

# Continuous Mathematical Methods, emphasizing Machine Learning

Ron Fedkiw<sup>\*1</sup>, Yilin Zhu<sup>\*23</sup>, Winnie Lin<sup>\*3</sup>, Jane Wu<sup>\*3</sup>

<sup>\*</sup> Stanford University, <sup>1</sup> Slide Design and Content (and Instructor), <sup>2</sup> Slide Illustrator, <sup>3</sup> Teaching Assistant

# Table of Contents

- Unit 1: Introduction
- Unit 2: Linear Systems
- Unit 3: Understanding Matrices
- Unit 4: Special Matrices
- Unit 5: Iterative Solvers
- Unit 6: Local Approximations
- Unit 7: Curse of Dimensionality
- Unit 8: Least Squares
- Unit 9: Basic Optimization
- Unit 10: Solving Least Squares
- Unit 11: Zero Singular Values
- Unit 12: Regularization
- Unit 13: Optimization
- Unit 14: Nonlinear Systems
- Unit 15: 1D Root Finding
- Unit 16: 1D Optimization
- Unit 17: Computing Derivatives
- Unit 18: Avoiding Derivatives
- Unit 19: Descent Methods
- Unit 20: Momentum Methods
- Appendix: Notation

# Unit 1

# What is Learning?

# What is Learning?

- There are lots of answers to this question, and explanations often become philosophical
- A more practical question might be:

What can we teach/train a person, animal, or machine to do?

# Example: Addition “+”

- How is addition taught in schools?
  - Memorize rules for pairs of numbers from the set  $\{0,1,2,3,4,5,6,7,8,9\}$
  - Memorize redundant rules collectively, for efficiency, e.g.  $0+x=x$
  - Learn to treat powers of 10 implicitly, e.g.  $12+34=46$  because  $1+3=4$  and  $2+4=6$
  - Learn to carry when the sum of two numbers is larger than 9
  - Learn to add larger sets of numbers by considering them one pair at a time
  - Learn how to treat negative numbers
  - Learn how to treat decimals and fractions
  - Learn how to treat irrational numbers

# Knowledge Based Systems (KBS)

Contains two parts:

## 1) Knowledge Base

- Explicit knowledge or facts
- Often populated by an expert (expert systems)

## 2) Inference Engine

- Way of reasoning about the facts in order to generate new facts
- Typically follows the rules of Mathematical Logic

# KBS Approach to Addition

- Rule:  $x$  and  $y$  commute
- Start with  $x$  and  $y$  as single digits, and record all  $x + y$  outcomes as facts (using addition)
- Add rules to deal with numbers with more than one digit by pulling out powers of 10
- Add rules for negative numbers, decimals, fractions, irrationals, etc.
- Mimics human learning (or at least human **teaching**)
- This is a discrete approach, and it has no inherent error

# Machine Learning (ML)

Contains two parts:

## 1) Training Data

- Data Points - typically as domain/range pairs
- Hand labeled by a user, measured from the environment, or generated procedurally

## 2) Model

- Derived from Training Data in order to estimate new data points minimizing errors
- Uses Algorithms, Statistical Reasoning, Rules, Networks, Etc.

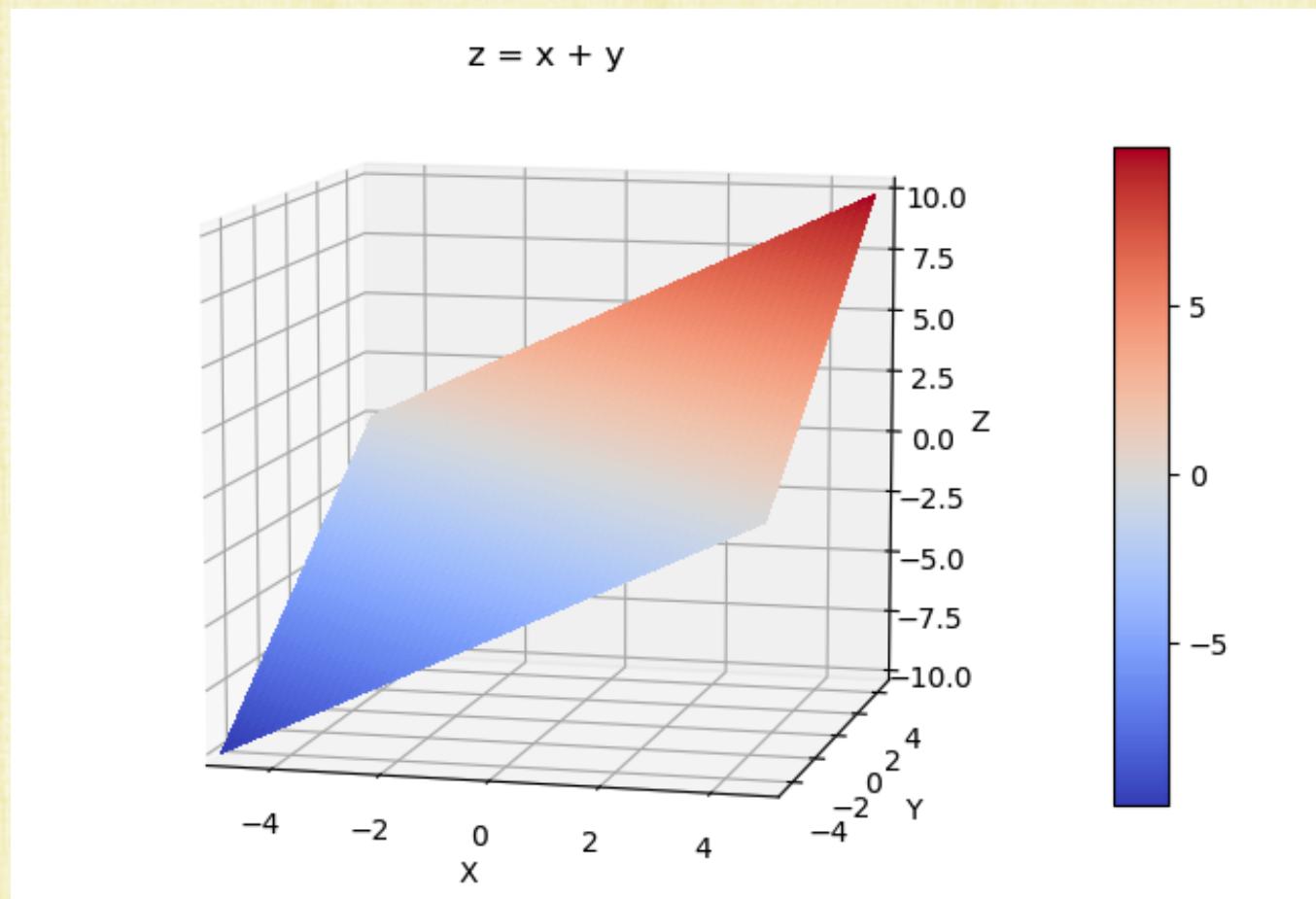
# KBS vs. ML

- KBS and ML can be seen as the discrete math and continuous math approaches (respectively) to the same problem
- KBS's Knowledge Base serves the same role as ML's Training Data
- Logic is the algorithm used to discover new discrete facts for KBS, whereas many numerical algorithms/methods are used to approximate continuous facts/data for ML
  - Logic (in particular) happens to be especially useful for discrete facts
- ML, derived from continuous math, will tend to have inherent approximation errors

# ML Approach to Addition

- Make a  $2D$  domain in  $R^2$ , and a  $1D$  range in  $R^1$  for the addition function
- As training data, choose a number of input points  $(x_i, y_i)$  with output  $x_i + y_i$
- Plot the  $3D$  points  $(x_i, y_i, x_i + y_i)$  and determine a model function  $z = f(x, y)$  that best approximates the training data
- Turns out that the plane  $z = x + y$  exactly fits the training data
  - Only need 3 training points to determine this plane
- Don't need special rules for negative numbers, decimals, fractions, irrationals such as  $\sqrt{2}$  and  $\pi$ , etc.
- However, small errors in the training data lead to a slightly incorrect plane, which has quite large errors far away from the training data
- This can be alleviated to some degree by adding training data where one wants smaller errors (and computing the best fitting plane to all the training data)

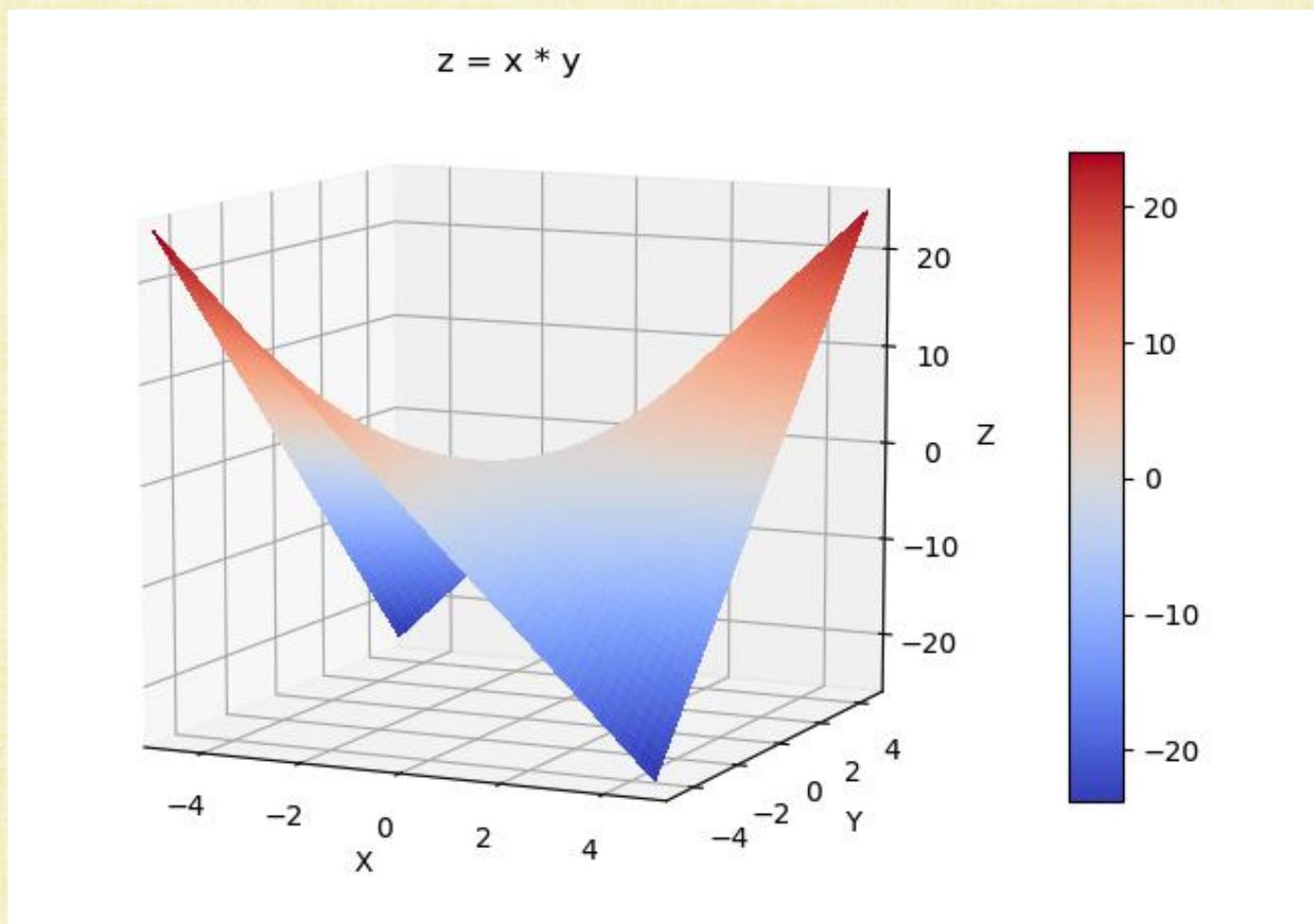
# ML Approach to Addition



# Example: Multiplication “ $*$ ”

- KBS creates new rules for  $x * y$ , utilizing the rules from addition too
- ML utilizes a set of 3D points  $(x_i, y_i, x_i * y_i)$  as training data, and the model function  $z = x * y$  can be found to exactly fit the training data
- However, one may claim that it is “cheating” to use an inherently represented floating point operation (i.e., multiplication) as the model

# ML Approach to Multiplication



# Example: Unknown Operation “#”

- KBS fails!
- How can KBS create rules for  $x\#y$  when we don't even know what # means?
- This is the case for many real-world phenomena that are not fully understood
- However, sometimes it is possible to get some examples of  $x\#y$
- That is, through experimentation or expert knowledge, can discover  $z_i = x_i \# y_i$  for some number of pairs  $(x_i, y_i)$
- Subsequently, these known (or estimated) 3D points  $(x_i, y_i, z_i)$  can be used as training data to determine a model function  $z = f(x, y)$  that approximately fits the data

# Determining the Model Function

- How does one determine  $z = f(x, y)$  near the training data, so that it robustly predicts/infers  $\hat{z}$  for new inputs  $(\hat{x}, \hat{y})$  not contained in the training data?
- How does one minimize the effect of inaccuracies or noise in the training data?
- Caution: away from the training data, the model function  $f$  is likely to be highly inaccurate (**extrapolation is ill-posed**)

# Nearest Neighbor

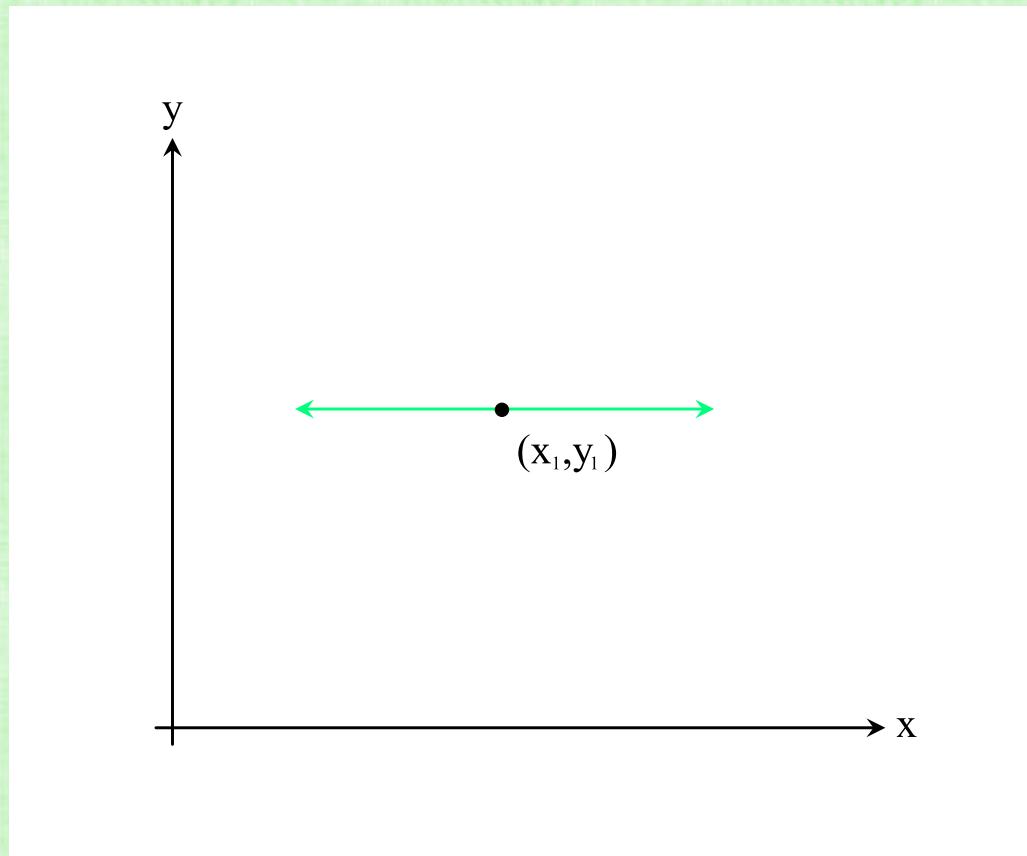
- If asked to multiply 51.023 times 298.5, one might quickly estimate that 50 times 300 is 15,000
- This is a nearest neighbor algorithm, relying on nearby data where the answer is known, better known, or more easy to come by
- Given  $(\hat{x}, \hat{y})$ , find the closest (Euclidean distance) training data  $(x_i, y_i)$  and return the associated  $z_i$  (with error  $\|z_i - \hat{z}\|$ )
- This represents  $z = f(x, y)$  as a piecewise constant function with discontinuities on the boundaries of Voronoi regions around the training data
- This is the simplest possible Machine Learning algorithm (a piecewise constant function), and it works in an arbitrary number of dimensions

# Data Interpolation

- In order to elucidate various concepts, let's consider the interpolation of data in more detail
- Let's begin with a very simple case with  $1D$  inputs and  $1D$  outputs, i.e.  $y = f(x)$

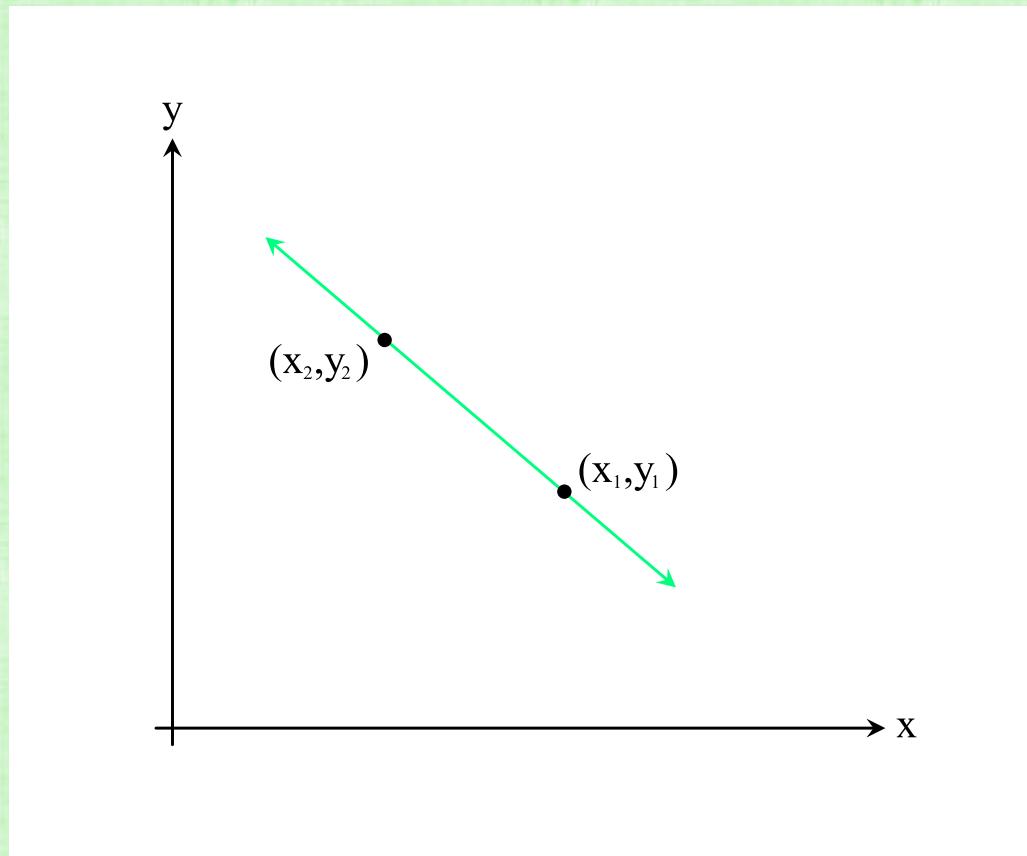
# Polynomial Interpolation

- Given 1 data point, one can (at best) draw a constant function



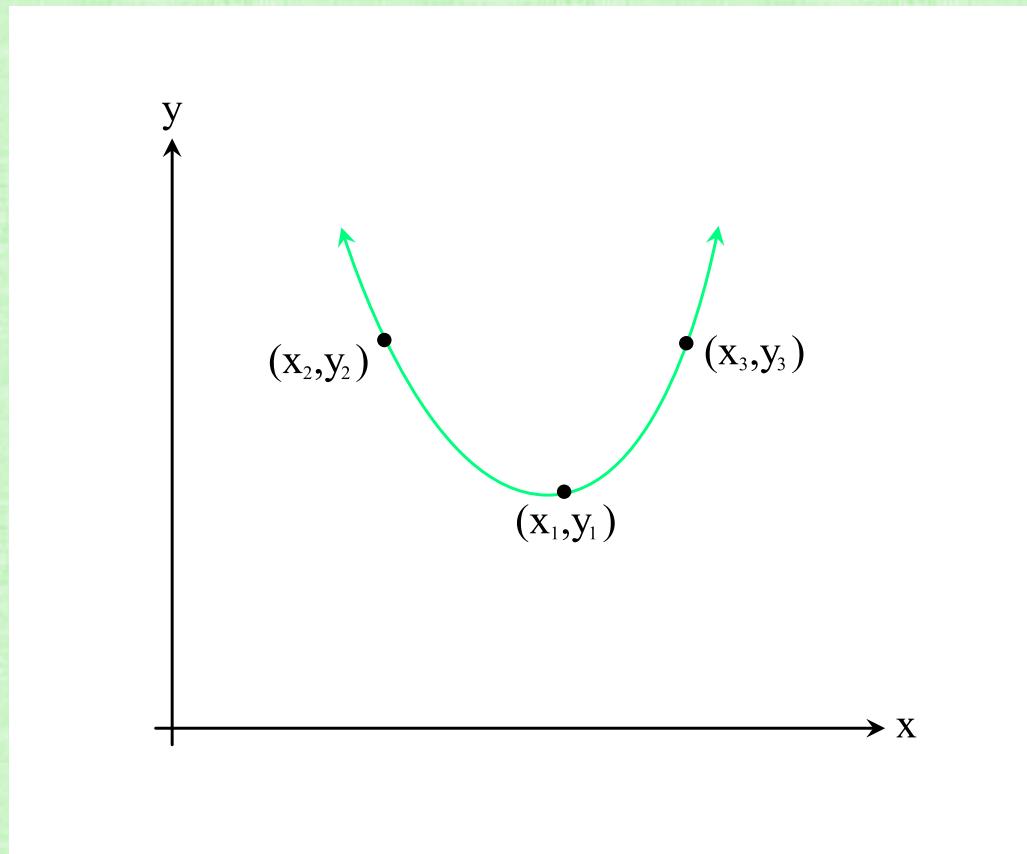
# Polynomial Interpolation

- Given 2 data points, one can (at best) draw a linear function



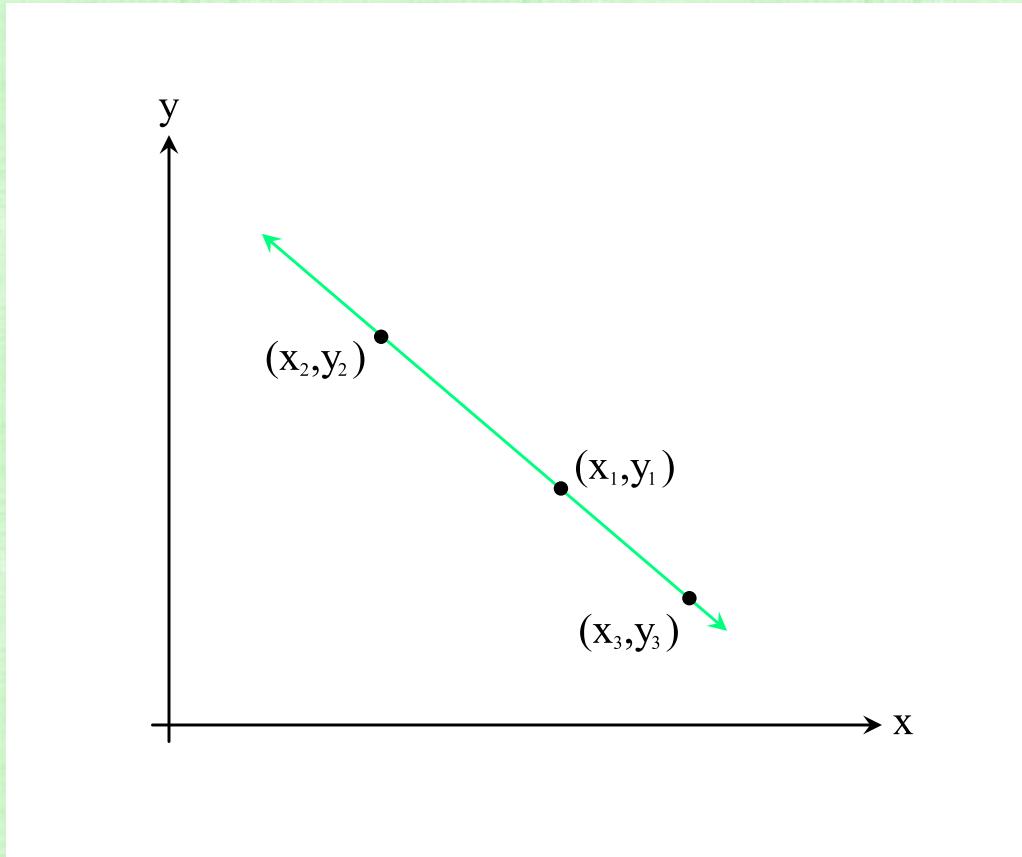
# Polynomial Interpolation

- Given 3 data points, one can (at best) draw a quadratic function



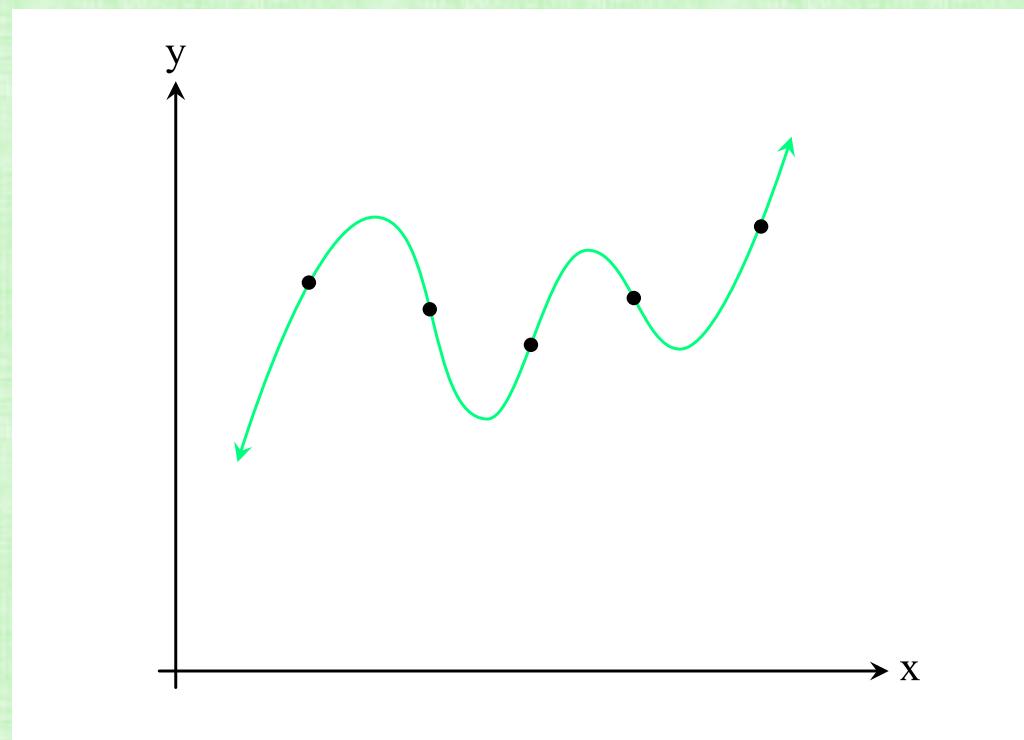
# Polynomial Interpolation

- Unless all 3 points are on the same line, in which case one can only draw a linear function



