

# Lecture 4: Math Review

# Administrative

HW0 out, due 2/3

HW1 out soon

# This Lecture: Math Review

Two goals for today:

- Math with computers  $\neq$  Math
- Practical math you need to know but may not have been taught

~~M | ISA MATHEMATICS  
UNIVERSITY OF MICHIGAN~~

# This Lecture: Goal

- Not a “Linear algebra in one lecture” – that’s impossible.
- *You should have seen this before*
- Aimed at reviving your knowledge and plugging any gaps
- Aimed at giving you intuitions
- **I’ll intentionally go fast**; intent is for slides to be a reference for you to review later

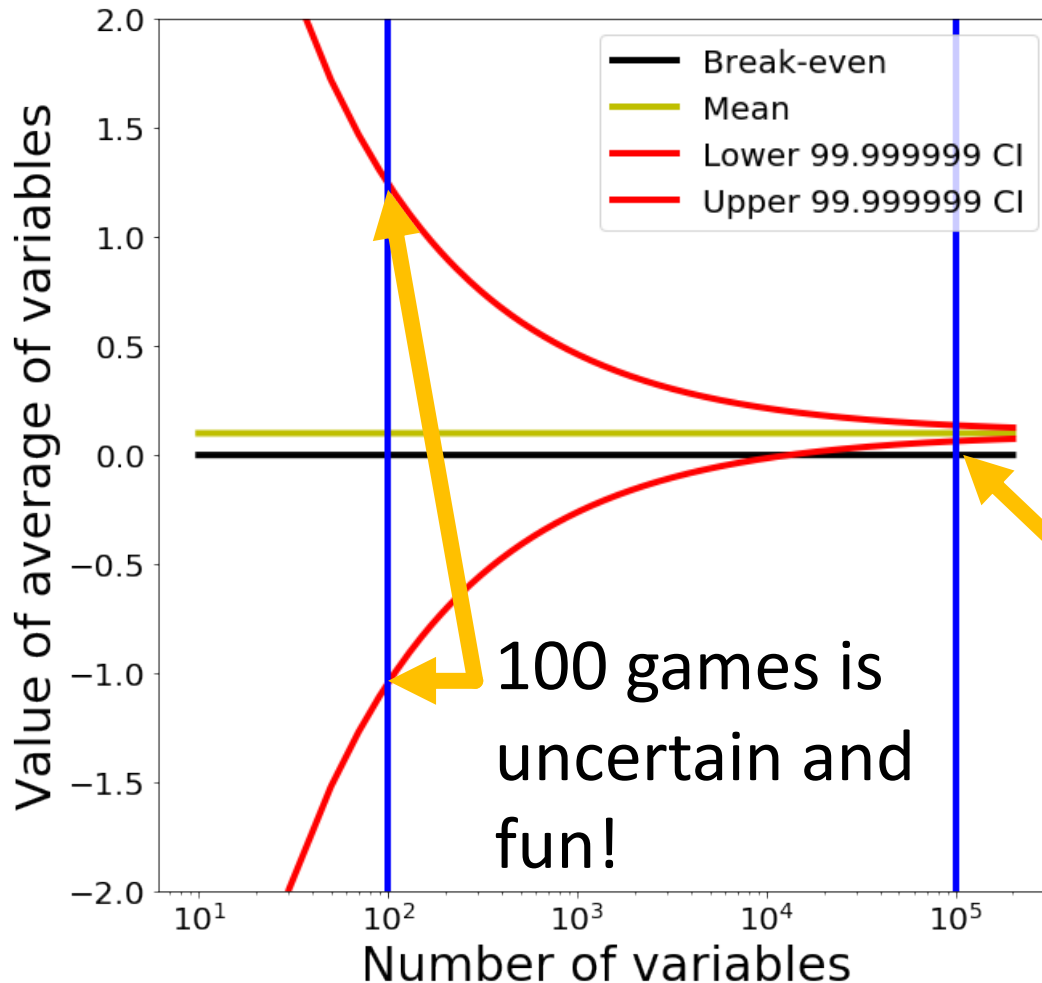
# Floating Point Arithmetic

# Adding Numbers

- **$1 + 1 = ?$**
- Suppose  $x_i$  is normally distributed with mean  $\mu$  and standard deviation  $\sigma$  for  $1 \leq i \leq N$
- **How is the average, or  $\hat{\mu} = \frac{1}{N} \sum_{i=1}^N x_i$ , distributed (qualitatively), in terms of variance?**
- *The Free Drinks in Vegas Theorem:*  
 $\hat{\mu}$  has mean  $\mu$  and standard deviation  $\frac{\sigma}{\sqrt{N}}$ .

# Free Drinks in Vegas

Each game/variable has mean \$0.10, std \$2



100K games is guaranteed profit: 99.999999% lowest value is \$0.064. \$0.01 for drinks \$0.054 for profits

# Let's make it big

- Suppose we take the average of 50M normally distributed numbers (mean=31, std=1)
- **Theory:** Average is a normally distributed random variable with mean 31 and std  $\frac{1}{\sqrt{50M}} \approx 10^{-5}$

- **Practice:**

```
#include <iostream>
#include <random>

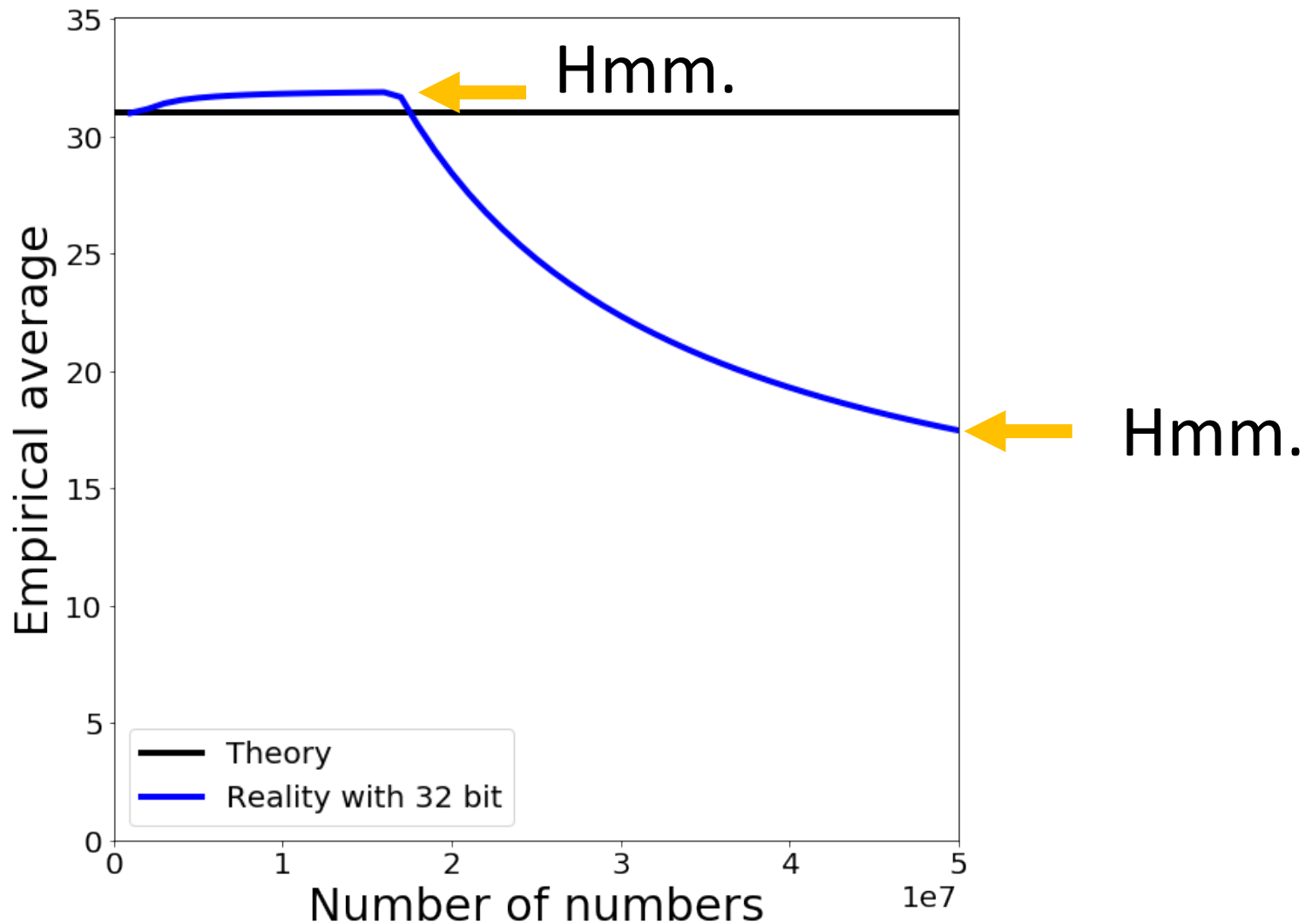
int main() {
    std::default_random_engine engine;
    std::normal_distribution<float> normal(31, 1);
    const int N = 50000000;
    float avg = 0.0f;
    for (int i = 0; i < N; ++i) {
        avg += normal(engine);
    }
    avg /= N;
    std::cout << avg << std::endl;
    return 0;
}
```

**Result:**

17.4695??





# Trying it out



# What is a number?

$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	
1	0	1	1	1	0	0	1	185
$128 + 32 + 16 + 8 + 1 =$								185

# Adding two numbers

$2^8$	$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	
	1	0	1	1	1	0	0	1	185
<b>+</b>	0	1	1	0	1	0	0	1	105
<hr/>									
1	0	0	1	0	0	0	1	0	34
									
Carry Flag	Result								

*“Integers” on a computer are integers modulo  $2^k$*

# Some Gotchas

$$32 + (3 / 4) \times 40 = 32$$

$$32 + (3 \times 40) / 4 = 62$$

**Why?**

Underflow

No Underflow

$$32 + (3 / 4) \times 40 =$$

$$32 + 0 \quad \times 40 =$$

$$32 + 0 \quad =$$

$$32$$

$$32 + (3 \times 40) / 4 =$$

$$32 + 120 \quad / 4 =$$

$$32 + 30 \quad =$$

$$62$$

*Ok – you have to multiply before dividing*

# Some Gotchas

math  $32 + (9 \times 40) / 10 = 68$

Should be:  
 $9 \times 4 = 36$

uint8  $32 + (9 \times 40) / 10 = 42$

## Overflow

$32 + 9 \times 40 / 10 =$

$32 + 104 / 10 =$

$32 + 10 =$

42

## Why 104?

$9 \times 40 = 360$

$360 \% 256 = 104$

# What is a number?

$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	
1	0	1	1	1	0	0	1	185

**How can we do fractions?**

$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	$2^{-1}$	$2^{-2}$	
1	0	1	1	1	0	0	1	45.25

45      0.25

# Fixed-Point Arithmetic

$$\begin{array}{cccccccc} 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 & 2^{-1} & 2^{-2} \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{array} \quad 45.25$$

**What's the largest number we can represent?**

**63.75 – Why?**

**How precisely can we measure at 63?**

0.25

**How precisely can we measure at 0?**

0.25

Fine for many purposes but for science, seems silly

# Floating Point Numbers

Sign (S)

Exponent (E)

Fraction (F)

1

0 1 1 1

0 0 1

1

7

1

-1

$2^{7-7} = 2^0 = 1$

$1 + 1/8 = 1.125$

$$\left(-1^S\right)\left(2^{E+bias}\right)\left(1 + \frac{F}{2^3}\right)$$

Bias: allows exponent to be negative (bias = -127 for float32)

Note: fraction = significant = mantissa;

exponents of all ones or all zeros are special numbers



# Floating Point Numbers

$$(-1^S)(2^{E+bias})\left(1 + \frac{F}{2^3}\right)$$

Sign	Exponent	Fraction	
1	0 1 1 1	0 0 0	$-2^0 \times 1.00 = -1$
		0 0 1	$-2^0 \times 1.125 = -1.125$
		0 1 0	$-2^0 \times 1.25 = -1.25$
		...	
		1 1 0	$-2^0 \times 1.75 = -1.75$
		1 1 1	$-2^0 \times 1.875 = -1.875$

-1      7-7=0  
 \*(-bias)\*

# Floating Point Numbers

$$(-1^S)(2^{E+bias})\left(1 + \frac{F}{2^3}\right)$$

Sign	Exponent	Fraction	
1	1 0 0 1	0 0 0	$-2^2 \times 1.00 = -4$
		0 0 1	$-2^2 \times 1.125 = -4.5$
		0 1 0	$-2^2 \times 1.25 = -5$
		...	
		1 1 0	$-2^2 \times 1.75 = -7$
		1 1 1	$-2^2 \times 1.875 = -7.5$

-1

9-7=2  
\*(-bias)\*

# Floating Point Numbers

Sign	Exponent	Fraction	
1	0 1 1 1	0 0 0	$-2^0 \times 1.00 = -1$
		0 0 1	$-2^0 \times 1.125 = -1.125$
1	1 0 0 1	0 0 0	$-2^2 \times 1.00 = -4$
		0 0 1	$-2^2 \times 1.125 = -4.5$

Gap between numbers is *relative*, not absolute

# Adding Floating Point Numbers

	Sign	Exponent	Fraction	
	1	0 1 1 0	0 0 0	$-2^{-1} \times 1.00 = -0.5$
+	1	1 0 0 1	0 0 0	$-2^2 \times 1.00 = -4$
<hr/>				
	1	1 0 0 1	0 0 1	$-2^2 \times 1.125 = -4.5$

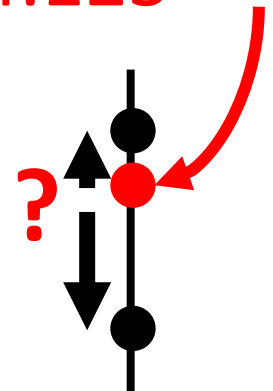
Actual implementation is complex

# Adding Floating Point Numbers

	Sign	Exponent	Fraction	
	1	0 1 0 0	0 0 0	$-2^{-3} \times 1.00 = -0.125$
+	1	1 0 0 1	0 0 0	$-2^2 \times 1.00 = -4$

$-2^2 \times 1.03125 = -4.125$

1	1 0 0 1	0 0 0	$-2^2 \times 1.00 = -4$
1	1 0 0 1	0 0 1	$-2^2 \times 1.125 = -4.5$

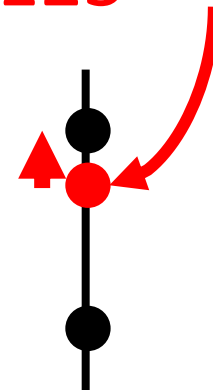


# Adding Floating Point Numbers

	Sign	Exponent	Fraction	
	1	0 1 0 0	0 0 0	$-2^{-3} \times 1.00 = -0.125$
+	1	1 0 0 1	0 0 0	$-2^2 \times 1.00 = -4$

$-2^2 \times 1.03125 = -4.125$

1	1 0 0 1	0 0 0	$-2^2 \times 1.00 = -4$
---	---------	-------	-------------------------



For a and b, these can happen

$$a + b = a \quad a + b - a \neq b$$

# Real Floating Point Numbers

IEEE 754 Single Precision / Single / float32

8 bits

$$2^{127} \approx 10^{38}$$

23 bits

$\approx 7$  decimal digits



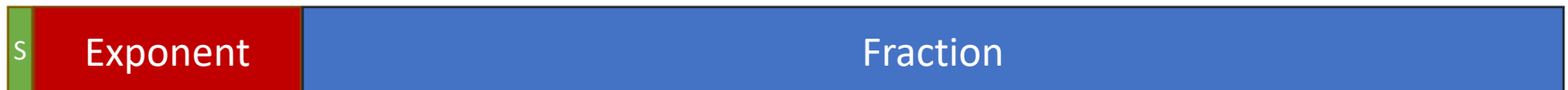
IEEE 754 Double Precision / Double / float64

11 bits

$$2^{1023} \approx 10^{308}$$

52 bits

$\approx 15$  decimal digits



# Real Floating Point Numbers

IEEE 754 Half Precision / Half / float16

5 bits  
 $2^{32} \approx 10^9$

10 bits  
 $\approx 3$  decimal digits



Brain Floating Point / bfloat16

8 bits  
 $2^{127} \approx 10^{38}$

7 bits  
 $\approx 2$  decimal digits

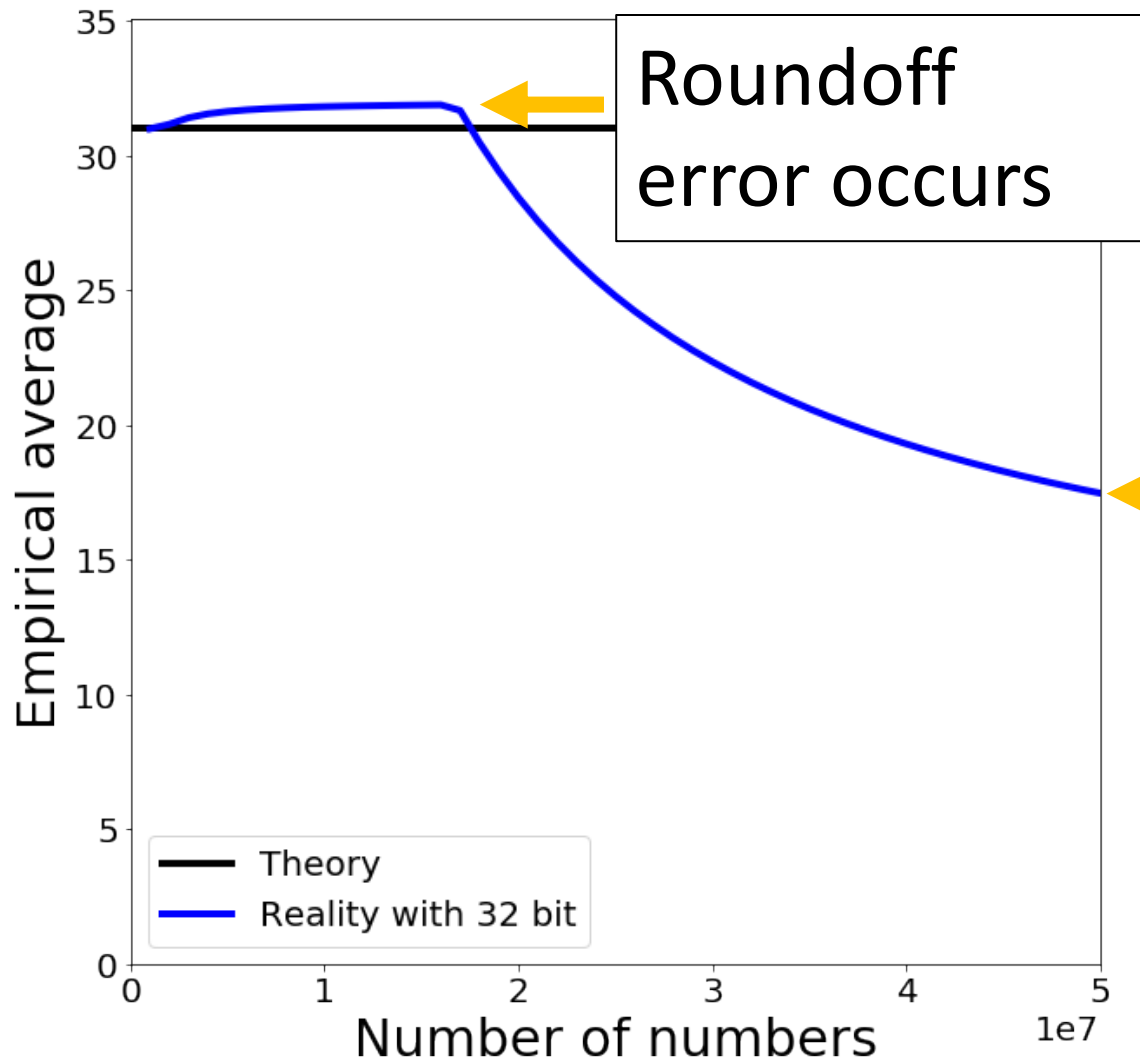


Same range as FP32, but reduced precision



# Trying it out

**Recall:** Average of many Gaussian random variables



$a+b=a \rightarrow$   
numerator is stuck,  
denominator isn't

# Things to Remember

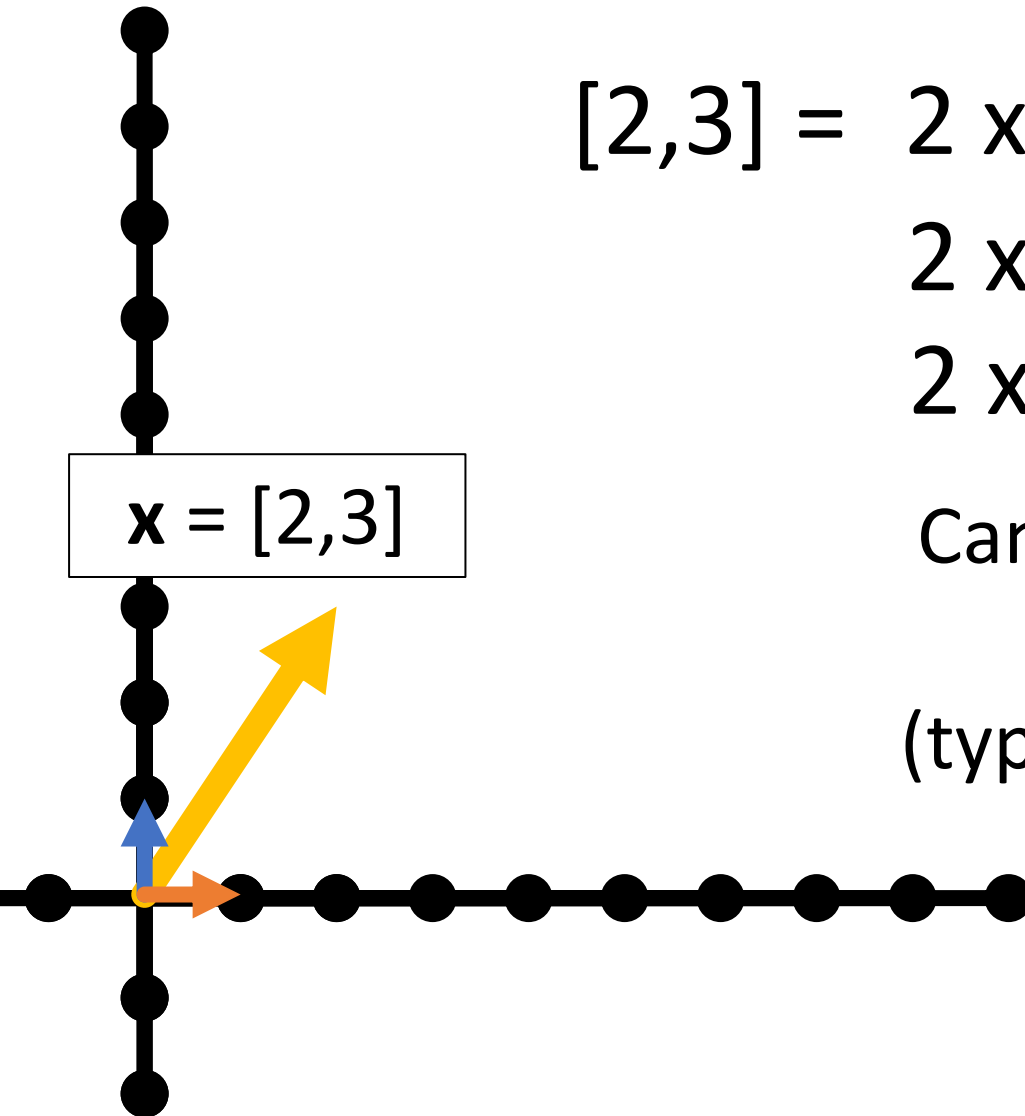
- Computer numbers aren't math numbers
- Overflow, accidental zeros, roundoff error, and basic equalities are almost certainly incorrect for some values
- Floating point defaults and numpy try to protect you.
- Generally safe to use a double and use built-in-functions in numpy (not necessarily others!)
- Spooky behavior = look for numerical issues

# Vectors

# Operations You Should Know

- Scale (vector, scalar  $\rightarrow$  vector)
- Add (vector, vector  $\rightarrow$  vector)
- Magnitude (vector  $\rightarrow$  scalar)
- Dot product (vector, vector  $\rightarrow$  scalar)
  - $\mathbf{v} \cdot \mathbf{w} = \|\mathbf{v}\| \|\mathbf{w}\| \cos \theta$
  - Dot products are projection / angles
  - Orthogonal vectors have dot product 0
- Cross product (vector, vector  $\rightarrow$  vector)
  - Only in 3 dimensions!
  - Output is orthogonal to both inputs
  - Vectors facing same direction have cross product  $\mathbf{0}$
- You can **never** mix vectors of different sizes

# Vectors



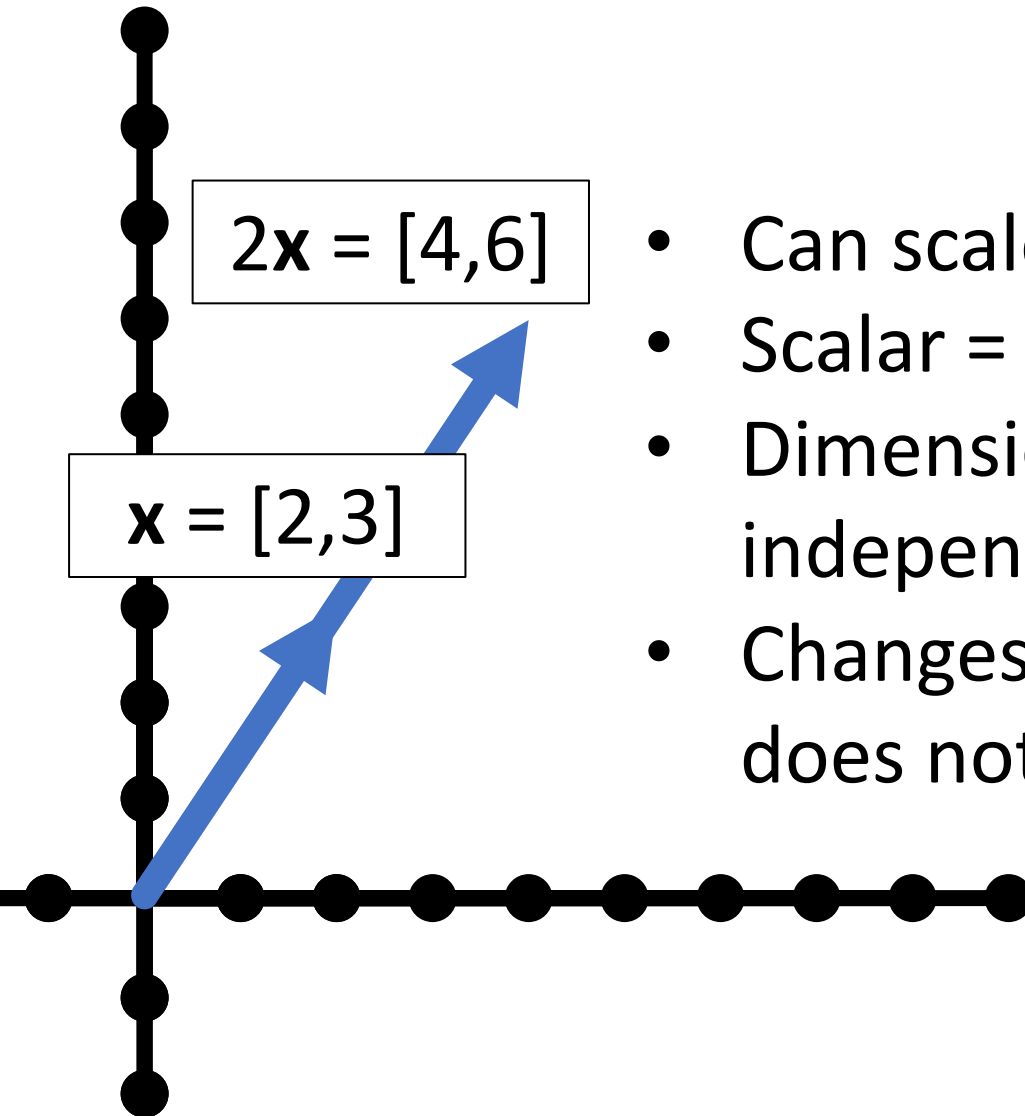
$$[2,3] = 2 \times [1,0] + 3 \times [0,1]$$

$$2 \times \rightarrow + 3 \times \uparrow$$

$$2 \times e_1 + 3 \times e_2$$

Can be arbitrary # of  
dimensions  
(typically denoted  $\mathbb{R}^n$ )

