
y = f.(x)
```

```
c = map(0:m) do i # Calculate coeffs  
    num = sum( y[k] * T(i, z[k])  
               for k in 1:m )  
    den = sum( T(i, z[k])^2  
               for k in 1:m )  
    return num/den  
end  
  
fh(x) = sum(  
    # evaluate approx  
    ci * T(i, 2 * (x-a)/(b-a) - 1)  
    for (ci, i) in zip(c, 0:n)  
)
```

Chebyshev fixes Runge
 $n = 20$



Chebyshev in Practice

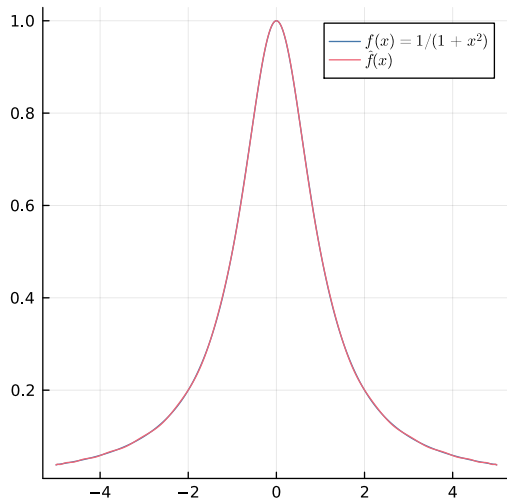
```
m = n + 1
```

```
T(n, x) = cos(n * acos(x))  
z = [-cos((2k - 1)/(2m) * pi) for k = 1:m]  
x = (z .+ 1) .* (b - a)/2 .+ a  
y = f.(x)
```

```
c = map(0:m) do i # Calculate coefs  
    num = sum( y[k] * T(i, z[k])  
               for k in 1:m )  
    den = sum( T(i, z[k])^2  
               for k in 1:m )  
    return num/den  
end  
  
fh(x) = sum( # evaluate approx  
    ci * T(i, 2 * (x-a)/(b-a) - 1)  
    for (ci, i) in zip(c, 0:n)  
)
```

Chebyshev fixes Runge

$n = 30$



Chebyshev Interpolation

When to use?

- ▶ Chebyshev interpolation works really well if you are sure that your function is defined everywhere and smooth
- ▶ The smoother it is, the better Chebyshev approximation performs (faster convergence)
- ▶ Sometimes it has trouble at the boundary
 - ▶ This can be fixed by using a different set of points x_i that include the boundary node
 - ▶ This is called the **expanded Chebyshev array** – you can look this up if you need it
- ▶ The bigger trouble arises when you have functions that are not bounded: if you have a utility function that goes to $-\infty$ when $c \rightarrow 0$, this can cause serious problems for Chebyshev polynomials
- ▶ Or functions that have kinks (discontinuous derivatives): all of the convergence guarantees go out the window

Chebyshev Interpolation

When to use?

- ▶ Chebyshev interpolation works really well if you are sure that your function is defined everywhere and smooth
- ▶ The smoother it is, the better Chebyshev approximation performs (faster convergence)
- ▶ Sometimes it has trouble at the boundary
 - ▶ This can be fixed by using a different set of points x_i that include the boundary node
 - ▶ This is called the **expanded Chebyshev array** – you can look this up if you need it
- ▶ The bigger trouble arises when you have functions that are not bounded: if you have a utility function that goes to $-\infty$ when $c \rightarrow 0$, this can cause serious problems for Chebyshev polynomials
- ▶ Or functions that have kinks (discontinuous derivatives): all of the convergence guarantees go out the window

Chebyshev Interpolation

When to use?