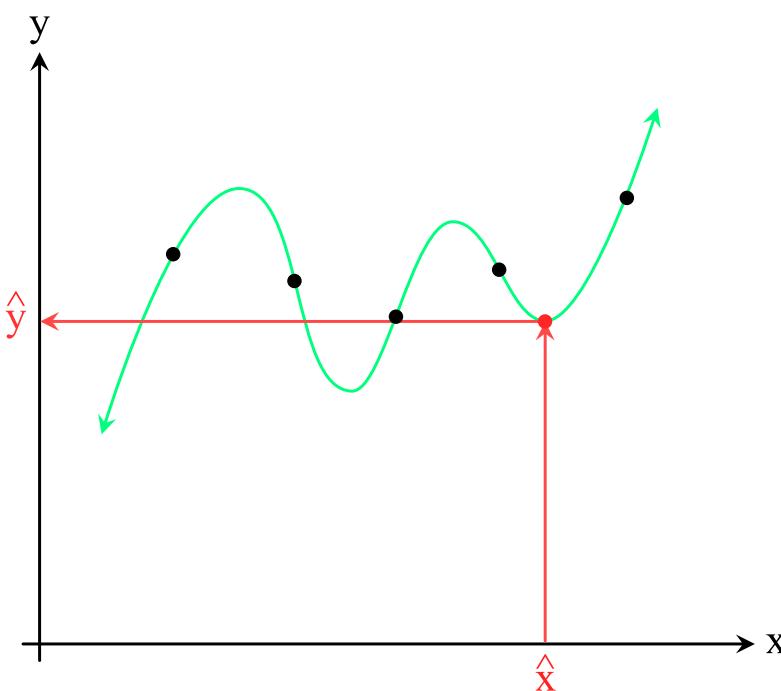
# Polynomial Interpolation

- Given  $m$  data points, one can (at best) draw a unique  $m - 1$  degree polynomial that goes through all of them
  - As long as they are not degenerate, like 3 points on a line



# Overfitting

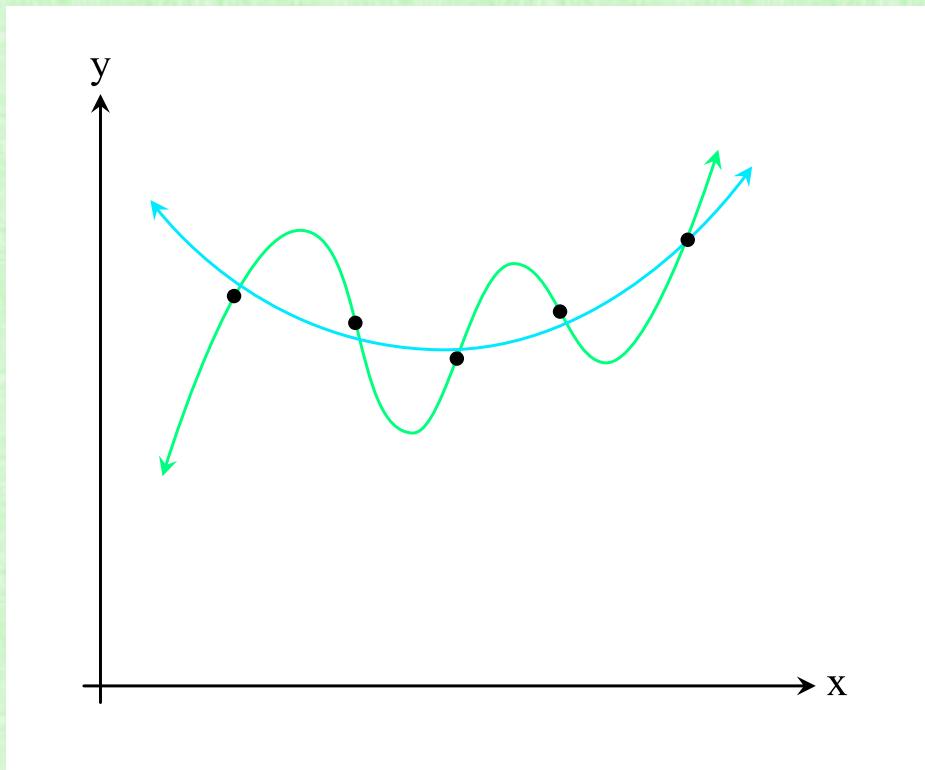
- Given a new input  $\hat{x}$ , the interpolating polynomial infers/predicts an output  $\hat{y}$  that may be far from what one may expect



- Interpolating polynomials are smooth (continuous function and derivatives)
- Thus, they wiggle/overshoot in between data points (so that they can smoothly turn back and hit the next point)
- Overly forcing polynomials to exactly hit every data point is called overfitting (overly fitting to the data)
- It results in inference/predictions that can vary wildly from the training data

# Regularization

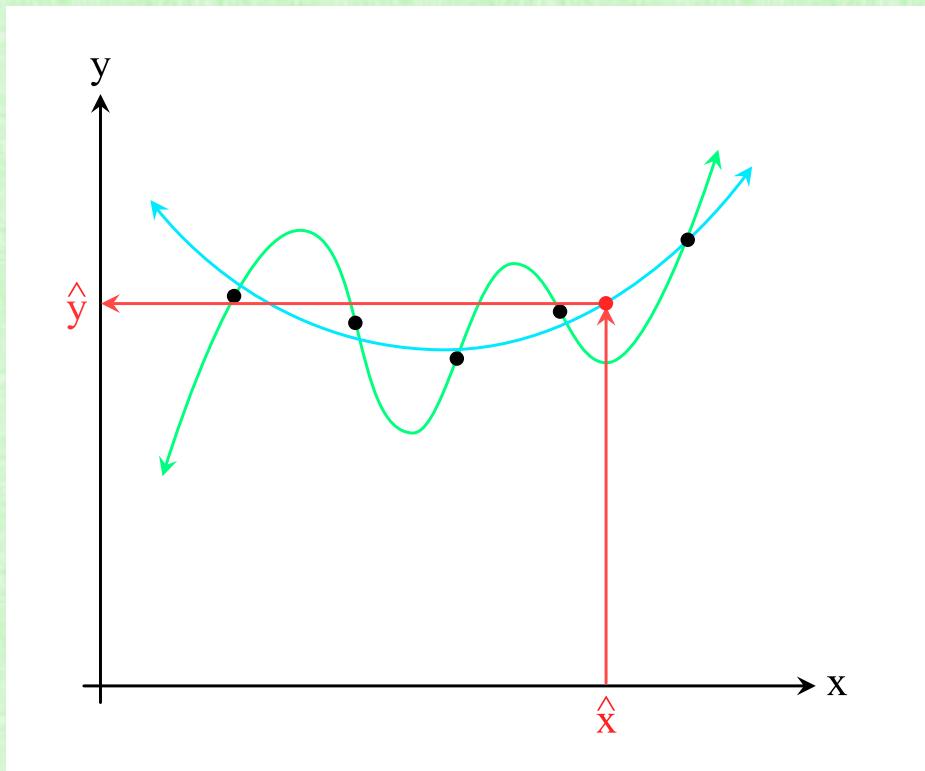
- Using a lower order polynomial that doesn't (can't) exactly fit the data points provides some degree of regularization



- A regularized interpolant contains intentional errors in the interpolation, missing some/all of the data points
- However, this hopefully makes the function more predictable/smooth in between the data points
- The data points themselves may contain noise/error, so it is not clear whether they should be interpolated exactly anyways

# Regularization

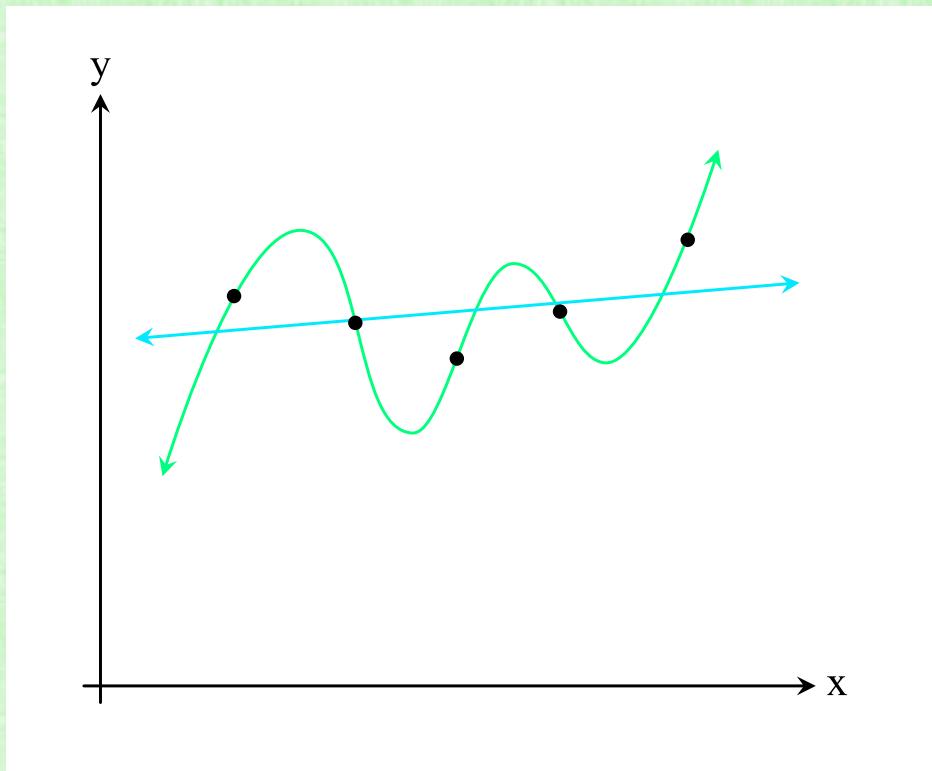
- Given  $\hat{x}$ , the regularized interpolant infers/predicts a more reasonable  $\hat{y}$



- There is a trade-off between sacrificing accuracy on fitting the original input data, and obtaining better accuracy on inference/prediction for new inputs

# Underfitting

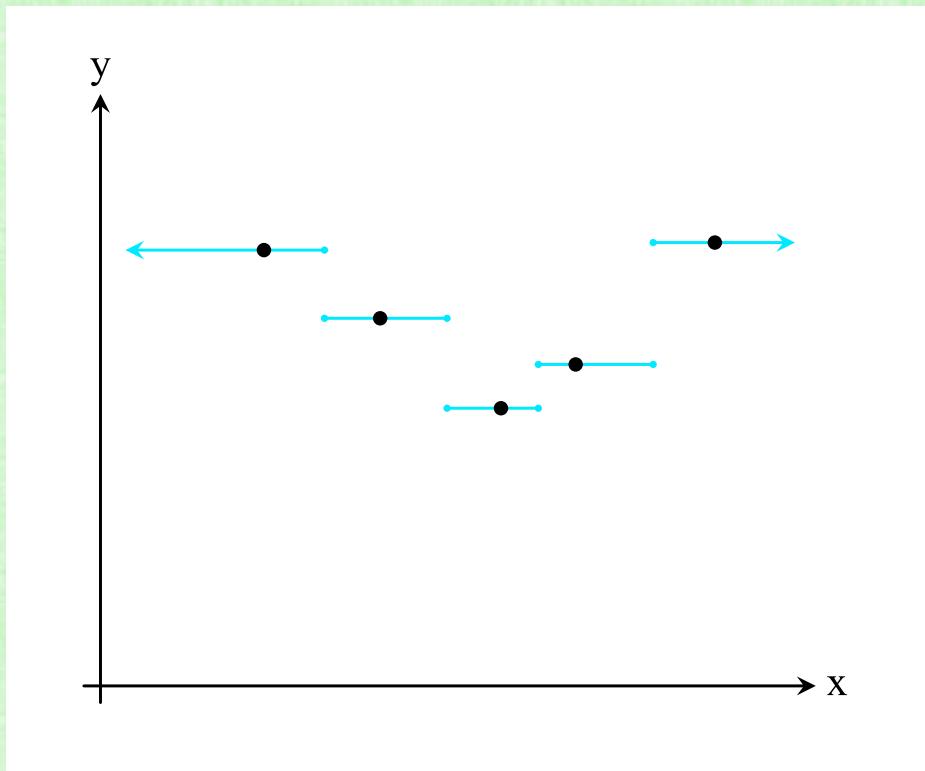
- Using too low of an order polynomial causes one to miss the data by too much



- A linear function doesn't capture the essence of this data as well as a quadratic function does
- Choosing too simple of a model function or regularizing too much prevents one from properly representing the data

# Nearest Neighbor

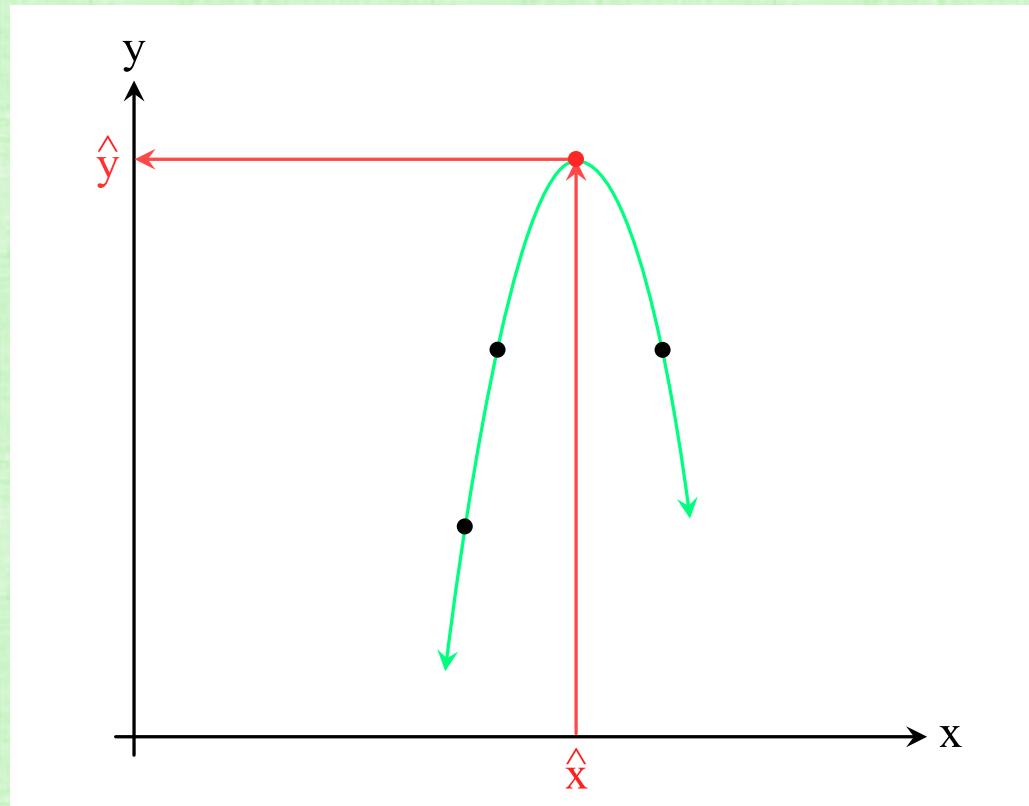
- Piecewise-constant interpolation on this data (equivalent to nearest neighbor)



- The reasonable behavior of the piecewise constant (nearest neighbor) function stresses the importance of approximating data locally
- We will address Local Approximations in Unit 6

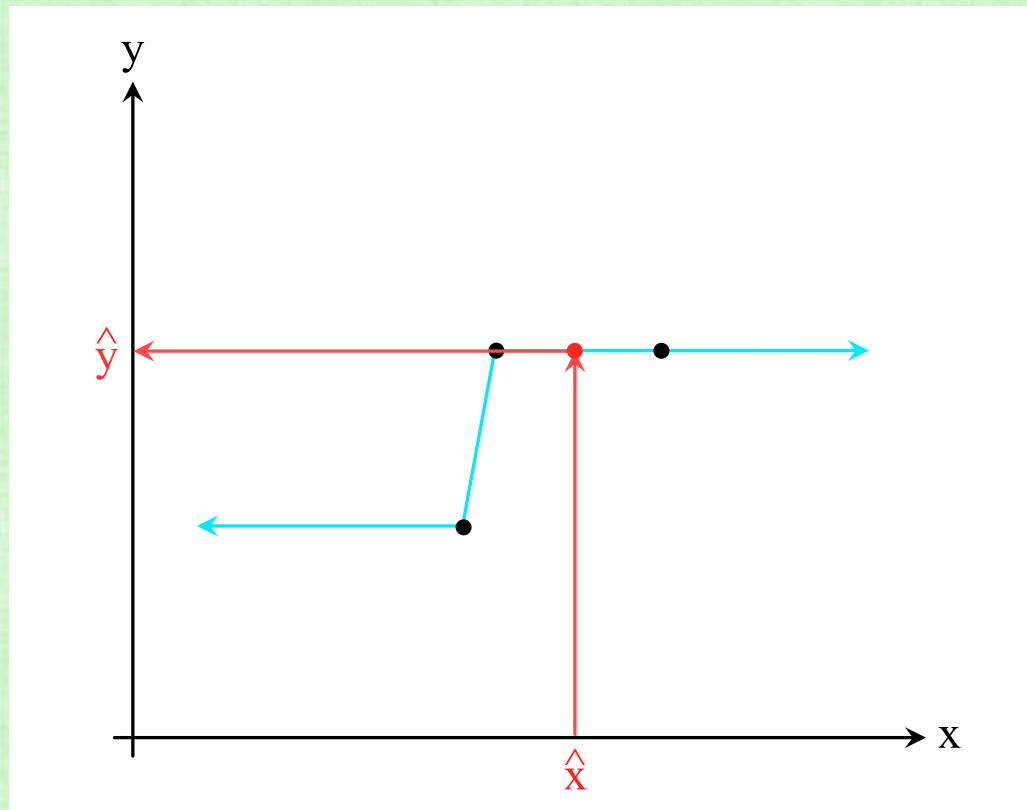
# Caution: Overfitting

- Higher order polynomials tend to oscillate wildly, but even a simple quadratic polynomial can overfit by quite a bit



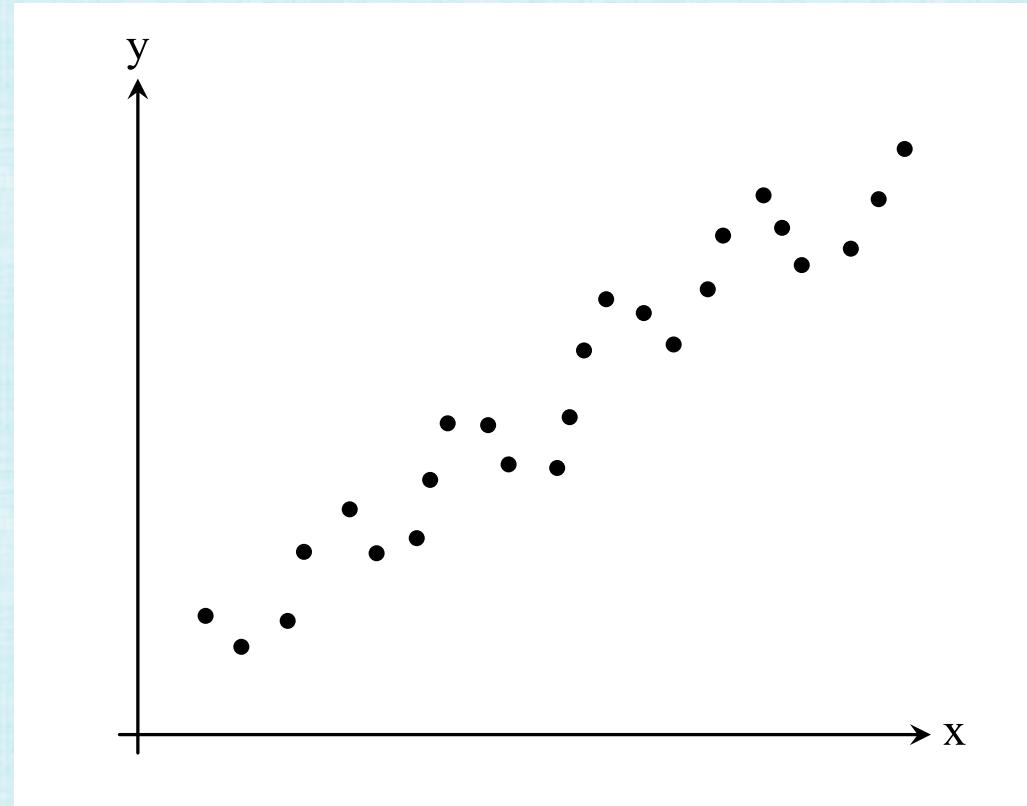
# Caution: Overfitting

- A piecewise linear approach works much better on this data



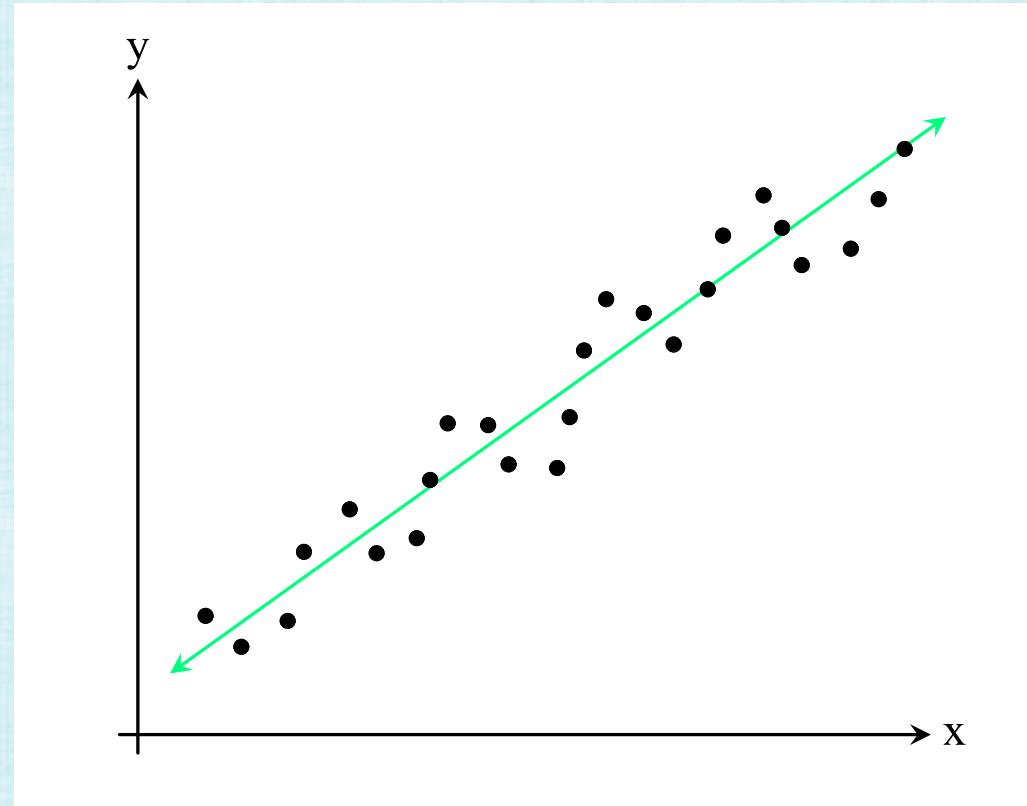
# Noisy Data

- There may be many sources of error in data, so it can be unwise to attempt to fit data too closely



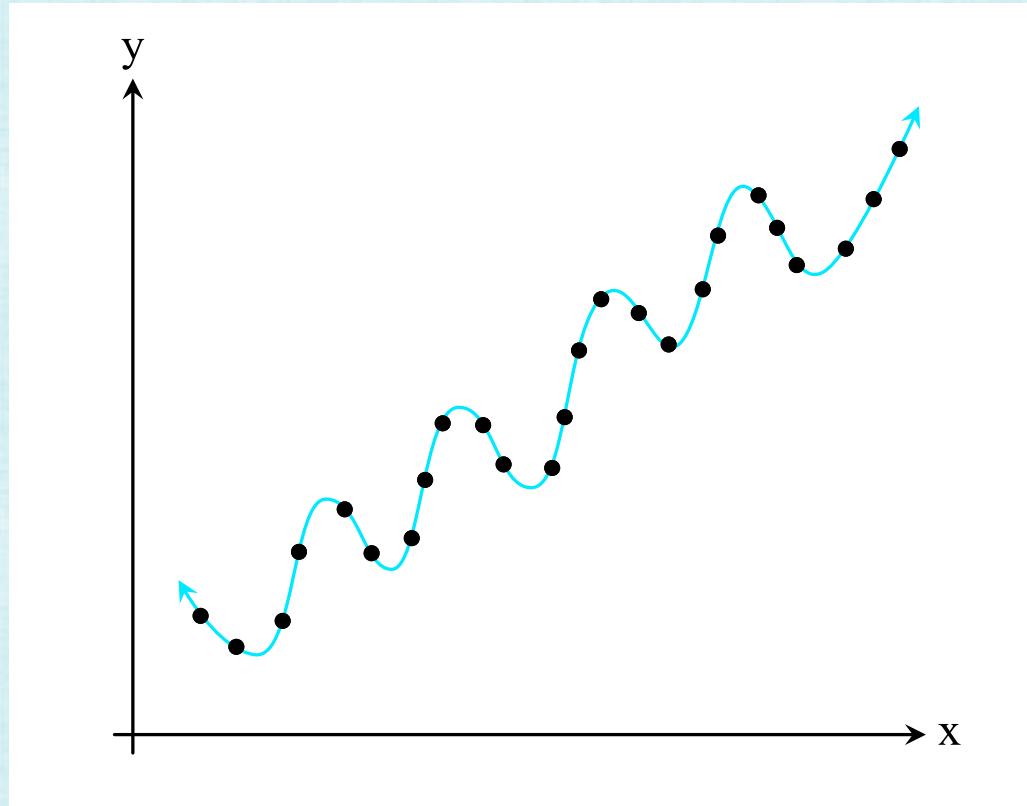
# Linear Regression

- One commonly fits a low order model to such data, while minimizing some metric of mis-interpolating or mis-representing the data



# Noise vs. Features

- But how can one differentiate between noise and features?