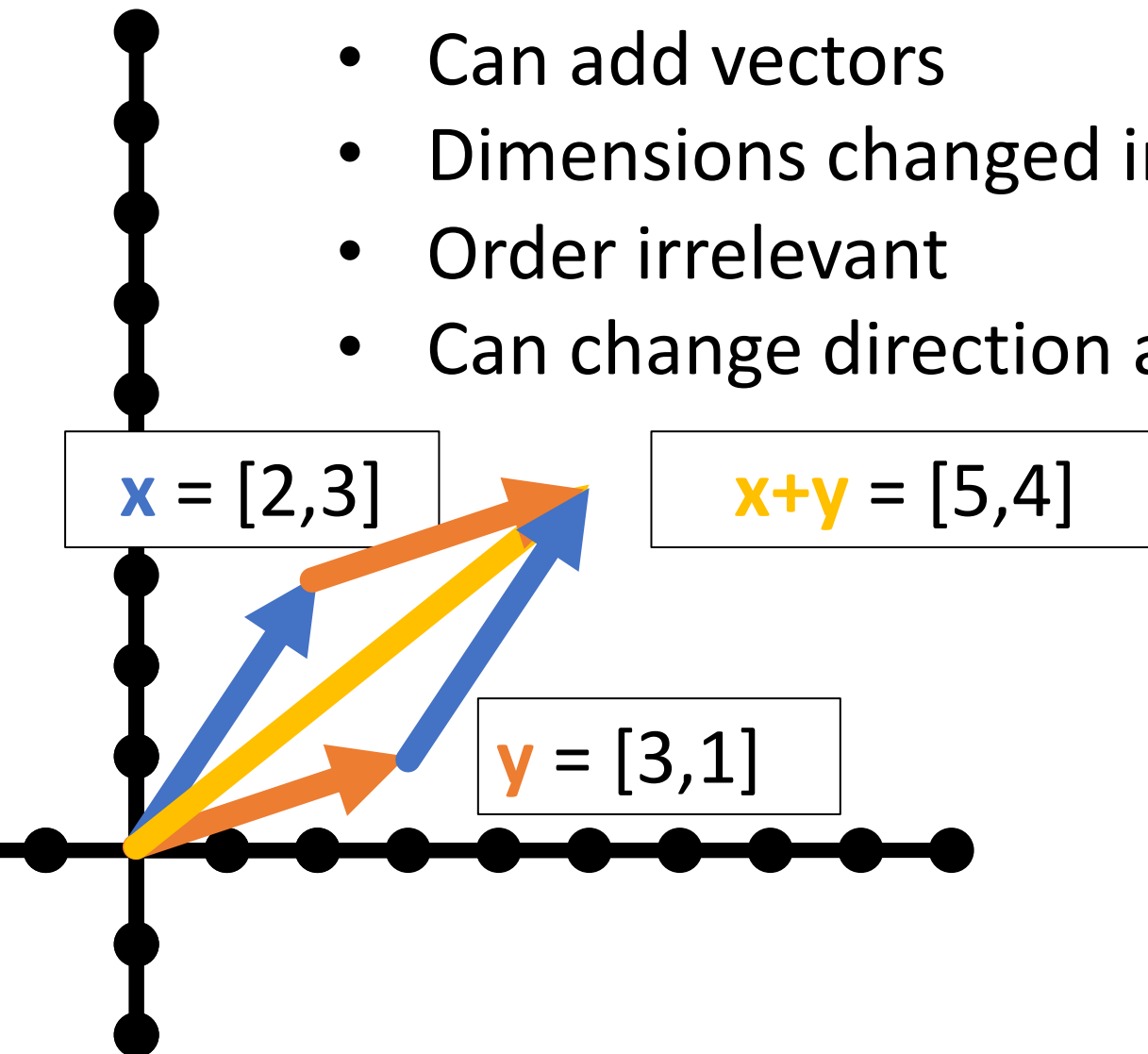
# Scaling Vectors



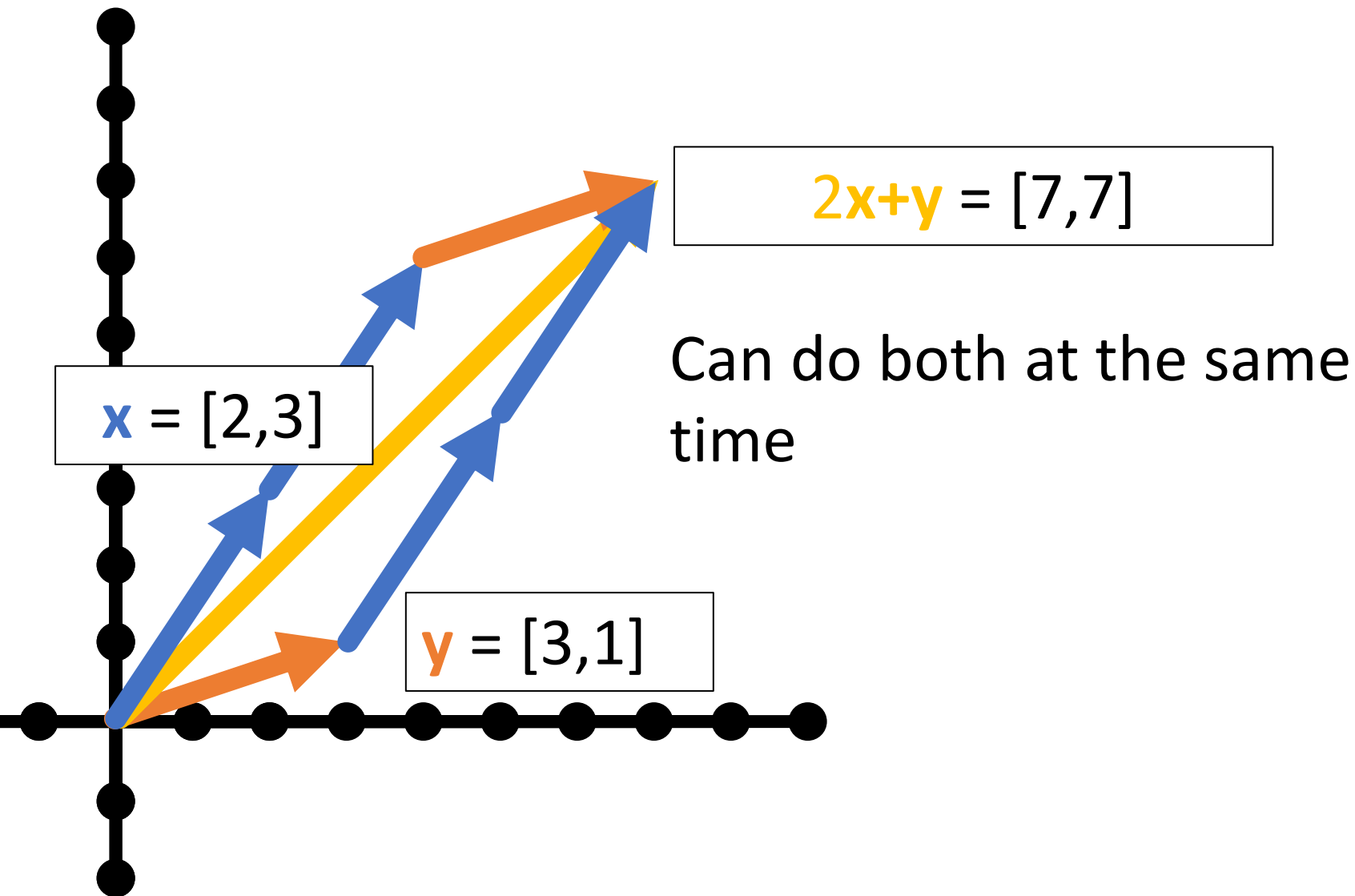
- Can scale vector by a *scalar*
- Scalar = single number
- Dimensions changed independently
- Changes *magnitude / length*, does not change *direction*.

# Adding Vectors

- Can add vectors
- Dimensions changed independently
- Order irrelevant
- Can change direction and magnitude



# Scaling and Adding





# Measuring Length

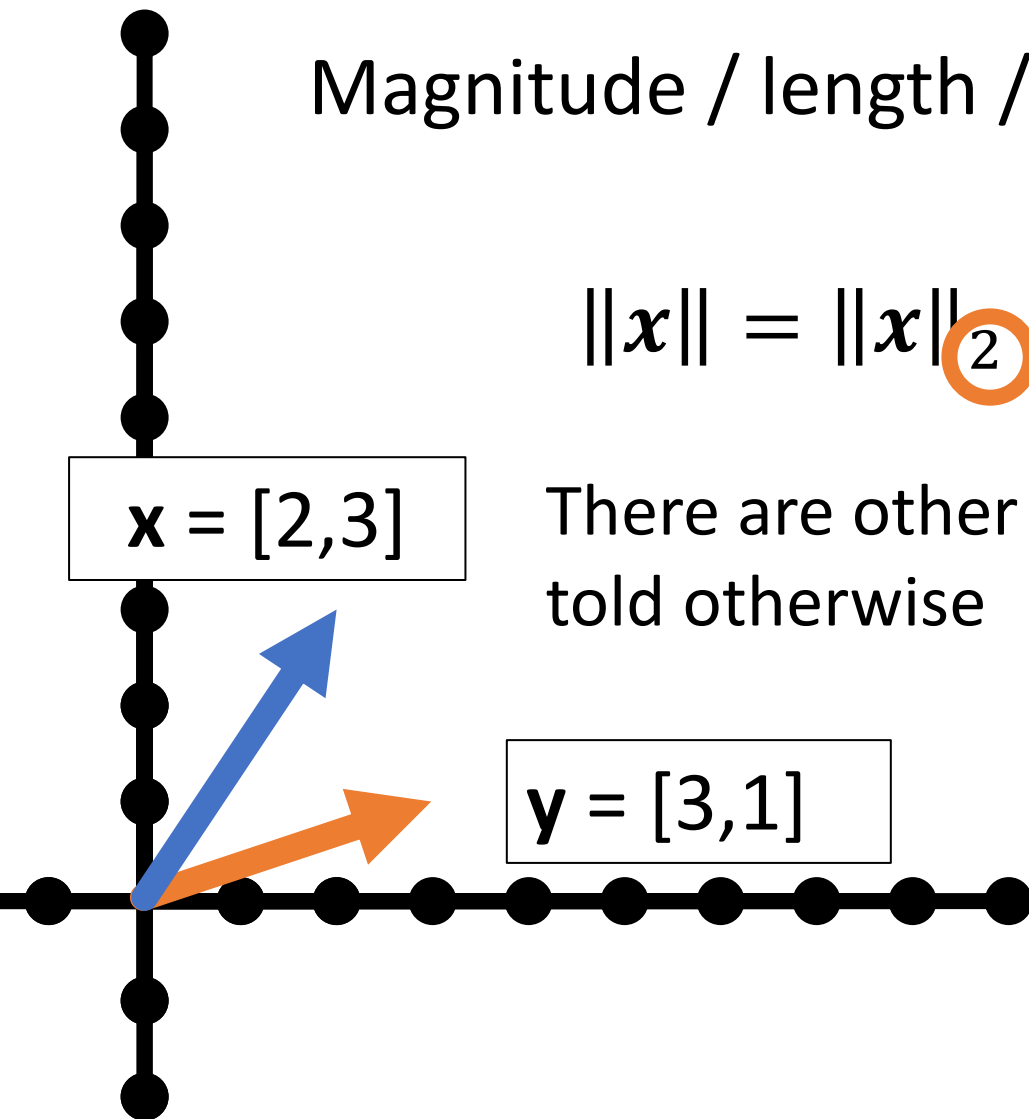
Magnitude / length / (L2) norm of vector

$$\|\mathbf{x}\| = \|\mathbf{x}\|_2 = \left( \sum_i^n x_i^2 \right)^{1/2}$$

There are other norms; assume L2 unless told otherwise

$$\|\mathbf{x}\|_2 = \sqrt{13}$$
$$\|\mathbf{y}\|_2 = \sqrt{10}$$

Why?



# Normalizing a Vector

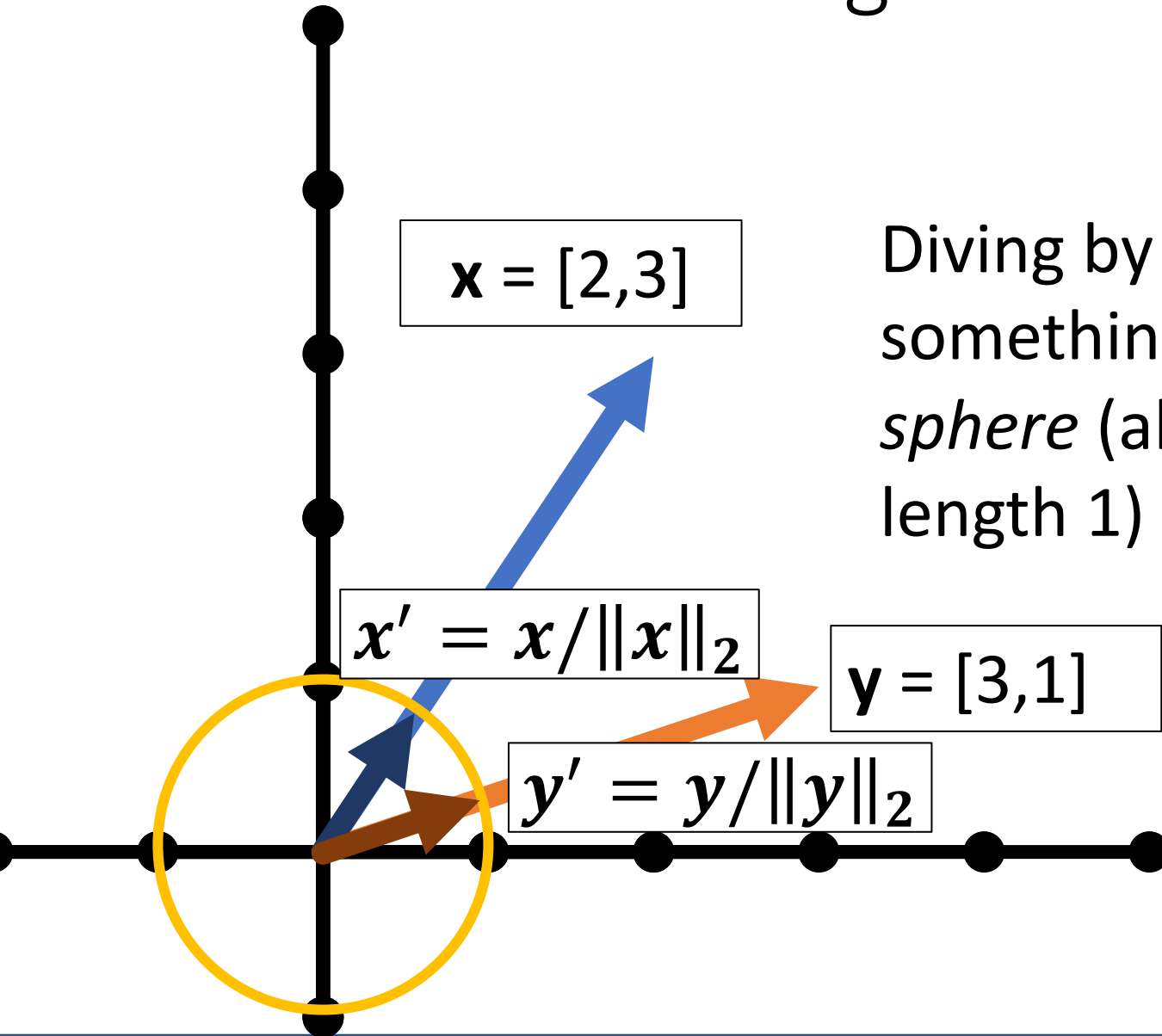
$$\mathbf{x} = [2, 3]$$

Dividing by norm gives something on the *unit sphere* (all vectors with length 1)

$$\mathbf{x}' = \mathbf{x} / \|\mathbf{x}\|_2$$

$$\mathbf{y} = [3, 1]$$

$$\mathbf{y}' = \mathbf{y} / \|\mathbf{y}\|_2$$

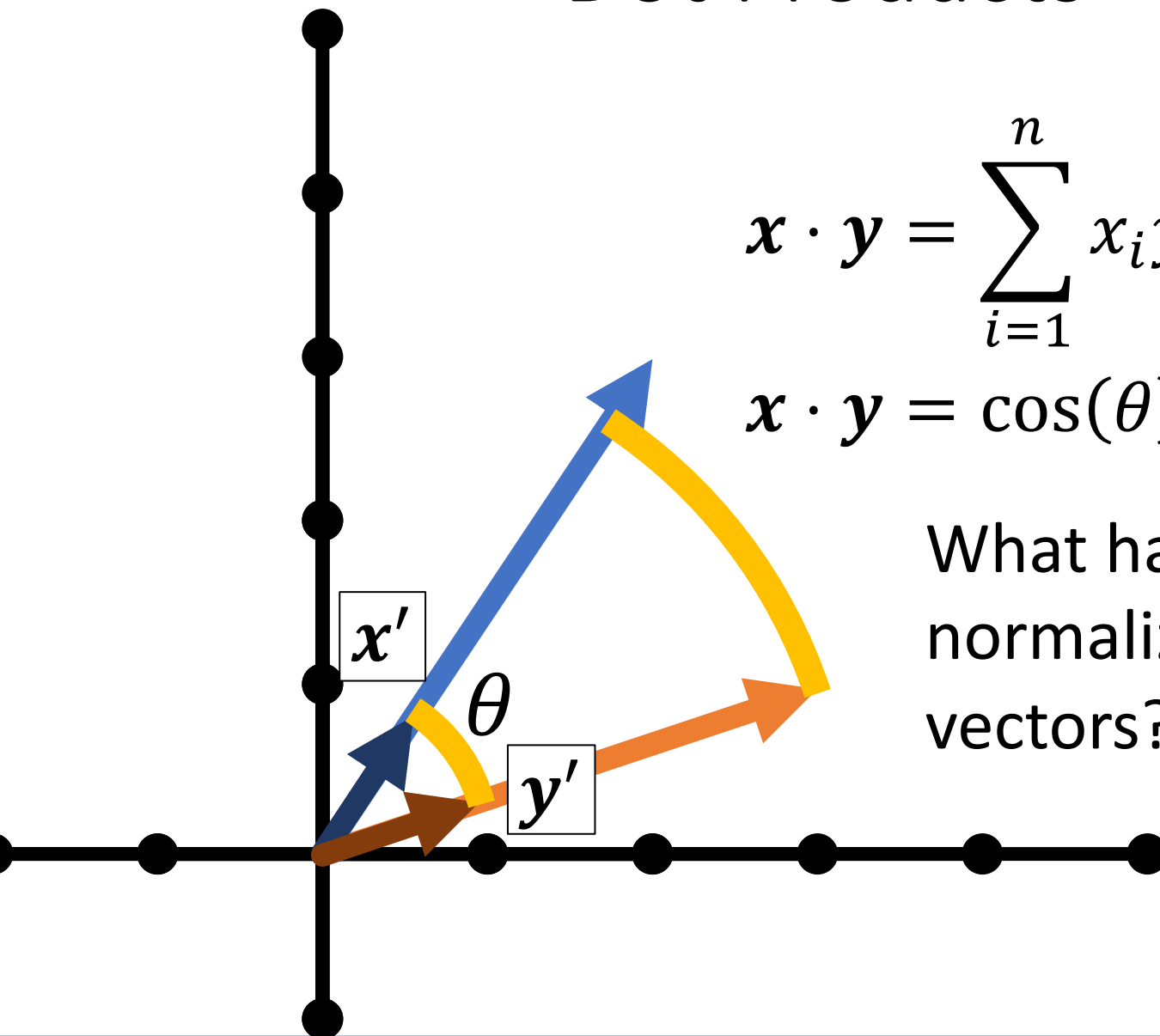


# Dot Products

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

$$\mathbf{x} \cdot \mathbf{y} = \cos(\theta) \|\mathbf{x}\| \|\mathbf{y}\|$$

What happens with  
normalized / unit  
vectors?



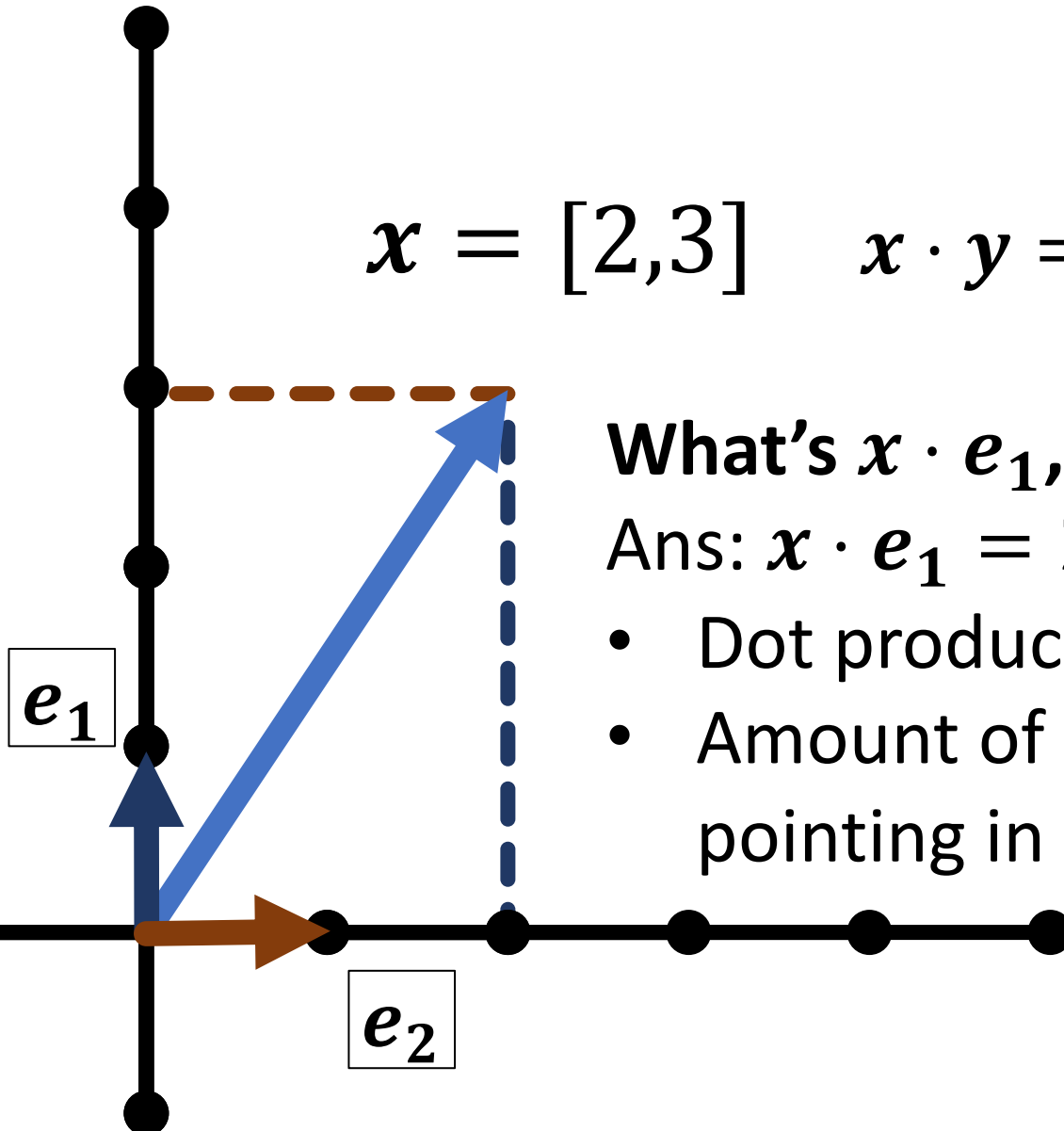
# Dot Products

$$\mathbf{x} = [2, 3] \quad \mathbf{x} \cdot \mathbf{y} = \sum_i^n x_i y_i$$

What's  $\mathbf{x} \cdot \mathbf{e}_1$ ,  $\mathbf{x} \cdot \mathbf{e}_2$ ?

Ans:  $\mathbf{x} \cdot \mathbf{e}_1 = 2$  ;  $\mathbf{x} \cdot \mathbf{e}_2 = 3$

- Dot product is projection
- Amount of  $\mathbf{x}$  that's also pointing in direction of  $\mathbf{y}$

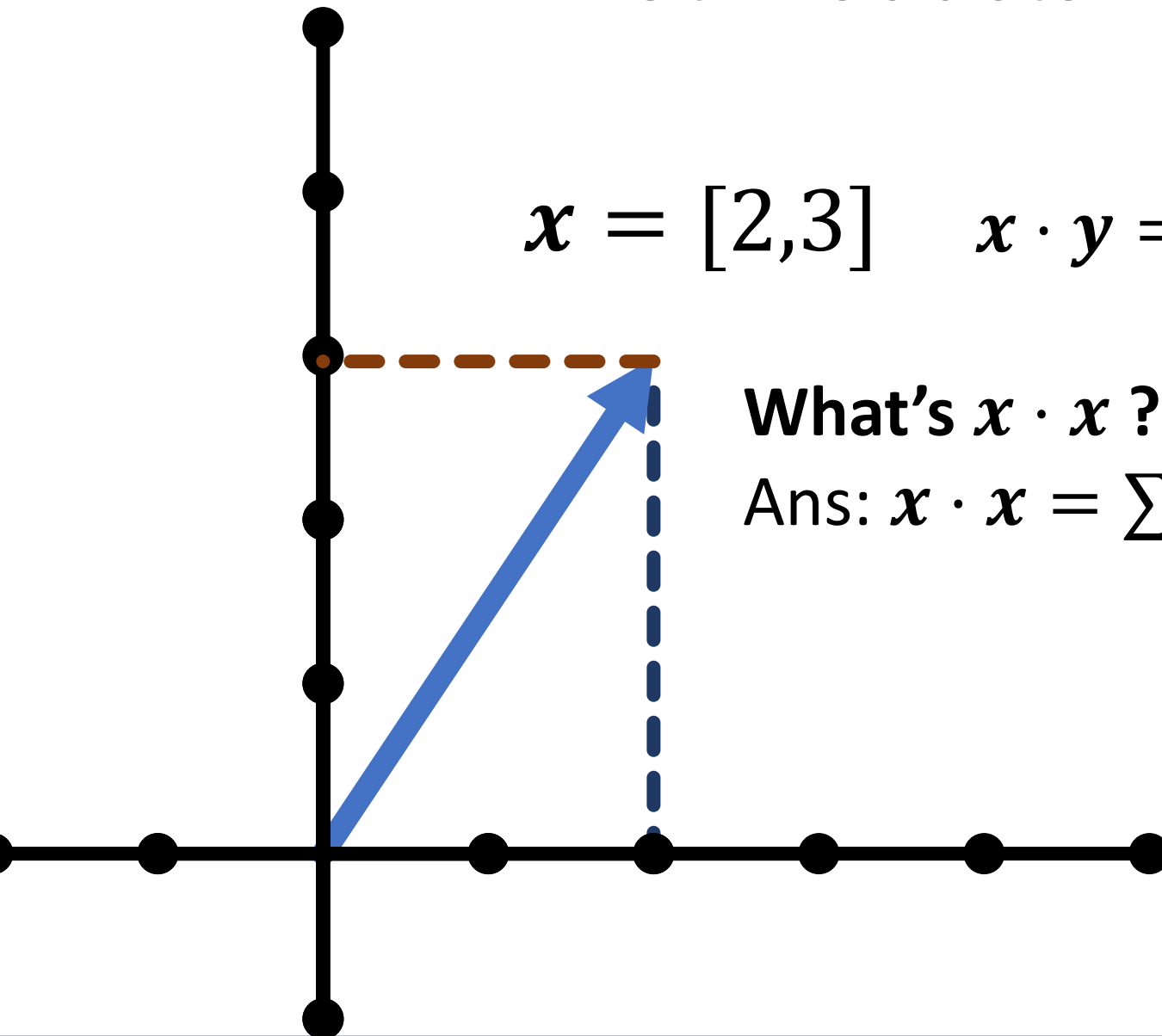


# Dot Products

$$\mathbf{x} = [2, 3] \quad \mathbf{x} \cdot \mathbf{y} = \sum_i^n x_i y_i$$

What's  $\mathbf{x} \cdot \mathbf{x}$  ?