- ▶ Chebyshev interpolation works really well if you are sure that your function is defined everywhere and smooth
- ▶ The smoother it is, the better Chebyshev approximation performs (faster convergence)
- ▶ Sometimes it has trouble at the boundary
 - ▶ This can be fixed by using a different set of points x_i that include the boundary node
 - ▶ This is called the **expanded Chebyshev array** – you can look this up if you need it
- ▶ The bigger trouble arises when you have functions that are not bounded: if you have a utility function that goes to $-\infty$ when $c \rightarrow 0$, this can cause serious problems for Chebyshev polynomials
- ▶ Or functions that have kinks (discontinuous derivatives): all of the convergence guarantees go out the window

Section 3

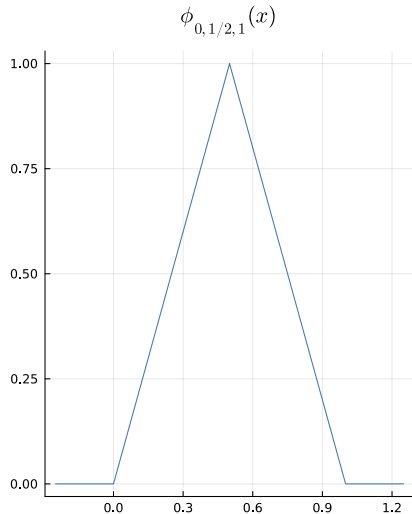
Linear Interpolation and Splines

Linear Interpolation

- ▶ Rather than use a family of polynomials that are defined everywhere, we can try polynomials that are more limited in scope
- ▶ In particular let's consider the piecewise linear functions (functions which look linear on any subinterval)
 - ▶ This is literally what you get if you just draw straight lines between the points on the graph
- ▶ Our prototypical piecewise linear function will be the “hat” function on $[x_1, x_2]$

$$\phi_{x_1, x_m, x_2}(x) = \begin{cases} \frac{x-x_1}{x_m-x_1} & \text{if } x_1 \leq x \leq x_m \\ 1 - \frac{x-x_m}{x_2-x_m} & \text{if } x_m < x \leq x_2 \\ 0 & \text{otherwise} \end{cases}$$

- ▶ You can think of x_m as the point where ϕ attains its maximum value 1



Linear Interpolant

- ▶ Suppose we have a function $f : [a, b] \rightarrow \mathbb{R}$ and have the data points $\{(x_i, y_i)\}_{i=1}^n$.
- ▶ How do we construct our linear interpolant?
- ▶ Let's add up the appropriate "hat" functions:

- ▶ For each i , let $\phi^i(x) = \phi_{x_{i-1}, x_i, x_{i+1}}(x)$

- ▶ This is putting a little hat function over every data point we have

We have to be careful at the edges. Pick any $x_0 < x_1$ and $x_{n+1} > x_n$ so that this definition works

- ▶ Take a look closely at ϕ^i . For all $1 < i < n$:

$$\phi^i(x_{i-1}) = 0$$

$$\phi^i(x_i) = 1$$

$$\phi^i(x_{i+1}) = 0$$

- ▶ Define $\hat{f}(x) := \sum_{i=1}^n c_i \phi^i(x)$ for some coefficients c_i
 - ▶ Let's evaluate $\hat{f}$ at each x_j :

$$\hat{f}(x_j) = \underbrace{\sum_{i=1}^n c_i \phi^i(x_j)}_{\text{All 0 when } i \neq j} = c_j \phi^j(x_j) = c_j \quad (8)$$

Linear Interpolant

- ▶ Suppose we have a function $f : [a, b] \rightarrow \mathbb{R}$ and have the data points $\{(x_i, y_i)\}_{i=1}^n$.
- ▶ How do we construct our linear interpolant?
- ▶ Let's add up the appropriate "hat" functions:
 - ▶ For each i , let $\phi^i(x) = \phi_{x_{i-1}, x_i, x_{i+1}}(x)$
 - ▶ This is putting a little hat function over every data point we have

We have to be careful at the edges. Pick any $x_0 < x_1$ and $x_{n+1} > x_n$ so that this definition works

- ▶ Take a look closely at ϕ^i . For all $1 < i < n$:

$$\phi^i(x_{i-1}) = 0$$

$$\phi^i(x_i) = 1$$

$$\phi^i(x_{i+1}) = 0$$

- ▶ Define $\hat{f}(x) := \sum_{i=1}^n c_i \phi^i(x)$ for some coefficients c_i
- ▶ Let's evaluate $\hat{f}$ at each x_j :

$$\hat{f}(x_j) = \underbrace{\sum_{i=1}^n c_i \phi^i(x_j)}_{\text{All 0 when } i \neq j} = c_j \phi^j(x_j) = c_j \quad (8)$$

Linear Interpolant

- ▶ Suppose we have a function $f : [a, b] \rightarrow \mathbb{R}$ and have the data points $\{(x_i, y_i)\}_{i=1}^n$.