# Noise vs. Features

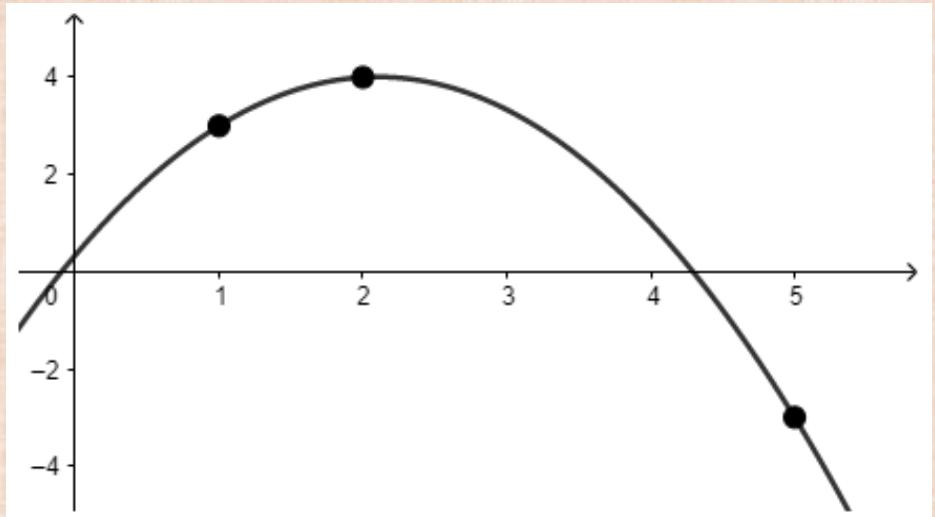
- When training a neural network, split the available data into 3 sets
- E.g., 80% training data, 10% model validation data, and 10% test data
- Training data is used to train the neural network
  - An interpolating function is fit to the training data (potentially overfitting it)
- When considering features vs. noise, overfitting, etc., model validation data is used to select the best model function or the best fitting strategy
  - Compare inference/prediction on model validation data to the known answers
- Finally, when disseminating results advocating the “best” validated model, inferencing on the test data gives some idea as to how well that validated model might generalize to unseen data
  - Competitions on unseen data have become a good way to stop “cheating” on test data

# Monomial Basis for Polynomial Interpolation

- Given  $m$  data points  $(x_i, y_i)$ , find the unique polynomial that passes through them:  $y = c_1 + c_2x + c_3x^2 + \cdots + c_m x^{m-1}$
- Write an equation for each data point, note that the equations are linear, and put into matrix form
- For example, consider  $(1,3), (2,4), (5, -3)$  and a quadratic polynomial

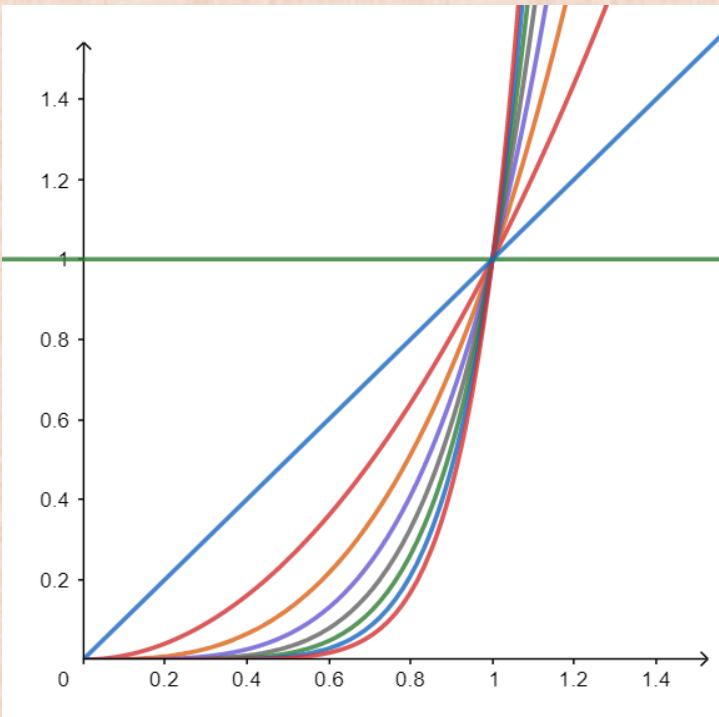
Then,  $\begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 5 & 25 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 3 \\ 4 \\ -3 \end{pmatrix}$  gives

$$\begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 1/3 \\ 7/2 \\ -5/6 \end{pmatrix} \text{ and } f(x) = \frac{1}{3} + \frac{7}{2}x - \frac{5}{6}x^2$$



# Monomial Basis for Polynomial Interpolation

- In general, solve  $Ac = y$  where  $A$  (the Vandermonde matrix) has a row for each data point of the form  $(1 \quad x_i \quad x_i^2 \quad \dots \quad x_i^{m-1})$

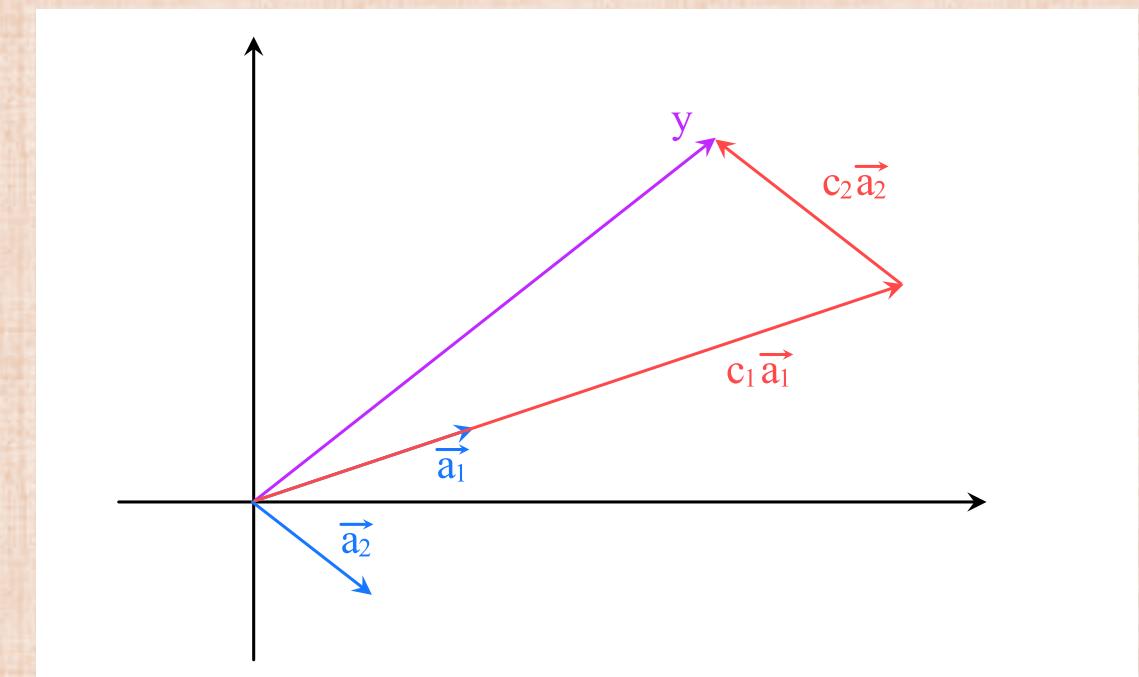
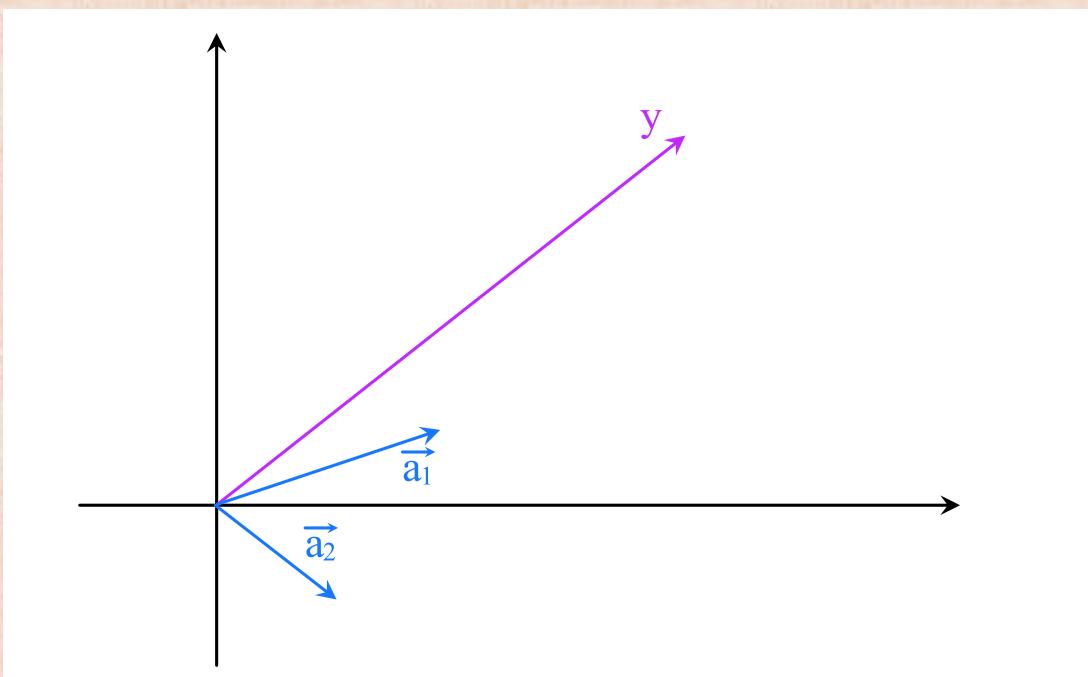


- Polynomials look more similar at higher powers
- This makes the rightmost columns of a Vandermonde matrix tend to become more parallel
  - Round-off errors and other numerical approximations exacerbate this
  - More parallel columns make the matrix less invertible, and thus it becomes more difficult to solve for the parameters  $c_k$
  - Too nearly parallel columns make the matrix **ill-posed** and unsolvable using a computer

$$f(x) = 1, x, x^2, x^3, x^4, x^5, x^6, x^7, x^8$$

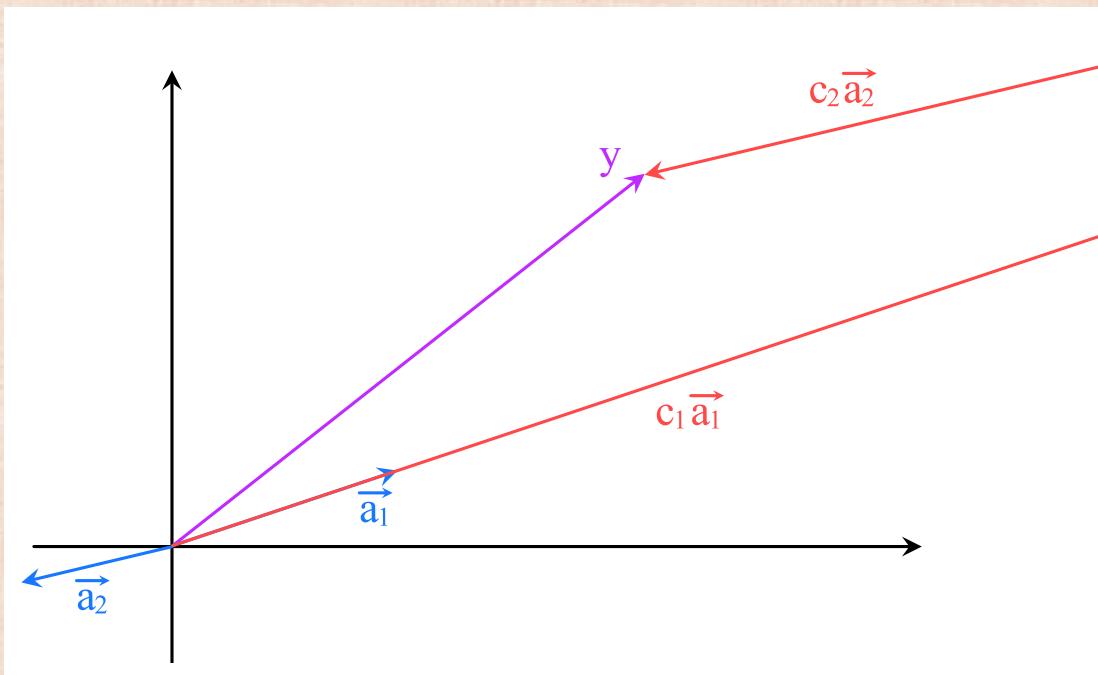
# Matrix Columns as Vectors

- Let the  $k$ -th column of  $A$  be vector  $a_k$ , so  $Ac = y$  is equivalent to  $\sum_k c_k a_k = y$
- That is, find a linear combination of the columns of  $A$  that gives the right hand side vector  $y$



# Matrix Columns as Vectors

- As columns become more parallel, the values of  $c$  become arbitrarily large, ill-conditioned, and prone to error



- In this example, the (red) column vectors go too far to the right and back in order to (fully) illustrate

# Singular Matrices

- If two columns of a matrix are parallel, they may be combined in an infinite number of ways while still obtaining the same result
  - Thus, the problem does not have a unique solution
- In addition, the  $n$  columns of  $A$  span at most an  $n - 1$  dimensional subspace
  - So, the range of  $A$  is at most  $n - 1$  dimensional
- If the right hand side vector is not contained in this  $n - 1$  dimensional subspace, then the problem has no solution
  - otherwise, there are infinite solutions

# Singular Matrices

- If any column of a matrix is a linear combination of other columns, they may be combined in an infinite number of ways while still obtaining the same result
  - Thus, the problem does not have a unique solution
- In addition, the  $n$  columns of  $A$  span at most an  $n - 1$  dimensional subspace
  - So, the range of  $A$  is at most  $n - 1$  dimensional
- If the right hand side vector is not contained in this  $n - 1$  dimensional subspace, then the problem has no solution
  - otherwise, there are infinite solutions

# Near Singular Matrices

- With limited numerical precision, one struggles to obtain accuracy when columns (or linear combinations of columns) are too parallel to each other
- That is, analytically invertible matrices may not be computationally invertible
- We use the concept of a condition number to describe how hard or easy it is to solve a problem computationally (on a computer)

# Approximation Errors

- **Modeling errors** – Parts of a problem under consideration might be ignored. E.g., when simulating solids/fluids, sometimes frictional/viscous effects are not included.
- **Empirical constants** – Some numbers are unknown, and measured with limited precision. Others may be known more accurately, but limited precision hinders the ability to express them. E.g. Avogadro's number, the speed of light in a vacuum, the charge on an electron, Planck's constant, Boltzmann's constant, pi, etc. (Note that the speed of light is 299792458 m/s exactly, so we are ok for double precision but not for single precision.)

# Approximation Errors

- **Rounding Errors:** Even integer calculations lead to floating point numbers, e.g.  $5/2=2.5$ , and floating point calculations frequently admit rounding errors, e.g.  $1./3.=.3333333\dots$  cannot be expressed on the computer. Machine precision is  $10^{-7}$  for single precision and  $10^{-16}$  for double precision.
- **Truncation errors** – Also called discretization errors. These occur in the mathematical approximation of an equation as opposed to an approximation of the physics (modeling errors). E.g. one (often) cannot take a derivative/integral exactly on a computer, and instead approximates them (recall Simpson's rule from Calculus).

# Approximation Errors

- **Inaccurate inputs** – Often, one is only concerned with part of a calculation, where a given set of inputs is used to produce outputs. Those inputs may have previously been subjected to any of the errors listed above, and thus may already have limited accuracy. This has implications for various algorithms. E.g., if inputs are only accurate to 4 decimal places, it probably doesn't make sense to carry out an algorithm to an accuracy of 8 decimal places.

# Computational Approach

- **Condition Number:** A problem is ill-conditioned if small changes in inputs lead to large changes in the outputs. Large condition numbers are bad (sensitive), and small condition numbers are good (insensitive). If the relative changes in the inputs and outputs are identical, the condition number is 1.
  - E.g. Near parallel columns in a matrix lead to a poor (and large) condition number!
- **Stability and Accuracy:** It is appropriate to aim to solve well-conditioned problems on a computer; then, stability and accuracy matter:
  - Stability refers to whether or not an algorithm can complete itself in any meaningful way. Unstable algorithms tend to give wildly varying (explosive) results, usually leading to NaN's.
  - Stability alone does not indicate that the problem has been solved. One also needs to be concerned with the size of the errors, which could still be enormous (e.g. no significant digits correct). Accuracy refers to how close an answer is to the correct solution.

# Computational Approach

- A problem should be well-posed before even considering it computationally
- Computational Approach:
  - 1) Conditioning - formulate a well-conditioned approach
  - 2) Stability - devise a stable algorithm
  - 3) Accuracy - make the algorithm as accurate as is warranted/practical

# Vector Norms (Carefully)

- Consider the norm of a vector:  $\|x\|_2 = \sqrt{x_1^2 + \cdots + x_m^2}$
- Straightforward algorithm:

```
for (i=1,m) sum+=x(i)*x(i); return sqrt(sum);
```
- This can overflow MAX\_FLOAT/MAX\_DOUBLE for large  $m$
- Safer algorithm:

```
find z=max(abs(x(i)))  
for (i=1,m) sum+=sqr(x(i)/z); return z*sqrt(sum);
```

# Quadratic Formula (Carefully)

- Consider  $.0501x^2 - 98.78x + 5.015 = 0$ 
  - To 10 digits of accuracy:  $x \approx 1971.605916$  and  $x \approx .05077069387$

- Using 4 digits of accuracy in the quadratic formula gives:

$$\frac{98.78+98.77}{.1002} = 1972 \quad \text{and} \quad \frac{98.78-98.77}{.1002} = .0998$$

- The **second root is** completely **wrong** (in the leading significant digit!)
- De-rationalize:  $\frac{-b \pm \sqrt{b^2-4ac}}{2a}$  to  $\frac{2c}{-b \mp \sqrt{b^2-4ac}}$

- Using 4 digits of accuracy in this de-rationalized quadratic formula gives:

$$\frac{10.03}{98.78-98.77} = 1003 \quad \text{and} \quad \frac{10.03}{98.78+98.77} = .05077$$

- Now the second root is fine, but the **first root is wrong!**
- Conclusion: use one formula for each root

# Quadratic Formula (Carefully)

- *Did you know that this was an issue?*
- Imagine debugging code with the correct quadratic formula implementation and getting zero digits of accuracy on a test case!
- The specific sequence of operations performed in solving the quadratic formula can result in large errors.
- The operations themselves are not (necessarily) dangerous, but the specific order [aka the algorithm] can be.
  - E.g. Many removable singularities ( $\frac{x^2-4}{x-2}$  near  $x = 2$ ,  $\frac{\sin x}{x}$  near  $x = 0$ , etc.) need to be carefully evaluated on the computer.

# Polynomial Interpolation (Carefully)

- Given basis functions  $\phi$  and unknowns  $c$ :

$$y = c_1\phi_1 + c_2\phi_2 + \cdots + c_n\phi_n$$

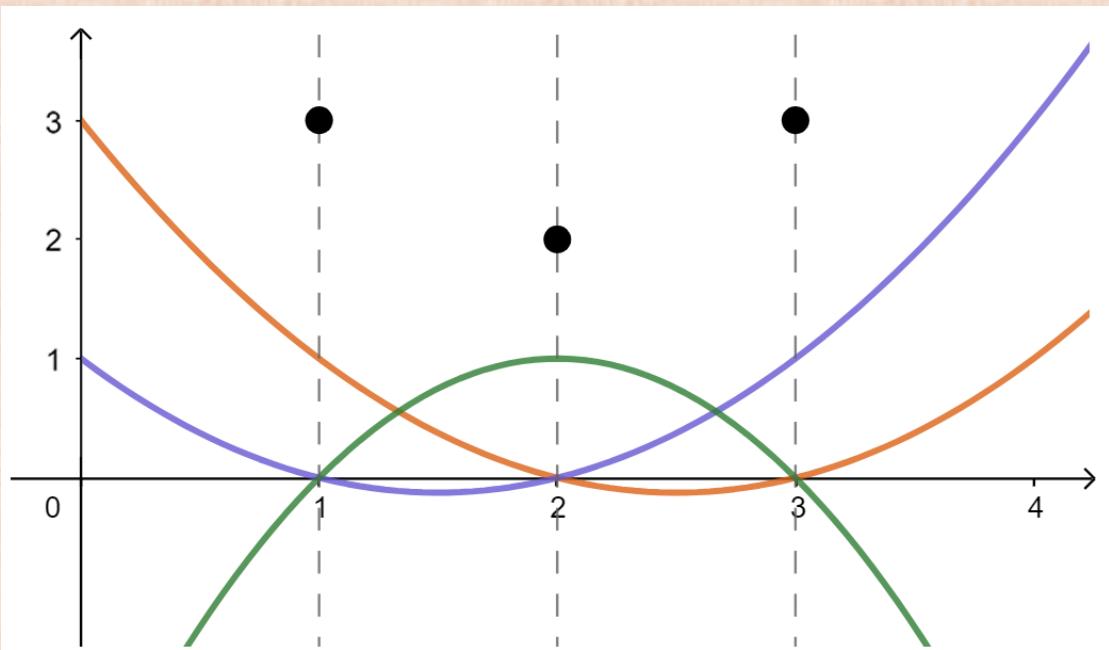
- Monomial basis:  $\phi_k(x) = x^{k-1}$
- The Vandermonde matrix may become near-singular and difficult to invert!