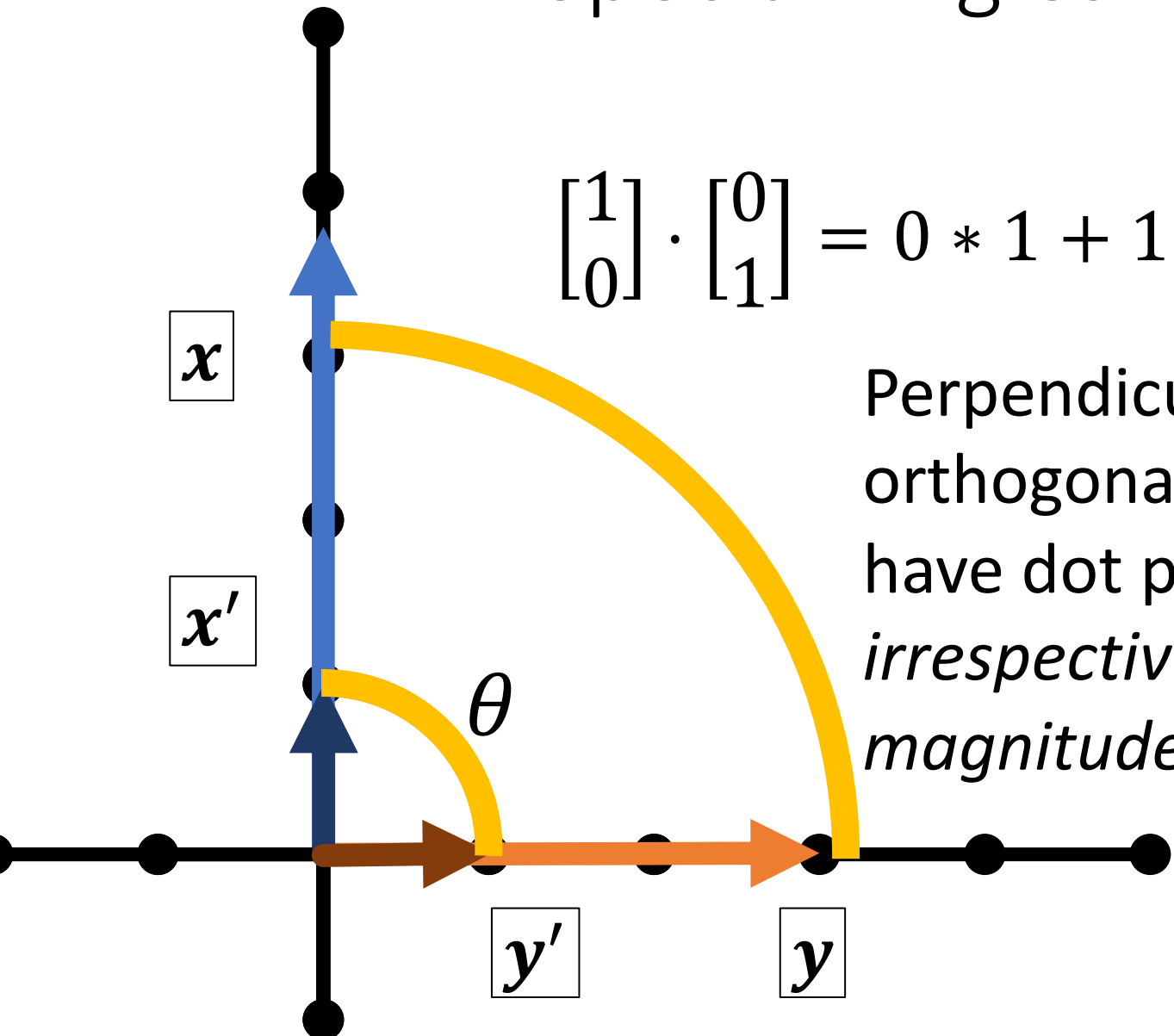
Ans:  $\mathbf{x} \cdot \mathbf{x} = \sum x_i x_i = \|\mathbf{x}\|_2^2$



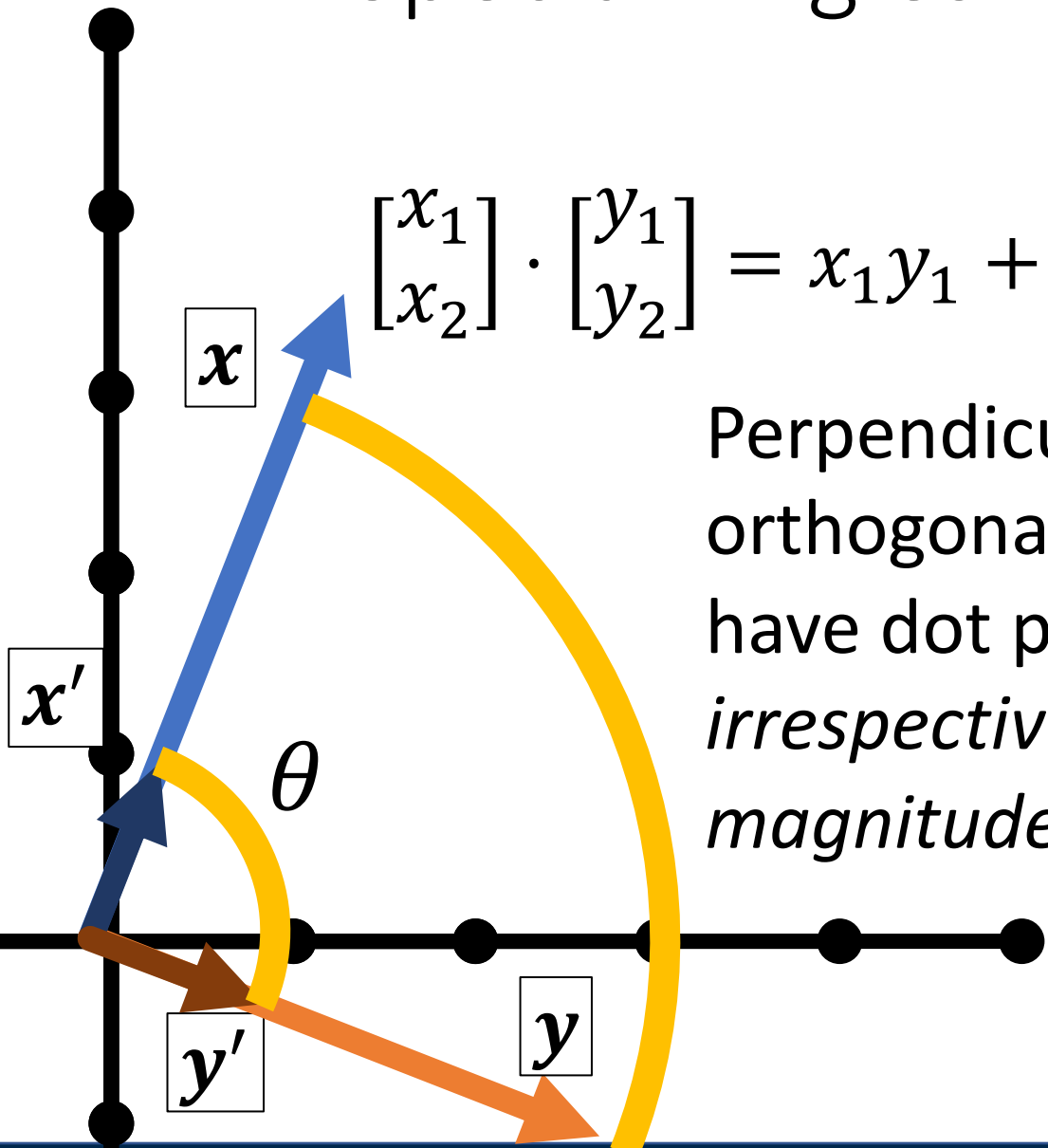
# Special Angles

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} = 0 * 1 + 1 * 0 = 0$$

Perpendicular /  
orthogonal vectors  
have dot product 0  
*irrespective of their  
magnitude*



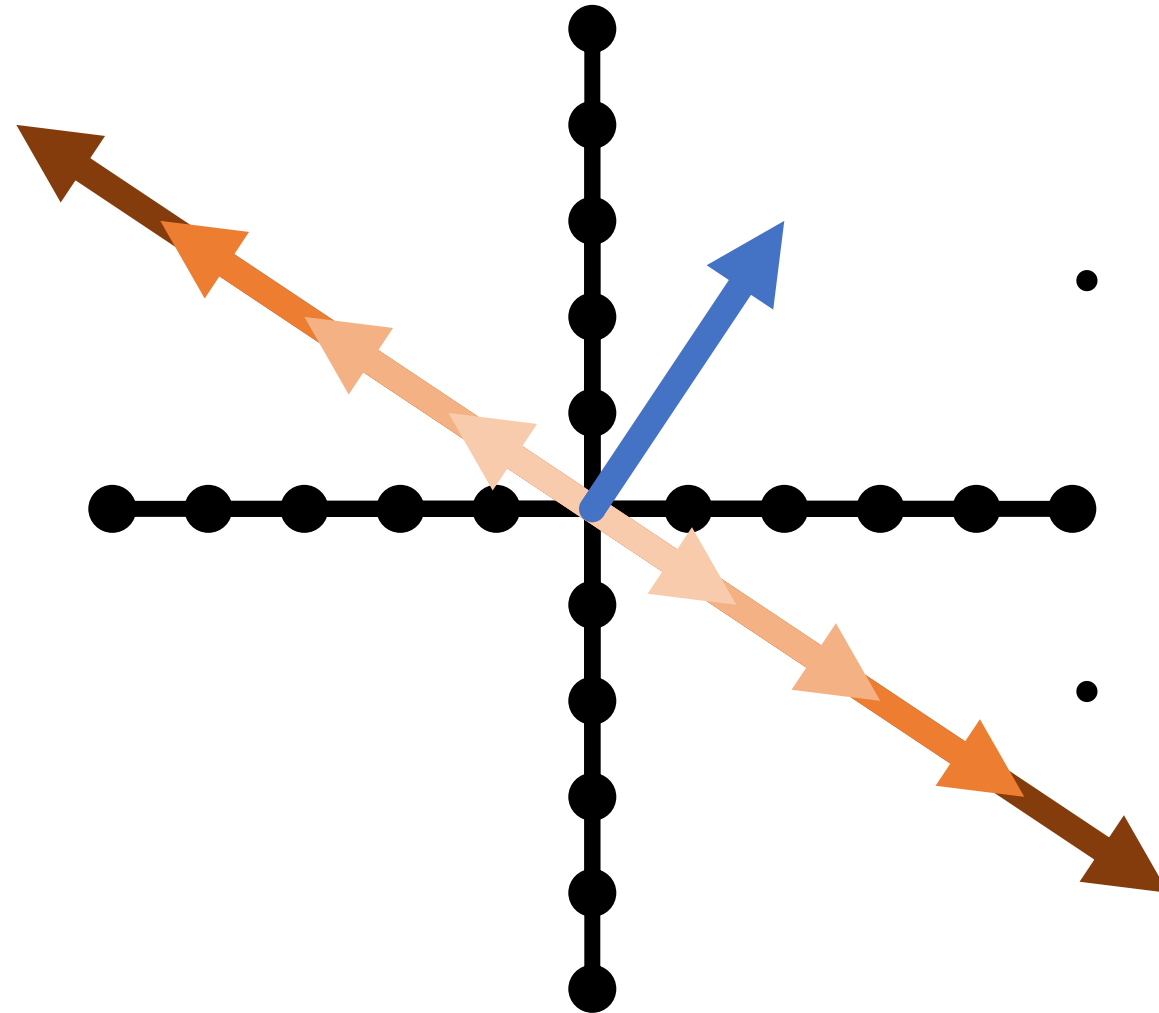
# Special Angles



$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = x_1 y_1 + x_2 y_2 = 0$$

Perpendicular /  
orthogonal vectors  
have dot product 0  
*irrespective of their  
magnitude*

# Orthogonal Vectors

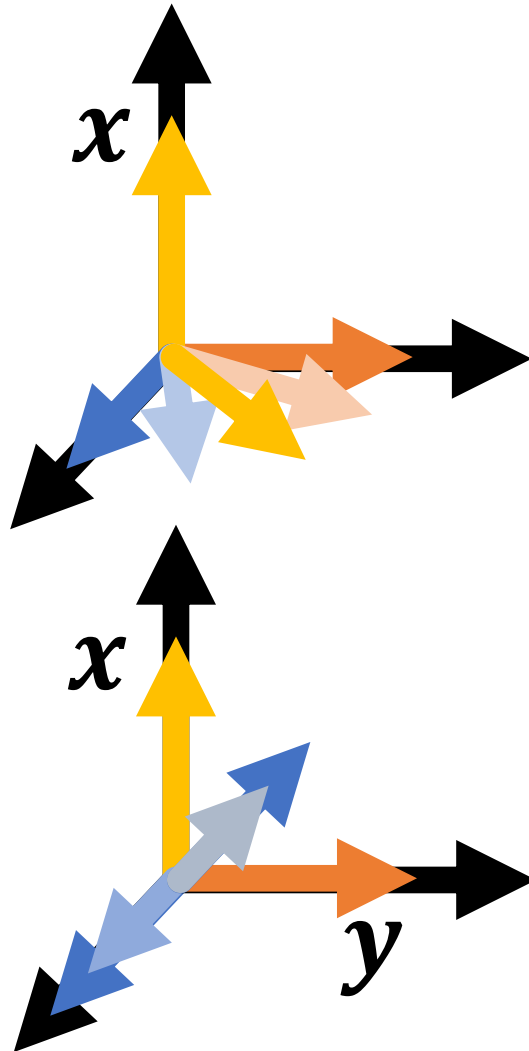


$$x = [2, 3]$$

- Geometrically, what's the set of vectors that are orthogonal to  $x$ ?
- A line  $[3, -2]$

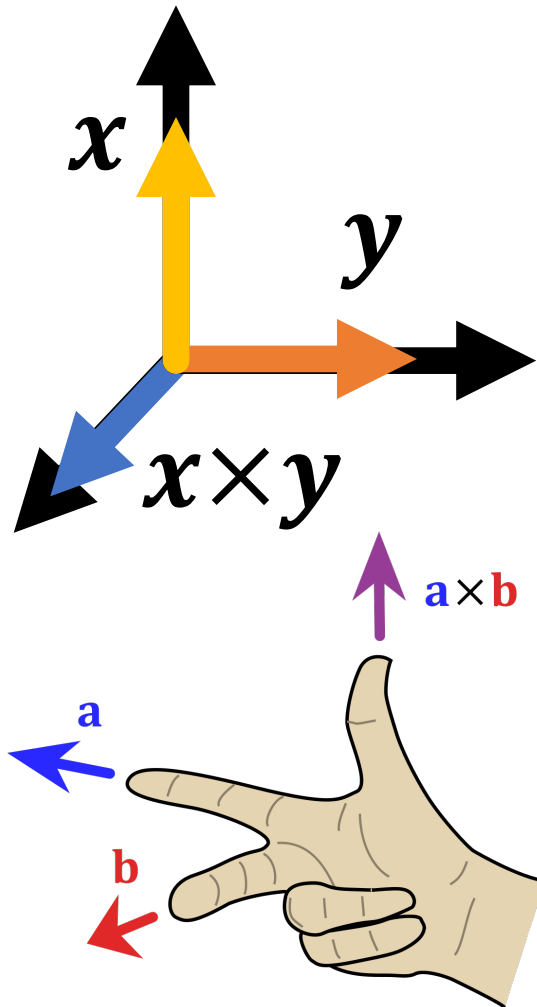


# Orthogonal Vectors



- What's the set of vectors that are orthogonal to  $x = [5,0,0]$ ?
- A plane/2D space of vectors/any vector  $[0, a, b]$
- What's the set of vectors that are orthogonal to  $x$  and  $y = [0,5,0]$ ?
- A line/1D space of vectors/any vector  $[0,0, b]$
- Ambiguity in *sign and magnitude*

# Cross Product



- Set  $\{z: z \cdot x = 0, z \cdot y = 0\}$  has an ambiguity in sign and magnitude
- Cross product  $x \times y$  is: (1) orthogonal to  $x, y$  (2) has sign given by right hand rule and (3) has magnitude given by area of parallelogram of  $x$  and  $y$
- **Important:** if  $x$  and  $y$  are the same direction or either is  $\mathbf{0}$ , then  $x \times y = \mathbf{0}$
- Only in 3D!
- (See wedge product for  $D \neq 3$ )

Image credit: Wikipedia.org

# Matrices

# Things you should know

- A **linear map** between vector spaces is a function satisfying:  $f(av + bw) = af(v) + bf(w)$  for all vectors  $v, w$  and all scalars  $a, b$
- A linear map from  $\mathbb{R}^N$  to  $\mathbb{R}^M$  can be represented by a **matrix** of shape  $M \times N$
- Given a matrix  $A \in \mathbb{R}^{M \times N}$  and a vector  $v \in \mathbb{R}^N$ , the **matrix vector product**  $Av$  is a vector in  $\mathbb{R}^M$  containing the dot products of  $v$  with the rows of  $A$
- Given matrices  $A \in \mathbb{R}^{M \times N}$  and  $B \in \mathbb{R}^{N \times P}$ , the **matrix-matrix product**  $AB$  is a matrix in  $\mathbb{R}^{M \times P}$  containing all dot products of  $A$ 's rows and  $B$ 's columns
- Matrix multiplication is **associative**:  $(AB)C = A(BC)$  but (in general) not **commutative**:  $AB \neq BA$

# Matrices

Horizontally concatenate  $n$ ,  $m$ -dim column vectors and you get a  $m \times n$  matrix  $A$  (here  $2 \times 3$ )

$$A = [\mathbf{v}_1, \dots, \mathbf{v}_n] = \begin{bmatrix} v_{1_1} & v_{2_1} & v_{3_1} \\ v_{1_2} & v_{2_2} & v_{3_2} \end{bmatrix}$$

**a** (scalar)  
lowercase  
undecorated

**a** (vector)  
lowercase  
**bold or arrow**

**A** (matrix)  
uppercase  
**bold**

# Matrices

Horizontally concatenate  $n$ ,  $m$ -dim column vectors and you get a  $m \times n$  matrix  $A$  (here  $2 \times 3$ )

$$A = [\mathbf{v}_1, \dots, \mathbf{v}_n] = \begin{bmatrix} v_{1_1} & v_{2_1} & v_{3_1} \\ v_{1_2} & v_{2_2} & v_{3_2} \end{bmatrix}$$

**Watch out:** In math, it's common to treat  $D$ -dim vector as a  $D \times 1$  matrix (column vector);  
In numpy these are different things

# Matrices

Transpose: flip rows / columns

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix}^T = [a \quad b \quad c] \quad (3 \times 1)^T = 1 \times 3$$

Vertically concatenate  $m$ ,  $n$ -dim row vectors and you get a  $m \times n$  matrix  $A$  (here  $2 \times 3$ )

$$A = \begin{bmatrix} \mathbf{u}_1^T \\ \vdots \\ \mathbf{u}_n^T \end{bmatrix} = \begin{bmatrix} u_{1_1} & u_{1_2} & u_{1_3} \\ u_{2_1} & u_{2_2} & u_{2_3} \end{bmatrix}$$

# Matrix-vector Product

$$\mathbf{y}_{2 \times 1} = \mathbf{A}_{2 \times 3} \mathbf{x}_{3 \times 1}$$
$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \mathbf{v}_1 & \mathbf{v}_2 & \mathbf{v}_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$\mathbf{y} = x_1 \mathbf{v}_1 + x_2 \mathbf{v}_2 + x_3 \mathbf{v}_3$$

*Linear combination of columns of  $\mathbf{A}$*



# Matrix-vector Product

$$\mathbf{y}_{2 \times 1} = \mathbf{A}_{2 \times 3} \mathbf{x}_{3 \times 1}$$

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \mathbf{u}_1^T \\ \mathbf{u}_2^T \end{bmatrix} \mathbf{x}$$

← 3 →

↑ 3 ↓

$$y_1 = \mathbf{u}_1^T \mathbf{x} \quad y_2 = \mathbf{u}_2^T \mathbf{x}$$

*Dot product between rows of  $\mathbf{A}$  and  $\mathbf{x}$*

# Matrix Multiplication

Generally:  $\mathbf{A}_{mn}$  and  $\mathbf{B}_{np}$  yield product  $(\mathbf{AB})_{mp}$

$$\mathbf{AB} = \begin{bmatrix} - & \mathbf{a}_1^T & - \\ & \vdots & \\ - & \mathbf{a}_m^T & - \end{bmatrix} \begin{bmatrix} | & & | \\ \mathbf{b}_1 & \cdots & \mathbf{b}_p \\ | & & | \end{bmatrix}$$

Yes – in  $\mathbf{A}$ , I'm referring to the rows, and in  $\mathbf{B}$ , I'm referring to the columns

# Matrix Multiplication

Generally:  $\mathbf{A}_{mn}$  and  $\mathbf{B}_{np}$  yield product  $(\mathbf{AB})_{mp}$

$$\mathbf{AB}_{ij} = \mathbf{a}_i^T \mathbf{b}_j$$
$$\mathbf{AB} = \begin{bmatrix} \mathbf{a}_1^T \\ \vdots \\ \mathbf{a}_m^T \end{bmatrix} \begin{bmatrix} \mathbf{b}_1 & \cdots & \mathbf{b}_p \end{bmatrix} = \begin{bmatrix} \mathbf{a}_1^T \mathbf{b}_1 & \cdots & \mathbf{a}_1^T \mathbf{b}_p \\ \vdots & \ddots & \vdots \\ \mathbf{a}_m^T \mathbf{b}_1 & \cdots & \mathbf{a}_m^T \mathbf{b}_p \end{bmatrix}$$

# Matrix Multiplication

- Dimensions must match
- Dimensions must match
- Dimensions must match
- (Associative):  $ABx = (A)(Bx) = (AB)x$
- (Not Commutative):  $ABx \neq (BA)x \neq (BxA)$

# Working with Matrices

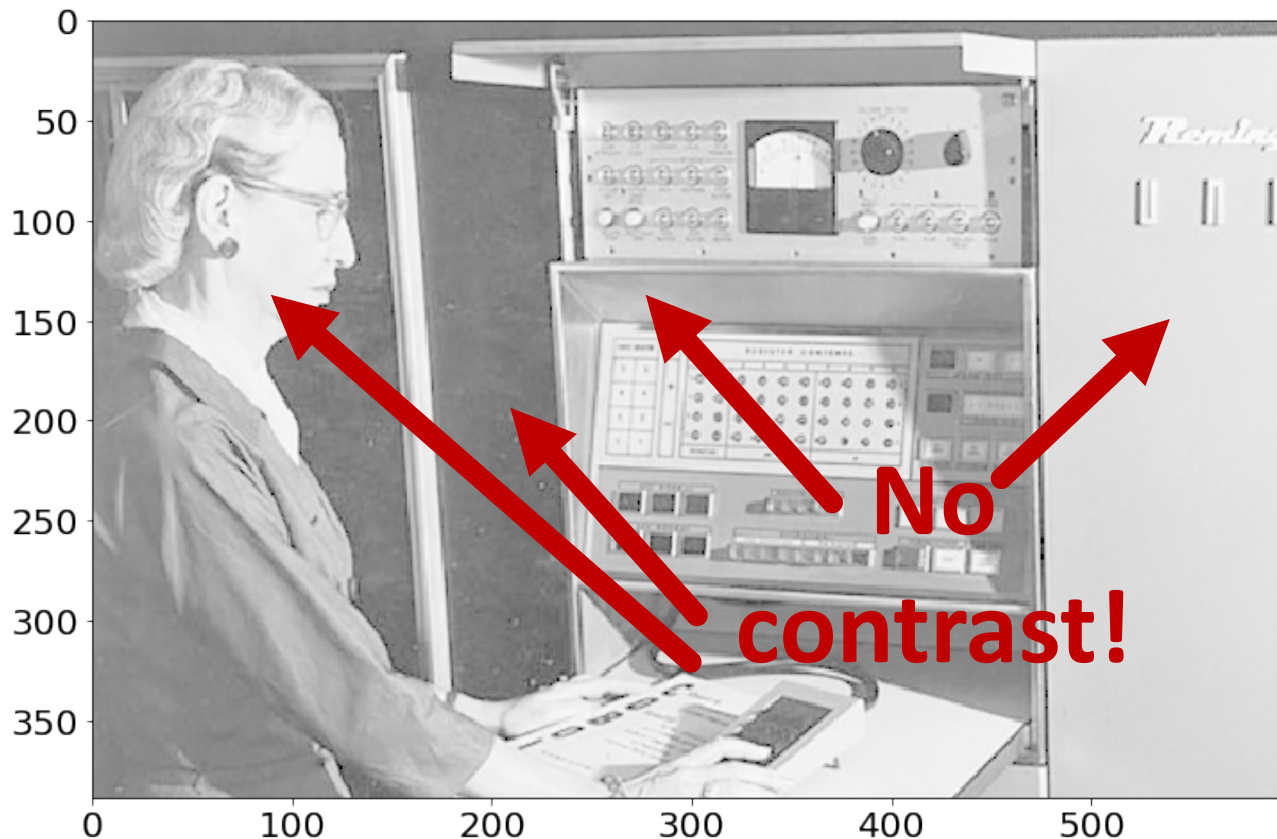
# Two uses for Matrices

1. Storing things in a rectangular array (e.g. images)
  - *Typical operations*: element-wise operations, convolution (which we'll cover later)
  - *Atypical operations*: almost anything you learned in a math linear algebra class
2. A linear operator that maps vectors to another space (**Ax**)
  - *Typical/Atypical*: reverse of above

# Images as Matrices

Suppose someone hands you this matrix.