- ▶ How do we construct our linear interpolant?
- ▶ Let's add up the appropriate "hat" functions:
 - ▶ For each i , let $\phi^i(x) = \phi_{x_{i-1}, x_i, x_{i+1}}(x)$
 - ▶ This is putting a little hat function over every data point we have

We have to be careful at the edges. Pick any $x_0 < x_1$ and $x_{n+1} > x_n$ so that this definition works

- ▶ Take a look closely at ϕ^i . For all $1 < i < n$:

$$\phi^i(x_{i-1}) = 0$$

$$\phi^i(x_i) = 1$$

$$\phi^i(x_{i+1}) = 0$$

- ▶ Define $\hat{f}(x) := \sum_{i=1}^n c_i \phi^i(x)$ for some coefficients c_i
- ▶ Let's evaluate $\hat{f}$ at each x_j :

$$\hat{f}(x_j) = \underbrace{\sum_{i=1}^n c_i \phi^i(x_j)}_{\text{All 0 when } i \neq j} = c_j \phi^j(x_j) = c_j \quad (8)$$

Linear Interpolant

- ▶ Suppose we have a function $f : [a, b] \rightarrow \mathbb{R}$ and have the data points $\{(x_i, y_i)\}_{i=1}^n$.
- ▶ How do we construct our linear interpolant?
- ▶ Let's add up the appropriate "hat" functions:

- ▶ For each i , let $\phi^i(x) = \phi_{x_{i-1}, x_i, x_{i+1}}(x)$

- ▶ This is putting a little hat function over every data point we have

We have to be careful at the edges. Pick any $x_0 < x_1$ and $x_{n+1} > x_n$ so that this definition works

- ▶ Take a look closely at ϕ^i . For all $1 < i < n$:

$$\phi^i(x_{i-1}) = 0$$

$$\phi^i(x_i) = 1$$

$$\phi^i(x_{i+1}) = 0$$

- ▶ Define $\hat{f}(x) := \sum_{i=1}^n c_i \phi^i(x)$ for some coefficients c_i

- ▶ Let's evaluate $\hat{f}$ at each x_j :

$$\hat{f}(x_j) = \underbrace{\sum_{i=1}^n c_i \phi^i(x_j)}_{\text{All 0 when } i \neq j} = c_j \phi^j(x_j) = c_j \quad (8)$$

Linear Interpolant

- ▶ Suppose we have a function $f : [a, b] \rightarrow \mathbb{R}$ and have the data points $\{(x_i, y_i)\}_{i=1}^n$.
- ▶ How do we construct our linear interpolant?
- ▶ Let's add up the appropriate "hat" functions:

- ▶ For each i , let $\phi^i(x) = \phi_{x_{i-1}, x_i, x_{i+1}}(x)$

- ▶ This is putting a little hat function over every data point we have

We have to be careful at the edges. Pick any $x_0 < x_1$ and $x_{n+1} > x_n$ so that this definition works

- ▶ Take a look closely at ϕ^i . For all $1 < i < n$:

$$\phi^i(x_{i-1}) = 0$$

$$\phi^i(x_i) = 1$$

$$\phi^i(x_{i+1}) = 0$$

- ▶ Define $\hat{f}(x) := \sum_{i=1}^n c_i \phi^i(x)$ for some coefficients c_i
- ▶ Let's evaluate $\hat{f}$ at each x_j :

$$\hat{f}(x_j) = \underbrace{\sum_{i=1}^n c_i \phi^i(x_j)}_{\text{All 0 when } i \neq j} = c_j \phi^j(x_j) = c_j \quad (8)$$

Linear Interpolation Solves a Linear System

- ▶ Let's impose our interpolation conditions

- ▶ We want $\hat{f}(x_i) = f(x_i) = y_i$ for all i

- ▶ That means eq. (8) implies

$$y_i = \hat{f}(x_i) = c_i \quad \text{for all } i \quad (9)$$

- ▶ This is a (really simple) system of linear equations:

$$\begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & \ddots & \\ & & & & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix} \quad (10)$$

- ▶ It's almost trivial, but we're going to come back to this when we discuss splines

Linear Interpolation

When to use it?