# Lagrange Basis for Polynomial Interpolation

- Basis functions:  $\phi_k(x) = \frac{\prod_{i \neq k} x - x_i}{\prod_{i \neq k} x_k - x_i}$
- Thus,  $\phi_k(x_k) = 1$
- Thus,  $\phi_k(x_i) = 0$  for  $i \neq k$
- As usual: write an equation for each point, note that the equations are linear, and put into matrix form
- Obtain  $Ac = y$  where  $A$  is the identity matrix (i.e.  $Ic = y$ ), so  $c = y$  trivially
- Easy to solve for  $c$ , but evaluation of the polynomial (with lots of terms) is expensive
  - i.e. inference is expensive

# Lagrange Basis for Polynomial Interpolation

- Consider data  $(1,3), (2,2), (3,3)$  with quadratic basis functions that are 1 at their corresponding data point and 0 at the other data points



- $\phi_1(x) = \frac{(x-2)(x-3)}{(1-2)(1-3)} = \frac{1}{2}(x-2)(x-3)$
- $\phi_1(1) = 1, \phi_1(2) = 0, \phi_1(3) = 0$
- $\phi_2(x) = \frac{(x-1)(x-3)}{(2-1)(2-3)} = -(x-1)(x-3)$
- $\phi_2(1) = 0, \phi_2(2) = 1, \phi_2(3) = 0$
- $\phi_3(x) = \frac{(x-1)(x-2)}{(3-1)(3-2)} = \frac{1}{2}(x-1)(x-2)$
- $\phi_3(1) = 0, \phi_3(2) = 0, \phi_3(3) = 1$

# Newton Basis for Polynomial Interpolation

- Basis functions:  $\phi_k(x) = \prod_{i=1}^{k-1} x - x_i$
- $Ac = y$  has a lower triangular  $A$  (as opposed to being dense or diagonal)
- Columns don't overlap, and it's not too expensive to evaluate/inference
- Can solve via a divided difference table:
  - Initially:  $f[x_i] = y_i$
  - Then, at each level, recursively:  $f[x_1, x_2, \dots, x_k] = \frac{f[x_2, x_3, \dots, x_k] - f[x_1, x_2, \dots, x_{k-1}]}{x_k - x_1}$
  - Finally:  $c_k = f[x_1, x_2, \dots, x_k]$
- As usual, high order polynomials still tend to be oscillatory
  - Using unequally spaced data points can help, e.g. Chebyshev points

# Summary: Polynomial Interpolation

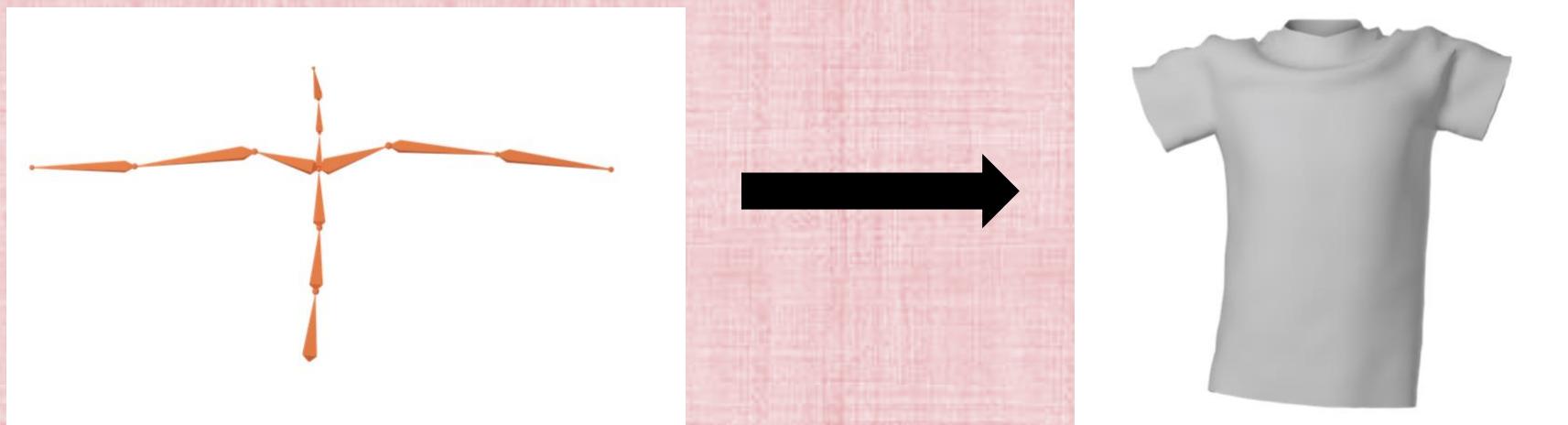
- Monomial/Lagrange/Newton basis all give the same exact unique polynomial
  - as one can see by multiplying out and collecting like terms
- But the representation used makes it easier/harder to both find the polynomial itself and subsequently evaluate the polynomial

# Representation Matters

- Consider: Divide CCX by VI
- As compared to: Divide 210 by 6
- See Chapter 15 on Representation Learning in the Deep Learning book

# Predict 3D Cloth Shape from Body Pose (Carefully)

- Input: pose parameters  $\theta$  are joint rotation matrices
  - 10 upper body joints with a  $3 \times 3$  rotation matrix for each gives a  $90D$  pose vector ( $30D$  when using quaternions)
  - global translation/rotation of root frame is ignored
- Output:  $3D$  cloth shape  $\varphi$ 
  - 3,000 vertices in a cloth triangle mesh gives a  $9,000D$  shape vector
- Function  $f: \mathbf{R}^{90} \rightarrow \mathbf{R}^{9000}$

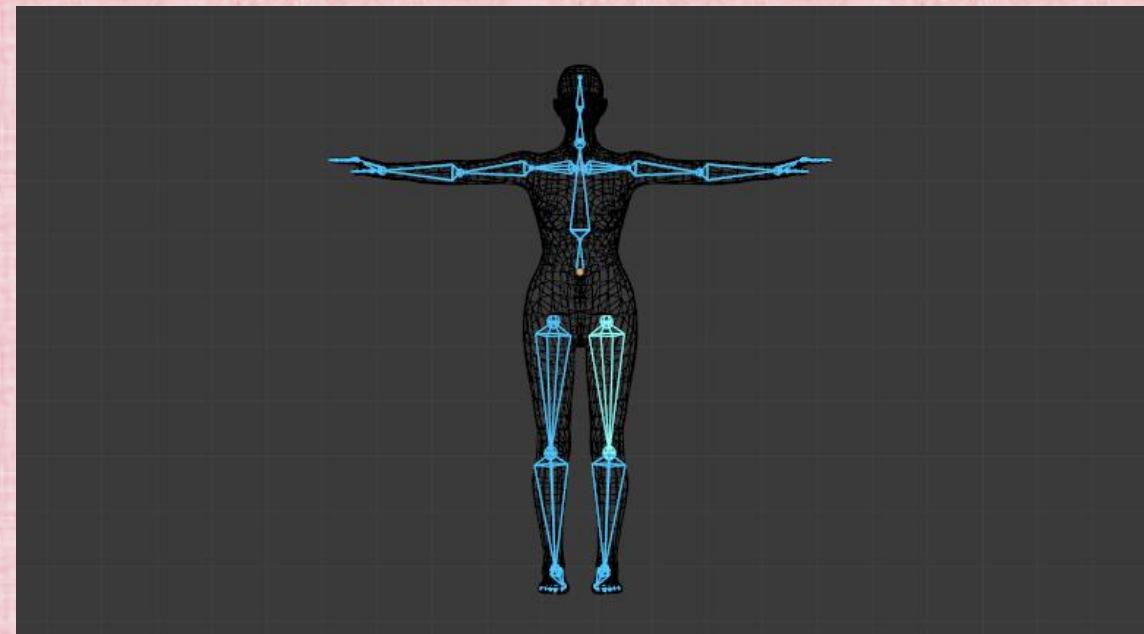


# Approach

- Given:  $m$  training data points  $(\theta_i, \varphi_i)$  generated from the true/approximated function  $\varphi_i = f(\theta_i)$ 
  - E.g. using physical simulation or computer vision techniques
- Goal: learn an  $\hat{f}$  that approximates  $f$ 
  - i.e.  $\hat{f}(\theta) = \hat{\varphi} \approx \varphi = f(\theta)$
- Issue: As joints rotate (rotation is highly nonlinear), cloth vertices move in complex nonlinear ways that are difficult to capture with a neural network
  - i.e. it is difficult to ascertain a suitable  $\hat{f}$
- How should the nonlinear rotations be handled?

# Aside: Procedural Skinning

- Deforms a body surface mesh to match a skeletal pose
  - well studied and widely used in graphics
- In the rest pose, associate each vertex of the body surface mesh with one or more nearby bones
- A weight from each bone dictates how much impact its position/orientation has on the vertex's position
- As the pose changes, bone changes dictate new positions for skin vertices



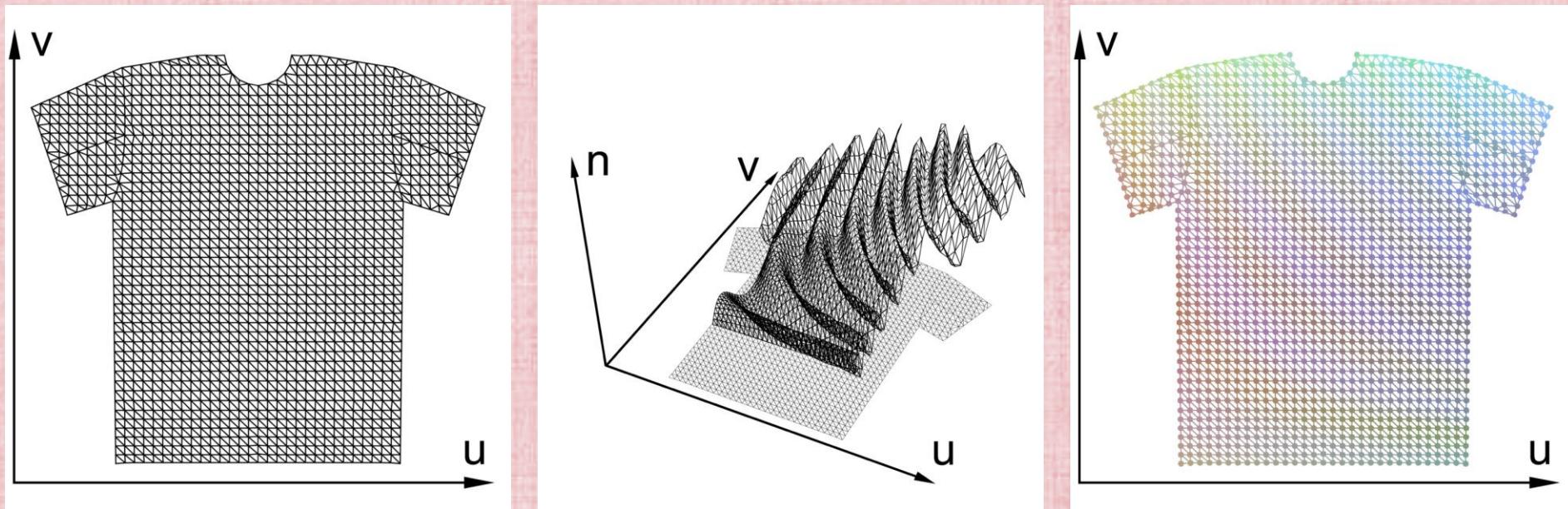
Credit: Blender website

# Leverage Procedural Skinning

- Leverage the plethora of prior work on procedural skinning to estimate the body surface mesh  $S$  based on pose parameters  $\theta$
- Then, represent the cloth mesh as offsets  $D(\theta)$  from the skinned mesh  $S(\theta)$
- Overall,  $\varphi = f(\theta) = S(\theta) + D(\theta)$ , where only  $D(\theta)$  needs to be learned
- The procedural skinning prior  $S(\theta)$  captures much of the nonlinearities, so that the remaining  $D(\theta)$  is a smoother function and thus easier to approximate/learn

# Pixel Based Cloth

- Assign  $(u, v)$  texture coordinates to the cloth triangle mesh
- Then, transfer the mesh into pattern/texture space (left)
- Store  $(u, v, n)$  offsets in the pattern/texture space (middle)
- Convert  $(u, v, n)$  offsets to RGB-triple color values or “pixels” (right)

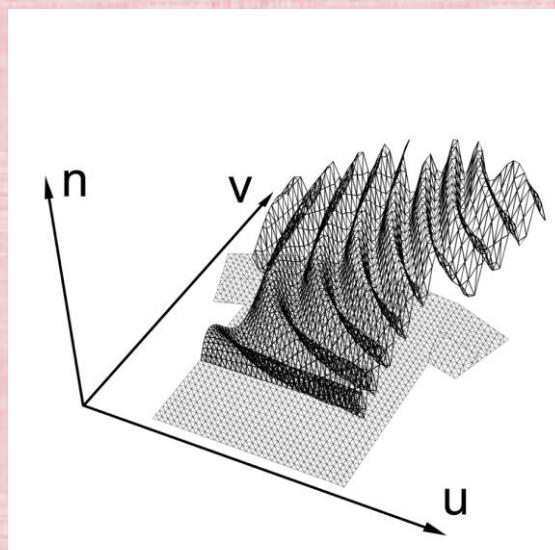
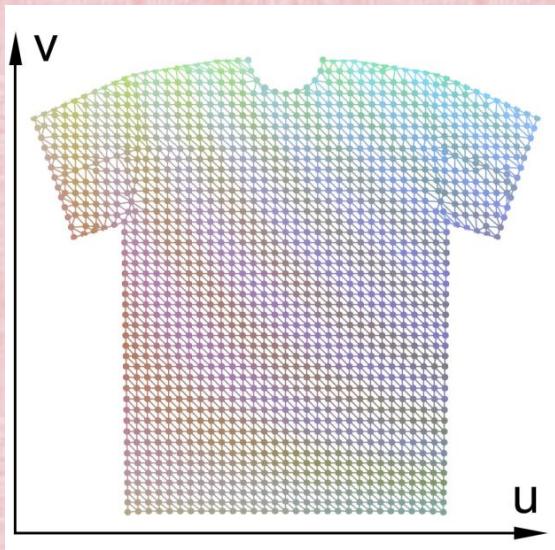


# Body Skinning of Cloth Pixels

- Shrink-wrap the cloth pixels (left) to the body triangle mesh (middle)
  - i.e. barycentrically embed cloth pixels to follow body mesh triangles
- As the body deforms, cloth pixels move with their parent triangles (right)
- Then, as a function of pose  $\theta$ , learn per-cloth-pixel  $(u, v, n)$  offsets  $D(\theta)$  from the skinned cloth pixels  $S(\theta)$  to the actual cloth mesh  $\varphi$

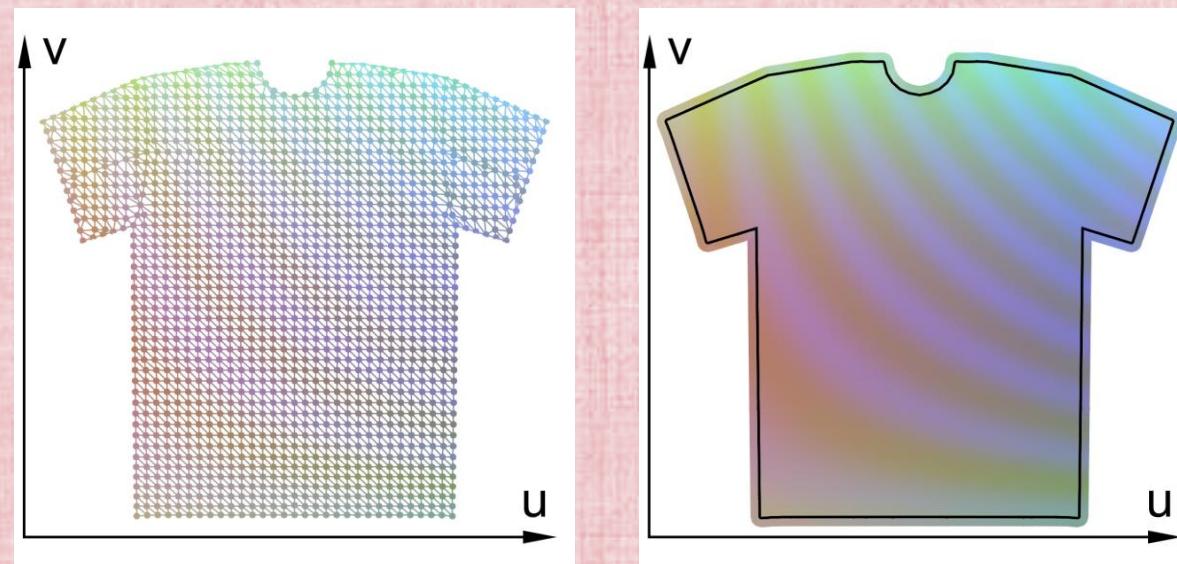


# RGB Values of Cloth Pixels Correspond to Offset Vectors



# Image Based Cloth

- Rasterize triangle vertex colors to standard 2D image pixels (in texture space)
- Function output becomes a (standard) 2D RGB image
- More continuous than cloth pixels (which have discrete mesh/graph topology)
- Then, can learn with Convolutional Neural Network (CNN) techniques



# Encode 3D Cloth Shapes as 2D Images

- For each pose in the training data, calculate per-vertex offsets and rasterize them into an image in pattern space
- Then, learn to predict an image from pose parameters, i.e. learn  $\hat{I}(\theta) \approx I(\theta)$
- Given an inferred  $\hat{I}(\theta)$ , interpolate to cloth pixels (vertices) and convert RGB values to offsets added to the skinned vertex positions:  $\hat{\varphi}(\theta) = S(\theta) + \psi(\hat{I}(\theta))$



# Unit 2

# Linear Systems

# Motivation

- “Matrices are bad, vector spaces are good”
  - That is, don’t think of matrices as a collection of numbers
  - Instead, think of columns as vectors in a high dimensional space
- We don’t have great intuition going from  $R^1$  to  $R^2$  to  $R^3$  to  $R^n$  (for large  $n$ )
- Thinking about vectors in high dimensional spaces is a good way of gaining intuition about what’s going on
- Linear algebra, as a mathematical area, contains a lot of machinery for dealing with, discussing, and gaining intuition about vectors in high dimensional spaces
- So, while we will cover the essentials of linear algebra, we will do it from the viewpoint of *understanding higher dimensional spaces*

# System of Linear Equations

- System of equations:  $3c_1 + 2c_2 = 6$  and  $-4c_1 + c_2 = 7$
- Matrix form:  $\begin{pmatrix} 3 & 2 \\ -4 & 1 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} 6 \\ 7 \end{pmatrix}$  or  $Ac = b$
- Given  $A$  and  $b$ , determine  $c$
- Theoretically, there is a unique solution, no solution, or infinite solutions
- Ideally, software would determine whether there was a unique solution, no solution, or infinite solutions. In the last case, it would list a parameterized family of solutions. Unfortunately, this is difficult to accomplish.
- Note: in this class,  $x$  will be typically be used for **data**, and  $c$  will typically be used for **unknowns** (such as for the unknown parameters of a neural network)

# “Zero”

- One of the basic issues that has to be confronted is the concept of “zero”
- When dealing with large numbers (e.g. Avogadro’s number:  $6.022e23$ ) zero can be quite large
  - E.g.  $6.022e23 - 1e7 = 6.022e23$  in double precision, making  $1e7$  behave like “zero”
- When dealing with small numbers (e.g.  $1e-23$ ), “zero” is much smaller
  - In this case, on the order of  $1e-39$  in double precision
- Mixing big and small numbers often wreaks havoc on algorithms
- So, we typically non-dimensionalize and normalize to make equations  $O(1)$  as opposed to  $O(\text{“big”})$  or  $O(\text{“small”})$

# Row/Column Scaling

- Consider: 
$$\begin{pmatrix} 3e6 & 2e10 \\ 1e-4 & 0 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} 5e10 \\ 6 \end{pmatrix}$$
- Row Scaling - divide first row by  $1e10$  to obtain:  
$$\begin{pmatrix} 3e-4 & 2 \\ 1e-4 & 0 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} 5 \\ 6 \end{pmatrix}$$
- Column Scaling - define a new variable  $c_3 = (1e-4)c_1$  to obtain:  
$$\begin{pmatrix} 3 & 2 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} c_3 \\ c_2 \end{pmatrix} = \begin{pmatrix} 5 \\ 6 \end{pmatrix}$$
- Final result is much easier to treat with finite precision arithmetic
- Solve for  $c_3$  and  $c_2$ , and then  $c_1 = (1e4)c_3$

# Transpose and Symmetry

- Elements of a matrix are often referred to by their row and column
- For example,  $a_{ik}$  is the element of matrix  $A$  in row  $i$  and column  $k$

- Transpose swaps the row and column of every entry

- $A^T$  moves element  $a_{ik}$  to row  $k$  column  $i$  (and vice versa)

- The matrix size changes when it's non-square:  
$$\begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}^T = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

Symmetric Matrices have  $A^T = A$  meaning that  $a_{ik} = a_{ki}$  for all  $i$  and  $k$

# Square Matrix

- A size  $mxn$  matrix has  $m$  rows and  $n$  columns
- For now, let's consider square  $nxn$  matrices
- We will consider (non-square) rectangular matrices with  $m \neq n$  a bit later

# Solvability

- Singular –  $A$  is singular when it is not invertible (does not have an inverse)
- Various ways of showing this:
  - At least one column is linearly dependent on others (as we have seen before)
  - The determinant is zero:  $\det A = 0$
  - $A$  has a nonempty null space, i.e.  $\exists c \neq 0$  with  $Ac = 0$
- Rank - maximum number of linearly independent columns
- Singular matrices have rank  $< n$  (the # of columns), i.e. rank-deficient, and have either no solution or infinite solutions
- A nonsingular square matrix has an inverse:  $AA^{-1} = A^{-1}A = I$ 
  - so  $Ac = b$  can be solved for  $c$  via  $c = A^{-1}b$
- *Note: we typically do not compute the inverse, but instead have a solution algorithm that exploits its existence*

# Matrices as Vectors (an example)

- Recall  $Ac = \sum_k c_k a_k$  where the  $a_k$  are the columns of  $A$
- Consider  $Ac = 0$  or  $\sum_k c_k a_k = 0$
- If one column is a linear combination of others, then the linear combination weights can be used to obtain  $Ac = 0$  with  $c$  nonzero
  - This nonzero  $c$  is in the null space of  $A$ , and  $A$  is singular
- Conversely, if the only solution to  $Ac = 0$  is  $c$  identically 0, then no column is linearly dependent on the others
  - Thus  $A$  is nonsingular

# Diagonal Matrices

- All off-diagonal entries are 0
- Equations are decoupled, and easy to solve
- E.g.  $\begin{pmatrix} 5 & 0 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} 10 \\ -1 \end{pmatrix}$  has  $5c_1 = 10$  and  $2c_2 = -1$  so  $c_1 = 2$  and  $c_2 = -.5$
- A zero on the diagonal indicates a singular system, which has no solution (e.g.  $0c_1 = 10$ ) or infinite solutions (e.g.  $0c_1 = 0$ )
- The determinant of a diagonal matrix is obtained by multiplying all the diagonal elements together
- Thus, a 0 on the diagonal implies a zero determinant and a singular matrix

# Upper Triangular Matrices

- All entries below the diagonal are 0
  - Nonsingular when the diagonal elements are all nonzero
    - Determinant is obtained by multiplying all the diagonal elements together
  - Solve via back substitution
- 
- E.g. consider  $\begin{pmatrix} 5 & 3 & 1 \\ 0 & 1 & -1 \\ 0 & 0 & 5 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 10 \\ 10 \end{pmatrix}$
  - Start at the bottom with  $5c_3 = 10$  or  $c_3 = 2$ , and move upwards one row at a time. Next,  $c_2 - c_3 = 10$  or  $c_2 - 2 = 10$  or  $c_2 = 12$ . Then,  $5c_1 + 3c_2 + c_3 = 0$  or  $5c_1 + 3(12) + 2 = 0$  or  $c_1 = -38/5 = -7.6$

# Lower Triangular Matrices

- All entries above the diagonal are 0
- Nonsingular when the diagonal elements are all nonzero
  - Determinant is obtained by multiplying all the diagonal elements together
- Solve via forward substitution

- E.g. consider  $\begin{pmatrix} 5 & 0 & 0 \\ -1 & 1 & 0 \\ 1 & 3 & 5 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 10 \\ 10 \\ 0 \end{pmatrix}$
- Start at the top with  $5c_1 = 10$  or  $c_1 = 2$ , and move downwards one row at a time. Next,  $-c_1 + c_2 = 10$  or  $-2 + c_2 = 10$  or  $c_2 = 12$ . Then,  $c_1 + 3c_2 + 5c_3 = 0$  or  $2 + 36 + 5c_3 = 0$  or  $c_3 = -38/5 = -7.6$

# Elimination Matrix

- Standard basis vectors:  $\hat{e}_i = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$  with a 1 in the  $i$ -th row/entry
- Given  $\begin{pmatrix} a_{1k} \\ \vdots \\ a_{ik} \\ a_{i+1,k} \\ \vdots \\ a_{mk} \end{pmatrix}$ , define  $m_{ik} = \frac{1}{a_{ik}} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ a_{i+1,k} \\ \vdots \\ a_{mk} \end{pmatrix}$  and  $M_{ik} = I_{mxm} - m_{ik} \hat{e}_i^T$
- $M_{ik}$  is a size  $mxm$  elimination matrix that subtracts multiples of row  $i$  from rows  $> i$  in order to create zeroes in column  $k$

# Elimination Matrix

- Let  $a_1 = \begin{pmatrix} 2 \\ 4 \\ -2 \end{pmatrix}$
- $M_{11} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} - \frac{1}{2} \begin{pmatrix} 0 \\ 4 \\ -2 \end{pmatrix} (1 \quad 0 \quad 0) = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$  and  $M_{11}a_1 = \begin{pmatrix} 2 \\ 0 \\ 0 \end{pmatrix}$
- $M_{21} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} - \frac{1}{4} \begin{pmatrix} 0 \\ 0 \\ -2 \end{pmatrix} (0 \quad 1 \quad 0) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1/2 & 1 \end{pmatrix}$  and  $M_{21}a_1 = \begin{pmatrix} 2 \\ 4 \\ 0 \end{pmatrix}$

# Elimination Matrix Inverse

- Inverse of an elimination matrix is  $L_{ik} = M_{ik}^{-1} = I_{m \times m} + m_{ik} \hat{e}_i^T$
- $L_{ik}$  is a size  $m \times m$  elimination matrix that adds multiples of row  $i$  to rows  $> i$  in order to reverse the effect of  $M_{ik}$
- $L_{11} = M_{11}^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & 0 & 1 \end{pmatrix}$
- $L_{21} = M_{21}^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1/2 & 1 \end{pmatrix}$

# Combining Elimination Matrices

- $M_{i_1 k_1} M_{i_2 k_2} = I - m_{i_1 k_1} \hat{e}_{i_1}^T - m_{i_2 k_2} \hat{e}_{i_2}^T$  when  $i_1 < i_2$  but not when  $i_1 > i_2$

$$M_{11} M_{21} = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & 1/2 & 1 \end{pmatrix}$$

- $L_{i_1 k_1} L_{i_2 k_2} = I + m_{i_1 k_1} \hat{e}_{i_1}^T + m_{i_2 k_2} \hat{e}_{i_2}^T$  when  $i_1 < i_2$  but not when  $i_1 > i_2$

$$L_{11} L_{21} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & -1/2 & 1 \end{pmatrix}$$

# Gaussian Elimination

- Consider  $\begin{pmatrix} 2 & 4 & -2 \\ 4 & 9 & -3 \\ -2 & -3 & 7 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 8 \\ 10 \end{pmatrix}$
- $M_{11}A = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 & 4 & -2 \\ 4 & 9 & -3 \\ -2 & -3 & 7 \end{pmatrix} = \begin{pmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 1 & 5 \end{pmatrix}$  and  $M_{11}b = \begin{pmatrix} 2 \\ 4 \\ 12 \end{pmatrix}$
- $M_{22}M_{11}A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 1 & 5 \end{pmatrix} = \begin{pmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 0 & 4 \end{pmatrix}$  and  $M_{22}M_{11}b = \begin{pmatrix} 2 \\ 4 \\ 8 \end{pmatrix}$
- Then, solve the upper triangular  $\begin{pmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 0 & 4 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \\ 8 \end{pmatrix}$  via back substitution