**What's wrong with it?**

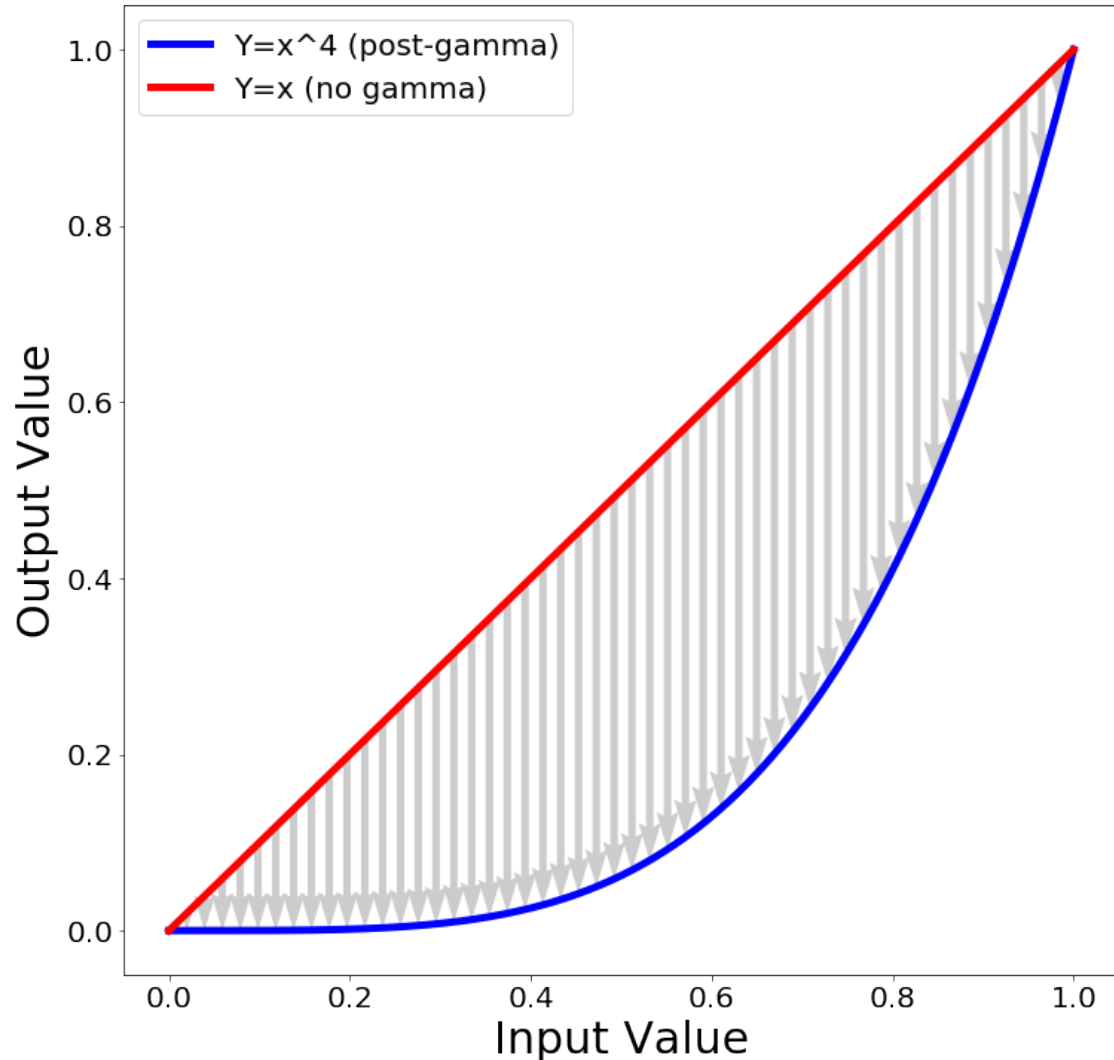


# Contrast: Gamma Curve

Typical way to change the contrast is to apply a nonlinear correction

$$\text{pixelvalue}^\gamma$$

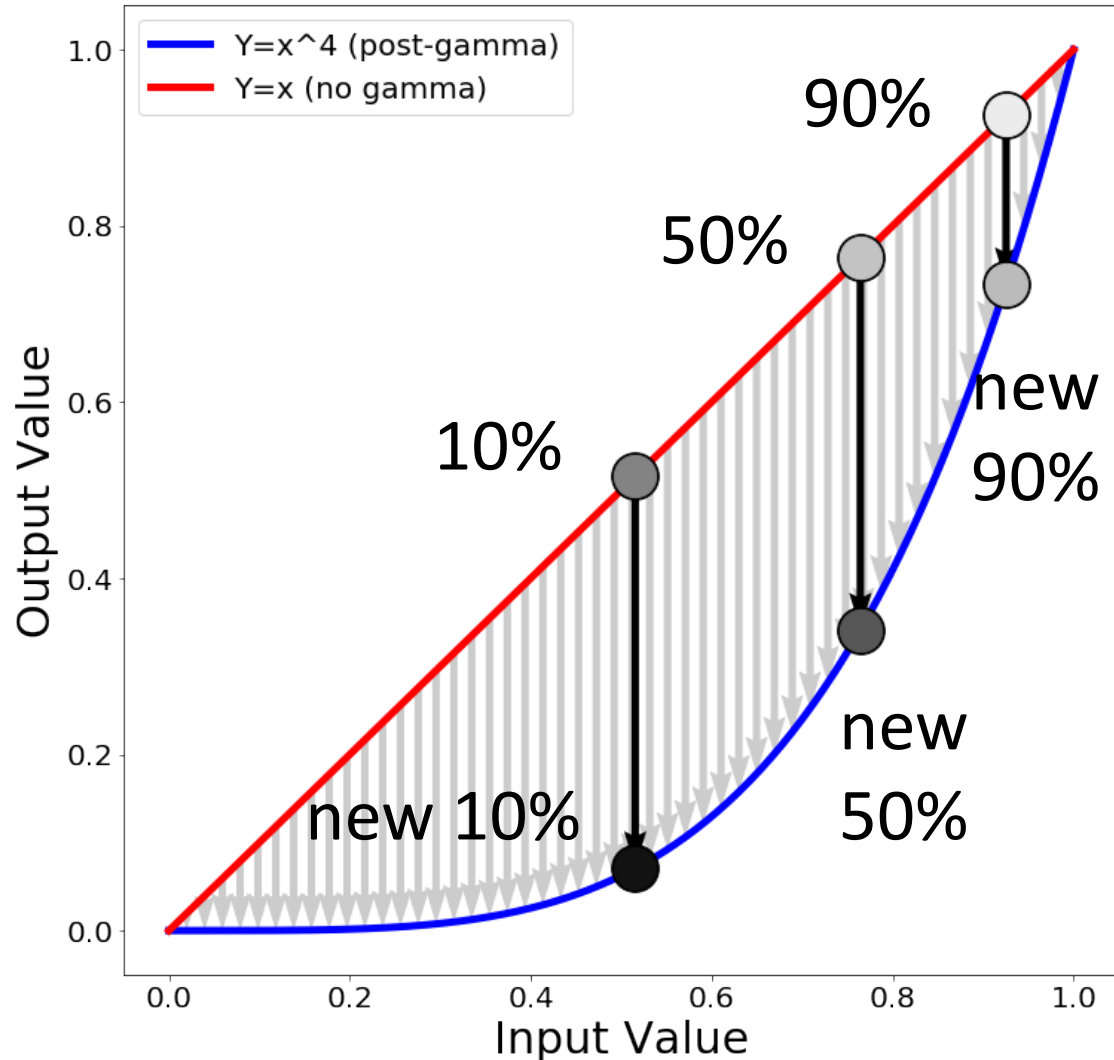
The quantity  $\gamma$  controls how much contrast gets added



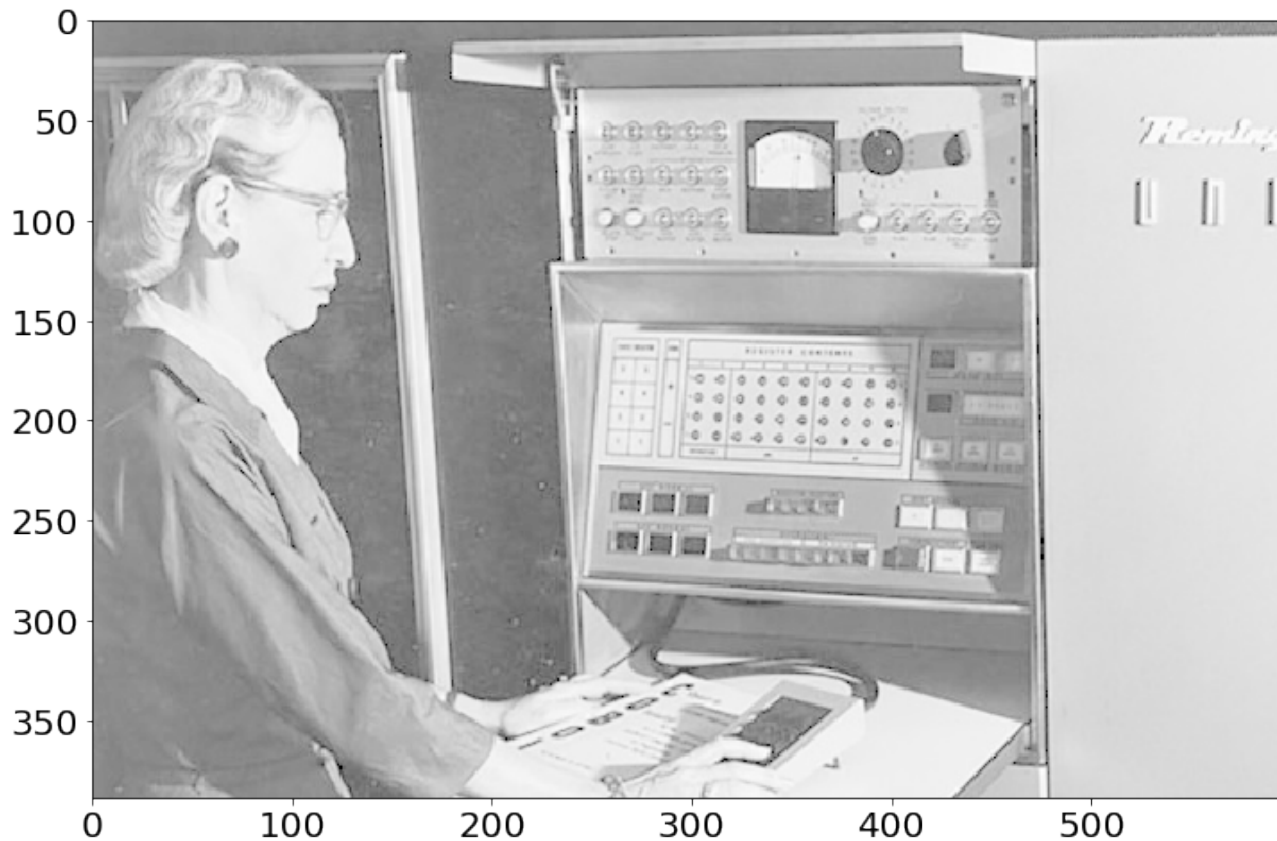


# Contrast: Gamma Curve

Now the darkest regions (10<sup>th</sup> pctile) are **much** darker than the moderately dark regions (50<sup>th</sup> pctile).

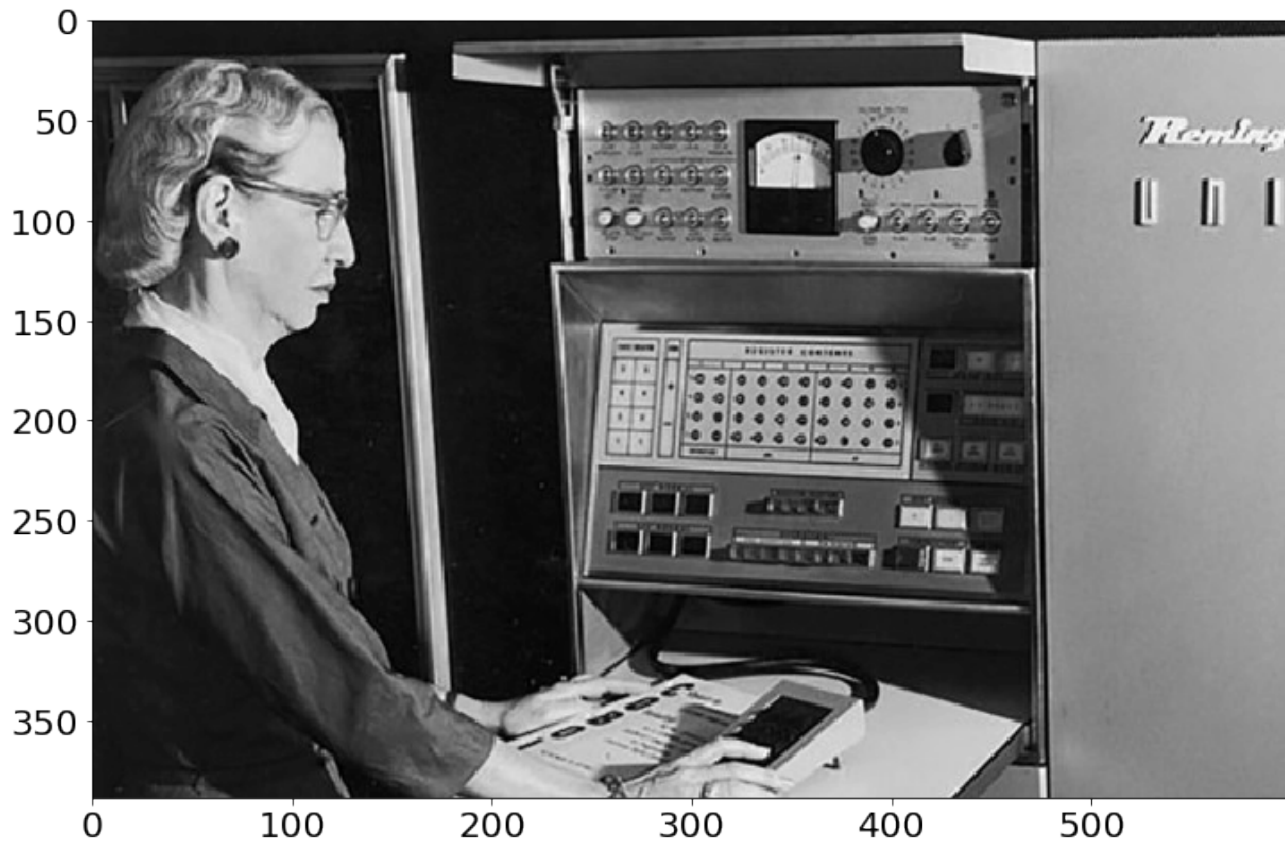


# Contrast: Gamma Correction



# Contrast: Gamma Correction

Phew! Much Better.



# Implementation

Python+Numpy (right way):

```
imNew = im**4
```

Python+Numpy (slow way – **why?**):

```
imNew = np.zeros(im.shape)
for y in range(im.shape[0]):
    for x in range(im.shape[1]):
        imNew[y,x] = im[y,x]**expFactor
```

# Elementwise Operations

Element-wise power – beware notation

$$(\mathbf{A}^p)_{ij} = A_{ij}^p$$

“Hadamard Product” / Element-wise multiplication

$$(\mathbf{A} \odot \mathbf{B})_{ij} = A_{ij} * B_{ij}$$

Element-wise division

$$(\mathbf{A}/\mathbf{B})_{ij} = \frac{A_{ij}}{B_{ij}}$$

# Sums Across Axes

Let  $\mathbf{A}$  be a matrix  
of shape  $(N, 2)$ :

`A = np.random.randn(N, 2)`

$$\mathbf{A} = \begin{bmatrix} x_1 & y_1 \\ \vdots & \vdots \\ x_n & y_n \end{bmatrix}$$

Sum over rows gives  
vector of shape  $(2,)$

`A.sum(axis=0)`

$$\Sigma(\mathbf{A}, 0) = \left[ \sum_{i=1}^n x_i, \sum_{i=1}^n y_i \right]$$

Sum over columns gives  
vector of shape  $(N,)$

`A.sum(axis=1)`

$$\Sigma(\mathbf{A}, 1) = \begin{bmatrix} x_1 + y_1 \\ \vdots \\ x_n + y_n \end{bmatrix}$$

*Note – libraries distinguish between N-D column vector and Nx1 matrix.*

# Operations they don't teach

You Probably Saw Matrix Addition

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} + \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} a + e & b + f \\ c + g & d + h \end{bmatrix}$$

**What is this? FYI:  $e$  is a scalar**

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} + e = \begin{bmatrix} a + e & b + e \\ c + e & d + e \end{bmatrix}$$

# Broadcasting

If you want to be pedantic and proper, you expand  $e$  by multiplying a matrix of 1s (denoted  $\mathbf{1}$ )

$$\begin{aligned} \begin{bmatrix} a & b \\ c & d \end{bmatrix} + e &= \begin{bmatrix} a & b \\ c & d \end{bmatrix} + \mathbf{1}_{2 \times 2} e \\ &= \begin{bmatrix} a & b \\ c & d \end{bmatrix} + \begin{bmatrix} e & e \\ e & e \end{bmatrix} \end{aligned}$$

Many smart matrix libraries do this automatically.  
This is the source of many bugs.



# Broadcasting Example

Given: Matrix  $\mathbf{P}$  of shape  $(N, 2)$  vector  $\mathbf{v}$  of shape  $(2, 1)$

Want: Difference matrix  $\mathbf{D}$  of shape  $(N, 2)$

$$\mathbf{P} = \begin{bmatrix} x_1 & y_1 \\ \vdots & \vdots \\ x_N & y_N \end{bmatrix} \quad \mathbf{v} = \begin{bmatrix} a \\ b \end{bmatrix} \quad \mathbf{D} = \begin{bmatrix} x_1 - a & y_1 - b \\ \vdots & \vdots \\ x_N - a & y_N - b \end{bmatrix}$$

$$\mathbf{P} - \mathbf{v}^T = \begin{bmatrix} x_1 & y_1 \\ \vdots & \vdots \\ x_N & y_N \end{bmatrix} - \begin{bmatrix} a & b \\ \vdots & \vdots \\ a & b \end{bmatrix} \quad \begin{array}{l} \text{Blue stuff is} \\ \text{assumed /} \\ \text{broadcast} \end{array}$$

# Broadcasting Rules

Suppose we have numpy arrays  $x$  and  $y$ .  
How will they broadcast?

1. Write down the **shape** of each array as a tuple of integers:  
For example:  $x: (10,)$   $y: (20, 10)$
2. If they have different numbers of dimensions, **prepend** with ones until they have the same number of dimensions  
For example:  $x: (10,)$   $y: (20, 10)$   $\rightarrow$   $x: (1, 10)$   $y: (20, 10)$
3. Compare each dimension. There are 3 cases:
  - (a) Dimension match. Everything is good
  - (b) Dimensions don't match, but one is =1.  
"Duplicate" the smaller array along that axis to match
  - (c) Dimensions don't match, neither are =1. Error!

# Broadcasting Examples

```
x = np.ones(10, 20)
y = np.ones(20)
z = x + y
print(z.shape)
(10, 20)
```

```
x = np.ones(10, 20)
y = np.ones(10, 1)
z = x + y
print(z.shape)
(10, 20)
```

```
x = np.ones(10, 20)
y = np.ones(10)
z = x + y
print(z.shape)
ERROR
```

```
x = np.ones(1, 20)
y = np.ones(10, 1)
z = x + y
print(z.shape)
(10, 20)
```

# Tensors

**Scalar:** Just one number

**Vector:** 1D list of numbers

**Matrix:** 2D grid of numbers

**Tensor:** N-dimensional grid of numbers  
(Lots of other meanings in math, physics)

# Broadcasting with Tensors

The same broadcasting rules apply to tensors with any number of dimensions!

```
x = np.ones(30)
y = np.ones(20, 1)
z = np.ones(10, 1, 1)
w = x + y + z
print(w.shape)

(10, 20, 30)
```

# Vectorization

Writing code without explicit loops:  
use broadcasting, matrix multiply,  
and other (optimized) numpy  
primitives instead

# Vectorization Example

- Suppose I have two sets of (D-dimensional) vectors  $\{\mathbf{x}_1, \dots, \mathbf{x}_N\}$  and  $\{\mathbf{y}_1, \dots, \mathbf{y}_M\}$  and I want to compute all pairwise distances  $d_{i,j} = \|\mathbf{x}_i - \mathbf{y}_j\|$
- Identity:  $\|\mathbf{x} - \mathbf{y}\|^2 = \|\mathbf{x}\|^2 + \|\mathbf{y}\|^2 - 2\mathbf{x}^T \mathbf{y}$
- Or:  $\|\mathbf{x} - \mathbf{y}\| = (\|\mathbf{x}\|^2 + \|\mathbf{y}\|^2 - 2\mathbf{x}^T \mathbf{y})^{1/2}$

# Vectorization Example

$$\mathbf{X} = \begin{bmatrix} - & \mathbf{x}_1 & - \\ & \vdots & \\ - & \mathbf{x}_N & - \end{bmatrix} \quad \mathbf{Y} = \begin{bmatrix} - & \mathbf{y}_1 & - \\ & \vdots & \\ - & \mathbf{y}_M & - \end{bmatrix} \quad \mathbf{Y}^T = \begin{bmatrix} | & & | \\ \mathbf{y}_1 & \cdots & \mathbf{y}_M \\ | & & | \end{bmatrix}$$

Compute a  $N \times 1$  vector  
of norms  
(can also do  $M \times 1$ )

$$\Sigma(\mathbf{X}^2, \mathbf{1}) = \begin{bmatrix} \|\mathbf{x}_1\|^2 \\ \vdots \\ \|\mathbf{x}_N\|^2 \end{bmatrix}$$

Compute a  $N \times M$  matrix  
of dot products

$$(\mathbf{X}\mathbf{Y}^T)_{ij} = \mathbf{x}_i^T \mathbf{y}_j$$



# Vectorization Example

$$\mathbf{D} = \left( \Sigma(\mathbf{X}^2, \mathbf{1}) + \Sigma(\mathbf{Y}^2, \mathbf{1})^T - 2\mathbf{X}\mathbf{Y}^T \right)^{1/2}$$

$$\begin{bmatrix} \|\mathbf{x}_1\|^2 \\ \vdots \\ \|\mathbf{x}_N\|^2 \end{bmatrix} + [\|\mathbf{y}_1\|^2 \quad \cdots \quad \|\mathbf{y}_M\|^2]$$

$$\begin{bmatrix} \|\mathbf{x}_1\|^2 + \|\mathbf{y}_1\|^2 & \cdots & \|\mathbf{x}_1\|^2 + \|\mathbf{y}_M\|^2 \\ \vdots & \ddots & \vdots \\ \|\mathbf{x}_N\|^2 + \|\mathbf{y}_1\|^2 & \cdots & \|\mathbf{x}_N\|^2 + \|\mathbf{y}_M\|^2 \end{bmatrix}$$

**Why?**

$$(\Sigma(\mathbf{X}^2, \mathbf{1}) + \Sigma(\mathbf{Y}^2, \mathbf{1})^T)_{ij} = \|\mathbf{x}_i\|^2 + \|\mathbf{y}_j\|^2$$

# Vectorization Example

$$\mathbf{D} = \left( \Sigma(\mathbf{X}^2, 1) + \Sigma(\mathbf{Y}^2, 1)^T - 2\mathbf{X}\mathbf{Y}^T \right)^{1/2}$$

$$\mathbf{D}_{ij} = \|\mathbf{x}_i\|^2 + \|\mathbf{y}_j\|^2 + 2\mathbf{x}_i^T \mathbf{y}_j$$

Numpy code:

```
XNorm = np.sum(X**2, axis=1, keepdims=True)
YNorm = np.sum(Y**2, axis=1, keepdims=True)
D = (XNorm+YNorm.T-2*np.dot(X, Y.T))**0.5
```

Get in the habit of thinking about shapes as tuples.

Suppose X is (N, D), Y is (M, D):

# Vectorization Example

$$\mathbf{D} = \left( \Sigma(\mathbf{X}^2, 1) + \Sigma(\mathbf{Y}^2, 1)^T - 2\mathbf{X}\mathbf{Y}^T \right)^{1/2}$$

$$\mathbf{D}_{ij} = \|\mathbf{x}_i\|^2 + \|\mathbf{y}_j\|^2 + 2\mathbf{x}_i^T \mathbf{y}_j$$

Numpy code:  $(N, 1)$

```
XNorm = np.sum(X**2, axis=1, keepdims=True)
```

```
YNorm = np.sum(Y**2, axis=1, keepdims=True)
```

```
D = (XNorm+YNorm.T-2*np.dot(X, Y.T))**0.5
```

Get in the habit of thinking about shapes as tuples.

Suppose X is (N, D), Y is (M, D):

# Vectorization Example

$$\mathbf{D} = \left( \Sigma(\mathbf{X}^2, 1) + \Sigma(\mathbf{Y}^2, 1)^T - 2\mathbf{X}\mathbf{Y}^T \right)^{1/2}$$

$$\mathbf{D}_{ij} = \|\mathbf{x}_i\|^2 + \|\mathbf{y}_j\|^2 + 2\mathbf{x}_i^T \mathbf{y}_j$$

Numpy code:  $(N, 1)$   $(M, 1)$

```
XNorm = np.sum(X**2, axis=1, keepdims=True)
```

```
YNorm = np.sum(Y**2, axis=1, keepdims=True)
```

```
D = (XNorm+YNorm.T-2*np.dot(X, Y.T))**0.5
```

Get in the habit of thinking about shapes as tuples.

Suppose X is (N, D), Y is (M, D):

# Vectorization Example

$$\mathbf{D} = \left( \Sigma(\mathbf{X}^2, 1) + \Sigma(\mathbf{Y}^2, 1)^T - 2\mathbf{X}\mathbf{Y}^T \right)^{1/2}$$

$$\mathbf{D}_{ij} = \|\mathbf{x}_i\|^2 + \|\mathbf{y}_j\|^2 + 2\mathbf{x}_i^T \mathbf{y}_j$$

Numpy code: (N, 1) (M, 1) (N, M)

```
XNorm = np.sum(X**2, axis=1, keepdims=True)
```

```
YNorm = np.sum(Y**2, axis=1, keepdims=True)
```

```
D = (XNorm+YNorm.T-2*np.dot(X, Y.T))**0.5
```

Get in the habit of thinking about shapes as tuples.

Suppose X is (N, D), Y is (M, D):

# Vectorization Example

$$\mathbf{D} = \left( \Sigma(\mathbf{X}^2, 1) + \Sigma(\mathbf{Y}^2, 1)^T - 2\mathbf{X}\mathbf{Y}^T \right)^{1/2}$$

$$\mathbf{D}_{ij} = \|\mathbf{x}_i\|^2 + \|\mathbf{y}_j\|^2 + 2\mathbf{x}_i^T \mathbf{y}_j$$

Numpy code:  $(N, 1)$   $(M, 1)$   $(N, M)$   $(N, M)$

```
XNorm = np.sum(X**2, axis=1, keepdims=True)
```

```
YNorm = np.sum(Y**2, axis=1, keepdims=True)
```

```
D = (XNorm+YNorm.T-2*np.dot(X, Y.T))**0.5
```

Get in the habit of thinking about shapes as tuples.