- ▶ Linear interpolation is a great fallback if you have a badly behaved function
 - ▶ E.g., kinks, poles, etc...
- ▶ It's simple and easy to implement: it's basically our mental model anyway
- ▶ You'll never be confused about why it's doing what it's doing
- ▶ Downsides:
 - ▶ Slow convergence – you often need way more grid points to get a good approximation
 - ▶ Not differentiable at the data points – sometimes an optimizer will get stuck on a kink and you will get a poor solution

Linear Interpolation

When to use it?

- ▶ Linear interpolation is a great fallback if you have a badly behaved function
 - ▶ E.g., kinks, poles, etc...
- ▶ It's simple and easy to implement: it's basically our mental model anyway
- ▶ You'll never be confused about why it's doing what it's doing
- ▶ Downsides:
 - ▶ Slow convergence – you often need way more grid points to get a good approximation
 - ▶ Not differentiable at the data points – sometimes an optimizer will get stuck on a kink and you will get a poor solution

Piecewise Cubic Approximation: Cubic Splines

- ▶ With piecewise linear functions, the problem is that they're not smooth enough
- ▶ What if we tried the same approach, but with a cubic polynomial on each sub-interval?
- ▶ Suppose for every interval $[x_{i-1}, x_i]$ we want our approximation to be a cubic polynomial:

$$\hat{f}(x) = a_i + b_i x + c_i x^2 + d_i x^3 \quad \text{for } x \in [x_{i-1}, x_i], \text{ and for all } i$$

- ▶ We have several conditions we want:

$$\begin{array}{ll} \text{Interpolation:} & y_i = a_i + b_i x_i + c_i x_i^2 + d_i x_i^3 \\ & \text{for } i = 1, \dots, n \end{array} \quad (11)$$

$$\begin{array}{ll} \text{Continuity:} & y_i = a_{i+1} + b_{i+1} x_i + c_{i+1} x_i^2 + d_{i+1} x_i^3 \\ & \text{for } i = 0, \dots, n-1 \end{array} \quad (12)$$

$$\begin{array}{ll} \text{Continuous } \hat{f}' : & b_i + 2c_i x_i + 3d_i x_i^2 = b_{i+1} + 2c_{i+1} x_i + 3d_{i+1} x_i^2 \\ & \text{for } i = 1, \dots, n-1 \end{array} \quad (13)$$

$$\begin{array}{ll} \text{Continuous } \hat{f}'' : & 2c_i + 6d_i x_i = 2c_{i+1} + 6d_{i+1} x_i \\ & \text{for } i = 1, \dots, n-1 \end{array} \quad (14)$$

Piecewise Cubic Approximation: Cubic Splines

- ▶ With piecewise linear functions, the problem is that they're not smooth enough
- ▶ What if we tried the same approach, but with a cubic polynomial on each sub-interval?
- ▶ Suppose for every interval $[x_{i-1}, x_i]$ we want our approximation to be a cubic polynomial:

$$\hat{f}(x) = a_i + b_i x + c_i x^2 + d_i x^3 \quad \text{for } x \in [x_{i-1}, x_i], \text{ and for all } i$$

- ▶ We have several conditions we want:

Interpolation:
$$y_i = a_i + b_i x_i + c_i x_i^2 + d_i x_i^3 \quad (11)$$

for $i = 1, \dots, n$

Continuity:
$$y_i = a_{i+1} + b_{i+1} x_i + c_{i+1} x_i^2 + d_{i+1} x_i^3 \quad (12)$$

for $i = 0, \dots, n-1$

Continuous $\hat{f}'$:
$$b_i + 2c_i x_i + 3d_i x_i^2 = b_{i+1} + 2c_{i+1} x_i + 3d_{i+1} x_i^2 \quad (13)$$

for $i = 1, \dots, n-1$

Continuous $\hat{f}''$:
$$2c_i + 6d_i x_i = 2c_{i+1} + 6d_{i+1} x_i \quad (14)$$

for $i = 1, \dots, n-1$

Piecewise Cubic Approximation: Cubic Splines

- ▶ With piecewise linear functions, the problem is that they're not smooth enough
- ▶ What if we tried the same approach, but with a cubic polynomial on each sub-interval?