# LU Factorization

- Gaussian Elimination gives an upper triangular  $U = M_{n-1,n-1} \cdots M_{22}M_{11}A$
- Using inverses,  $A = L_{11}L_{22} \cdots L_{n-1,n-1}M_{n-1,n-1} \cdots M_{22}M_{11}A = L_{11}L_{22} \cdots L_{n-1,n-1}U$
- Since  $L_{i_1 i_1} L_{i_2 i_2} = I + m_{i_1 i_1} \hat{e}_{i_1}^T + m_{i_2 i_2} \hat{e}_{i_2}^T$  when  $i_1 < i_2$ ,  $L = L_{11}L_{22} \cdots L_{n-1,n-1}$  is lower triangular and  $A = LU$

- Here  $L = L_{11}L_{22} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & 1 & 1 \end{pmatrix}$

$$A = \begin{pmatrix} 2 & 4 & -2 \\ 4 & 9 & -3 \\ -2 & -3 & 7 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 0 & 4 \end{pmatrix} = LU$$

# LU Factorization

- Factorizing  $A = LU$  facilitates solving  $Ac = b$
- In order to solve  $Lc = b$ , define an auxiliary variable  $\hat{c} = Uc$
- First, solve  $L\hat{c} = b$  for  $\hat{c}$  via forward substitution
- Second, solve  $Uc = \hat{c}$  for  $c$  via back substitution
- Note: the LU factorization is only computed once, and then can be used afterwards on many right hand side ( $b$ ) vectors

# Pivoting

- $A = \begin{pmatrix} 0 & 4 \\ 4 & 9 \end{pmatrix}$  requires division by **zero** in order to create  $M_{11}$
- (Partial) Pivoting - swap rows to use the largest (magnitude) element in the column under consideration
  - Don't forget to swap the right hand side  $b$  too
- Full Pivoting swap rows and columns to use **the largest possible element**
  - Don't forget to change the order of the unknowns  $c$
- When considering column  $k$ , can only swap with rows/columns  $\geq k$

# Permutation Matrix

- Constructed by switching the 2 rows of  $I$  that one wants swapped
- E.g.  $P_{13} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$ , and  $P_{13}A$  swaps the first and third rows of  $A$
- Permutation matrices are their own inverses (swapping again restores the rows)
- Switching rows  $i_1$  and  $i_2$  moves a 1 from  $a_{i_1 i_1}$  to  $a_{i_2 i_1}$  as well as from  $a_{i_2 i_2}$  to  $a_{i_1 i_2}$ , preserving symmetry (i.e.  $P_{i_1 i_2}^T = P_{i_1 i_2}$ )
- To swap the first and third unknowns:  $Ac = AP_{13}P_{13}c = (AP_{13})(P_{13}c)$  where  $P_{13}c$  swaps the unknowns and  $AP_{13}$  swaps the columns (to see this, consider  $(AP_{13})^{TT} = (P_{13}A^T)^T$  which swaps the rows of  $A^T$ )

# Full Pivoting

- Let  $P_{r_i}$  be the permutation matrix that (potentially) switches row  $i$  with a row  $> i$
- Let  $P_{c_k}$  be the permutation matrix that (potentially) switches column  $k$  with a col  $> k$
- Then full pivoting can be written as:

$$(M_{n-1,n-1}P_{r_{n-1}} \cdots M_{22}P_{r_2}M_{11}P_{r_1}AP_{c_1}P_{c_2} \cdots P_{c_{n-1}})(P_{c_{n-1}} \cdots P_{c_2}P_{c_1}c)$$

- Once known,  $P_r = P_{r_{n-1}} \cdots P_{r_2}P_{r_1}$  and  $P_c = P_{c_{n-1}} \cdots P_{c_2}P_{c_1}$  can be used to do all the permutations ahead of time (the resulting matrix doesn't require pivoting)
- $Ac = b$  becomes  $(P_rAP_c^T)(P_c c) = P_r b$  or  $A_P c_P = b_P$ ; then,  $A_P = L_P U_P$  can be computed without pivoting
- Subsequently, given any right hand side  $b$ , solve  $L_P U_P c_P = P_r b$  to find  $c_P$  using forward/back substitution, and then  $c = P_c^T c_P$

# Sparsity

- Most large matrices (of interest) operate on variables that only interact with a sparse set of other variables
- This makes the matrix sparse (as opposed to dense), i.e. most entries are identically 0
- However, the inverse of a sparse matrix can contain an unwieldy amount of non-zero entries
- E.g. the 3D Poisson equation on a relatively small  $100^3$  Cartesian grid has an unknown for each of the  $10^6$  grid points
- For each unknown, the discretized Poisson equation depends on the unknown itself and its 6 immediate Cartesian grid neighbors
- Thus, the size  $10^6 \times 10^6$  matrix has only  $7 \times 10^6$  nonzero entries
- But, the inverse potentially has  $10^{12}$  nonzero entries!

# Computing the Inverse

- When  $A$  is relatively small (and dense), one might compute  $A^{-1}$
- Since  $AA^{-1} = I$ , the solution  $c_k$  to  $Ac_k = \hat{e}_k$  is the  $k$ -th column of  $A^{-1}$
- First, compute  $A_P = L_P U_P$  as usual
- Then, solve  $Ac_k = \hat{e}_k$  repeatedly ( $n$  times, once for each column)

# Unit 3

# Understanding Matrices

# Eigensystems

- Eigenvalues - special directions  $v_k$  in which a matrix only applies scaling
- Eigenvalues - the amount  $\lambda_k$  of that scaling
- Right Eigenvectors (or simply eigenvectors) satisfy  $A v_k = \lambda_k v_k$ 
  - **Eigenvalues represent directions**, so  $A(\alpha v_k) = \lambda_k(\alpha v_k)$  is also true for all  $\alpha$
- Left Eigenvectors satisfy  $u_k^T A = \lambda_k u_k^T$  (or  $A^T u_k = \lambda_k u_k$ )
- Diagonal matrices have eigenvalues on the diagonal, and eigenvectors  $\hat{e}_k$

$$\begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = 2 \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = 3 \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

- Upper/lower triangular matrices also have eigenvalues on the diagonal

$$\begin{pmatrix} 2 & 1 \\ 0 & 3 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = 2 \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} 2 & 1 \\ 0 & 3 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = 3 \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

# Complex Numbers

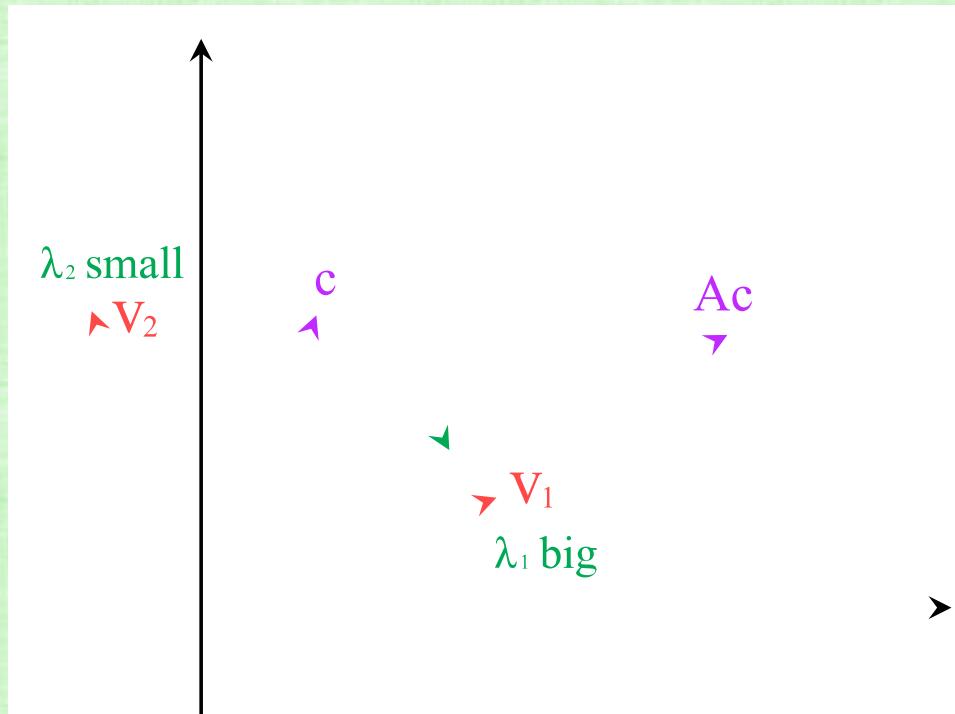
- Complex numbers may appear in both eigenvalues and eigenvectors

$$\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ i \end{pmatrix} = i \begin{pmatrix} 1 \\ i \end{pmatrix}$$

- Recall: complex conjugate:  $(a + bi)^* = a - bi$
- Hermitian Matrix:  $A^{*T} = A$  (often,  $A^{*T}$  is written as  $A^H$ )
  - $A\boldsymbol{\nu} = \lambda\boldsymbol{\nu}$  implies  $(A\boldsymbol{\nu})^{*T} = (\lambda\boldsymbol{\nu})^{*T}$  or  $\boldsymbol{\nu}^{*T} A = \lambda^* \boldsymbol{\nu}^{*T}$
  - Using this,  $A\boldsymbol{\nu} = \lambda\boldsymbol{\nu}$  implies  $\boldsymbol{\nu}^{*T} A \boldsymbol{\nu} = \boldsymbol{\nu}^{*T} \lambda \boldsymbol{\nu}$  or  $\lambda^* \boldsymbol{\nu}^{*T} \boldsymbol{\nu} = \lambda \boldsymbol{\nu}^{*T} \boldsymbol{\nu}$  or  $\lambda^* = \lambda$
  - Thus, Hermitian matrices have  $\lambda \in \mathbb{R}$  (no complex eigenvalues)
- Symmetric real-valued matrices have real-valued eigenvalues/eigenvectors
  - However, complex eigenvectors work too, e.g.  $A(\alpha\boldsymbol{\nu}_k) = \lambda_k(\alpha\boldsymbol{\nu}_k)$  with  $\alpha$  complex

# Spatial Deformation

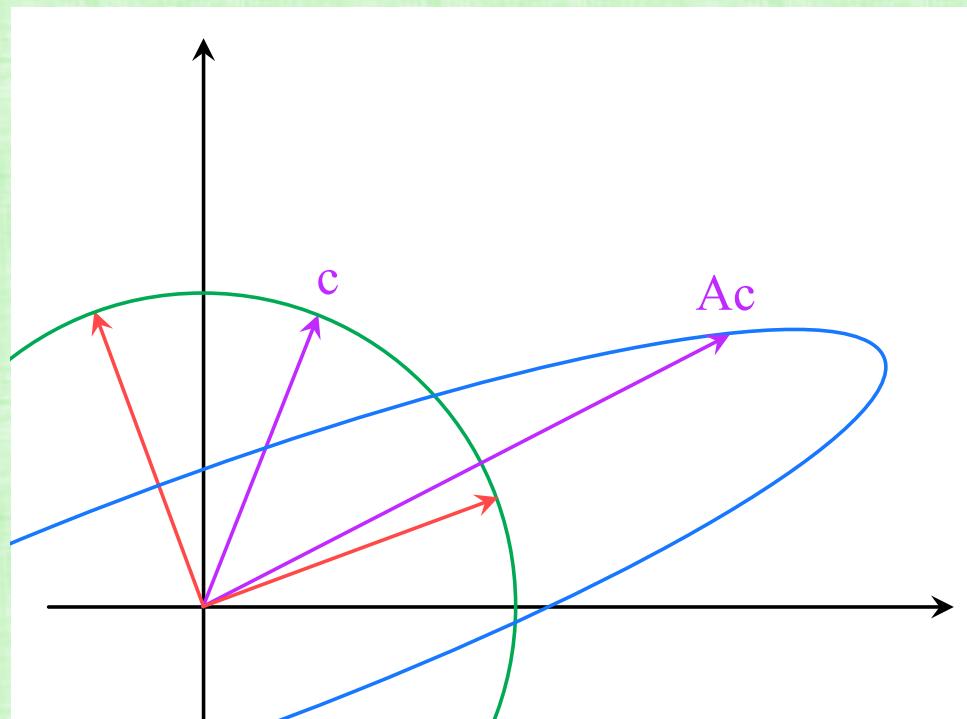
- Let  $c = \sum_k \alpha_k v_k$ , so that  $Ac = \sum_k \alpha_k A v_k = \sum_k (\alpha_k \lambda_k) v_k$
- Then,  $A$  tilts  $c$  away from directions with smaller eigenvalues and towards directions with larger eigenvalues



- Large  $\lambda_k$  stretch components in their associated  $v_k$  directions
- Small  $\lambda_k$  squish components in their associated  $v_k$  directions
- Negative  $\lambda_k$  flip the sign of components in their associated  $v_k$  directions

# Spatial Deformation

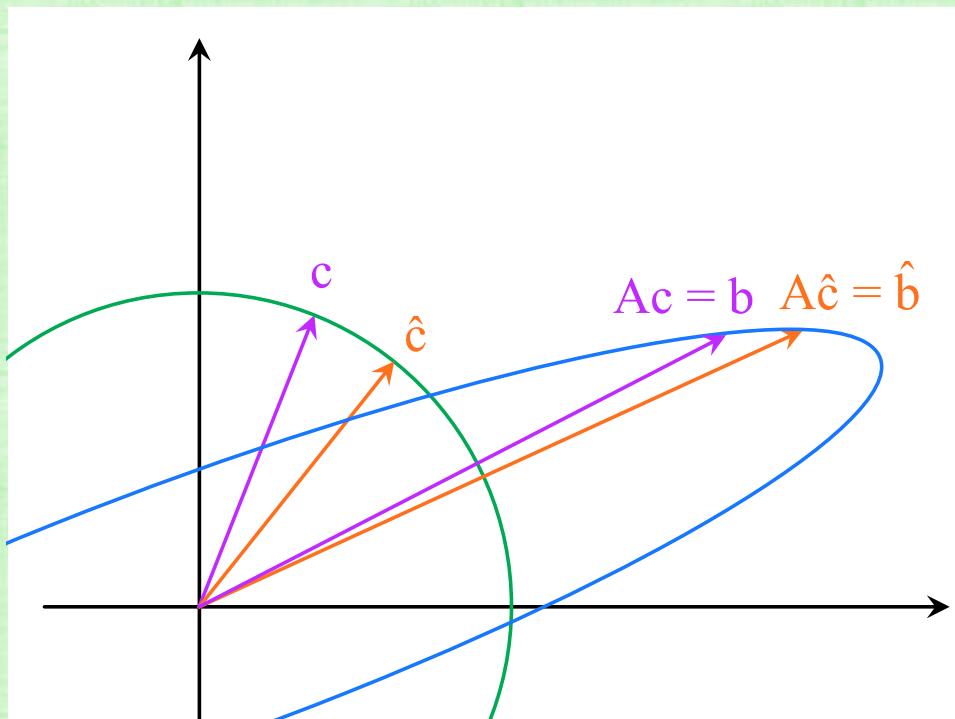
- Consider every point on the unit circle (green) as a vector  $c = \sum_k \alpha_k v_k$ , and remap each point via  $Ac = \sum_k (\alpha_k \lambda_k) v_k$



- The remapped shape (blue) is more elliptical than the original circle (green)
- The circle is stretched along the axis with the larger eigenvalue, and compressed along the axis with the smaller eigenvalue
- The larger the ratio of eigenvalues, the more elliptical the circle becomes
- This is true for all circles (representing all points in the plane)

# Solving Linear Systems

- Perturb the right hand side from  $b$  to  $\hat{b}$ , and solve  $A\hat{c} = \hat{b}$  to find  $\hat{c}$
- Note how  $c$  and  $\hat{c}$  are more separated than  $b$  and  $\hat{b}$ , i.e. the solution is more perturbed than the right hand side is



- Small changes in the right hand side lead to larger changes in the solution
- **Small algorithmic errors are also amplified**, since they change  $A^{-1}b$  to  $\hat{A}^{-1}\hat{b}$  which is similar to changing  $A^{-1}b$  to  $A^{-1}\hat{b}$
- The amount of amplification is proportional to the ratio of the eigenvalues

# Preconditioning

- Suppose  $A$  has large eigenvalue ratios that make  $Ac = b$  difficult to solve
- Suppose one had an approximate guess for the inverse, i.e an  $\hat{A}^{-1} \approx A^{-1}$
- Then, transform  $Ac = b$  into  $\hat{A}^{-1}Ac = \hat{A}^{-1}b$  or  $\hat{I}c = \tilde{b}$ 
  - Typically, a bit more involved than this, but conceptually the same
- $\hat{I}$  is not the identity, so there is still more work to do in order to find  $c$
- However,  $\hat{I}$  has eigenvalues with similar magnitudes (clusters work too), making  $\hat{I}c = \tilde{b}$  far easier to solve than a poorly conditioned  $Ac = b$

Preconditioning works GREAT!

- It is best to re-scale ellipsoids along eigenvector axes, but scaling along the coordinate axes (diagonal/Jacobi preconditioning) can work well too

# Rectangular Matrices (Rank)

- An  $m \times n$  rectangular matrix has  $m$  rows and  $n$  columns
- (Note: these comments also hold for square matrices with  $m = n$ )
- The columns span a space, and the unknowns are weights on each column (recall  $Ac = \sum_k c_k a_k$  )
- A matrix with  $n$  columns has maximum rank  $n$
- The actual rank depends on how many of the columns are linearly independent from one another
- Each column has length  $m$  (which is the number of rows)
- Thus, the columns live in an  $m$  dimensional space, and at best can span that whole space
- That is, there is a maximum of  $m$  independent columns (that could exist)
- Overall, a matrix **at most** has rank equal to the minimum of  $m$  and  $n$
- Both considerations are based on looking at the columns (which are scaled by the unknowns)

# Rows vs. Columns

- One can find discussions on rows, row spaces, etc. that are used for various purposes
- Although these are fine discussions in regards to matrices/mathematics, they are unnecessary for an intuitive understanding of high dimensional vector spaces (and as such can be ignored)
- The number of columns is identical to number of variables, which depends on the parameters of the problem
  - E.g. the unknown parameters that govern a neural network architecture
- The number of rows depends on the amount of data used, and adding/removing data does not intrinsically effect the nature of the problem
  - E. g. it does not change the network architecture, but merely perturbs the ascertained values of the unknown parameters

# Singular Value Decomposition (SVD)

- Factorization of any size  $m \times n$  matrix:  $A = U\Sigma V^T$
- $\Sigma$  is  $m \times n$  diagonal with non-negative diagonal entries (called singular values)
- $U$  is  $m \times m$  orthogonal,  $V$  is  $n \times n$  orthogonal (their columns are called singular vectors)
  - Orthogonal matrices have orthonormal columns (an orthonormal basis), so their transpose is their inverse. They preserve inner products, and thus are rotations, reflections, and combinations thereof
  - If  $A$  has complex entries, then  $U$  and  $V$  are unitary (conjugate transpose is their inverse)
- Introduced and rediscovered many times: Beltrami 1873, Jordan 1875, Sylvester 1889, Autonne 1913, Eckart and Young 1936. Pearson introduced principal component analysis (PCA) in 1901, which uses SVD. Numerical methods by Chan, Businger, Golub, Kahan, etc.

# (Rectangular) Diagonal Matrices

- All off-diagonal entries are 0
  - Diagonal entries are  $a_{kk}$ , and off diagonal entries are  $a_{ki}$  with  $k \neq i$
- E.g.  $\begin{pmatrix} 5 & 0 \\ 0 & 2 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 10 \\ -1 \\ \alpha \end{pmatrix}$  has  $5c_1 = 10$  and  $2c_2 = -1$ , so  $c_1 = 2$  and  $c_2 = -.5$ 
  - Note that  $\alpha \neq 0$  imposes a “no solution” condition (even though  $c_1$  and  $c_2$  are well-specified)
- E.g.  $\begin{pmatrix} 5 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 10 \\ -1 \\ 0 \end{pmatrix}$  has  $5c_1 = 10$  and  $2c_2 = -1$ , so  $c_1 = 2$  and  $c_2 = -.5$ 
  - Note that there are “infinite solutions” for  $c_3$  (even though  $c_1$  and  $c_2$  are well-specified)
- A zero on the diagonal indicates a singular system, which has either no solution (e.g.  $0c_1 = 10$ ) or infinite solutions (e.g.  $0c_1 = 0$ )

# Singular Value Decomposition (SVD)

- $A^T A = V \Sigma^T U^T U \Sigma V^T = V (\Sigma^T \Sigma) V^T$ , so  $(A^T A)v = \lambda v$  gives  $(\Sigma^T \Sigma)(V^T v) = \lambda(V^T v)$
- $\Sigma^T \Sigma$  is  $n \times n$  diagonal with eigenvectors  $\hat{e}_k$ , so  $\hat{e}_k = V^T v$  and  $v = V \hat{e}_k$
- That is, the columns of  $V$  are the eigenvectors of  $A^T A$
- $AA^T = U \Sigma V^T V \Sigma^T U^T = U (\Sigma \Sigma^T) U^T$ , so  $(AA^T)v = \lambda v$  gives  $(\Sigma \Sigma^T)(U^T v) = \lambda(U^T v)$
- $\Sigma \Sigma^T$  is  $m \times m$  diagonal with eigenvectors  $\hat{e}_k$ , so  $\hat{e}_k = U^T v$  and  $v = U \hat{e}_k$
- That is, the columns of  $U$  are the eigenvectors of  $AA^T$
- When  $m \neq n$ , either  $\Sigma^T \Sigma$  or  $\Sigma \Sigma^T$  is larger and contains extra zeros on the diagonal
- Their other diagonal entries are the squares of the singular values
- That is, the singular values are the (non-negative) square roots of the non-extra eigenvalues of  $A^T A$  and  $AA^T$
- Note that both  $A^T A$  and  $AA^T$  are symmetric positive semi-definite, and thus easy to work with
- E.g. symmetry means their eigensystem (and thus the SVD) has no complex numbers when  $A$  doesn't

# Example (Tall Matrix)

- Consider size  $4 \times 3$  matrix  $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix}$
- Label the columns  $a_1 = \begin{pmatrix} 1 \\ 4 \\ 7 \\ 10 \end{pmatrix}$ ,  $a_2 = \begin{pmatrix} 2 \\ 5 \\ 8 \\ 11 \end{pmatrix}$ ,  $a_3 = \begin{pmatrix} 3 \\ 6 \\ 9 \\ 12 \end{pmatrix}$
- Since  $a_1$  and  $a_2$  point in different directions,  $A$  is at least rank 2
- $a_3 = 2a_2 - a_1$ , so the third column is in the span of the first two columns
- Thus,  $A$  is only rank 2 (not rank 3)

# Example (SVD)

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix} =$$

$$\begin{pmatrix} .141 & .825 & -.420 & -.351 \\ .344 & .426 & .298 & .782 \\ .547 & .028 & .644 & -.509 \\ .750 & -.371 & -.542 & .079 \end{pmatrix} \begin{pmatrix} 25.5 & 0 & 0 \\ 0 & 1.29 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} .504 & .574 & .644 \\ -.761 & -.057 & .646 \\ .408 & -.816 & .408 \end{pmatrix}$$

- Singular values are 25.5, 1.29, and 0
- Singular value of 0 indicates that the matrix is rank deficient
- The rank of a matrix is equal to its number of nonzero singular values

# Derivation from $A^T A$ and $AA^T$

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix} =$$

$$\begin{pmatrix} .141 & .825 & -.420 & -.351 \\ .344 & .426 & .298 & .782 \\ .547 & .028 & .644 & -.509 \\ .750 & -.371 & -.542 & .079 \end{pmatrix} \begin{pmatrix} 25.5 & 0 & 0 \\ 0 & 1.29 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} .504 & .574 & .644 \\ -.761 & -.057 & .646 \\ .408 & -.816 & .408 \end{pmatrix}$$

- $A^T A$  is size  $3x3$  and has 3 eigenvectors (seen in  $V$ )
- The square roots of the 3 eigenvalues of  $A^T A$  are seen in  $\Sigma$  (color coded to the eigenvectors)
- $AA^T$  is size  $4x4$  and has 4 eigenvectors (seen in  $U$ )
- The square roots of 3 of the eigenvalues of  $AA^T$  are seen in  $\Sigma$ 
  - The 4<sup>th</sup> eigenvalue of  $AA^T$  is an extra eigenvalue of 0

# Understanding $Ac$

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix} =$$

$$\begin{pmatrix} .141 & .825 & -.420 & -.351 \\ .344 & .426 & .298 & .782 \\ .547 & .028 & .644 & -.509 \\ .750 & -.371 & -.542 & .079 \end{pmatrix} \begin{pmatrix} 25.5 & 0 & 0 \\ 0 & 1.29 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} .504 & .574 & .644 \\ -.761 & -.057 & .646 \\ .408 & -.816 & .408 \end{pmatrix}$$

- $A$  maps from  $R^3$  to  $R^4$
- $Ac$  first projects  $c \in R^3$  onto the 3 basis vectors in  $V$
- Then, the associated singular values (diagonally) scale the results
- Lastly, those scaled results are used as weights on the basis vectors in  $U$