Suppose X is (N, D), Y is (M, D):

# Vectorization Example

$$\mathbf{D} = \left( \Sigma(\mathbf{X}^2, 1) + \Sigma(\mathbf{Y}^2, 1)^T - 2\mathbf{X}\mathbf{Y}^T \right)^{1/2}$$

$$\mathbf{D}_{ij} = \|\mathbf{x}_i\|^2 + \|\mathbf{y}_j\|^2 + 2\mathbf{x}_i^T \mathbf{y}_j$$

Numpy code: (N, 1) (M, 1) (N, M) (N, M)

```
XNorm = np.sum(X**2, axis=1, keepdims=True)
```

```
YNorm = np.sum(Y**2, axis=1, keepdims=True)
```

```
D = (XNorm + YNorm.T - 2*np.dot(X, Y.T))**0.5
```

Get in the habit of thinking about shapes as tuples.

Suppose X is (N, D), Y is (M, D):

# Vectorization Example

$$\mathbf{D} = \left( \Sigma(\mathbf{X}^2, 1) + \Sigma(\mathbf{Y}^2, 1)^T - 2\mathbf{X}\mathbf{Y}^T \right)^{1/2}$$

$$\mathbf{D}_{ij} = \|\mathbf{x}_i\|^2 + \|\mathbf{y}_j\|^2 + 2\mathbf{x}_i^T \mathbf{y}_j$$

Numpy code: (N, 1) (M, 1) (N, M) (N, M)

```
XNorm = np.sum(X**2, axis=1, keepdims=True)
```

```
YNorm = np.sum(Y**2, axis=1, keepdims=True)
```

```
D = (XNorm+YNorm.T-2*np.dot(X, Y.T))**0.5
```

Get in the habit of thinking about shapes as tuples.

Suppose X is (N, D), Y is (M, D):



# Vectorization Example

$$\mathbf{D} = \left( \Sigma(\mathbf{X}^2, 1) + \Sigma(\mathbf{Y}^2, 1)^T - 2\mathbf{X}\mathbf{Y}^T \right)^{1/2}$$

$$\mathbf{D}_{ij} = \|\mathbf{x}_i\|^2 + \|\mathbf{y}_j\|^2 + 2\mathbf{x}_i^T \mathbf{y}_j$$

Numpy code:

```
XNorm = np.sum(X**2, axis=1, keepdims=True)
YNorm = np.sum(Y**2, axis=1, keepdims=True)
D = (XNorm + YNorm.T - 2*np.dot(X, Y.T))**0.5
```

\*May have to make sure this is at least 0 (sometimes roundoff issues happen)

# Does Vectorization Matter?

Computing pairwise distances between 300 and 400  
128-dimensional vectors

1. for x in X, for y in Y, using native python: 9s
2. for x in X, for y in Y, using numpy to compute distance: 0.8s
3. vectorized: 0.0045s (~2000x faster than 1, 175x faster than 2)

*Expressing things in primitives that are optimized is usually faster*

*Even more important with special hardware like GPUs or TPUs!*

# Linear Algebra

# Things you should know:

- A set of vectors are **linearly independent** if you can't write one as a linear combination of the others
- The **rank** of a matrix is the number of linearly independent columns (or rows)
- An  $N \times N$  matrix with rank  $N$  is **nonsingular** and behaves nicely: has an inverse, spans the full output space
- A **symmetric matrix** is its own transpose:  $A^T = A$
- A **rotation matrix** has its transpose as its inverse:  $RR^T = R^T R = I$

# Linear Independence

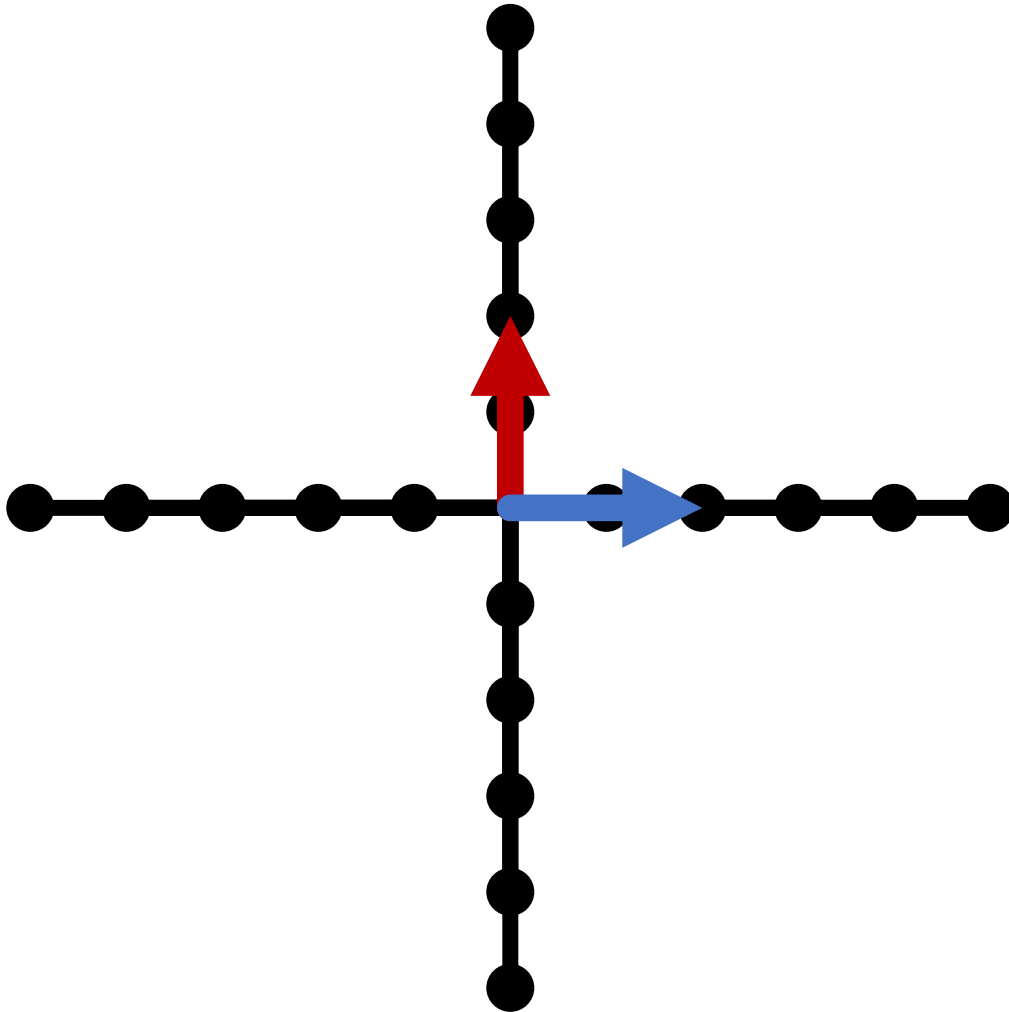
A set of vectors is **linearly independent** if you can't write one as a linear combination of the others.

**Suppose:**  $a = \begin{bmatrix} 0 \\ 0 \\ 2 \end{bmatrix}$   $b = \begin{bmatrix} 0 \\ 6 \\ 0 \end{bmatrix}$   $c = \begin{bmatrix} 5 \\ 0 \\ 0 \end{bmatrix}$

$$x = \begin{bmatrix} 0 \\ 0 \\ 4 \end{bmatrix} = \quad y = \begin{bmatrix} 0 \\ -2 \\ 1 \end{bmatrix} = \frac{1}{2}a - \frac{1}{3}b$$

- Is the set  $\{a,b,c\}$  linearly independent?
- Is the set  $\{a,b,x\}$  linearly independent?
- Max # of independent 3D vectors?

# Span



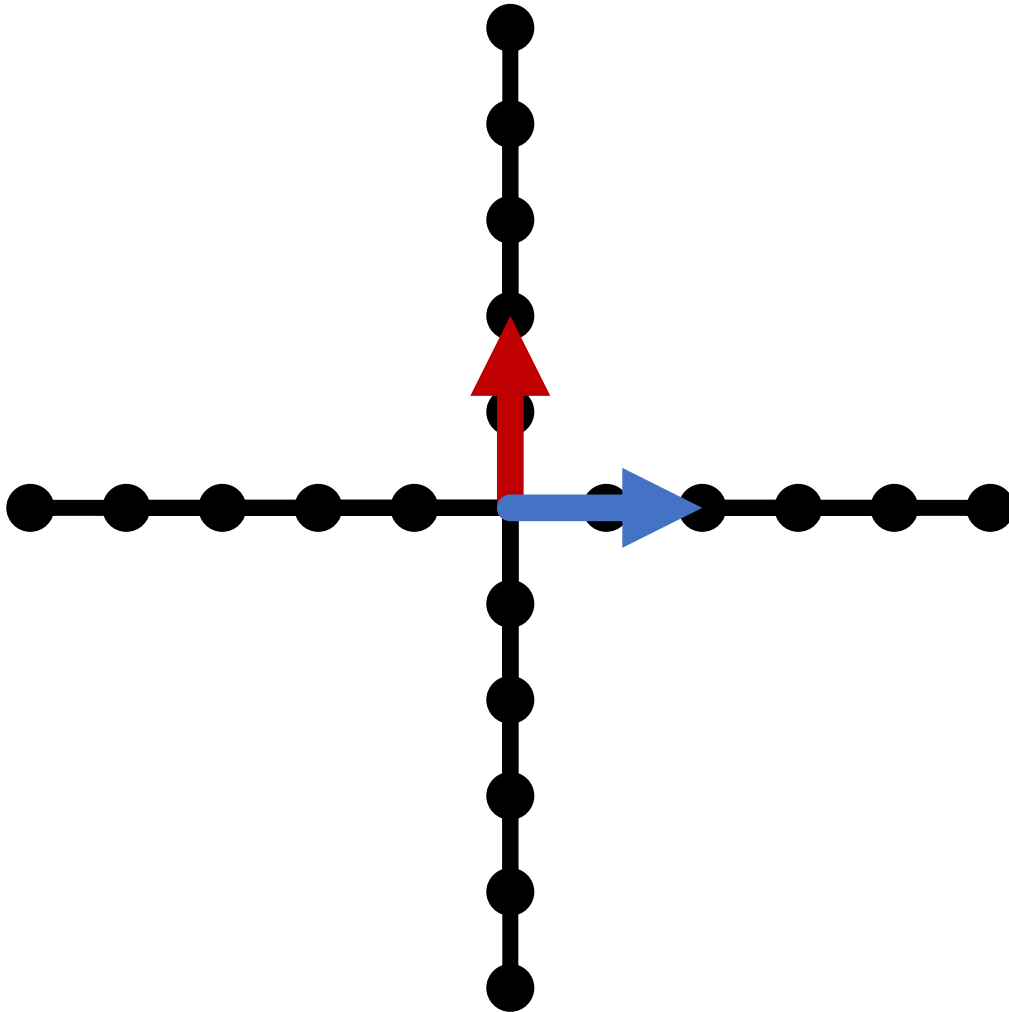
Span: all linear combinations of a set of vectors

$\text{Span}(\{\uparrow\}) =$   
 $\text{Span}(\{[0,2]\}) = ?$

All vertical lines through origin =  
 $\{\lambda[0,1]: \lambda \in R\}$

Is **blue** in **{red}**'s span?

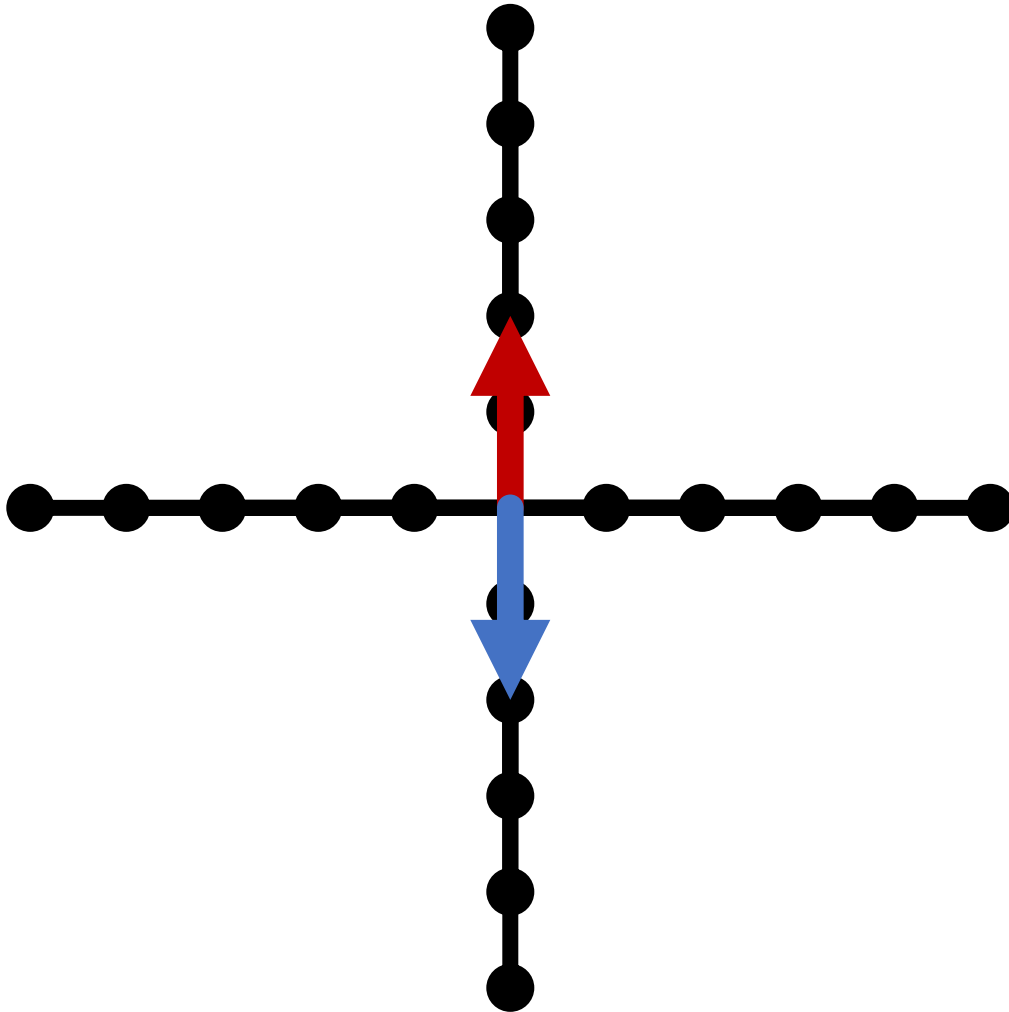
# Span



Span: all linear combinations of a set of vectors

$$\text{Span}(\{\uparrow, \rightarrow\}) = ?$$

# Span



Span: all linear combinations of a set of vectors

$$\text{Span}(\{\uparrow, \downarrow\}) = ?$$



# Matrix-Vector Product

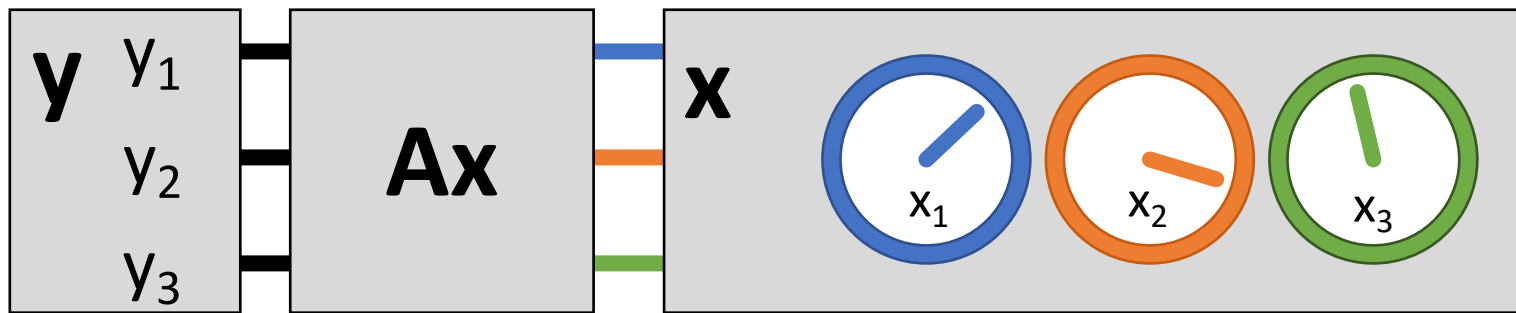
$$\mathbf{Ax} = \begin{bmatrix} | & & | \\ \mathbf{c}_1 & \cdots & \mathbf{c}_n \\ | & & | \end{bmatrix} \mathbf{x}$$

Right-multiplying  $\mathbf{A}$  by  $\mathbf{x}$   
mixes columns of  $\mathbf{A}$   
according to entries of  $\mathbf{x}$

- The output space of  $f(\mathbf{x}) = \mathbf{Ax}$  is constrained to be the *span* of the columns of  $\mathbf{A}$ .
- Can't output things you can't construct out of your columns

# An Intuition

$$\mathbf{y} = \mathbf{A}\mathbf{x} = \begin{bmatrix} | & | & | \\ \mathbf{c}_1 & \mathbf{c}_2 & \mathbf{c}_n \\ | & | & | \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$



$\mathbf{x}$  – knobs on machine (e.g., fuel, brakes)

$\mathbf{y}$  – state of the world (e.g., where you are)

$\mathbf{A}$  – machine (e.g., your car)

# Linear Independence

Suppose the columns of 3x3 matrix  $\mathbf{A}$  are *not* linearly independent ( $c_1, \alpha c_1, c_2$  for instance)

$$\mathbf{y} = \mathbf{A}\mathbf{x} = \begin{bmatrix} | & | & | \\ \mathbf{c}_1 & \alpha\mathbf{c}_1 & \mathbf{c}_2 \\ | & | & | \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

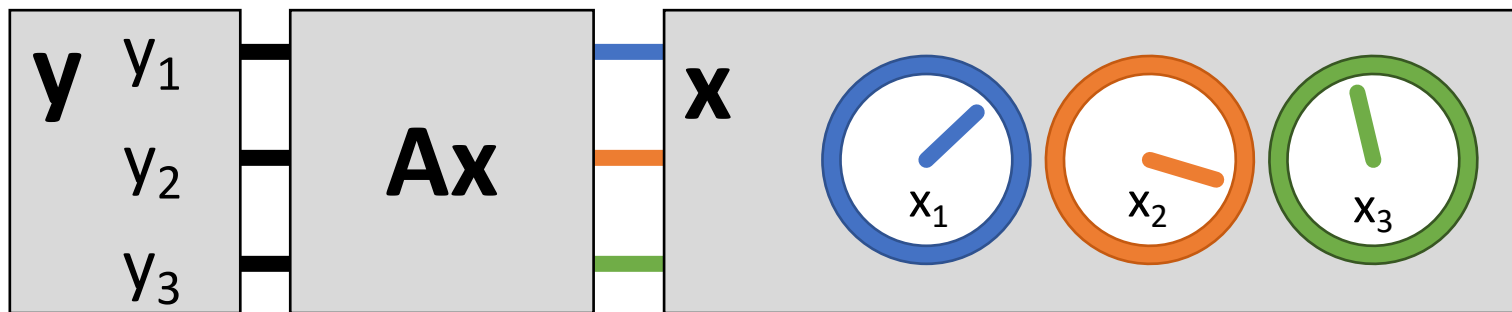
$$\mathbf{y} = x_1\mathbf{c}_1 + \alpha x_2\mathbf{c}_1 + x_3\mathbf{c}_2$$

$$\mathbf{y} = (x_1 + \alpha x_2)\mathbf{c}_1 + x_3\mathbf{c}_2$$

# Linear Independence Intuition

Knobs of  $\mathbf{x}$  are redundant. Even if  $\mathbf{y}$  has 3 outputs, you can only control it in two directions

$$\mathbf{y} = (x_1 + \alpha x_2)\mathbf{c}_1 + x_3\mathbf{c}_2$$



# Linear Independence

Recall:  $\mathbf{Ax} = (x_1 + \alpha x_2)\mathbf{c}_1 + x_3\mathbf{c}_2$

$$\mathbf{y} = \mathbf{A} \begin{bmatrix} x_1 + \beta \\ x_2 - \beta/\alpha \\ x_3 \end{bmatrix} = \left( x_1 + \cancel{\beta} + \alpha x_2 - \alpha \cancel{\frac{\beta}{\alpha}} \right) \mathbf{c}_1 + x_3 \mathbf{c}_2$$

- Can write  $\mathbf{y}$  an infinite number of ways by adding  $\beta$  to  $\mathbf{x}_1$  and subtracting  $\frac{\beta}{\alpha}$  from  $\mathbf{x}_2$
- Or, given a vector  $\mathbf{y}$  there's not a unique vector  $\mathbf{x}$  s.t.  $\mathbf{y} = \mathbf{Ax}$
- Not all  $\mathbf{y}$  have a corresponding  $\mathbf{x}$  s.t.  $\mathbf{y} = \mathbf{Ax}$  (assuming  $\mathbf{c}_1$  and  $\mathbf{c}_2$  have dimension  $\geq 3$ )

# Linear Independence

$$A\mathbf{x} = (x_1 + \alpha x_2)\mathbf{c}_1 + x_3\mathbf{c}_2$$

$$\mathbf{y} = A \begin{bmatrix} \beta \\ -\beta/\alpha \\ 0 \end{bmatrix} = \left( \beta - \alpha \frac{\beta}{\alpha} \right) \mathbf{c}_1 + 0\mathbf{c}_2$$

- What else can we cancel out?
- An infinite number of non-zero vectors  $\mathbf{x}$  can map to a zero-vector  $\mathbf{y}$
- Called the **right null-space** of  $A$ .

# Rank

- Rank of a  $n \times n$  matrix  $\mathbf{A}$  – number of linearly independent columns (**or rows**) of  $\mathbf{A}$  / the dimension of the span of the columns
- Matrices with *full rank* ( $n \times n$ , rank  $n$ ) behave nicely: can be inverted, span the full output space, are one-to-one.
- Matrices with *full rank* are machines where every knob is useful and every output state can be made by the machine

# Matrix Inverses

- Given  $\mathbf{y} = \mathbf{A}\mathbf{x}$ ,  $\mathbf{y}$  is a linear combination of columns of  $\mathbf{A}$  proportional to  $\mathbf{x}$ . If  $\mathbf{A}$  is full-rank, we should be able to invert this mapping.
- Given some  $\mathbf{y}$  (output) and  $\mathbf{A}$ , what  $\mathbf{x}$  (inputs) produced it?
- $\mathbf{x} = \mathbf{A}^{-1}\mathbf{y}$
- Note: if you don't need to compute it, **never ever compute it**. Solving for  $\mathbf{x}$  is much faster and stable than obtaining  $\mathbf{A}^{-1}$ .

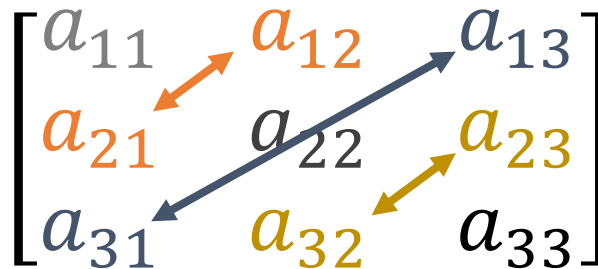
**Bad:** `y = np.linalg.inv(A).dot(y)`

**Good:** `y = np.linalg.solve(A, y)`



# Symmetric Matrices

- Symmetric:  $A^T = A$  or  $A_{ij} = A_{ji}$
- Have **lots** of special properties



A 3x3 matrix is shown with elements  $a_{11}, a_{12}, a_{13}$  in the first row,  $a_{21}, a_{22}, a_{23}$  in the second row, and  $a_{31}, a_{32}, a_{33}$  in the third row. A blue diagonal arrow points from  $a_{11}$  to  $a_{33}$ . An orange arrow points from  $a_{12}$  to  $a_{21}$ . A yellow arrow points from  $a_{23}$  to  $a_{32}$ .

Any matrix of the form  $A = X^T X$  is symmetric.