- ▶ Suppose for every interval $[x_{i-1}, x_i]$ we want our approximation to be a cubic polynomial:

$$\hat{f}(x) = a_i + b_i x + c_i x^2 + d_i x^3 \quad \text{for } x \in [x_{i-1}, x_i], \text{ and for all } i$$

- ▶ We have several conditions we want:

Interpolation:

$$y_i = a_i + b_i x_i + c_i x_i^2 + d_i x_i^3 \quad (11)$$

for $i = 1, \dots, n$

Continuity:

$$y_i = a_{i+1} + b_{i+1} x_i + c_{i+1} x_i^2 + d_{i+1} x_i^3 \quad (12)$$

for $i = 0, \dots, n-1$

Continuous $\hat{f}'$:

$$b_i + 2c_i x_i + 3d_i x_i^2 = b_{i+1} + 2c_{i+1} x_i + 3d_{i+1} x_i^2 \quad (13)$$

for $i = 1, \dots, n-1$

Continuous $\hat{f}''$:

$$2c_i + 6d_i x_i = 2c_{i+1} + 6d_{i+1} x_i \quad (14)$$

for $i = 1, \dots, n-1$

Piecewise Cubic Approximation: Cubic Splines

- ▶ With piecewise linear functions, the problem is that they're not smooth enough
- ▶ What if we tried the same approach, but with a cubic polynomial on each sub-interval?
- ▶ Suppose for every interval $[x_{i-1}, x_i]$ we want our approximation to be a cubic polynomial:

$$\hat{f}(x) = a_i + b_i x + c_i x^2 + d_i x^3 \quad \text{for } x \in [x_{i-1}, x_i], \text{ and for all } i$$

- ▶ We have several conditions we want:

Interpolation:
$$y_i = a_i + b_i x_i + c_i x_i^2 + d_i x_i^3 \quad (11)$$

for $i = 1, \dots, n$

Continuity:
$$y_i = a_{i+1} + b_{i+1} x_i + c_{i+1} x_i^2 + d_{i+1} x_i^3 \quad (12)$$

for $i = 0, \dots, n-1$

Continuous $\hat{f}'$:
$$b_i + 2c_i x_i + 3d_i x_i^2 = b_{i+1} + 2c_{i+1} x_i + 3d_{i+1} x_i^2 \quad (13)$$

for $i = 1, \dots, n-1$

Continuous $\hat{f}''$:
$$2c_i + 6d_i x_i = 2c_{i+1} + 6d_{i+1} x_i \quad (14)$$

for $i = 1, \dots, n-1$

Piecewise Cubic Approximation: Cubic Splines

- ▶ With piecewise linear functions, the problem is that they're not smooth enough
- ▶ What if we tried the same approach, but with a cubic polynomial on each sub-interval?
- ▶ Suppose for every interval $[x_{i-1}, x_i]$ we want our approximation to be a cubic polynomial:

$$\hat{f}(x) = a_i + b_i x + c_i x^2 + d_i x^3 \quad \text{for } x \in [x_{i-1}, x_i], \text{ and for all } i$$

- ▶ We have several conditions we want:

Interpolation:
$$y_i = a_i + b_i x_i + c_i x_i^2 + d_i x_i^3 \quad (11)$$

for $i = 1, \dots, n$

Continuity:
$$y_i = a_{i+1} + b_{i+1} x_i + c_{i+1} x_i^2 + d_{i+1} x_i^3 \quad (12)$$

for $i = 0, \dots, n-1$

Continuous $\hat{f}'$:
$$b_i + 2c_i x_i + 3d_i x_i^2 = b_{i+1} + 2c_{i+1} x_i + 3d_{i+1} x_i^2 \quad (13)$$

for $i = 1, \dots, n-1$

Continuous $\hat{f}''$:
$$2c_i + 6d_i x_i = 2c_{i+1} + 6d_{i+1} x_i \quad (14)$$

for $i = 1, \dots, n-1$

Piecewise Cubic Approximation: Cubic Splines

- ▶ With piecewise linear functions, the problem is that they're not smooth enough
- ▶ What if we tried the same approach, but with a cubic polynomial on each sub-interval?