# Understanding $Ac$

$$\begin{aligned}
 Ac &= \begin{pmatrix} .141 & .825 & -.420 & -.351 \\ .344 & .426 & .298 & .782 \\ .547 & .028 & .644 & -.509 \\ .750 & -.371 & -.542 & .079 \end{pmatrix} \begin{pmatrix} 25.5 & 0 & 0 \\ 0 & 1.29 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} .504 & .574 & .644 \\ -.761 & -.057 & .646 \\ .408 & -.816 & .408 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} \\
 &= \begin{pmatrix} .141 & .825 & -.420 & -.351 \\ .344 & .426 & .298 & .782 \\ .547 & .028 & .644 & -.509 \\ .750 & -.371 & -.542 & .079 \end{pmatrix} \begin{pmatrix} 25.5 & 0 & 0 \\ 0 & 1.29 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} v_1^T c \\ v_2^T c \\ v_3^T c \end{pmatrix} \\
 &= \begin{pmatrix} .141 & .825 & -.420 & -.351 \\ .344 & .426 & .298 & .782 \\ .547 & .028 & .644 & -.509 \\ .750 & -.371 & -.542 & .079 \end{pmatrix} \begin{pmatrix} \sigma_1 v_1^T c \\ \sigma_2 v_2^T c \\ \sigma_3 v_3^T c \\ 0 \end{pmatrix} \\
 &= u_1 \sigma_1 v_1^T c + u_2 \sigma_2 v_2^T c + u_3 \sigma_3 v_3^T c + u_4 0
 \end{aligned}$$

- $Ac$  projects  $c$  onto the basis vectors in  $V$ , scales by the associated singular values, and uses those results as weights on the basis vectors in  $U$

# Extra Dimensions

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix} =$$
$$\begin{pmatrix} .141 & .825 & -.420 & -.351 \\ .344 & .426 & .298 & .712 \\ .547 & .028 & .644 & -.09 \\ .750 & -.371 & -.542 & .79 \end{pmatrix} \begin{pmatrix} 25.5 & 0 & 0 \\ 0 & 1.29 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} .504 & .574 & .644 \\ -.761 & -.057 & .646 \\ .408 & -.816 & .408 \end{pmatrix}$$

- The 3D space of vector inputs can only span a 3D subspace of  $R^4$
- The last (green) column of  $U$  represents the unreachable dimension, orthogonal to the range of  $A$ , and is always multiplied by 0
- One can delete this column and the associated portion of  $\Sigma$  (and still obtain a valid factorization)

# Zero Singular Values

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix} =$$
$$\begin{pmatrix} .141 & .825 & -& .20 & -& .51 \\ .344 & .426 & -& .98 & -& .72 \\ .547 & .028 & -& .64 & -& .09 \\ .750 & -.371 & -& .542 & -& .79 \end{pmatrix} \begin{pmatrix} 25.5 & 0 & -& \\ 0 & 1.29 & -& \\ 0 & 0 & -& \\ 0 & 0 & -& \end{pmatrix} \begin{pmatrix} .504 & .574 & .644 \\ -.761 & -.057 & .646 \\ .408 & -.816 & .408 \end{pmatrix}$$

- The 3<sup>rd</sup> singular value is 0, so  $A$  has a 1D null space that reduces the 3D input vectors to only 2 dimensions
- The associated (pink) terms make no contribution to the final result, and can also be deleted (still obtaining a valid factorization)
- The first 2 columns of  $U$  span the 2D subset of  $R^4$  that comprises the range of  $A$

# Approximating $A$

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix} \approx$$

$$\begin{pmatrix} .141 & .325 & - .20 & .51 \\ .344 & .26 & .98 & .72 \\ .547 & .928 & .64 & -.09 \\ .750 & -.371 & -.542 & .79 \end{pmatrix} \begin{pmatrix} 25.5 & 0 & 0 \\ 0 & 1.1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} .504 & .574 & .644 \\ -.761 & .057 & .646 \\ .408 & -.816 & .408 \end{pmatrix}$$

- The first singular value is much bigger than the second, and so represents the vast majority of what  $A$  does (note, the vectors in  $U$  and  $V$  are unit length)
- Thus, one could approximate  $A$  quite well by only using the terms associated with the largest singular value
- This is not a valid factorization, but an **approximation** (and the idea behind PCA)

# Summary

- The columns of  $V$  that do not correspond to “nonzero” singular values form an orthonormal basis for the null space of  $A$
- The remaining columns of  $V$  form an orthonormal basis for the space perpendicular to the null space of  $A$  (parameterizing meaningful inputs)
- The columns of  $U$  corresponding to “nonzero” singular values form an orthonormal basis for the range of  $A$
- The remaining columns of  $U$  form an orthonormal basis for the (unattainable) space perpendicular to the range of  $A$
- One can drop the columns of  $U$  and  $V$  that do not correspond to “nonzero” singular values and still obtain a valid factorization of  $A$
- One can drop the columns of  $U$  and  $V$  that correspond to “small/smaller” singular values and still obtain a reasonable approximation of  $A$

# Example (Wide Matrix)

$$A = \begin{pmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{pmatrix} =$$
$$\begin{pmatrix} .504 & -.761 & .408 \\ .574 & -.057 & -.816 \\ .644 & .646 & .408 \end{pmatrix} \begin{pmatrix} 25.5 & 0 & 0 & 0 \\ 0 & 1.29 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} .141 & .344 & .547 & .750 \\ .825 & .426 & .028 & -.371 \\ -.420 & .298 & .644 & -.542 \\ -.351 & .782 & -.509 & .079 \end{pmatrix}$$

- $A$  maps from  $R^4$  to  $R^3$  and so has at least a 1D null space (**green**)
- The 3<sup>rd</sup> singular value is **0**, and the associated (**pink**) terms make no contribution to the final result

## Example (Wide Matrix)

$$A = \begin{pmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{pmatrix} =$$
$$\begin{pmatrix} .504 & -.761 & .413 \\ .574 & -.057 & -.826 \\ .644 & .646 & .413 \end{pmatrix} \begin{pmatrix} 25.5 & 0 \\ 0 & 1.29 \end{pmatrix} \begin{pmatrix} .141 & .344 & .547 & .750 \\ .825 & .426 & .028 & -.371 \\ \text{---} & \text{---} & \text{---} & \text{---} \\ \text{---} & \text{---} & \text{---} & \text{---} \end{pmatrix}$$

- Only a 2D subspace of  $R^4$  matters, with the rest of  $R^4$  in the null space of  $A$
- Only a 2D subspace of  $R^3$  is in the range of  $A$

# Notes

- The SVD is often unwieldy for computational purposes
- However, replacing matrices by their SVD can be quite useful/enlightening for theoretical pursuits
- Moreover, its theoretical underpinnings are often used to devise computational algorithms
- The SVD is unique under certain assumptions, such as all  $\sigma_k \geq 0$  and in descending order
- However, one can make both a  $\sigma_k$  and its associated column in  $U$  negative for an “alternate SVD” (see e.g. “Invertible Finite Elements For Robust Simulation of Large Deformation”, Irving et al. 2004)

# SVD Construction (Important Detail)

- Let  $A = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$  so that  $A^T A = AA^T = I$ , and thus  $U = V = \Sigma = I$
- But  $A \neq U\Sigma V^T = I$  **What's wrong?**
- Given a column vector  $v_k$  of  $V$ ,  $\textcolor{green}{Av}_k = U\Sigma V^T v_k = U\Sigma \hat{e}_k = U\sigma_k \hat{e}_k = \sigma_k \textcolor{green}{u}_k$  where  $u_k$  is the corresponding column of  $U$
- $\textcolor{green}{Av}_1 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \textcolor{green}{u}_1$  but  $\textcolor{green}{Av}_2 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ -1 \end{pmatrix} \neq \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \textcolor{green}{u}_2$
- Since  $U$  and  $V$  are orthonormal, their columns are unit length
- However, there are still two choices for the direction of each column
- Multiplying  $u_2$  by  $-1$  to get  $u_2 = \begin{pmatrix} 0 \\ -1 \end{pmatrix}$  makes  $U = A$ , and thus  $A = U\Sigma V^T$  as desired

# SVD Construction (Important Detail)

- An orthogonal matrix has determinant equal to  $\pm 1$ , where  $-1$  indicates a reflection of the coordinate system
- If  $\det V = -1$ , flip the direction of any column to make  $\det V = 1$  (so  $V$  does not contain a reflection)
- Then, for each  $v_k$ , compare  $Av_k$  to  $\sigma_k u_k$  and flip the direction of  $u_k$  when necessary in order to make  $Av_k = \sigma_k u_k$
- $\det U = \pm 1$  and may contain a reflection
- When  $\det U = -1$ , one can flip the sign of the smallest singular value in  $\Sigma$  to be negative, whilst also flipping the direction of the corresponding column in  $U$  so that  $\det U = 1$
- This embeds the reflection into  $\Sigma$  and is called the polar-SVD (Irving et al. 2004)

# Solving Linear Systems

- $Ac = b$  becomes  $U\Sigma V^T c = b$  or  $\Sigma(V^T c) = (U^T b)$  or  $\Sigma\hat{c} = \hat{b}$
- The unknowns  $c$  are remapped into the space spanned by  $V$ , and the right hand side  $b$  is remapped into the space spanned by  $U$
- **Every matrix is a diagonal matrix, when viewed in the right space**
- Solve the diagonal system  $\Sigma\hat{c} = \hat{b}$  by dividing the entries of  $\hat{b}$  by the singular values  $\sigma_k$ ; then,  $c = V\hat{c}$
- The SVD transforms the problem into an inherently diagonal space with eigenvectors along the coordinate axes
- Circles becoming ellipses (discussed earlier) is still problematic
  - Eccentricity is caused by ratios of singular values (since  $U$  and  $V$  are orthogonal matrices)

# Condition Number

- The condition number of  $A$  is  $\frac{\sigma_{max}}{\sigma_{min}}$  and measures closeness to being singular
- For a square matrix, it measures the difficulty in solving  $Ac = b$
- For a rectangular (and square) matrix, it measures how close the columns are to being linearly dependent
  - For a wide (rectangular) matrix, it ignores the extra columns that are guaranteed to be linearly dependent (which is fine, because the associated variables lack any data)
- The condition number does not depend on the right hand side
- The condition number is always bigger than 1, and approaches  $\infty$  for nearly singular matrices
- Singular matrices have condition number equal to  $\infty$ , since  $\sigma_{min} = 0$

# Singular Matrices

- Diagonalize  $Ac = b$  to  $\Sigma(V^T c) = (U^T b)$ , e.g.  $\begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix} \begin{pmatrix} \hat{c}_1 \\ \hat{c}_2 \end{pmatrix} = \begin{pmatrix} \hat{b}_1 \\ \hat{b}_2 \end{pmatrix}$  with  $\hat{c}_1 = \frac{\hat{b}_1}{\sigma_1}$ ,  $\hat{c}_2 = \frac{\hat{b}_2}{\sigma_2}$
- Suppose  $\sigma_1 \neq 0$  and  $\sigma_2 = 0$ ; then, there is no unique solution:
  - When  $\hat{b}_2 = 0$ , there are infinite solutions for  $\hat{c}_2$  (but  $\hat{c}_1$  is still uniquely determined)
  - When  $\hat{b}_2 \neq 0$ , there is no solution for  $\hat{c}_2$ , and  $b$  is not in the range of  $A$  (but  $\hat{c}_1$  is still uniquely determined)
- Consider:  $\begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} \hat{c}_1 \\ \hat{c}_2 \\ \hat{c}_3 \end{pmatrix} = \begin{pmatrix} \hat{b}_1 \\ \hat{b}_2 \\ \hat{b}_3 \end{pmatrix}$  with  $\hat{c}_1 = \frac{\hat{b}_1}{\sigma_1}$ ,  $\hat{c}_2 = \frac{\hat{b}_2}{\sigma_2}$ 
  - When  $\hat{b}_3 = 0$ , the last row adds no new information (one has extra redundant data)
  - When  $\hat{b}_3 \neq 0$ , the last row is false and there is no solution (but  $\hat{c}_1$  and  $\hat{c}_2$  are still uniquely determined)
- Consider:  $\begin{pmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \end{pmatrix} \begin{pmatrix} \hat{c}_1 \\ \hat{c}_2 \\ \hat{c}_3 \end{pmatrix} = \begin{pmatrix} \hat{b}_1 \\ \hat{b}_2 \\ \hat{b}_3 \end{pmatrix}$  with  $\hat{c}_1 = \frac{\hat{b}_1}{\sigma_1}$ ,  $\hat{c}_2 = \frac{\hat{b}_2}{\sigma_2}$ 
  - Infinite solutions work for  $\hat{c}_3$  (but  $\hat{c}_1$  and  $\hat{c}_2$  are still uniquely determined)

# Understanding Variables

- Consider any column  $k$  of  $\Sigma$
- When  $\sigma_k \neq 0$ , one can state a value for  $\hat{c}_k$
- When  $\sigma_k = 0$  or there is no  $\sigma_k$ , then there is no information in the data for  $\hat{c}_k$ 
  - This does not mean that other parameters cannot be adequately determined!
- Consider a row  $i$  of  $\Sigma$  that is identically zero
- When  $\hat{b}_i = 0$ , this row indicates that there is extra redundant data
- When  $\hat{b}_i \neq 0$ , this row indicates that there is conflicting information in the data
- Conflicting information doesn't necessarily imply that all is lost, i.e. "no solution"; rather, it might merely mean that the data contains a bit of noise
- Regardless, in spite of any conflicting information, the determinable  $\hat{c}_k$  represent the "best" that one can do

# Norms

- Common norms:  $\|c\|_1 = \sum_k |c_k|$ ,  $\|c\|_2 = \sqrt{\sum_k c_k^2}$ ,  $\|c\|_\infty = \max_k |c_k|$
- “All norms are interchangeable” is a **theoretically** valid statement (**only**)
- In practice, the “worst case scenario” ( $L^\infty$ ) and the “average” ( $L^1$ ,  $L^2$ , etc.) are not interchangeable
- E.g.  $(100 \text{ people} * 98.6^\circ + 1 \text{ person} * 105^\circ) / (101 \text{ people}) = 98.66^\circ$
- Their average temperature is  $98.66^\circ$ , but everything is not “ok”

# Matrix Norms

- Define the norm of a matrix  $\|A\| = \max_{c \neq 0} \frac{\|Ac\|}{\|c\|}$ , so:
  - $\|A\|_1$  is the maximum absolute value column sum
  - $\|A\|_\infty$  is the maximum absolute value row sum
  - $\|A\|_2$  is the square root of the maximum eigenvalue of  $A^T A$ , i.e. the maximum singular value of  $A$
- The condition number for solving (square matrix)  $Ac = b$  is  $\|A\|_2 \|A^{-1}\|_2$
- Since  $A^{-1} = V\Sigma^{-1}U^T$  where  $\Sigma^{-1}$  has diagonal entries  $\frac{1}{\sigma_k}$ ,  $\|A^{-1}\|_2 = \frac{1}{\sigma_{min}}$
- Thus,  $\|A\|_2 \|A^{-1}\|_2 = \frac{\sigma_{max}}{\sigma_{min}}$

# Unit 4

# Special Matrices

# (Strict) Diagonal Dominance

- The magnitude of each diagonal element is (either):
  - strictly larger than the sum of the magnitudes of all the other elements in its row
  - strictly larger than the sum of the magnitudes of all the other elements in its column
- One may row/column scale and permute rows/columns to achieve diagonal dominance (since it's just a rewriting of the equations)
  - Recall: choosing the form of the equations wisely is important
- E.g. consider  $\begin{pmatrix} 3 & -2 \\ 5 & 1 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} 9 \\ 4 \end{pmatrix}$
- Switch rows  $\begin{pmatrix} 5 & 1 \\ 3 & -2 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} 4 \\ 9 \end{pmatrix}$  and column scale  $\begin{pmatrix} 5 & -2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} c_1 \\ -.5c_2 \end{pmatrix} = \begin{pmatrix} 4 \\ 9 \end{pmatrix}$

# (Strict) Diagonal Dominance

- Strictly diagonally dominant (square) matrices are guaranteed to be non-singular
- Since  $\det(A) = \det(A^T)$ , either row or column diagonal dominance is enough
- Column diagonal dominance guarantees that pivoting is not required during *LU* factorization
- However, pivoting still improves robustness
- E.g. consider  $\begin{pmatrix} 4 & 3 \\ -2 & 50 \end{pmatrix}$  where 50 is more desirable than 4 for  $a_{11}$

# Inner Product

- Consider the space of all vectors with length  $m$
- The dot/inner product of two vectors is  $u \cdot v = \sum_i u_i v_i$
- The magnitude of a vector is  $\|v\|_2 = \sqrt{v \cdot v} (\geq 0)$
- Alternative notations:  $\langle u, v \rangle = u \cdot v = u^T v$
- Weighted inner product defined via an  $n \times n$  matrix  $A$
- $\langle u, v \rangle_A = u \cdot A v = u^T A v$
- Since  $\langle v, u \rangle_A = v^T A u = u^T A^T v$ , weighted inner products commute when  $A$  is symmetric
- The standard dot product uses identity matrix weighting:  $\langle u, v \rangle = \langle u, v \rangle_{I_{127}}$

# Definiteness

- Assume  $A$  is symmetric so that  $\langle u, v \rangle_A = \langle v, u \rangle_A$
- $A$  is positive definite if and only if  $\langle v, v \rangle_A = v^T A v > 0$  for  $\forall v \neq 0$
- $A$  is positive semi-definite if and only if  $\langle v, v \rangle_A = v^T A v \geq 0$  for  $\forall v \neq 0$
- We abbreviate with SPD and SP(S)D
- $A$  is negative definite if and only if  $\langle v, v \rangle_A = v^T A v < 0$  for  $\forall v \neq 0$
- $A$  is negative semi-definite if and only if  $\langle v, v \rangle_A = v^T A v \leq 0$  for  $\forall v \neq 0$
- If  $A$  is negative (semi) definite, then  $-A$  is positive (semi) definite (and vice versa)
- Thus, can convert such problems to SPD or SP(S)D
- $A$  is considered indefinite when it is neither positive/negative semi-definite

# Eigenvalues

- SPD matrices have all eigenvalues  $> 0$
- SP(S)D matrices have all eigenvalues  $\geq 0$
- Symmetric negative definite matrices have all eigenvalues  $< 0$
- Symmetric negative semi-definite matrices have all eigenvalues  $\leq 0$
- Indefinite matrices have both positive and negative eigenvalues

# Recall: SVD Construction (Unit 3)

- Let  $A = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$  so that  $A^T A = AA^T = I$ , and thus  $U = V = \Sigma = I$
- But  $A \neq U\Sigma V^T = I$  What's wrong?
- Given a column vector  $v_k$  of  $V$ ,  $\textcolor{red}{Av}_k = U\Sigma V^T v_k = U\Sigma \hat{e}_k = U\sigma_k \hat{e}_k = \sigma_k u_k$  where  $u_k$  is the corresponding column of  $U$
- $Av_1 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} = u_1$  but  $Av_2 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ -1 \end{pmatrix} \neq \begin{pmatrix} 0 \\ 1 \end{pmatrix} = u_2$
- Since  $U$  and  $V$  are orthonormal, their columns are unit length
- However, there are still two choices for the direction of each column
- Multiplying  $u_2$  by  $-1$  to get  $u_2 = \begin{pmatrix} 0 \\ -1 \end{pmatrix}$  makes  $U = A$ , and thus  $A = U\Sigma V^T$  as desired

# Symmetric Matrices (SVD)

- Since  $A^T A = AA^T = A^2$ , both the columns of  $U$  and the columns of  $V$  are eigenvectors of  $A^2$
- They are identical (but **potentially opposite**) directions:  $u_k = \pm v_k$
- Thus,  $A v_k = \sigma_k u_k$  implies  $A v_k = \pm \sigma_k v_k$
- That is, the  $v_k$  (and  $u_k$ ) are eigenvectors of  $A$  with eigenvalues  $\pm \sigma_k$
- Similar to the polar SVD, can pull negative signs out of the columns of  $U$  into the  $\sigma_k$  to obtain  $U = V$  and  $A = V \Lambda V^T$  as a modified SVD
- $A = V \Lambda V^T$  implies  $AV = V \Lambda$  which is the matrix form of the eigensystem of  $A$
- Here,  $\Lambda$  contains the positive and negative eigenvalues of  $A$

# SPD Matrices

- When  $A$  is SP(S)D,  $\Lambda = \Sigma$  and the standard SVD is  $A = V\Sigma V^T$  (i.e.  $U = V$ )
- The singular values are the (all positive) eigenvalues of  $A$  (since  $AV = V\Sigma$ )
- Construct  $V$  with  $\det V = 1$  (as usual), and all  $\sigma_k > 0$  implies that there are no reflections
- Since all  $\sigma_k > 0$ , SPD matrices have full rank and are invertible
- SP(S)D (and not SPD) has at least one  $\sigma_k = 0$  and a null space
- Often, one can use modified SPD techniques for SP(S)D matrices
- Unfortunately, indefinite matrices are significantly more challenging

# Making/Breaking Symmetry

- Row/column scaling can break symmetry:
  - Row scaling  $\begin{pmatrix} 5 & 3 \\ 3 & -4 \end{pmatrix}$  by  $-2$  gives a non-symmetric  $\begin{pmatrix} 5 & 3 \\ -6 & 8 \end{pmatrix}$
  - Additional column scaling by  $-2$  gives  $\begin{pmatrix} 5 & -6 \\ -6 & -16 \end{pmatrix}$
- Scaling the same row/column together in the same way preserves symmetry
- Important: a nonsymmetric matrix might be inherently symmetric when properly rescaled/rearranged

# Rules Galore

- There are many rules/theorems regarding special matrices (especially for SPD)
- It is important to be aware of reference material (and to look things up)
- Examples:
  - SPD matrices don't require pivoting during  $LU$  factorization
  - A symmetric (strictly) diagonally dominant matrix with positive diagonal entries is positive definite
  - Jacobi and Gauss-Seidel iteration converge when a matrix is strictly (or irreducibly) diagonally dominant
  - Etc.

# Cholesky Factorization

- SPD matrices have an  $LU$  factorization of  $LL^T$  and don't require elimination to find it
- Consider  $\begin{pmatrix} a_{11} & a_{21} \\ a_{21} & a_{22} \end{pmatrix} = \begin{pmatrix} l_{11} & 0 \\ l_{21} & l_{22} \end{pmatrix} \begin{pmatrix} l_{11} & l_{21} \\ 0 & l_{22} \end{pmatrix} = \begin{pmatrix} l_{11}^2 & l_{11}l_{21} \\ l_{11}l_{21} & l_{21}^2 + l_{22}^2 \end{pmatrix}$
- So  $l_{11} = \sqrt{a_{11}}$  and  $l_{21} = \frac{a_{21}}{l_{11}}$  and  $l_{22} = \sqrt{a_{22} - l_{21}^2}$   
 $\text{for}(j=1,n)\{$   
     $\text{for}(k=1,j-1) \text{for}(i=j,n) a_{ij} -= a_{ik}a_{jk};$   
     $a_{jj} = \sqrt{a_{jj}}; \text{for}(k=j+1,n) a_{kj}/= a_{jj};\}$   
\\" For each column j of the matrix  
\\" Loop over all previous columns k, and subtract a multiple of column k from the current column j  
\\" Take the square root of the diagonal entry, and scale column j by that value
- This algorithm factors the matrix “in place” replacing  $A$  with  $L$

# Incomplete Cholesky Preconditioner

- Cholesky factorization can be used to construct a preconditioner for a sparse matrix
- The full Cholesky factorization would fill in too many non-zero entries
- So, incomplete Cholesky preconditioning uses Cholesky factorization with the **caveat** that only the nonzero entries are modified (all zeros remain zeros)

# Symmetric Approximation

- For non-symmetric  $A$ , a symmetric  $\hat{A} = \frac{1}{2}(A + A^T)$  averages off-diagonal components
- Solving the symmetric  $\hat{A}c = b$  instead of the non-symmetric  $Ac = b$  gives a faster/easier (perhaps erroneous) approximation to a problem that might not require too much accuracy
- Alternatively, the inverse of the symmetric  $\hat{A}$  (or the notion thereof) may be used to devise a preconditioner for  $Ac = b$

# Unit 5

# Iterative Solvers

# Iterative vs. Direct Solvers

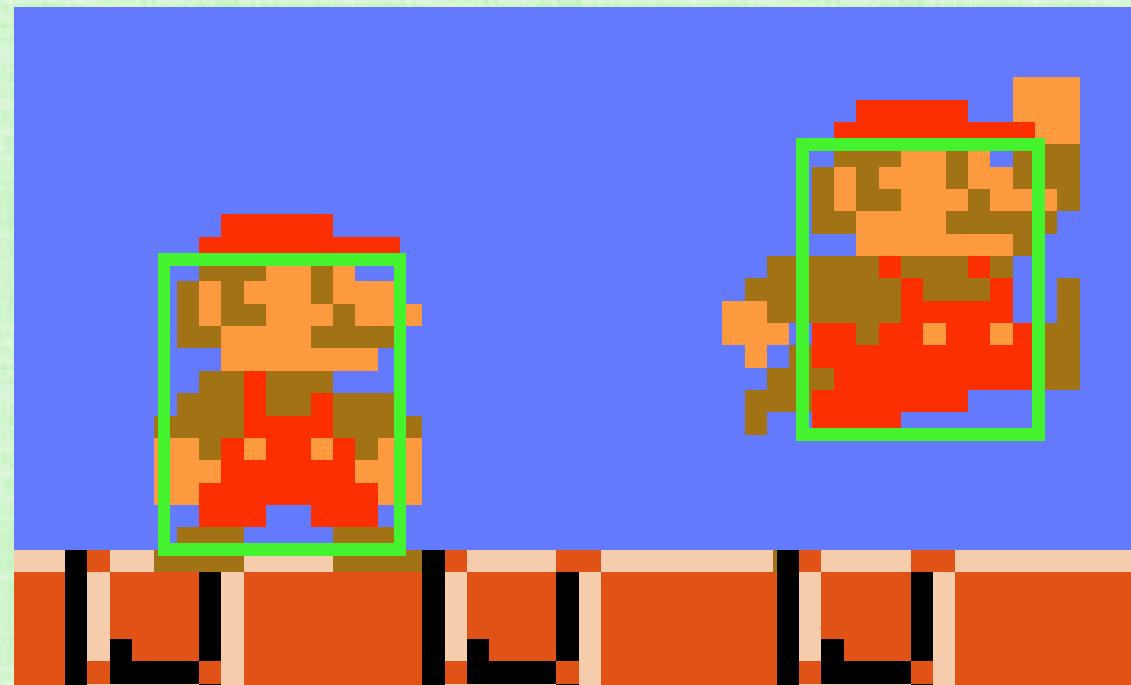
- Direct Solver/Method – closed form strategy, e.g. quadratic/Cardano formula, Gaussian Elimination for LU factorization, Cholesky factorization, etc.
- Iterative Solver/Method
  - start with an initial guess  $c^1$
  - use a recursive approach to improve that guess:  $c^2, c^3, c^4, \dots$
  - terminate based on a stopping criterion, e.g. when error is small  $\|c^q - c^{exact}\| \leq \epsilon$
- A direct method can be used to obtain an initial guess
- Iterative methods are great for sparse matrices, as they often can ignore 0 entries
  - E.g. by formulating the method via the matrix's action (multiplication) on a vector
- Direct solvers are more commonly used on dense matrices
- **Iterative solvers are used for training Neural Networks!**