Quick check:

$$A^T = (X^T X)^T$$
$$A^T = X^T (X^T)^T$$
$$A^T = X^T X$$

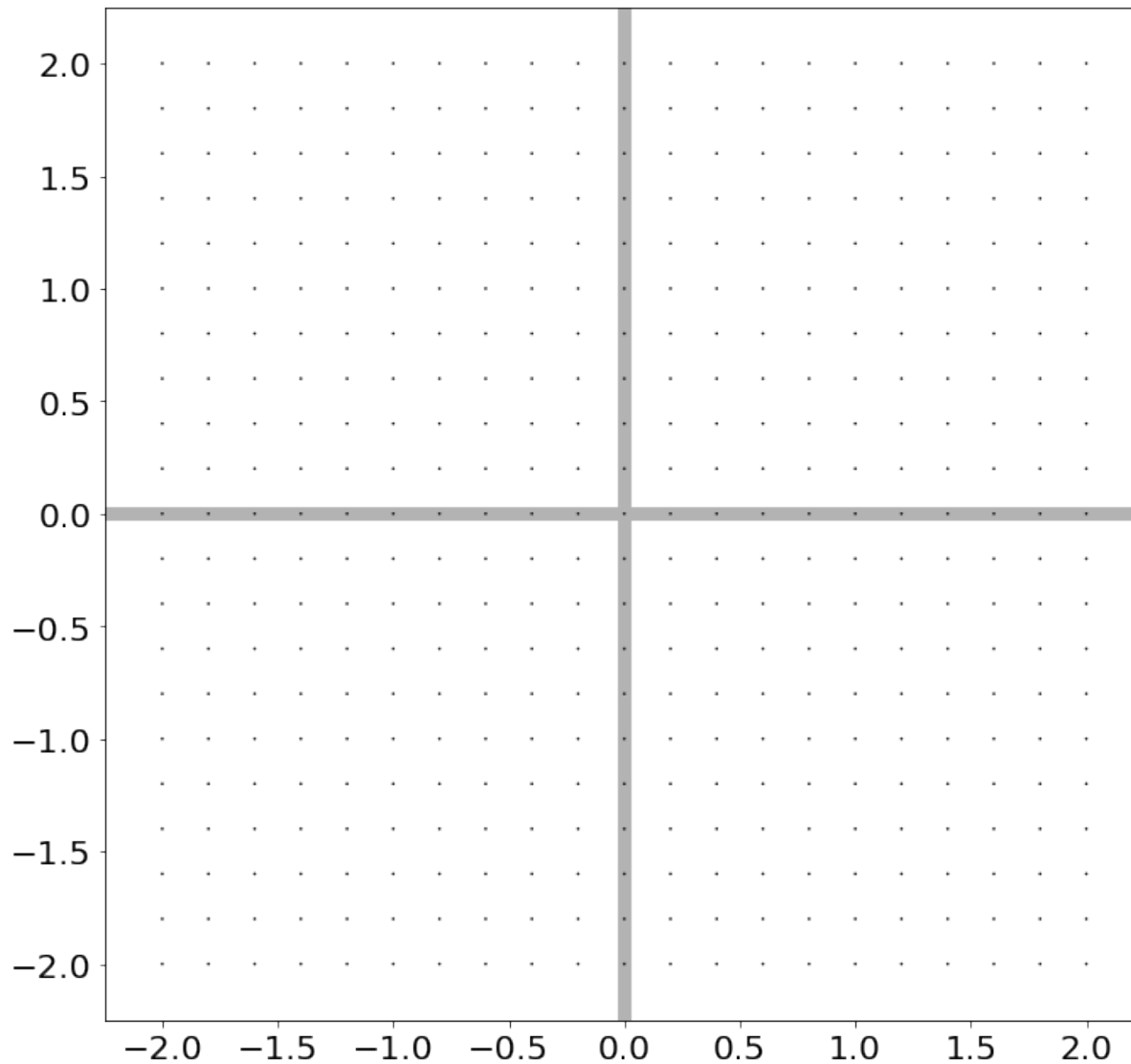
# Special Matrices: Rotations

$$\begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

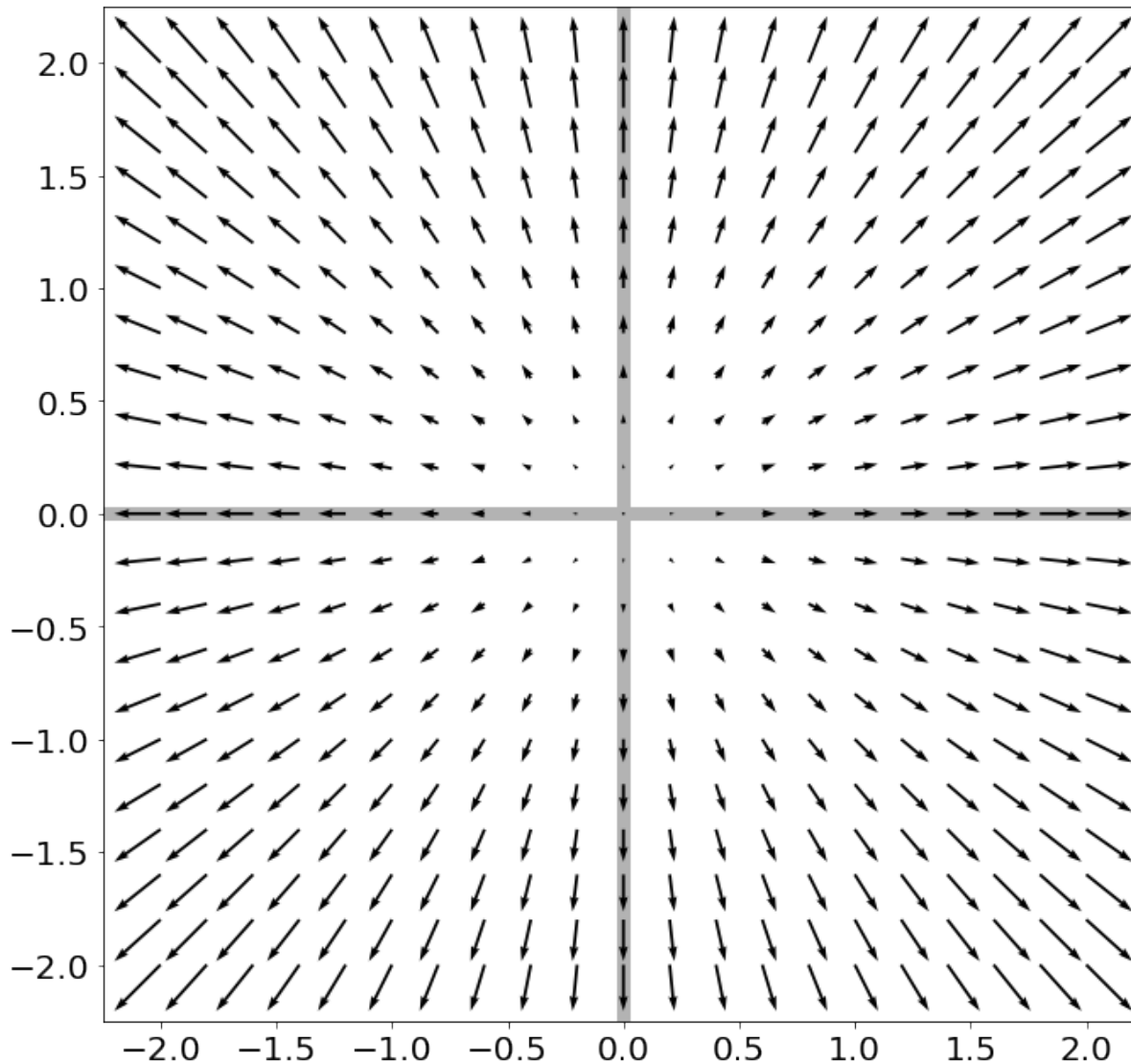
- Rotation matrices  $\mathbf{R}$  rotate vectors and **do not change vector L2 norms** ( $\|\mathbf{R}\mathbf{x}\|_2 = \|\mathbf{x}\|_2$ )
- Every row/column is unit norm
- Every row is linearly independent
- Transpose is inverse  $\mathbf{R}\mathbf{R}^T = \mathbf{R}^T\mathbf{R} = \mathbf{I}$
- Determinant is 1 (otherwise it's also a coordinate flip/reflection), eigenvalues are 1

# Eigensystems

- An eigenvector  $\mathbf{v}_i$  and eigenvalue  $\lambda_i$  of a matrix  $\mathbf{A}$  satisfy  $\mathbf{A}\mathbf{v}_i = \lambda_i\mathbf{v}_i$  ( $\mathbf{A}\mathbf{v}_i$  is scaled by  $\lambda_i$ )
- Vectors and values are always paired and typically you assume  $\|\mathbf{v}_i\|^2 = 1$
- Biggest eigenvalue of  $\mathbf{A}$  gives bounds on how much  $f(\mathbf{x}) = \mathbf{A}\mathbf{x}$  stretches a vector  $\mathbf{x}$ .
- Hints of what people really mean:
  - “Largest eigenvector” = vector w/ largest value
  - “Spectral” just means there’s eigenvectors somewhere

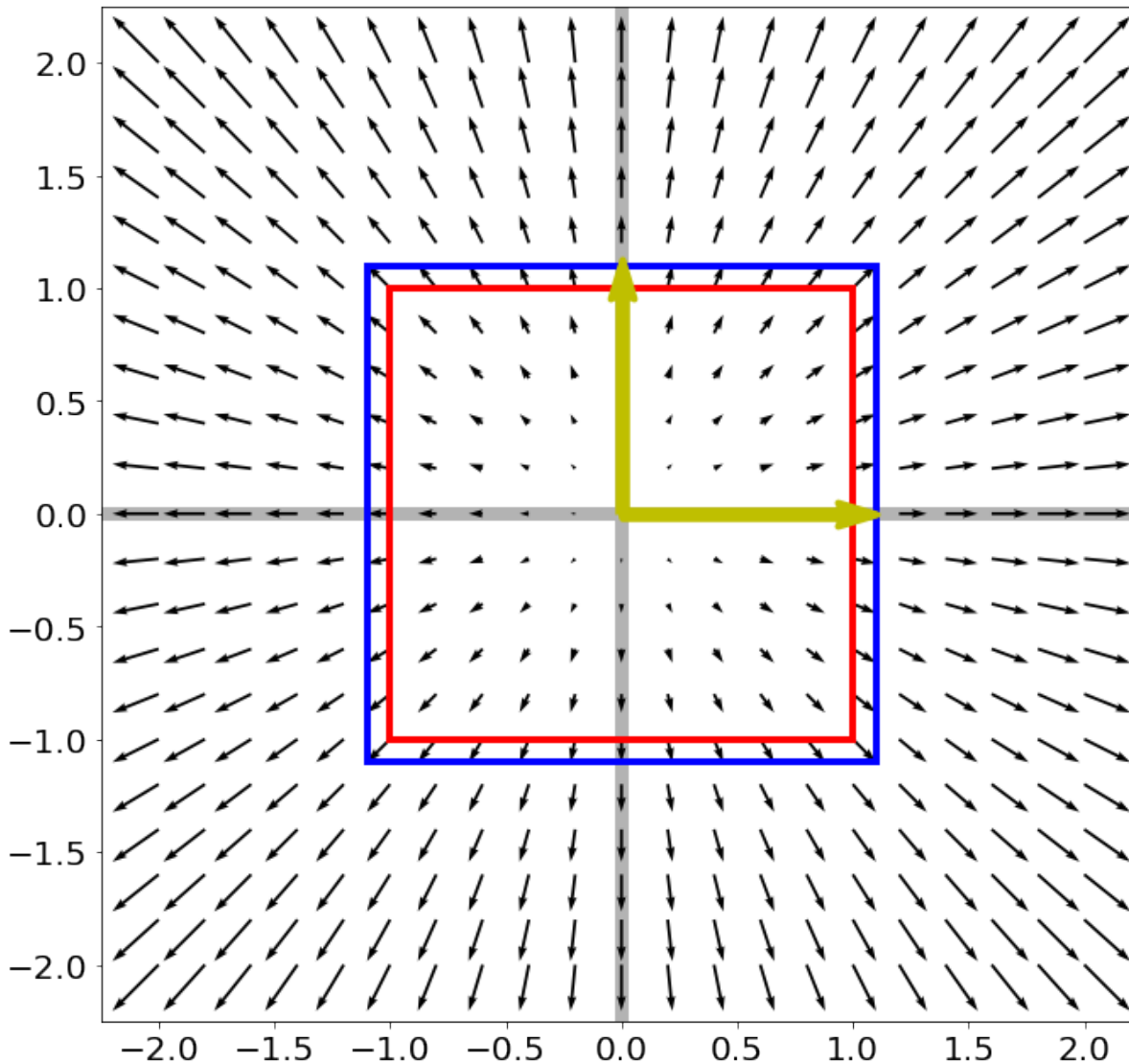


Suppose I have points in a grid



Now I apply  $f(\mathbf{x}) = \mathbf{Ax}$  to these points  
 Pointy-end:  $\mathbf{Ax}$  . Non-Pointy-End:  $\mathbf{x}$

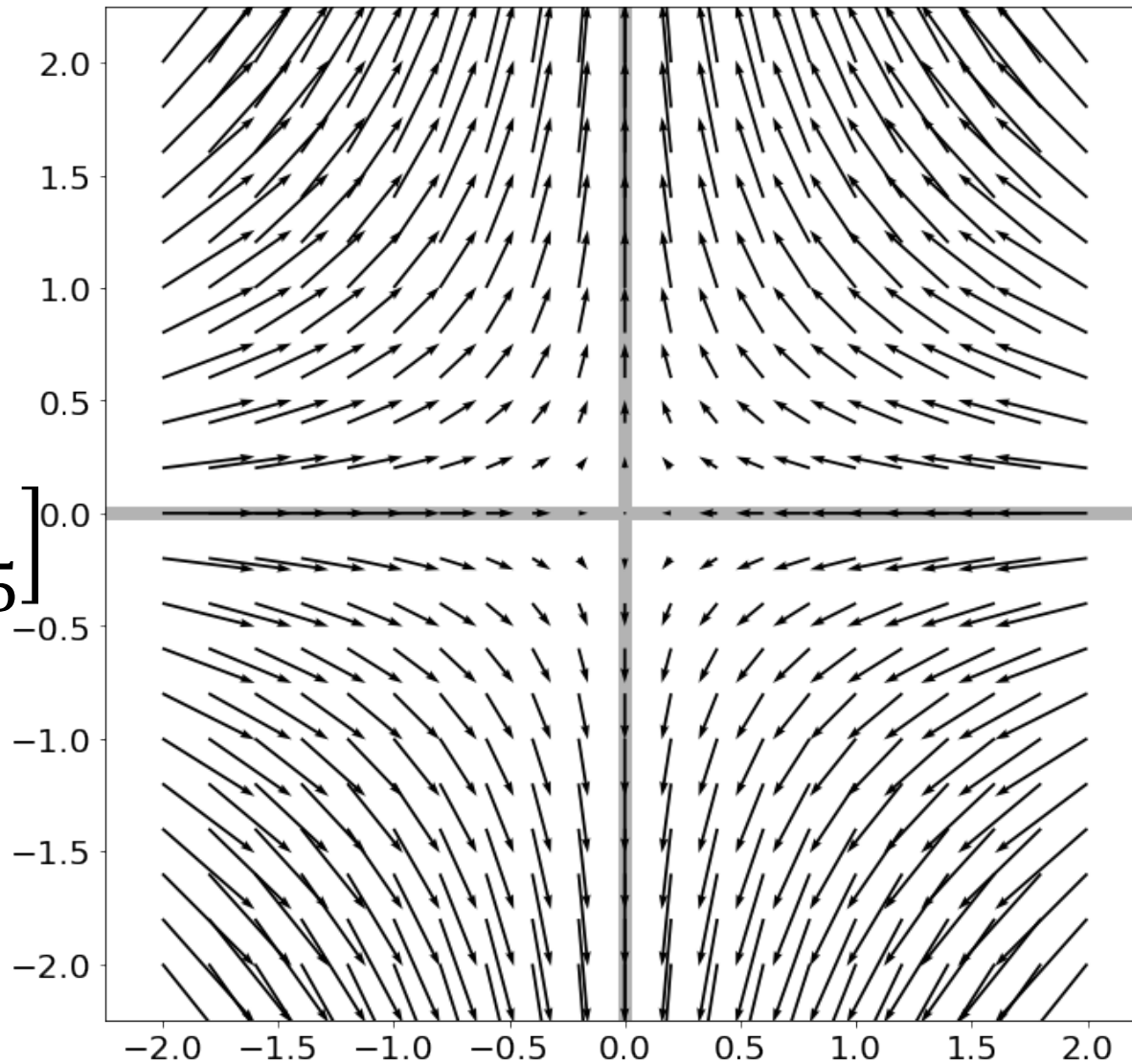
$$\mathbf{A} = \begin{bmatrix} 1.1 & 0 \\ 0 & 1.1 \end{bmatrix}$$



Red box – unit square, Blue box – after  $f(\mathbf{x}) = \mathbf{A}\mathbf{x}$ .

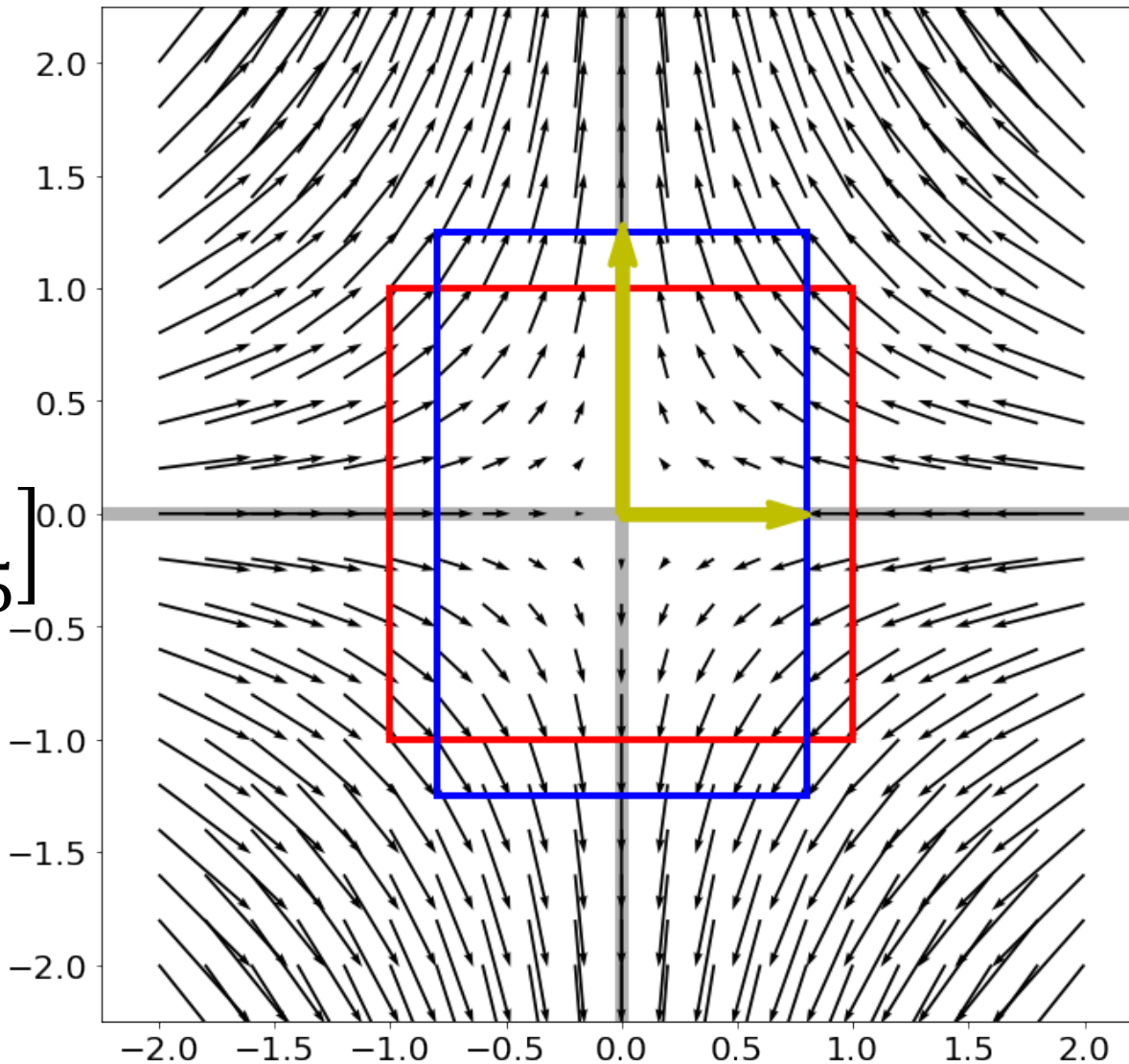
**What are the yellow lines and why?**

$$\mathbf{A} = \begin{bmatrix} 0.8 & 0 \\ 0 & 1.25 \end{bmatrix}$$



Now I apply  $f(\mathbf{x}) = \mathbf{A}\mathbf{x}$  to these points  
 Pointy-end:  $\mathbf{A}\mathbf{x}$  . Non-Pointy-End:  $\mathbf{x}$

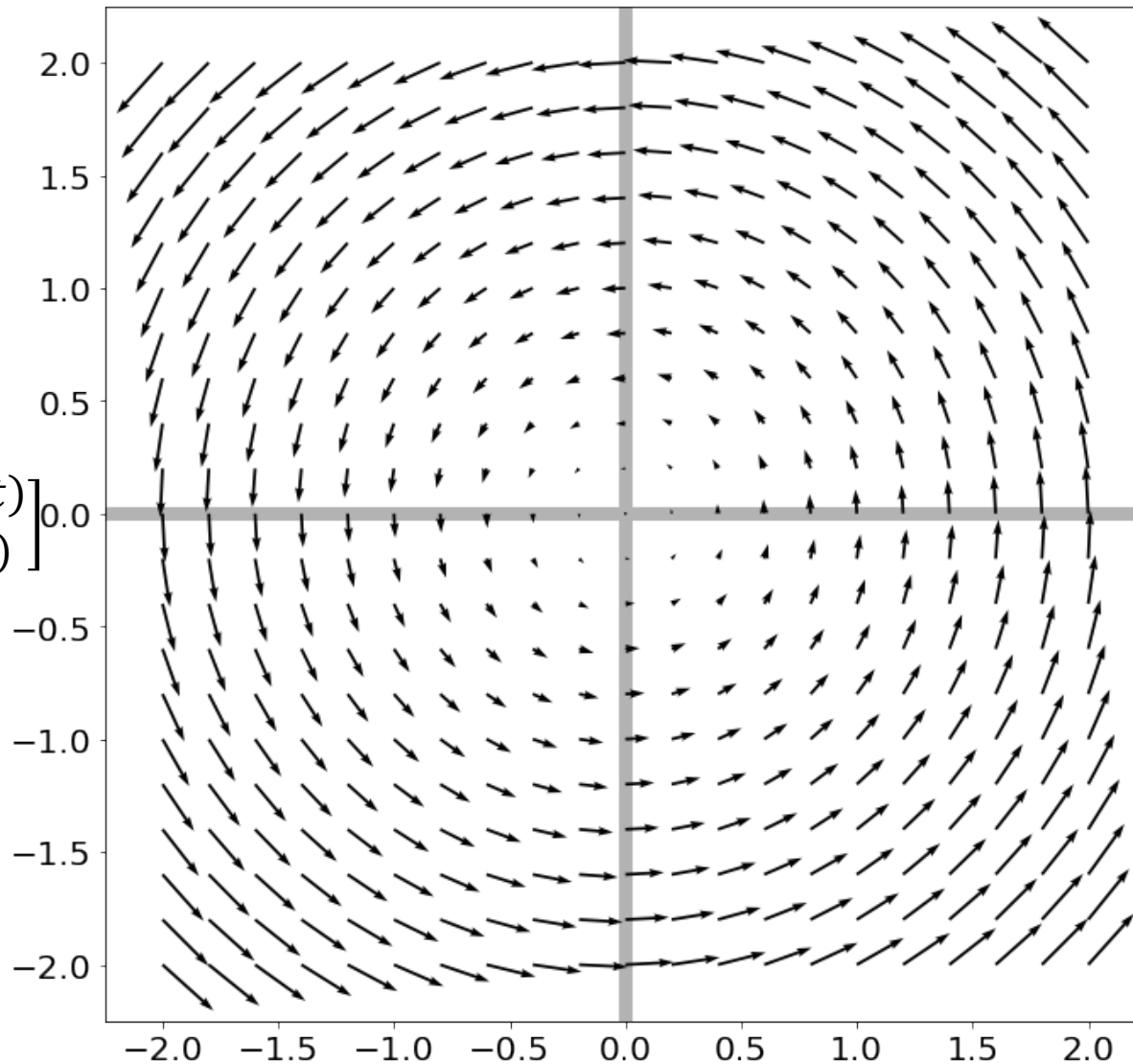
$$\mathbf{A} = \begin{bmatrix} 0.8 & 0 \\ 0 & 1.25 \end{bmatrix}$$



Red box – unit square, Blue box – after  $f(\mathbf{x}) = \mathbf{A}\mathbf{x}$ .  
**What are the yellow lines and why?**



$$\mathbf{A} = \begin{bmatrix} \cos(t) & -\sin(t) \\ \sin(t) & \cos(t) \end{bmatrix}$$



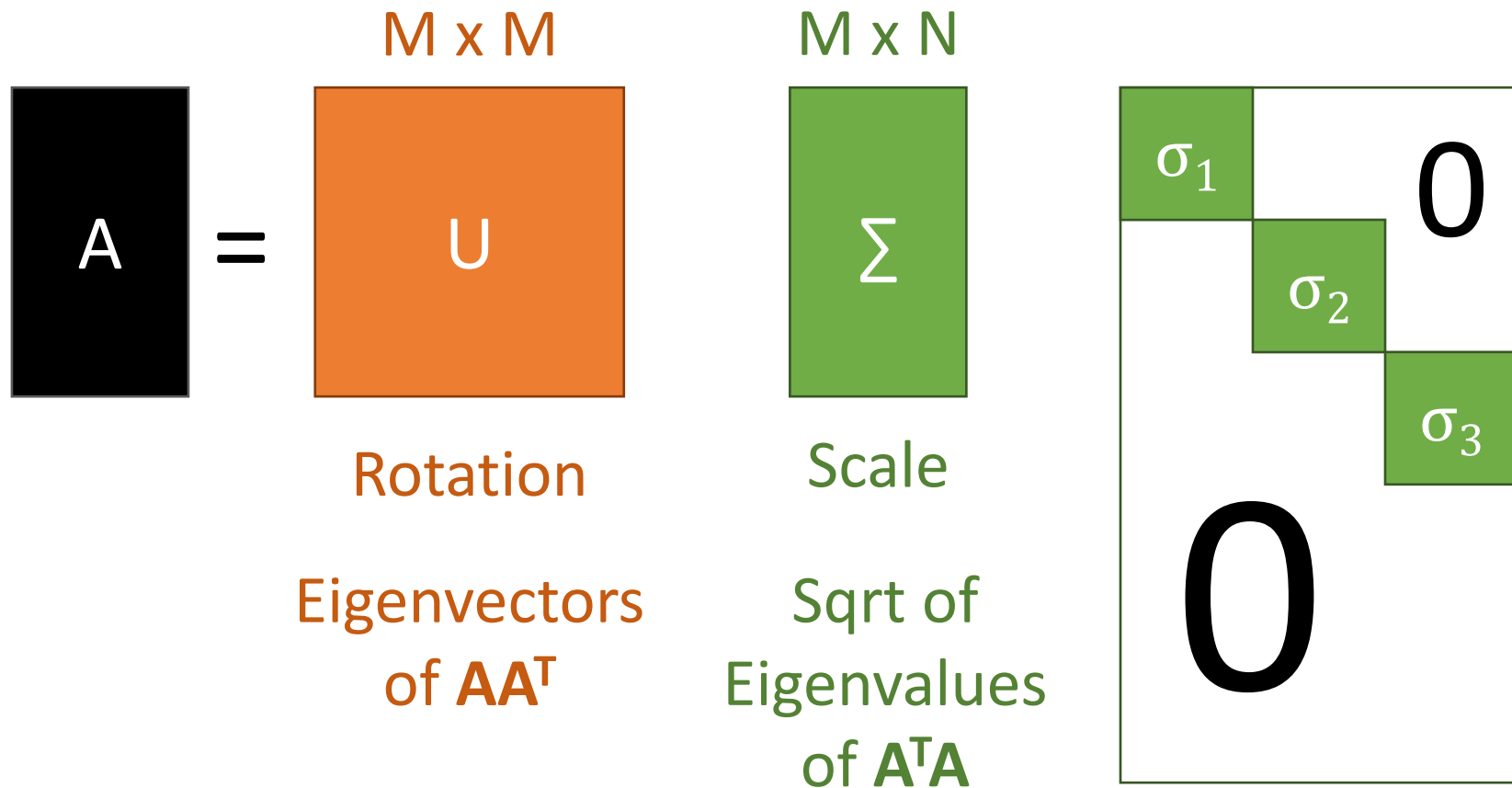
Red box – unit square, Blue box – after  $f(\mathbf{x}) = \mathbf{A}\mathbf{x}$ .  
**Can we draw any yellow lines?**

# Eigenvectors of Symmetric Matrices

- Always  $n$  mutually orthogonal eigenvectors with  $n$  (not necessarily) distinct eigenvalues
- For symmetric  $A$ , the eigenvector with the largest eigenvalue maximizes  $\frac{x^T A x}{x^T x}$  (smallest/min)
- So for unit vectors (where  $x^T x = 1$ ), that eigenvector maximizes  $x^T A x$
- A surprisingly large number of optimization problems rely on (max/min)imizing this