- ▶ Suppose for every interval $[x_{i-1}, x_i]$ we want our approximation to be a cubic polynomial:

$$\hat{f}(x) = a_i + b_i x + c_i x^2 + d_i x^3 \quad \text{for } x \in [x_{i-1}, x_i], \text{ and for all } i$$

- ▶ We have several conditions we want:

$$\begin{aligned} \text{Interpolation:} \quad & y_i = a_i + b_i x_i + c_i x_i^2 + d_i x_i^3 \\ & \text{for } i = 1, \dots, n \end{aligned} \tag{11}$$

$$\begin{aligned} \text{Continuity:} \quad & y_i = a_{i+1} + b_{i+1} x_i + c_{i+1} x_i^2 + d_{i+1} x_i^3 \\ & \text{for } i = 0, \dots, n-1 \end{aligned} \tag{12}$$

$$\begin{aligned} \text{Continuous } \hat{f}' : \quad & b_i + 2c_i x_i + 3d_i x_i^2 = b_{i+1} + 2c_{i+1} x_i + 3d_{i+1} x_i^2 \\ & \text{for } i = 1, \dots, n-1 \end{aligned} \tag{13}$$

$$\begin{aligned} \text{Continuous } \hat{f}'' : \quad & 2c_i + 6d_i x_i = 2c_{i+1} + 6d_{i+1} x_i \\ & \text{for } i = 1, \dots, n-1 \end{aligned} \tag{14}$$

Cubic splines solve a linear system

- ▶ So far this is all one big linear system of equations!
 - ▶ We know how to solve linear systems
 - ▶ Stack the conditions up in a matrix, and have the computer solve it
- ▶ We have $4n$ variables and $4n - 2$ equations
- ▶ Why did we lose two equations? Check back on the previous slide
 - ▶ Continuity of the derivatives is only imposed in the interior.
 - ▶ Need to make some assumptions about the derivatives of our approximation at the edges of our domain
 - ▶ These are called **boundary conditions**

Cubic splines solve a linear system

- ▶ So far this is all one big linear system of equations!
 - ▶ We know how to solve linear systems
 - ▶ Stack the conditions up in a matrix, and have the computer solve it
- ▶ We have $4n$ variables and $4n - 2$ equations
- ▶ Why did we lose two equations? Check back on the previous slide
 - ▶ Continuity of the derivatives is only imposed in the interior.
 - ▶ Need to make some assumptions about the derivatives of our approximation at the edges of our domain
 - ▶ These are called **boundary conditions**

Cubic splines solve a linear system

- ▶ So far this is all one big linear system of equations!
 - ▶ We know how to solve linear systems
 - ▶ Stack the conditions up in a matrix, and have the computer solve it
- ▶ We have $4n$ variables and $4n - 2$ equations
- ▶ Why did we lose two equations? Check back on the previous slide
 - ▶ Continuity of the derivatives is only imposed in the interior.
 - ▶ Need to make some assumptions about the derivatives of our approximation at the edges of our domain
 - ▶ These are called **boundary conditions**

Spline Boundary Conditions

- ▶ There are three main options:

- ▶ Natural spline: $\hat{f}'(x_0) = 0 = \hat{f}'(x_n)$

- ▶ Hermite Spline: $\hat{f}'(x_0) = y'_0$ and $\hat{f}'(x_n) = y'_n$

Assumes you know the true derivatives at the boundary

- ▶ Secant spline: $\hat{f}'(x_0) = \frac{\hat{f}(x_1) - \hat{f}(x_0)}{x_1 - x_0}$ and a similar condition for $\hat{f}'(x_n)$

Assumes a linear approximation of the derivative at the lower and upper bounds

- ▶ Which you choose depends on the specifics of the problem

- ▶ Often the *natural* spline is not a good fit if you know your function is strictly concave (like a utility function)

How to actually use splines?