# Issues with Direct Methods

- (Recall) Quadratic formula loses precision, and can fail, when  $-b \pm \sqrt{b^2 - 4ac}$  has catastrophic cancellation
  - The de-rationalized quadratic formula instead uses  $-b \mp \sqrt{b^2 - 4ac}$
  - Using one formula for each root avoids catastrophic cancellation
- Cardano's formula for the roots of a cubic equation suffers from similar issues, but there is no straightforward fix
- The computed roots/solutions too often have unacceptably high error
- To highlight the need for more accurate cubic roots, let's consider collision detection

# Hit Box

- In order to detect interactions between objects in video games, objects were assigned a hit box
- Anything inside an object's hit box can potentially interact with (i.e. hit) it



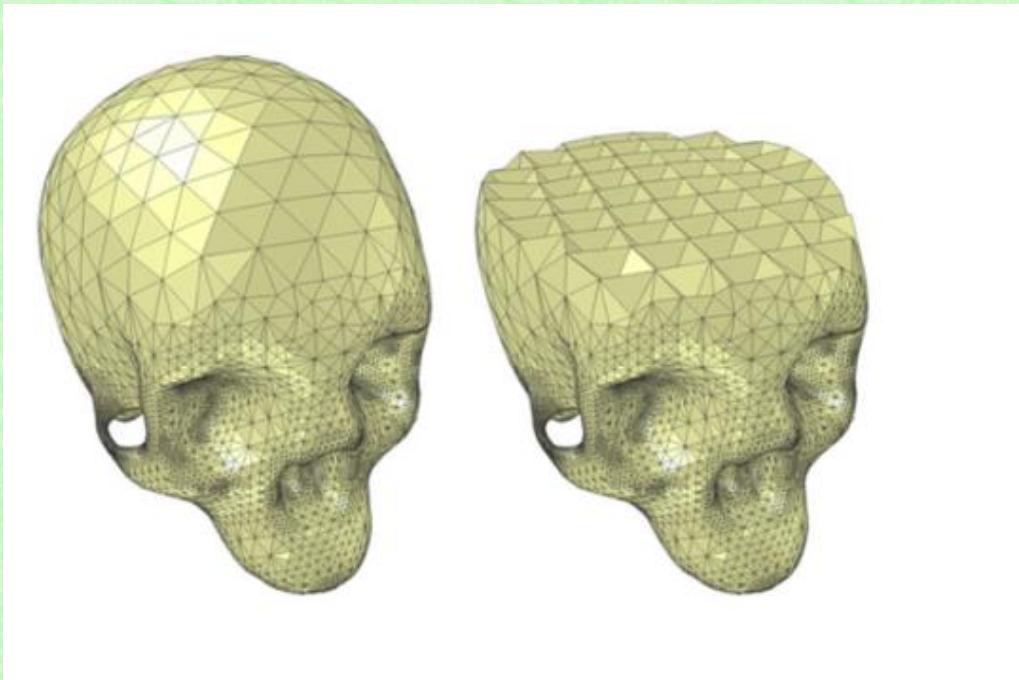
# Better Hit Boxes

- These evolved over time to more complicated shapes in both 2D and 3D
  - e.g. spheres, ellipsoids, capsules, etc.
- Anything inside any of an object's hit boxes can potentially interact with it



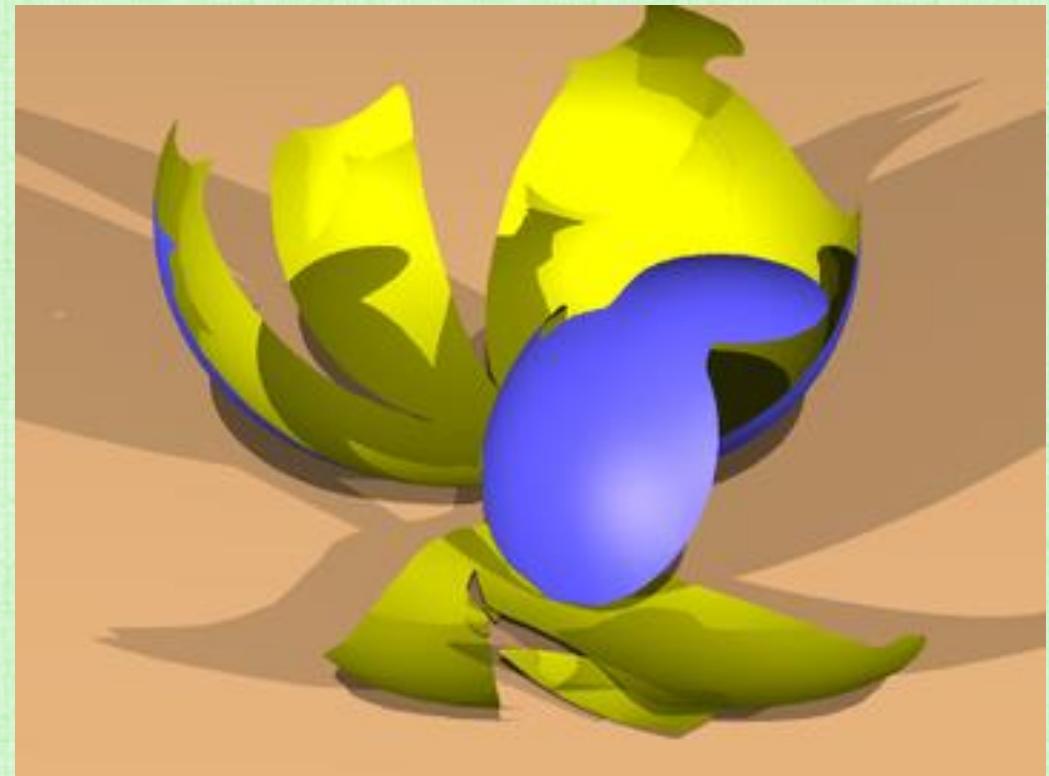
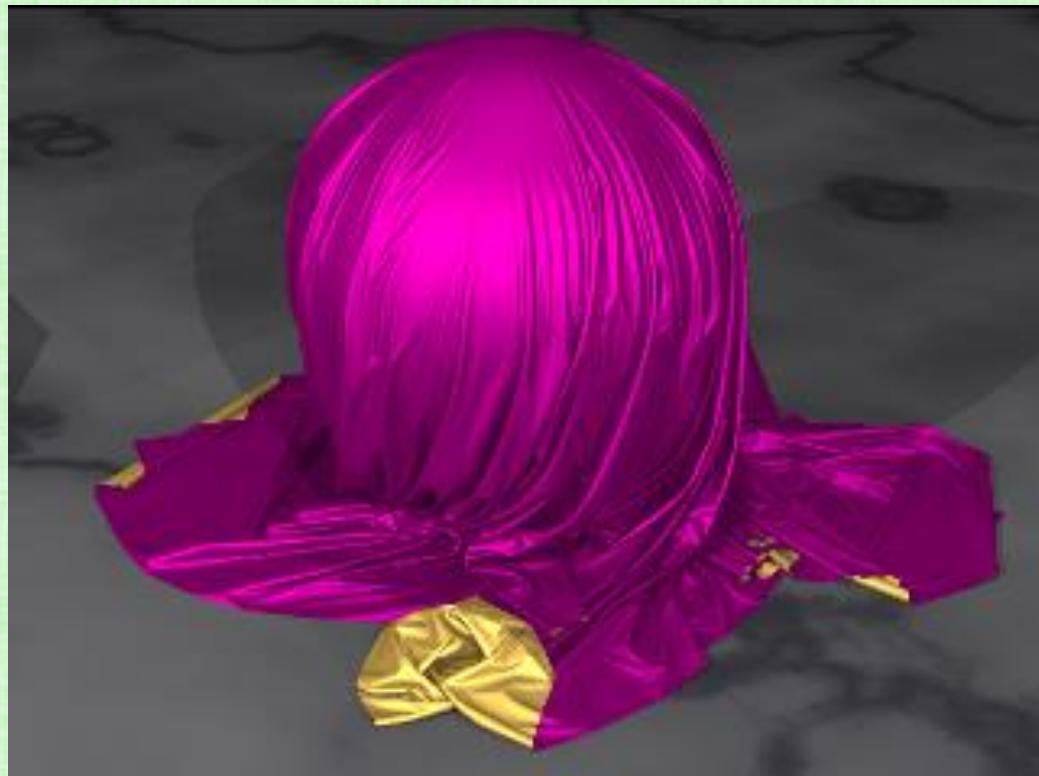
# Accurate Collision Detection

- More complex objects are often modeled by a triangulated surface mesh
- The interior can be filled with tetrahedra, or approximated with other objects
- Anything inside any of an object's interior structures can potentially interact with it



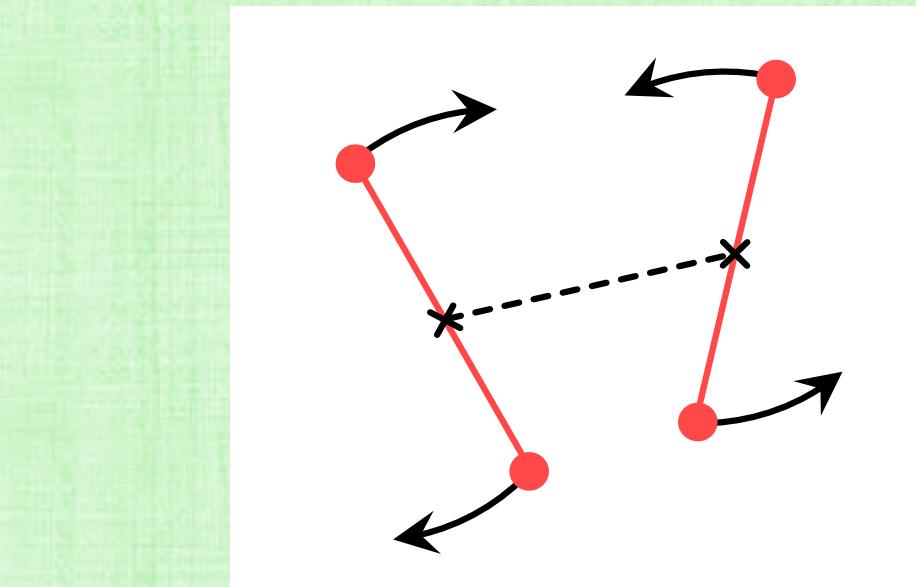
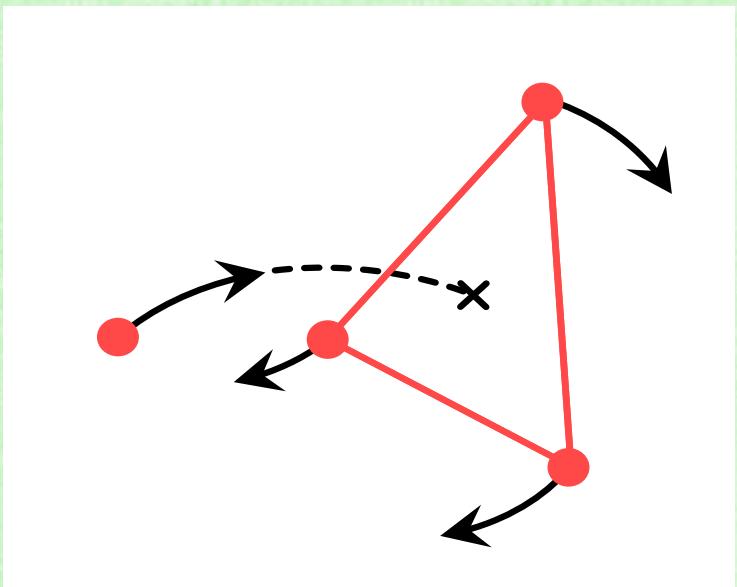
# Objects Without Interiors

- Very thin objects, such as cloth/shells, do not have an interior region
- One cannot use the same concept of inside to detect potential interactions



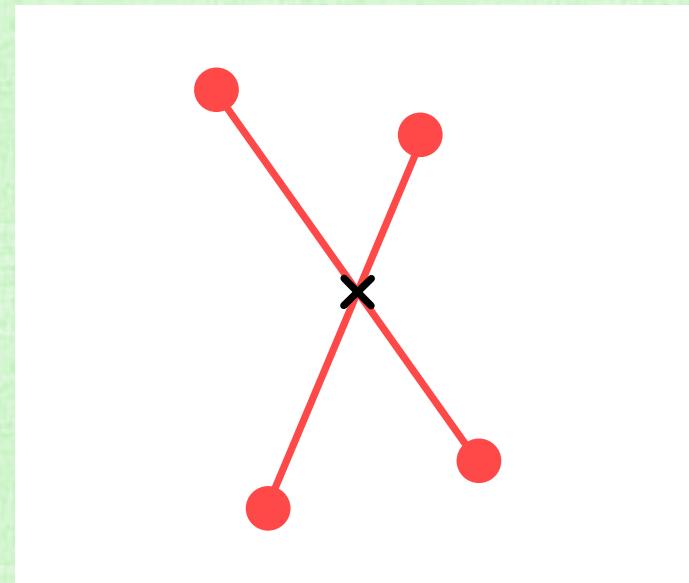
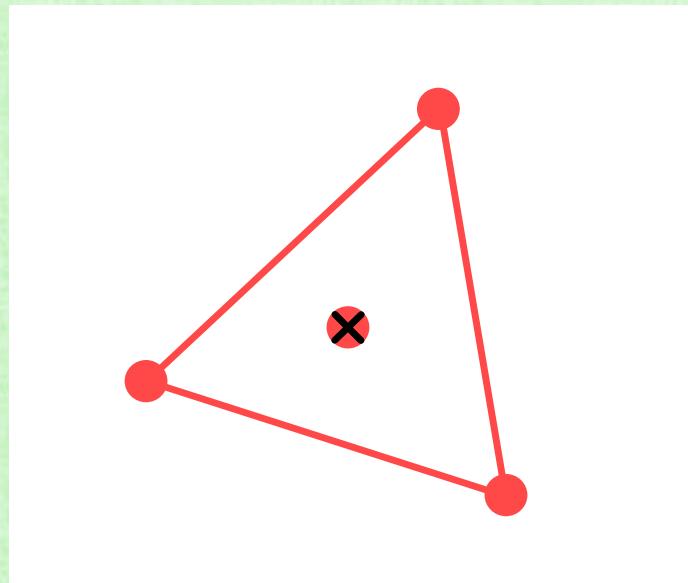
# Continuous Collision Detection (CCD)

- Model time varying trajectories of surface triangle vertices (or other geometry) to see if/when they collide with each other
- Doesn't depend on the existence of an interior region
- For triangles, there are two cases to consider: (1) Point-Face, (2) Edge-Edge



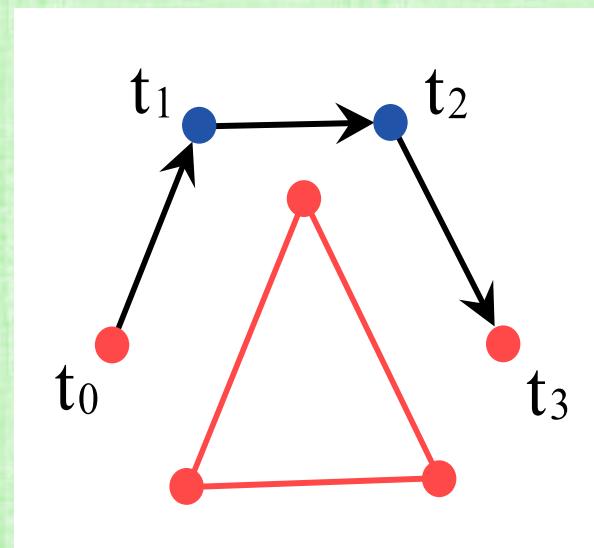
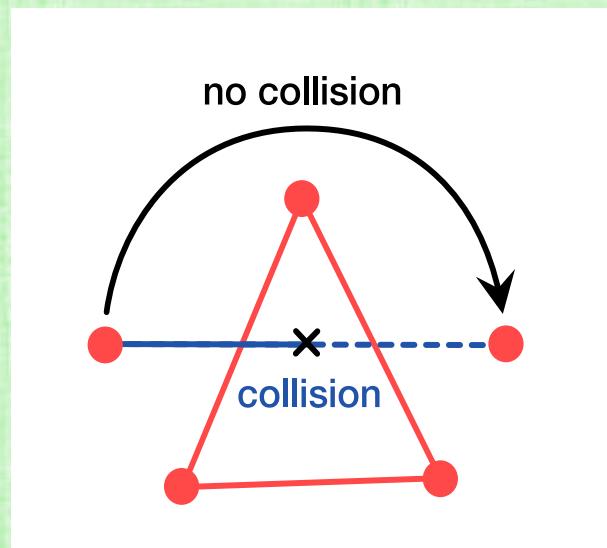
# Continuous Collision Detection (CCD)

- In both cases, the 4 relevant points need to become coplanar in order to (potentially) collide
- Once deemed coplanar, a second check determines whether: the lone point is inside the triangle (for Point-Face) or the two edges intersect (for Edge-Edge)



# Continuous Collision Detection (CCD)

- Consider time  $t_o$  to time  $t_f$  and assume all the points have constant velocities during that time interval:  $V_i(t_o)$  for  $i = 1, 2, 3, 4$
- The time evolving positions are:  $X_i(t) = X_i(t_o) + V_i(t_o)(t - t_o)$  for  $t \in [t_o, t_f]$
- Although their paths are (generally) curved, considering piecewise linear increments is sufficient for preventing self-intersecting states



# Continuous Collision Detection (CCD)

- Coplanarity occurs when  $X_4(t) - X_1(t)$ ,  $X_3(t) - X_1(t)$ , and  $X_2(t) - X_1(t)$  are not a basis for  $R^3$ , which can be checked by making them the columns of a 3x3 matrix and setting the determinant to zero (obtaining a cubic equation in  $t$ )
- Need to find the first root of this cubic equation in the interval  $[t_o, t_f]$
- Cubic equation solvers are so error prone that collisions are (very) often missed, and the cloth/shell ends up in a spurious self-intersecting state
- A very carefully devised/implemented iterative solver for cubic equations was able to detect all collisions:
  - It requires double precision (and fails too often in single precision)
  - See Bridson et al. “Robust Treatment of Collisions, Contact, and Friction for Cloth Animation” (2002)

# Residual (and Error)

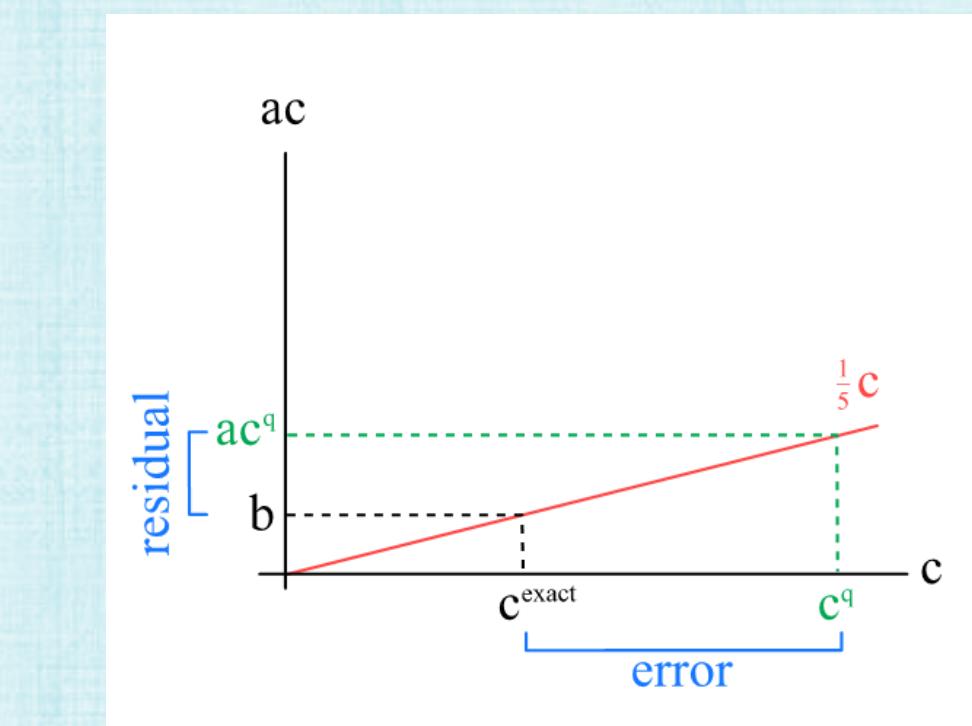
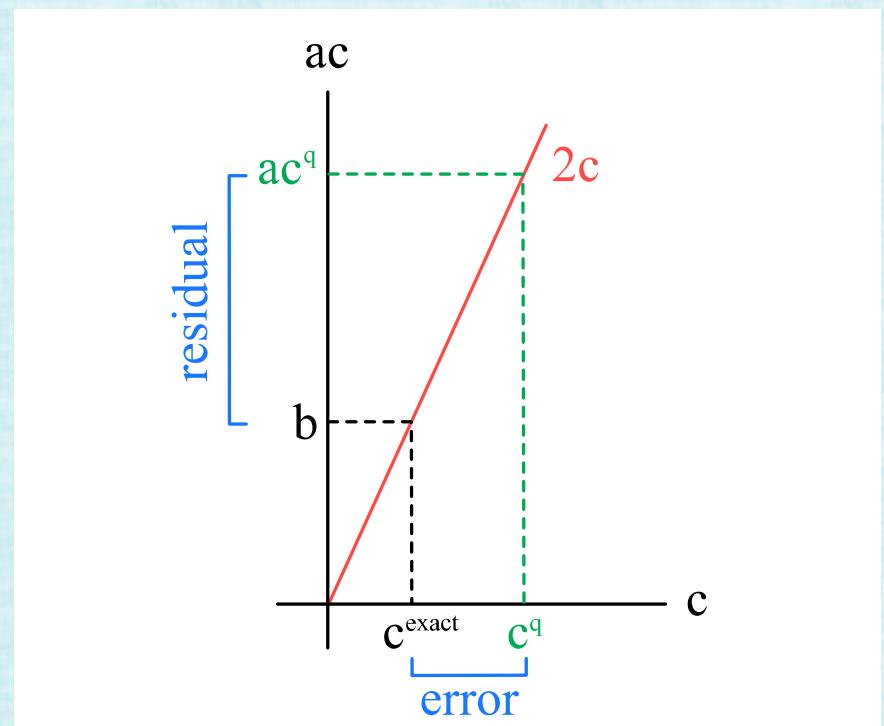
- When solving  $Ac = b$ , given a guess  $c^q$ , the residual is  $r^q = b - Ac^q$
- The residual measures the errors in the equations, not the error in the solution
- Given an error  $e^q = c^q - c^{exact}$ ,

$$r^q = b - Ac^q = Ac^{exact} - Ac^q = A(c^{exact} - c^q) = -Ae^q$$

- That is, the residual is the error transformed into the space that  $b$  lives in (the range of  $A$ )

# Residual (1D example)

- Consider a simple size  $1 \times 1$  matrix, i.e.  $[a]c = b$  with exact solution  $c = \frac{b}{a}$
- Since  $r^q = -ae^q$ , smaller  $a$  values lead to deceptively small residuals even when the error is large



# Residual

- "All matrices are diagonal matrices"
- And, diagonal matrices represent decoupled 1D scalar problems
- Using the SVD,  $r^q = -Ae^q$  becomes  $(U^T r^q) = -\Sigma(V^T e^q)$  which is a decoupled set of diagonal equations
- Each decoupled equation has the form  $\hat{r}_k^q = -\sigma_k \hat{e}_k^q$  (seen on the previous slide)
- Small  $\sigma_k$  lead to deceptively small residuals even when the error is large
- A small residual indicates a small error for larger singular values, but not for smaller singular values

# Line Search (in parameter space)

- Choose a search direction  $s^q$  and move some distance  $\alpha^q$  in that direction to find the next iterative guess:  $c^{q+1} = c^q + \alpha^q s^q$ 
  - There are various strategies for choosing  $\alpha^q$ , including the notion of safe sets that clamp its maximum magnitude
  - Subtract  $c^{exact}$  from both sides of this recursion to get  $e^{q+1} = e^q + \alpha^q s^q$ , and multiply through by  $-A$  to get  $r^{q+1} = r^q - \alpha^q A s^q$
- Optimally, one would follow  $s^q$  until no error was left in that direction, i.e. until the remaining error was orthogonal to  $s^q$ , i.e.  $e^{q+1} \cdot s^q = 0$
- The error is unknown (otherwise, the solution would be known), but one can instead progress until the residual is orthogonal to  $s^q$ , i.e.  $r^{q+1} \cdot s^q = 0$ 
  - Plugging in the recursion for  $r^{q+1}$  gives  $\alpha^q = \frac{s^q \cdot r^q}{s^q \cdot A s^q}$