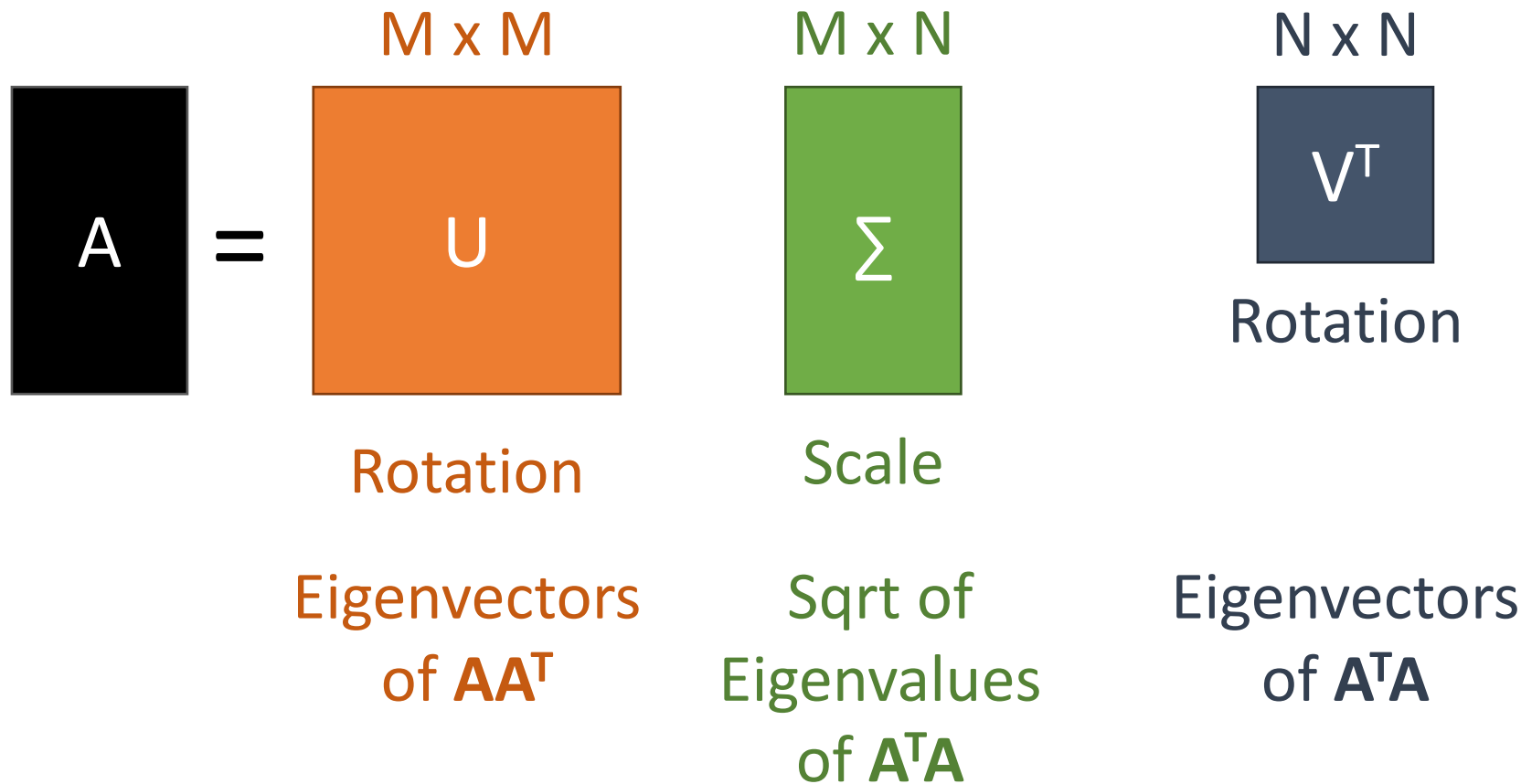
# Singular Value Decomposition

Can **always** write a  $m \times n$  matrix  $\mathbf{A}$  as:  $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$



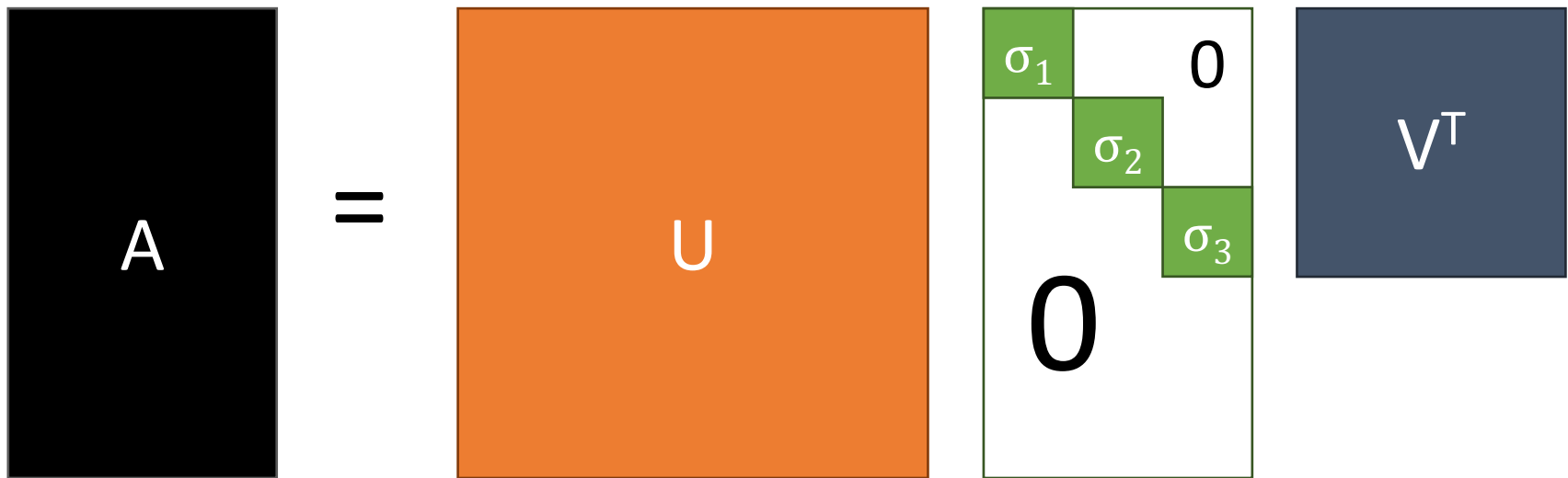
# Singular Value Decomposition

Can **always** write a  $m \times n$  matrix  $\mathbf{A}$  as:  $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$



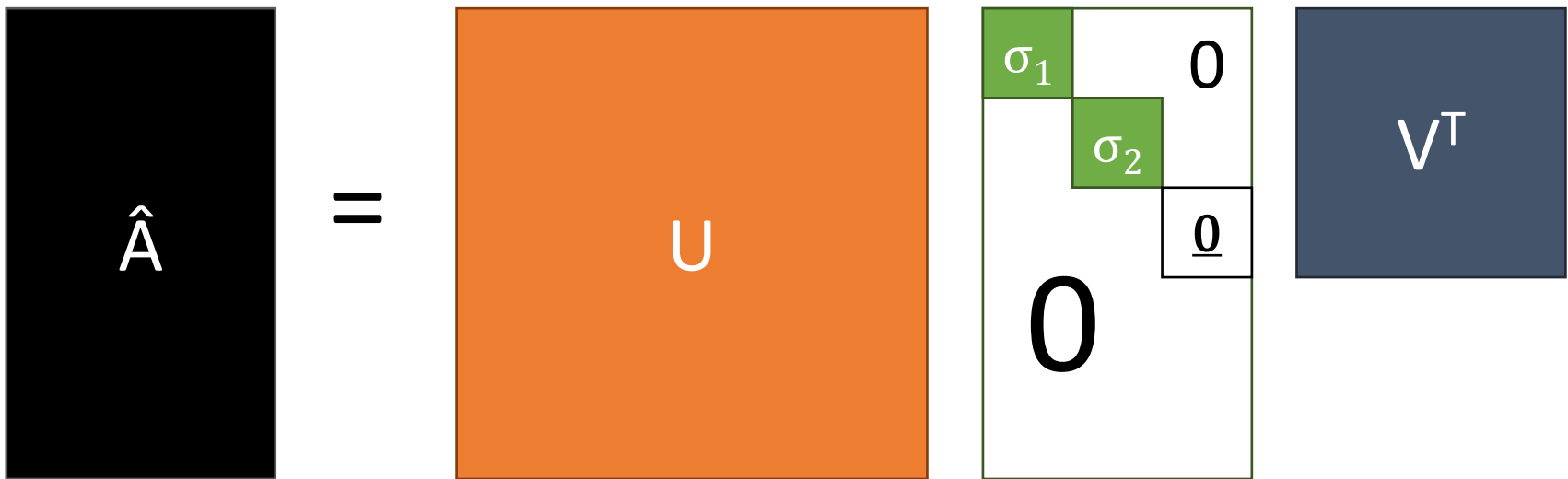
# Singular Value Decomposition

- *Every* matrix is a rotation, scaling, and rotation
- Number of non-zero singular values = rank / number of linearly independent vectors
- “Closest” matrix to **A** with a lower rank



# Singular Value Decomposition

- *Every* matrix is a rotation, scaling, and rotation
- Number of non-zero singular values = rank / number of linearly independent vectors
- “Closest” matrix to  $\mathbf{A}$  with a lower rank





# Least Squares

Given:  $\mathbf{y} \in \mathbb{R}^M$ ,  $\mathbf{A} \in \mathbb{R}^{M \times N}$

Find:  $\mathbf{v} \in \mathbb{R}^N$  such that  $\mathbf{A}\mathbf{v}$  is closest to  $\mathbf{y}$   
( $M > N$ , more equations than unknowns)

$$\boxed{\arg \min_{\mathbf{v}} \|\mathbf{y} - \mathbf{A}\mathbf{v}\|^2}$$

Solution satisfies  $(\mathbf{A}^T \mathbf{A})\mathbf{v}^* = \mathbf{A}^T \mathbf{y}$

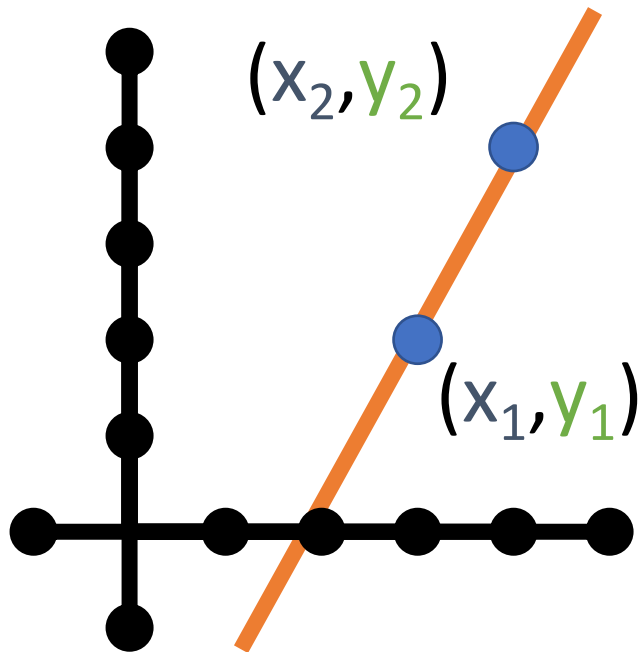
or

$$\mathbf{v}^* = (\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \mathbf{y}$$

*(Don't actually compute the inverse!)*



# Solving Least-Squares



Start with two points  $(x_i, y_i)$

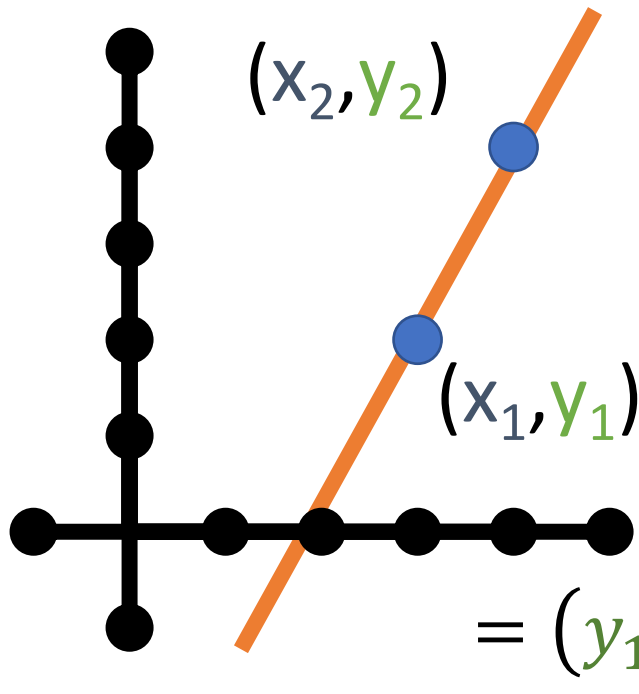
$$y = Av$$

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} x_1 & 1 \\ x_2 & 1 \end{bmatrix} \begin{bmatrix} m \\ b \end{bmatrix}$$

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} mx_1 + b \\ mx_2 + b \end{bmatrix}$$

We know how to solve this – invert  $A$  and find  $v$  (i.e.,  $(m, b)$  that fits points)

# Solving Least-Squares



Start with two points  $(x_i, y_i)$

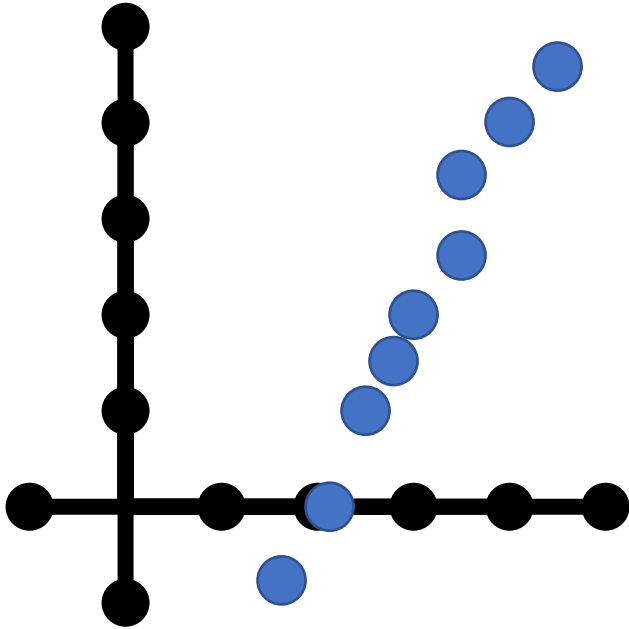
$$\mathbf{y} = \mathbf{A}\mathbf{v}$$

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} x_1 & 1 \\ x_2 & 1 \end{bmatrix} \begin{bmatrix} m \\ b \end{bmatrix}$$

$$\begin{aligned} \|\mathbf{y} - \mathbf{A}\mathbf{v}\|^2 &= \left\| \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} - \begin{bmatrix} mx_1 + b \\ mx_2 + b \end{bmatrix} \right\|^2 \\ &= (y_1 - (mx_1 + b))^2 + (y_2 - (mx_2 + b))^2 \end{aligned}$$

The sum of squared differences between  
**the actual value of  $\mathbf{y}$**  and  
**what the model says  $\mathbf{y}$  should be.**

# Solving Least-Squares



Suppose there are  $n > 2$  points

$$\mathbf{y} = \mathbf{A}\mathbf{v}$$

$$\begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix} = \begin{bmatrix} x_1 & 1 \\ \vdots & \vdots \\ x_N & 1 \end{bmatrix} \begin{bmatrix} m \\ b \end{bmatrix}$$

Compute  $\|\mathbf{y} - \mathbf{A}\mathbf{x}\|^2$  again

$$\|\mathbf{y} - \mathbf{A}\mathbf{v}\|^2 = \sum_{i=1}^n (y_i - (mx_i + b))^2$$

# Solving Least-Squares

Given  $\mathbf{y}$ ,  $\mathbf{A}$ , and  $\mathbf{v}$  with  $\mathbf{y} = \mathbf{A}\mathbf{v}$  overdetermined  
( $\mathbf{A}$  tall / more equations than unknowns)

We want to minimize  $\|\mathbf{y} - \mathbf{A}\mathbf{v}\|^2$ , or find:

$$\boxed{\arg \min_{\mathbf{v}} \|\mathbf{y} - \mathbf{A}\mathbf{v}\|^2}$$

*(The value of  $x$  that makes  
the expression smallest)*

$$\text{Solution satisfies } (\mathbf{A}^T \mathbf{A}) \mathbf{v}^* = \mathbf{A}^T \mathbf{y}$$

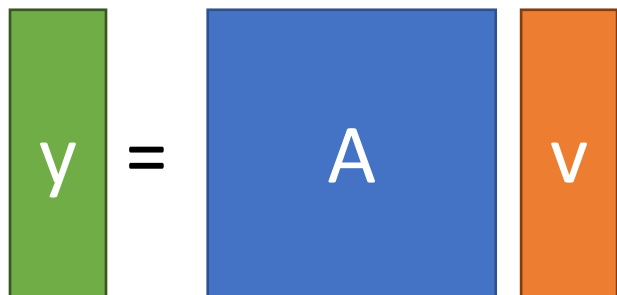
or

$$\mathbf{v}^* = (\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \mathbf{y}$$

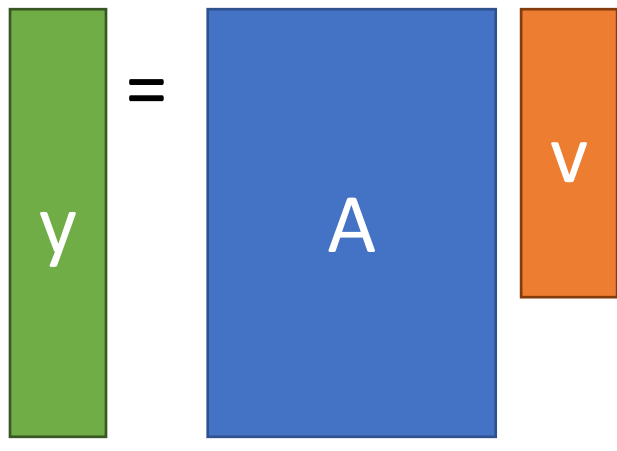
*(Don't actually compute the inverse!)*

# When is Least-Squares Possible?

Given  $\mathbf{y}$ ,  $\mathbf{A}$ , and  $\mathbf{v}$ . Want  $\mathbf{y} = \mathbf{A}\mathbf{v}$



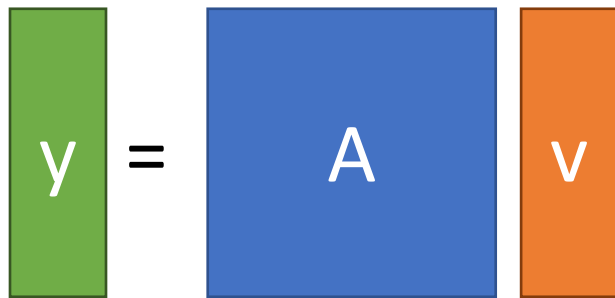
Want  $n$  outputs, have  $n$  knobs to fiddle with, every knob is useful if  $\mathbf{A}$  is full rank.



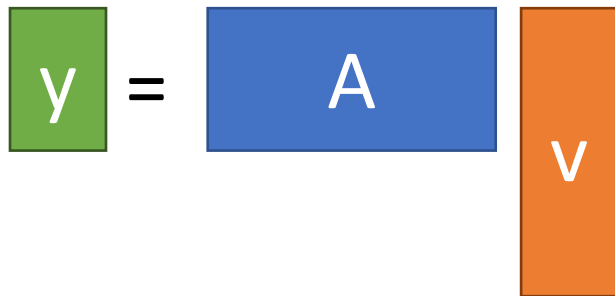
$\mathbf{A}$ : rows (outputs)  $>$  columns (knobs). Thus can't get precise output you want (not enough knobs). So settle for "closest" knob setting.

# When is Least-Squares Possible?

Given  $\mathbf{y}$ ,  $\mathbf{A}$ , and  $\mathbf{v}$ . Want  $\mathbf{y} = \mathbf{A}\mathbf{v}$



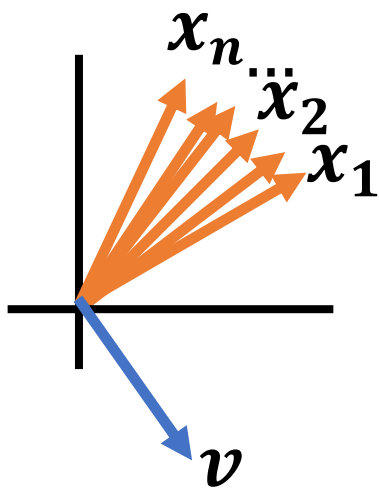
Want  $n$  outputs, have  $n$  knobs to fiddle with, every knob is useful if  $\mathbf{A}$  is full rank.



$\mathbf{A}$ : columns (knobs)  $>$  rows (outputs). Thus, any output can be expressed in infinite ways.

# Homogeneous Least-Squares

Given a set of unit vectors (aka directions)  $\mathbf{x}_1, \dots, \mathbf{x}_n$  and I want vector  $\mathbf{v}$  that is as orthogonal to all the  $\mathbf{x}_i$  as possible (for some definition of orthogonal)



Stack  $\mathbf{x}_i$  into  $\mathbf{A}$ , compute  $\mathbf{A}\mathbf{v}$

$$\mathbf{A}\mathbf{v} = \begin{bmatrix} - & \mathbf{x}_1^T & - \\ & \vdots & \\ - & \mathbf{x}_n^T & - \end{bmatrix} \mathbf{v} = \begin{bmatrix} \mathbf{x}_1^T \mathbf{v} \\ \vdots \\ \mathbf{x}_n^T \mathbf{v} \end{bmatrix} \begin{matrix} 0 \text{ if} \\ \text{orthog} \end{matrix}$$

Compute  $\|\mathbf{A}\mathbf{v}\|^2 = \sum_i (\mathbf{x}_i^T \mathbf{v})^2$

Sum of how orthog.  $\mathbf{v}$  is to each  $\mathbf{x}$

# Homogenous Least-Squares

- A lot of times, given a matrix  $\mathbf{A}$  we want to find the  $\mathbf{v}$  that minimizes  $\|\mathbf{A}\mathbf{v}\|^2$ .
- I.e., want  $\mathbf{v}^* = \arg \min_{\mathbf{v}} \|\mathbf{A}\mathbf{v}\|_2^2$
- What's a trivial solution?
- Set  $\mathbf{v} = \mathbf{0} \rightarrow \mathbf{A}\mathbf{v} = \mathbf{0}$
- Exclude this by forcing  $\mathbf{v}$  to have unit norm



# Homogenous Least-Squares

Let's look at  $\|\mathbf{A}\mathbf{v}\|_2^2$

$$\|\mathbf{A}\mathbf{v}\|_2^2 = \quad \text{Rewrite as dot product}$$

$$\|\mathbf{A}\mathbf{v}\|_2^2 = (\mathbf{A}\mathbf{v})^T (\mathbf{A}\mathbf{v}) \quad \text{Distribute transpose}$$

$$\|\mathbf{A}\mathbf{v}\|_2^2 = \mathbf{v}^T \mathbf{A}^T \mathbf{A} \mathbf{v} = \mathbf{v}^T (\mathbf{A}^T \mathbf{A}) \mathbf{v}$$

We want the vector minimizing this quadratic form

Where have we seen this?

# Homogenous Least-Squares

Ubiquitous tool in vision:

$$\arg \min_{\|v\|^2=1} \|Av\|^2$$

- (1) “Smallest”\* eigenvector of  $A^T A$   
(2) “Smallest” right singular vector of  $A$

For min → max, switch smallest → largest

\*Note:  $A^T A$  is positive semi-definite so it has all non-negative eigenvalues

# Derivatives

# Things to know

- Given a scalar-valued function  $f: \mathbb{R} \rightarrow \mathbb{R}$ , the **derivative**  $f'(x) \in \mathbb{R}$  of  $f$  at the point  $x \in \mathbb{R}$  is the rate at which the function changes at that point
- Given a vector-valued function  $f: \mathbb{R}^N \rightarrow \mathbb{R}$ , the **gradient**  $\nabla f(x) \in \mathbb{R}^N$  at the point  $x \in \mathbb{R}^N$  is the vector of all partial derivatives  $\frac{\partial f}{\partial x_i}(x)$
- The gradient points in the direction of greatest increase; its magnitude is the slope in that direction
- If  $x$  is a local minimum of  $f$ , then  $\nabla f(x) = \mathbf{0}$  (but the converse is not true!)

# Derivatives

Remember derivatives?

Derivative: rate at which a function  $f(x)$  changes at a point as well as the direction that increases the function

Given quadratic function  $f(x)$

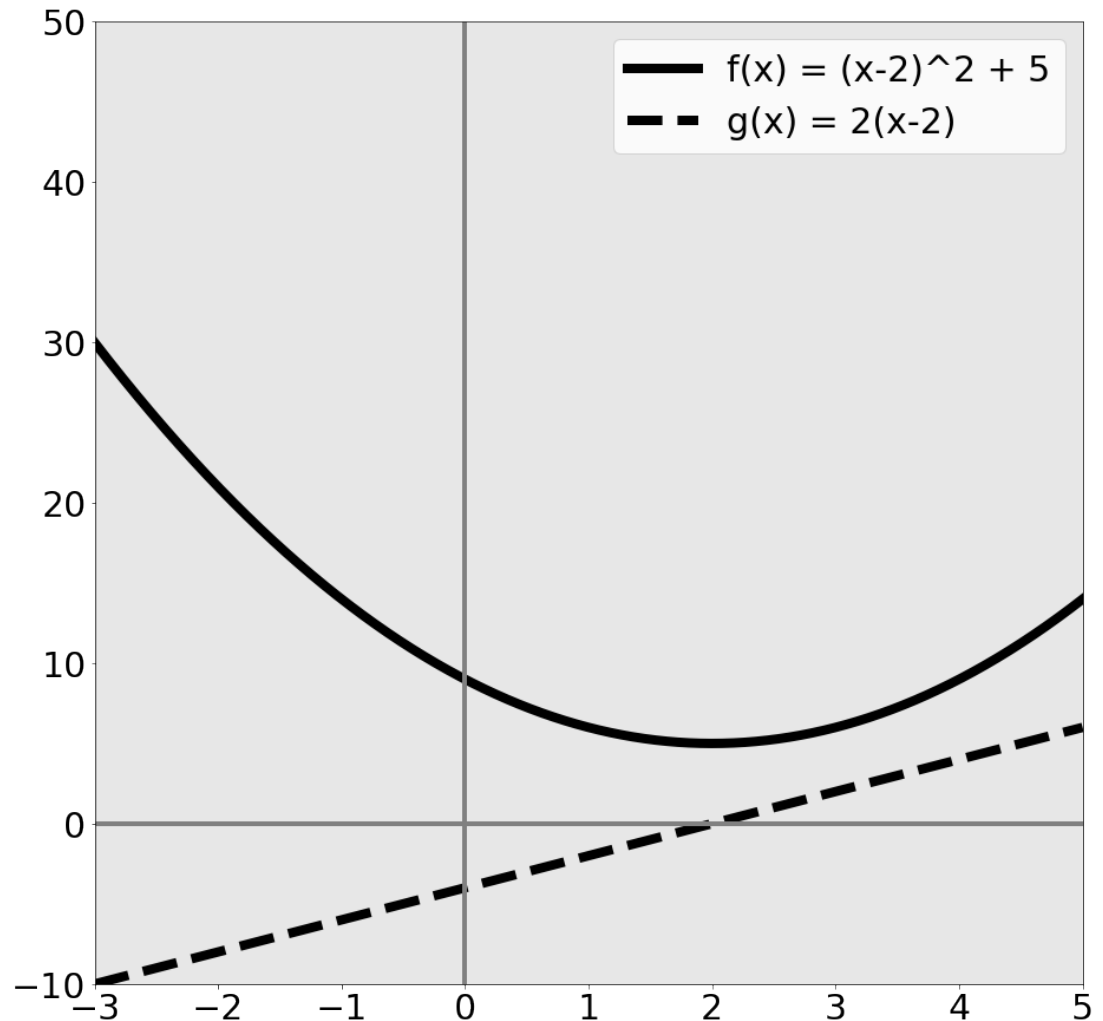
$$f(x, y) = (x - 2)^2 + 5$$

$f(x)$  is function

$$g(x) = f'(x)$$

aka

$$g(x) = \frac{d}{dx} f(x)$$



Given quadratic function  $f(x)$

$$f(x, y) = (x - 2)^2 + 5$$

What's special about  $x=2$ ?