- ▶ Unless you're explicitly asked, don't code these up yourself
- ▶ Extremely efficient implementations (for splines and linear interpolation) are available in `Interpolations.jl`
- ▶ There are also some other fun things you can try:
 - ▶ Shape preserving splines: these splines add extra conditions to ensure that the approximation will never have a curvature that does not match the input data
 - ▶ I.e, if your data is sampled from a strictly concave function, the resulting spline will also be strictly concave
- ▶ Crucially, all of these methods generalize quite nicely to multiple dimensions:
 - ▶ In `Interpolations.jl` it's the same functions for multidimensional splines

How to actually use splines?

- ▶ Unless you're explicitly asked, don't code these up yourself
- ▶ Extremely efficient implementations (for splines and linear interpolation) are available in `Interpolations.jl`
- ▶ There are also some other fun things you can try:
 - ▶ Shape preserving splines: these splines add extra conditions to ensure that the approximation will never have a curvature that does not match the input data
 - ▶ I.e, if your data is sampled from a strictly concave function, the resulting spline will also be strictly concave
- ▶ Crucially, all of these methods generalize quite nicely to multiple dimensions:
 - ▶ In `Interpolations.jl` it's the same functions for multidimensional splines

How to actually use splines?

- ▶ Unless you're explicitly asked, don't code these up yourself
- ▶ Extremely efficient implementations (for splines and linear interpolation) are available in `Interpolations.jl`
- ▶ There are also some other fun things you can try:
 - ▶ Shape preserving splines: these splines add extra conditions to ensure that the approximation will never have a curvature that does not match the input data
 - ▶ I.e, if your data is sampled from a strictly concave function, the resulting spline will also be strictly concave
- ▶ Crucially, all of these methods generalize quite nicely to multiple dimensions:
 - ▶ In `Interpolations.jl` it's the same functions for multidimensional splines

How to actually use splines?

- ▶ Unless you're explicitly asked, don't code these up yourself
- ▶ Extremely efficient implementations (for splines and linear interpolation) are available in `Interpolations.jl`
- ▶ There are also some other fun things you can try:
 - ▶ Shape preserving splines: these splines add extra conditions to ensure that the approximation will never have a curvature that does not match the input data
 - ▶ I.e, if your data is sampled from a strictly concave function, the resulting spline will also be strictly concave
- ▶ Crucially, all of these methods generalize quite nicely to multiple dimensions:
 - ▶ In `Interpolations.jl` it's the same functions for multidimensional splines

Section 4

Optional Content

Lagrange Interpolant is Unique

Theorem 1

Suppose we have data $\{(x_i, y_i) \mid i = 1, \dots, n\}$ where the x_i are all unique. There is a unique polynomial of degree $n - 1$ that interpolates these values.

Proof.

- ▶ Since the Vandermonde matrix V has full rank, we know that a solution $p(x)$ exists
- ▶ Suppose that $\hat{p}(x)$ is a polynomial of degree at most $n - 1$ which also interpolates these points.
- ▶ We know that since $\hat{p}$ interpolates our data, $\hat{p}(x_i) = p(x_i) = y_i$ for all i .
- ▶ This means that $g(x) = p(x) - \hat{p}(x)$ is a polynomial of degree at most $n - 1$ which has n distinct zeros (all of the data points).
- ▶ The only such polynomial is the zero polynomial, which implies that $p = \hat{p}$ □

Chebyshev Approximation Theorem

Theorem 2

Assume that $f : [-1, 1] \rightarrow \mathbb{R}$ has continuous k th derivatives. If

$$c_j \equiv \frac{2}{\pi} \int_{-1}^1 \frac{f(x) T_j(x)}{\sqrt{1-x^2}} dx$$

and

$$C_n(x) \equiv \frac{1}{2} c_0 + \sum_{j=1}^n c_j T_j(x)$$

Then there is a $B < \infty$ such that for all $n \geq 2$:

$$\|f - C_n\|_{\infty} \leq \frac{B \log n}{n^k}$$

- ▶ This means that for smooth enough functions, our Chebyshev approximation will converge uniformly (and rapidly) to the true function
- ▶ Note that $\|f\|_{\infty} := \max_x |f(x)|$