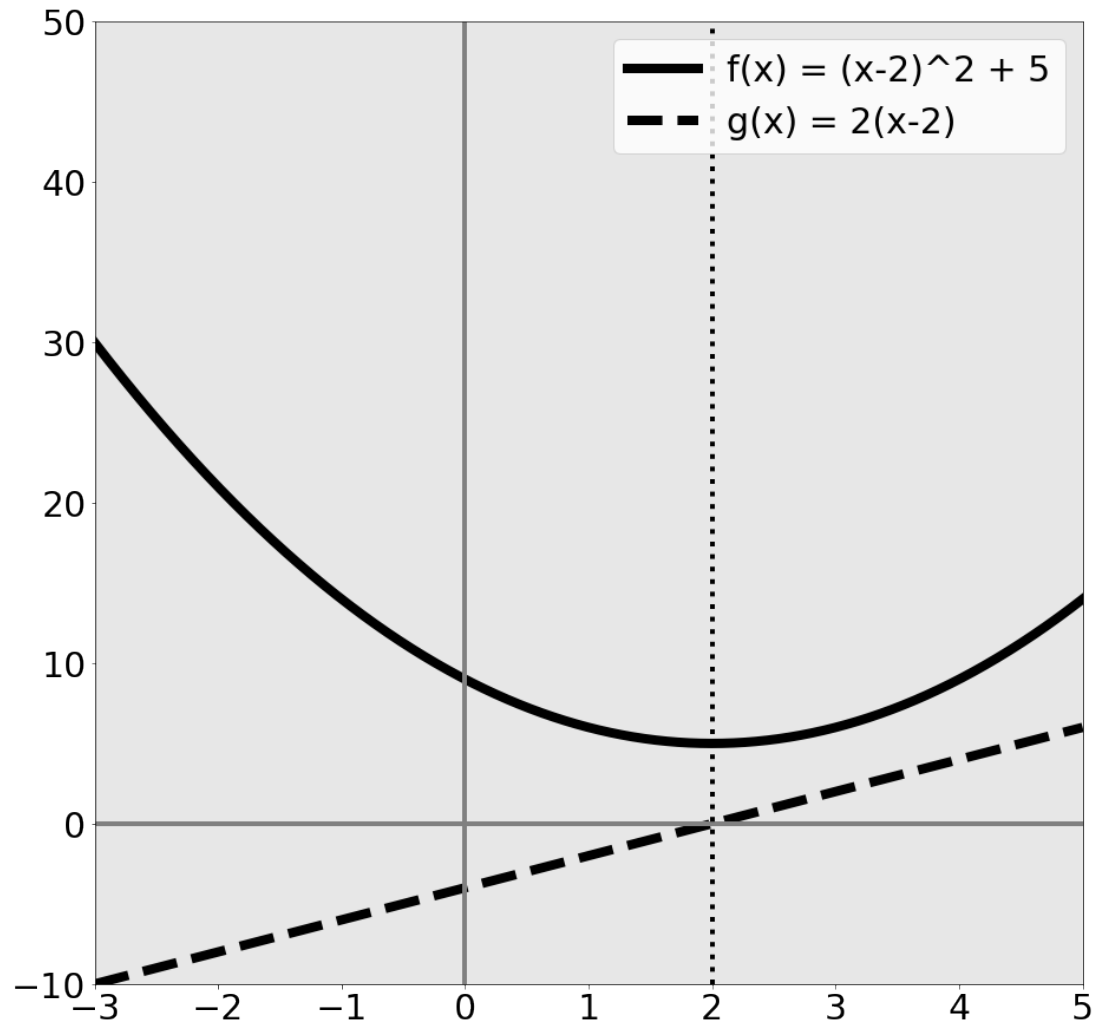
$f(x)$  minim. at 2

$$g(x) = 0 \text{ at } 2$$

$a = \text{minimum of } f \rightarrow$

$$g(a) = 0$$

Reverse is **not true**



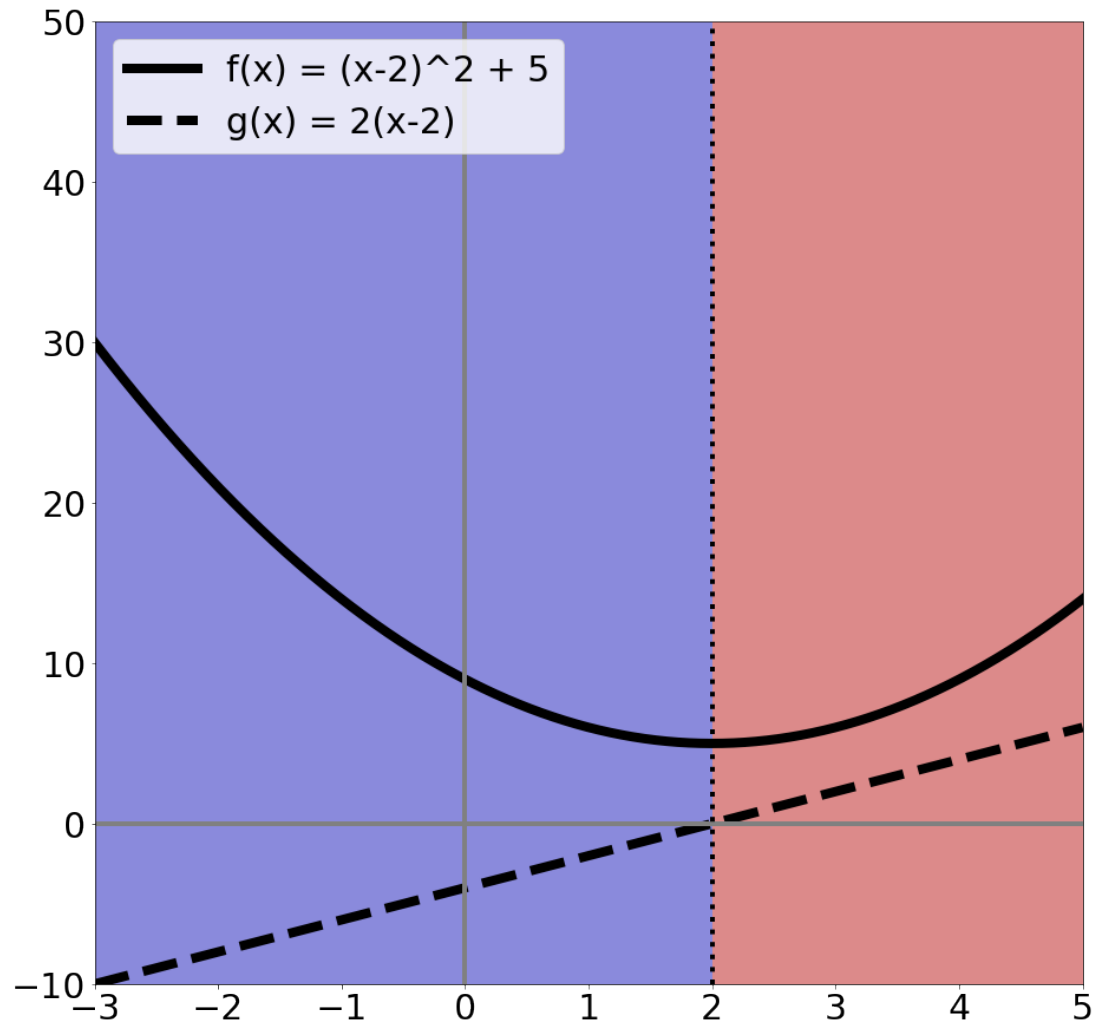
# Rates of change

$$f(x, y) = (x - 2)^2 + 5$$

Suppose I want to increase  $f(x)$  by changing  $x$ :

Blue area: move left  
Red area: move right

Derivative tells you direction of ascent and rate





# Calculus to Know

- Really need intuition
- Need chain rule
- Rest you should look up / use a computer algebra system / use a cookbook
- Partial derivatives (and that's it from multivariable calculus)

# Partial Derivatives

- Pretend other variables are constant, take a derivative. That's it.
- Make our function a function of two variables

$$f(x) = (x - 2)^2 + 5$$

$$\frac{\partial}{\partial x} f(x) = 2(x - 2) * 1 = 2(x - 2)$$

$$f_2(x, y) = (x - 2)^2 + 5 + (y + 1)^2$$

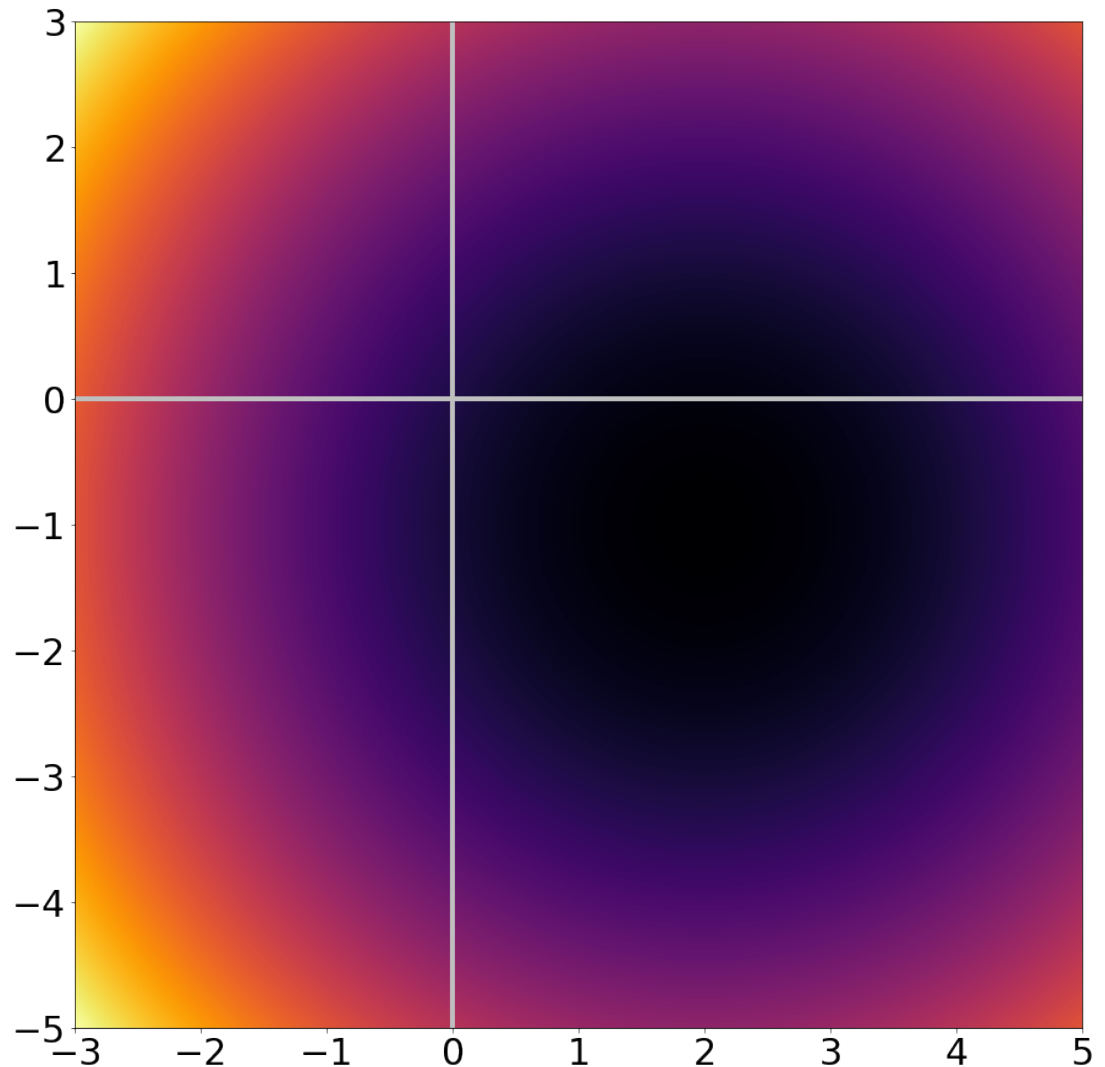
$$\frac{\partial}{\partial x} f_2(x) = 2(x - 2)$$

Pretend it's  
constant  $\rightarrow$   
derivative = 0

# Zooming Out

$$f_2(x, y) = (x - 2)^2 + 5 + (y + 1)^2$$

Dark =  $f(x, y)$  low  
Bright =  $f(x, y)$  high



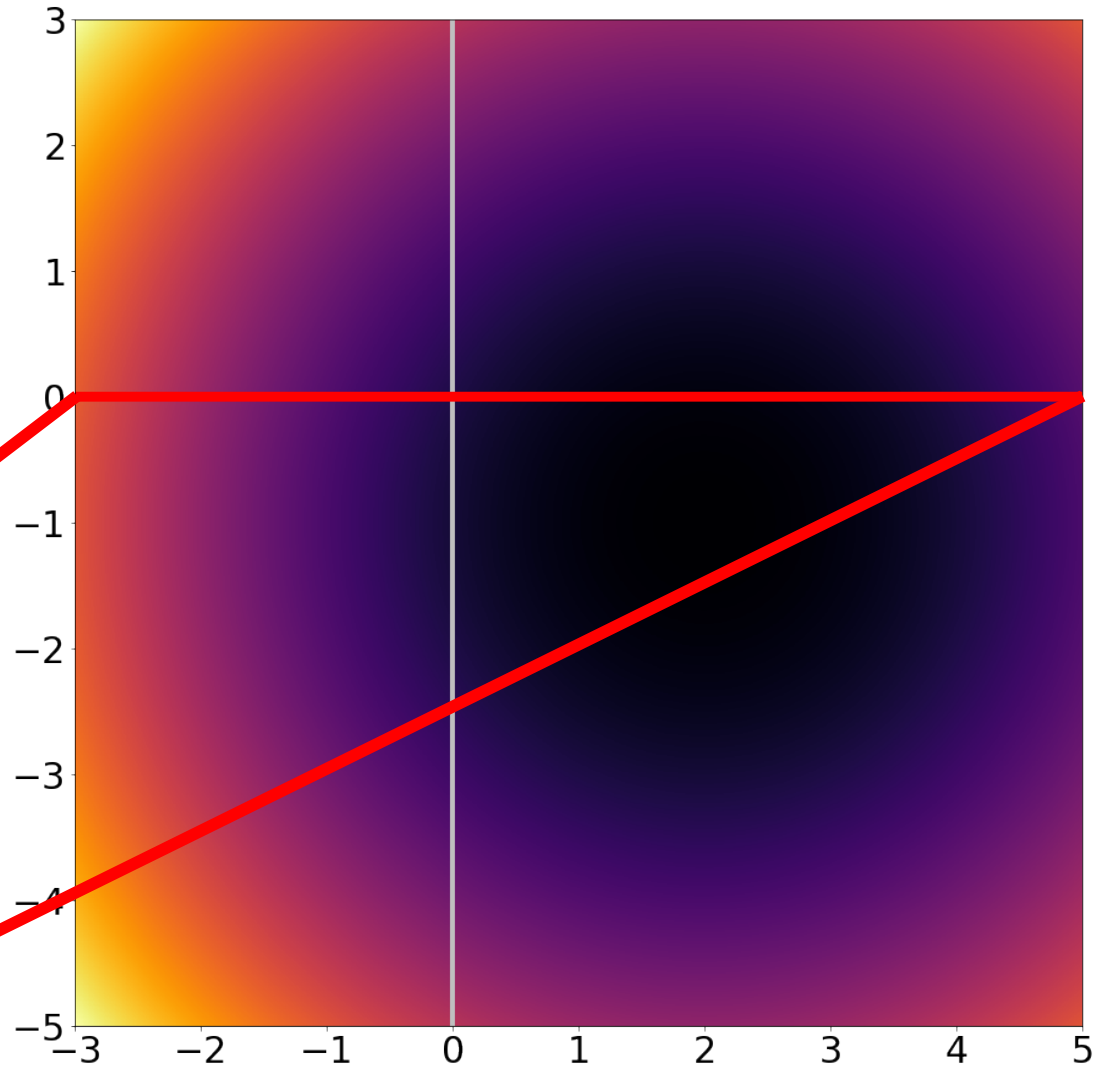
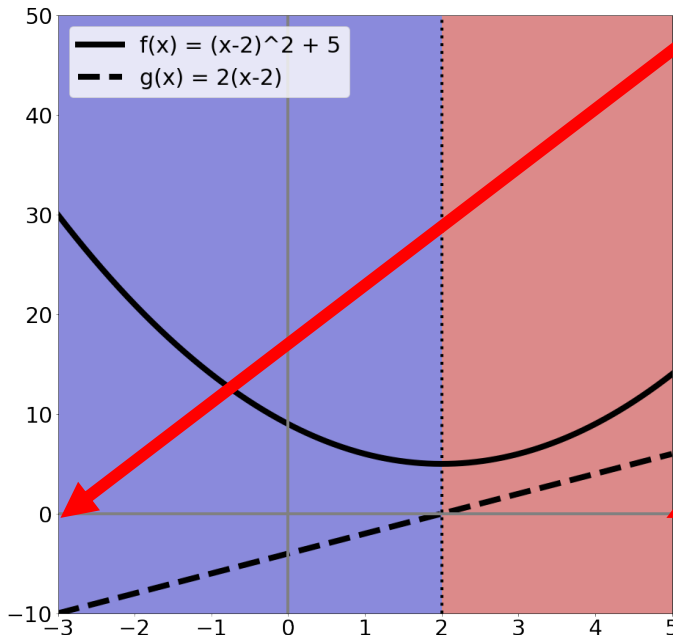
# Taking a slice of

$$f_2(x, y) = (x - 2)^2 + 5 + (y + 1)^2$$

Slice of  $y=0$  is the  
function from before:

$$f(x) = (x - 2)^2 + 5$$

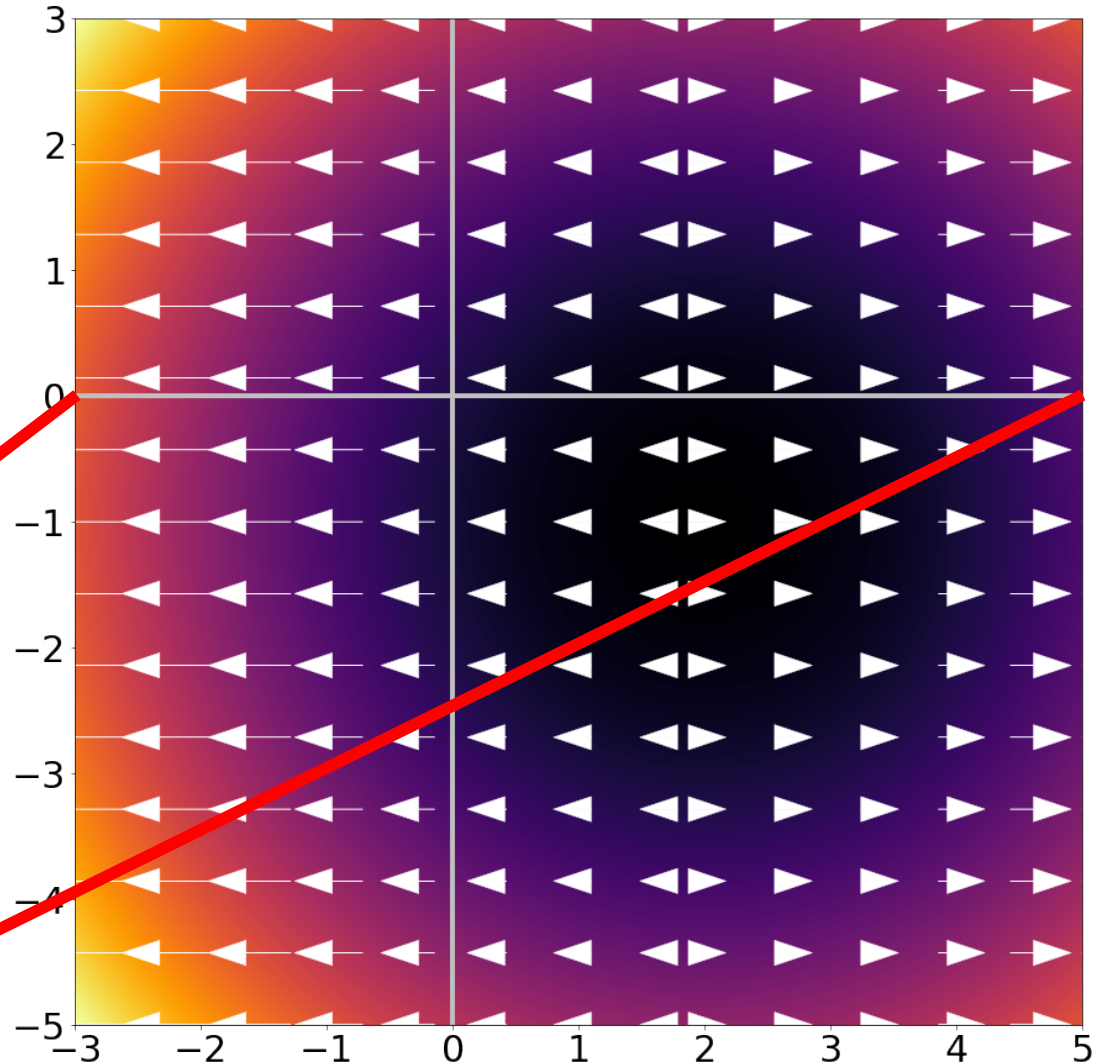
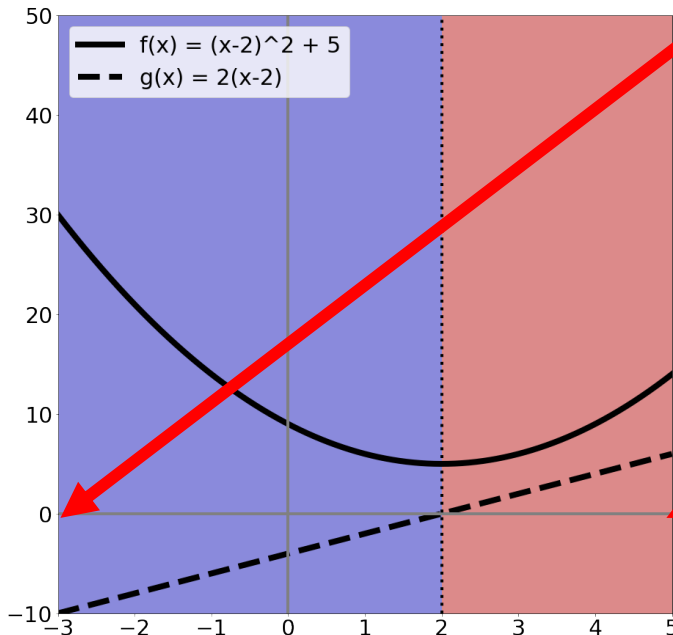
$$f'(x) = 2(x - 2)$$



# Taking a slice of

$$f_2(x, y) = (x - 2)^2 + 5 + (y + 1)^2$$

$\frac{\partial}{\partial x} f_2(x, y)$  is rate of change & direction in x dimension



# Zooming Out

$$f_2(x, y) = (x - 2)^2 + 5 + (y + 1)^2$$

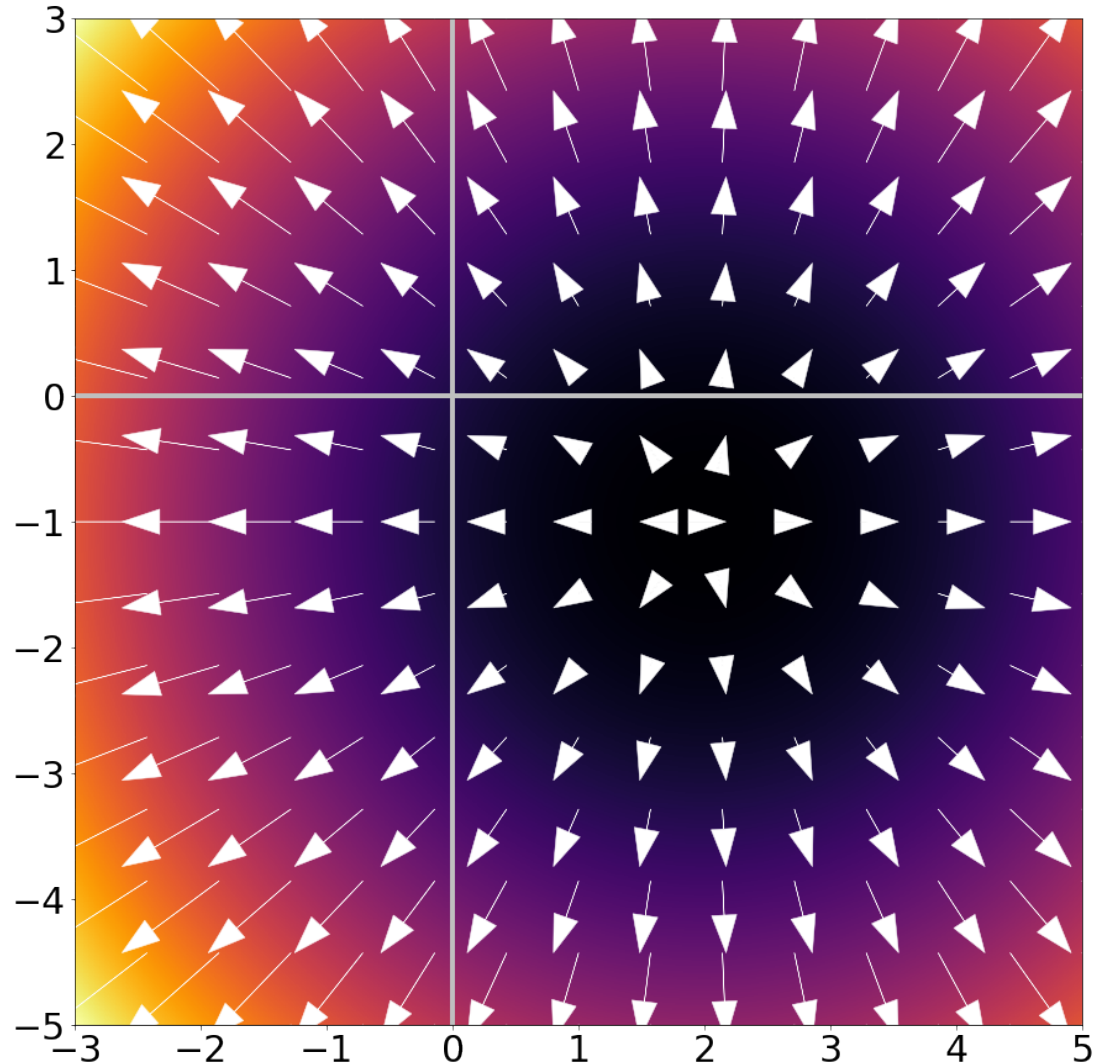
## Gradient/Jacobian:

Making a vector of

$$\nabla_f = \left[ \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right]$$

gives rate and direction  
of change.

Arrows point OUT of  
minimum / basin.



# What Should I Know?

- Gradients are simply partial derivatives per-dimension: if  $\mathbf{x}$  in  $f(\mathbf{x})$  has  $n$  dimensions,  $\nabla_f(\mathbf{x})$  has  $n$  dimensions
- Gradients point in direction of ascent and tell the rate of ascent
- If  $\mathbf{a}$  is minimum of  $f(\mathbf{x}) \rightarrow \nabla_f(\mathbf{a}) = \mathbf{0}$
- Reverse is not true, especially in high-dimensional spaces

# Recap

Lots of things today:

- Floating point numbers
- Vectors
- Matrices
- Broadcasting / Vectorization
- Linear Algebra
- Derivatives



# Next Time: Image Filtering