# Steepest Descent Method

- Steepest Descent chooses the search direction to be the steepest downhill direction, which turns out to be the residual, i.e.  $s^q = r^q$
- Iterate:  $r^q = b - Ac^q$ ,  $\alpha^q = \frac{r^q \cdot r^q}{r^q \cdot Ar^q}$ ,  $c^{q+1} = c^q + \alpha^q r^q$ , until  $r^q$  is considered small enough
- Note:  $r^q = b - Ac^q$  can be replaced with  $r^q = r^{q-1} - \alpha^{q-1} Ar^{q-1}$  where  $Ar^{q-1}$  had already been computed to find  $\alpha^{q-1}$ 
  - This eliminates one of the (possibly expensive) multiplications by  $A$
- Main Drawback: Steepest Descent repeatedly searches in the same directions too often, especially for higher condition number matrices (more on this later)

# Conjugate Gradients (CG) Method

- A very efficient/robust method for SPD systems
- Converges (theoretically) in  $n$ -steps for an  $n \times n$  matrix
  - Actually, converges in the number of steps equal to the number of distinct eigenvalues
  - Almost converges in the number of steps equal to the number of eigenvalue clusters
  - Thus, preconditioning makes a big difference (assuming it clusters eigenvalues)
- Motivation: choosing \***orthogonal**\* search directions would preclude repeatedly searching in the same directions (as Steepest Descent inefficiently does)
  - It turns out to be difficult to implement orthogonality
- Instead: choose search directions to be A-orthogonal
  - That is,  $\langle s^q, s^{\hat{q}} \rangle_A = 0$  for  $q \neq \hat{q}$ , instead of  $\langle s^q, s^{\hat{q}} \rangle = 0$

# Gram-Schmidt

- Orthogonalizes a set of vectors
- For each new vector, subtract its (weighted) dot product overlap with all prior vectors, making it orthogonal to them
- A-orthogonal Gram-Schmidt uses an A-weighted dot/inner product
- Given vector  $\bar{s}^q$ , subtract out the A-overlap with  $s^1$  to  $s^{q-1}$  so that the resulting vector  $s^q$  has  $\langle s^q, s^{\hat{q}} \rangle_A = 0$  for  $\hat{q} \in \{1, 2, \dots, q-1\}$
- That is,  $s^q = \bar{s}^q - \sum_{\hat{q}=1}^{q-1} \frac{\langle \bar{s}^q, s^{\hat{q}} \rangle_A}{\langle s^{\hat{q}}, s^{\hat{q}} \rangle_A} s^{\hat{q}}$  where the two non-normalized  $s^{\hat{q}}$  both require division by their norm (and  $\langle s^{\hat{q}}, s^{\hat{q}} \rangle_A = \|s^{\hat{q}}\|_A^2$ )
- Proof:  $\langle s^q, s^{\tilde{q}} \rangle_A = \langle \bar{s}^q, s^{\tilde{q}} \rangle_A - \frac{\langle \bar{s}^q, s^{\tilde{q}} \rangle_A}{\langle s^{\tilde{q}}, s^{\tilde{q}} \rangle_A} \langle s^{\tilde{q}}, s^{\tilde{q}} \rangle_A = 0$

# Conjugate Gradients (Error Analysis)

- In the A-orthogonal basis of search directions, the initial error is  $e^1 = \sum_{\hat{q}=1}^n \beta^{\hat{q}} s^{\hat{q}}$ ; so,  $\langle s^q, e^1 \rangle_A = \beta^q \langle s^q, s^q \rangle_A$
- Error recursion gives  $e^q = e^1 + \sum_{\hat{q}=1}^{q-1} \alpha^{\hat{q}} s^{\hat{q}}$ ; so,  $\langle s^q, e^q \rangle_A = \langle s^q, e^1 \rangle_A$
- Recall: progressing until  $r^{q+1} \cdot s^q = 0$  gave  $\alpha^q = \frac{s^q \cdot r^q}{s^q \cdot As^q}$  ( $= -\frac{\langle s^q, e^q \rangle_A}{\langle s^q, s^q \rangle_A}$ )
- Thus  $\alpha^q = -\beta^q$ ; and then,  $e^1 = \sum_{\hat{q}=1}^n (-\alpha^{\hat{q}}) s^{\hat{q}}$  and  $e^q = \sum_{\hat{q}=q}^n (-\alpha^{\hat{q}}) s^{\hat{q}}$  prove that the error is indeed cancelled out in  $n$  steps!
- Aside: For  $\tilde{q} < q$ , one has  $s^{\tilde{q}} \cdot r^q = -\langle s^{\tilde{q}}, e^q \rangle_A = 0$  implying that **the residual is orthogonal to all previous search directions** (not just the previous one)

# Conjugate Gradients (Gram-Schmidt)

- Choose candidate search directions  $\bar{S}^q = r^q$ , and make A-orthogonal via Gram-Schmidt
- That is,  $s^q = r^q - \sum_{\hat{q}=1}^{q-1} \frac{\langle r^q, s^{\hat{q}} \rangle_A}{\langle s^{\hat{q}}, s^{\hat{q}} \rangle_A} s^{\hat{q}}$
- Dot product with  $r^{\tilde{q}}$  to get:  $s^q \cdot r^{\tilde{q}} = r^q \cdot r^{\tilde{q}} - \sum_{\hat{q}=1}^{q-1} \frac{\langle r^q, s^{\hat{q}} \rangle_A}{\langle s^{\hat{q}}, s^{\hat{q}} \rangle_A} s^{\hat{q}} \cdot r^{\tilde{q}}$ 
  - When  $\tilde{q} > q$ , one has  $0 = r^q \cdot r^{\tilde{q}} + 0$  implying that the residuals are all orthogonal
  - When  $\tilde{q} = q$ , one has  $s^q \cdot r^q = r^q \cdot r^q + 0$ , so that  $\alpha^q = \frac{r^q \cdot r^q}{\langle s^q, s^q \rangle_A}$
- Starting with  $r^q = r^{q-1} - \alpha^{q-1} A s^{q-1}$
- Dot product with  $r^{\tilde{q}}$  to get:  $r^{\tilde{q}} \cdot r^q = r^{\tilde{q}} \cdot r^{q-1} - \alpha^{q-1} \langle r^{\tilde{q}}, s^{q-1} \rangle_A$ 
  - When  $\tilde{q} = q$ , one has  $r^q \cdot r^q = 0 - \alpha^{q-1} \langle r^q, s^{q-1} \rangle_A$  for the last term in the sum
  - When  $\tilde{q} > q$ , one has  $0 = 0 - \alpha^{q-1} \langle r^{\tilde{q}}, s^{q-1} \rangle_A$ , so only the last term in the sum is nonzero
- Finally,  $s^q = r^q + \frac{r^q \cdot r^q}{\alpha^{q-1} \langle s^{q-1}, s^{q-1} \rangle_A} s^{q-1} = r^q + \frac{r^q \cdot r^q}{r^{q-1} \cdot r^{q-1}} s^{q-1}$

# Conjugate Gradients (Method)

- Start with:  $s^1 = r^1 = b - Ac^1$
- Iterate:
  - $\alpha^q = \frac{r^q \cdot r^q}{\langle s^q, s^q \rangle_A}$
  - $c^{q+1} = c^q + \alpha^q s^q$  and  $r^{q+1} = r^q - \alpha^q As^q$  (both as usual)
  - $s^{q+1} = r^{q+1} + \frac{r^{q+1} \cdot r^{q+1}}{r^q \cdot r^q} s^q$
- Note: Gram-Schmidt drifts, making search directions less A-orthogonal over time; thus, occasionally throw out all search directions and start over with  $s^1 = r^1 = b - Ac^1$

# Non-Symmetric and Indefinite

- GMRES, MINRES, BiCGSTAB, etc...
- Generally speaking, iterative methods for non-symmetric and/or indefinite matrices are less stable, more error prone, and slower than CG on an SPD matrix

# Unit 6

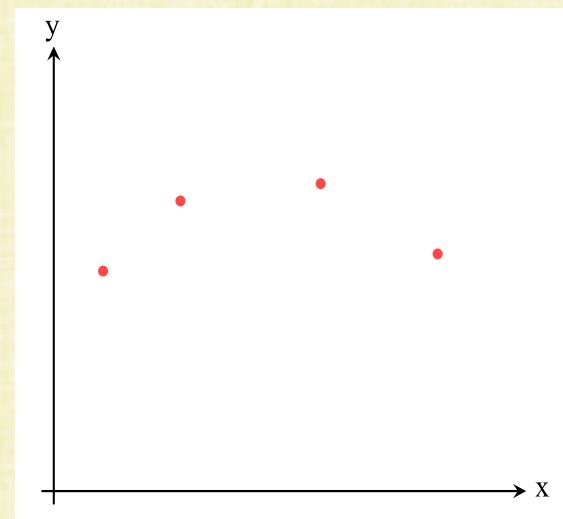
# Local Approximations

# Taylor Expansion

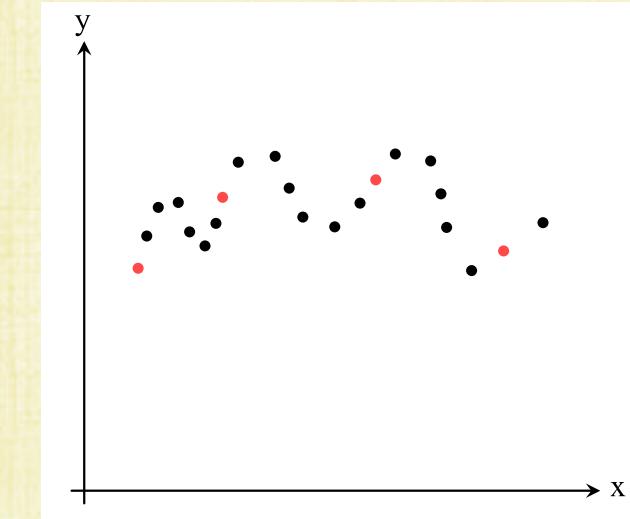
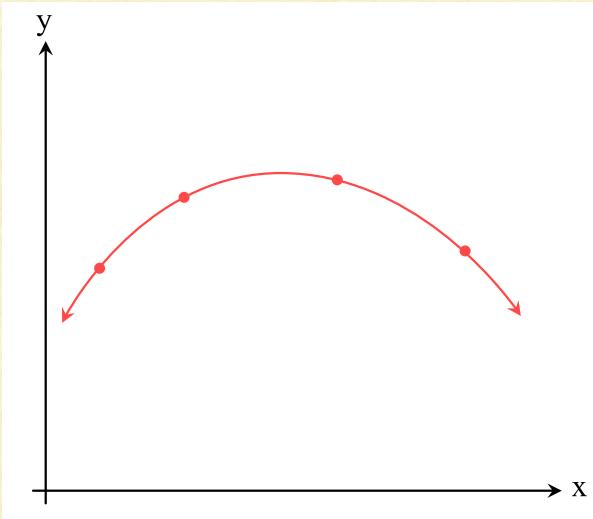
- $f(x + h) = \sum_{p=0}^{\infty} \frac{h^p}{p!} f^{(p)}(x) = \sum_{p=0}^{\hat{p}} \frac{h^p}{p!} f^{(p)}(x) + O(h^{\hat{p}+1})$
- Bounded derivatives would indicate that  $O(h^{\hat{p}+1}) \rightarrow 0$  as  $h \rightarrow 0$
- Examples:
  - $f(x + h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + O(h^4)$  forward difference
  - $f(x - h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(x) + O(h^4)$  backward difference
- Approximations (truncated Taylor expansions) become more valid as  $h \rightarrow 0$ 
  - $f(x + h) \approx f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x)$
  - $f(x - h) \approx f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(x)$

# Sampling

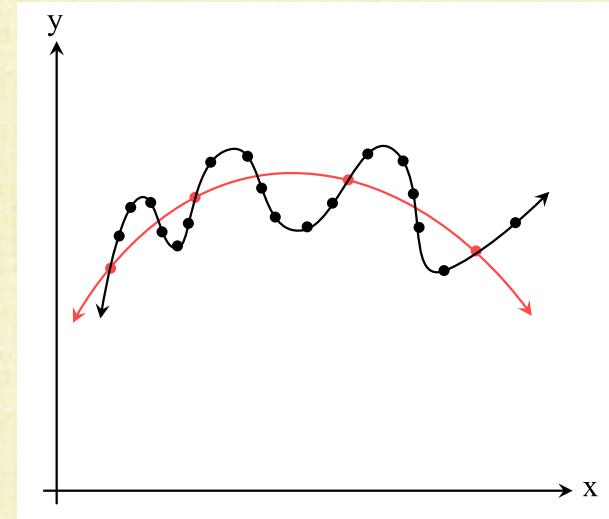
- Accurate approximation of a function is often limited by the amount of available data
- Given too few samples (left), one may "hallucinate" an incorrect function
- Adding more data allows better/proper feature resolution (right)
- Given "enough" sample points, a function tends to not vary too much in between them



under-resolved



resolved better with more data

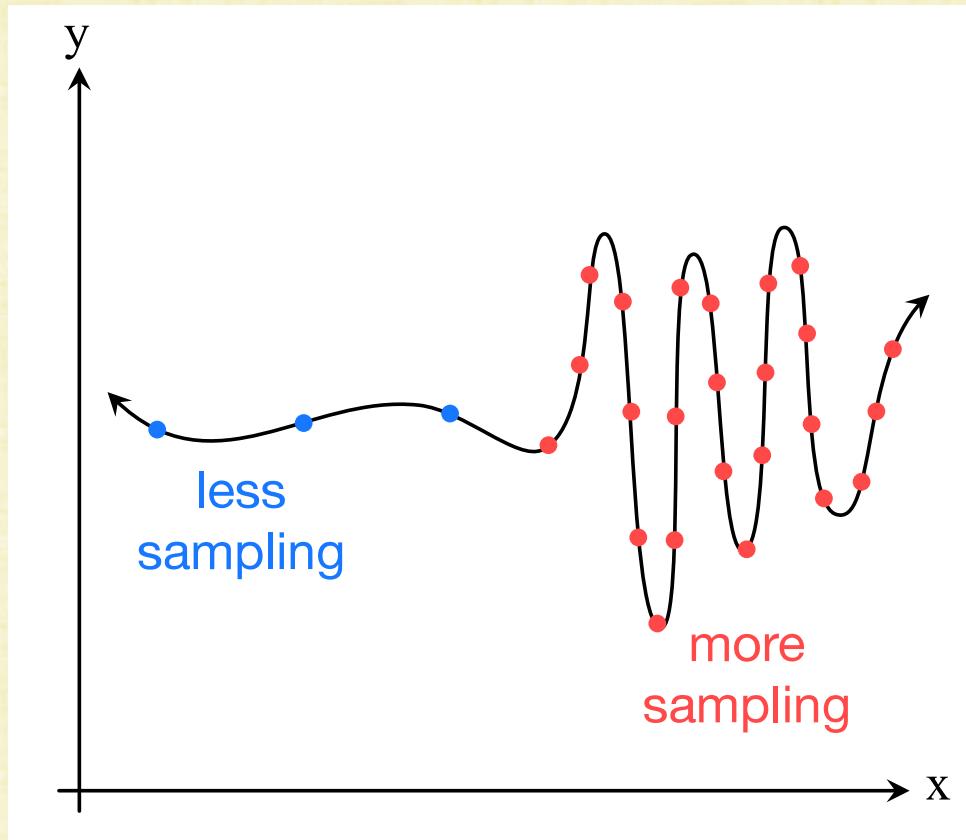


# Well-Resolved Functions

- The Taylor expansion approximates a function  $f$  at a new location  $x + h$  based on known information at a nearby point  $x$
- When the sample points are “closely” spaced, new locations are “close” to known sample points making  $h$  “small” enough
- However, large derivative values can overwhelm even a small  $h$
- Thus, functions with more variation need higher sampling rates
  - Similarly, smoother functions can utilize lower sampling rates
- Well-resolved functions have vanishing high order terms in their Taylor expansion making truncated Taylor expansions more valid

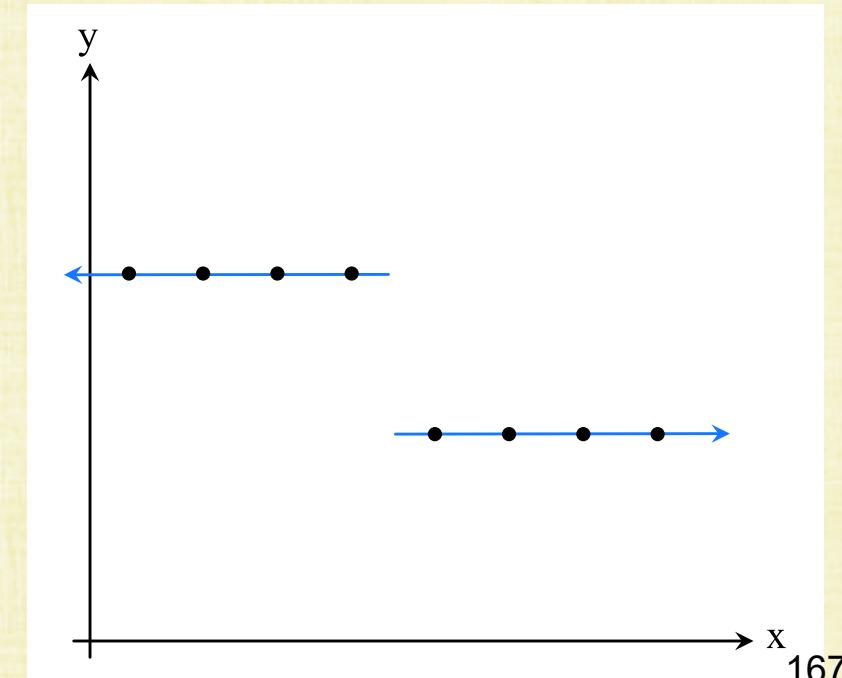
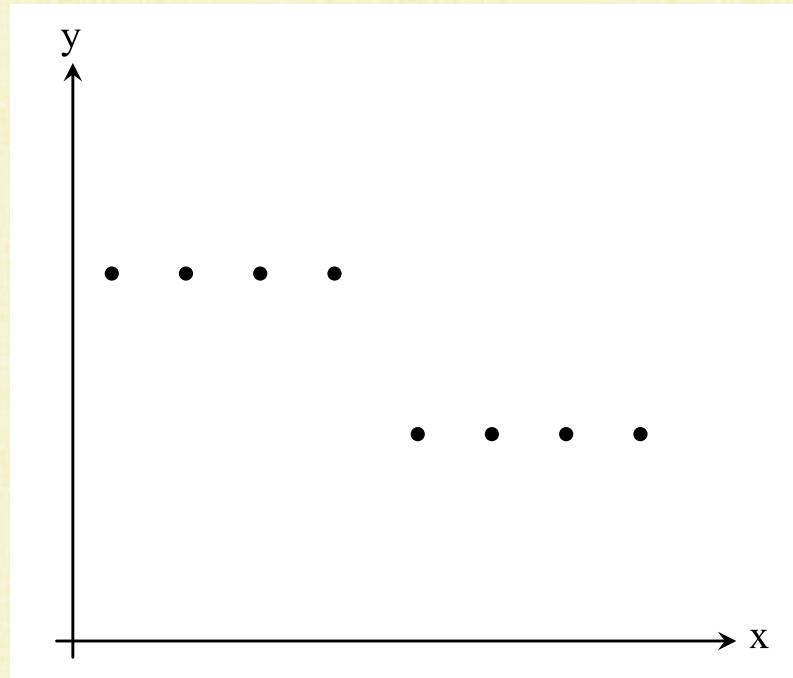
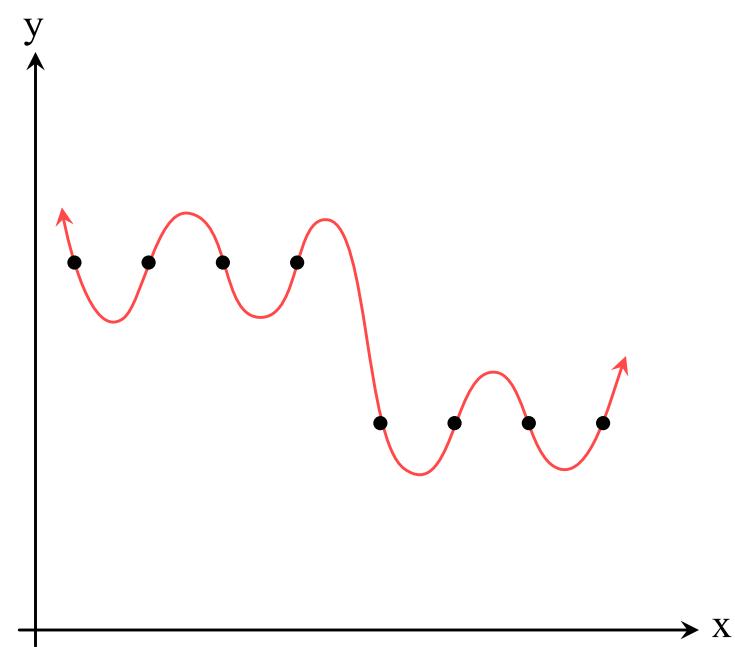
# Well-Resolved Functions

- Regions of a function with less/more variation require lower/higher sampling rates



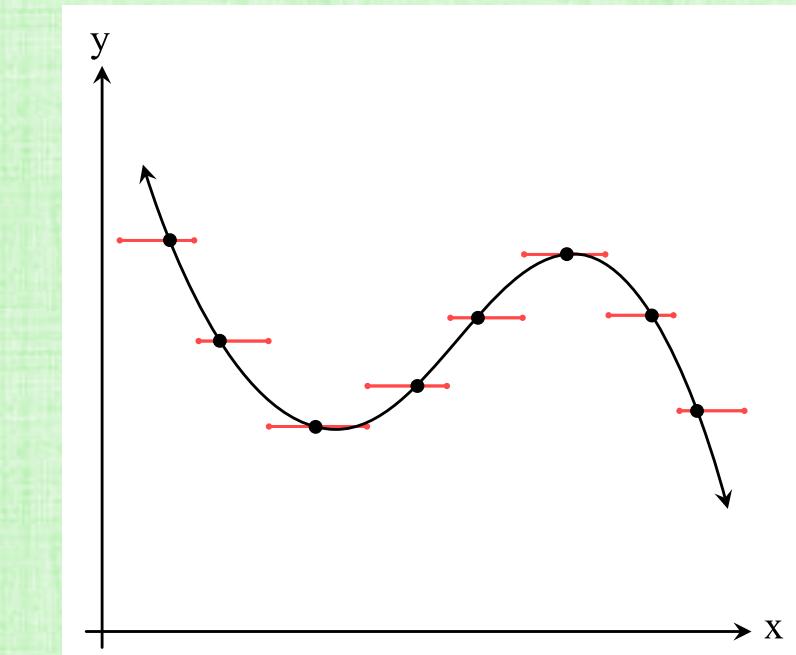
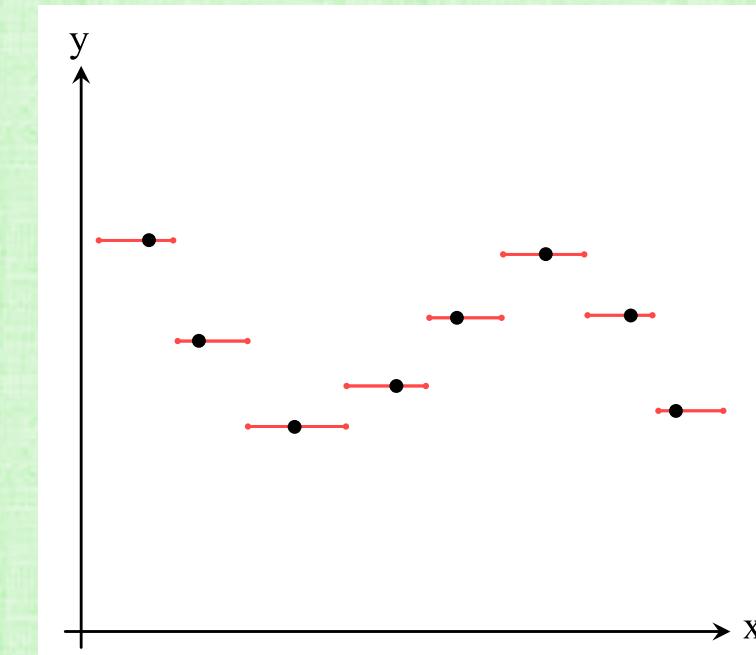
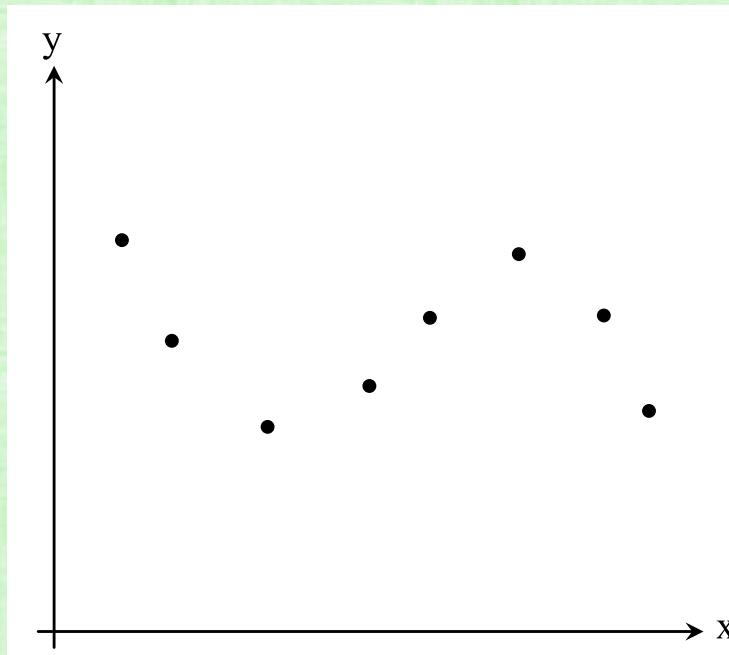
# Piecewise Approximation

- Piecewise approximation enables the use of simpler models to approximate (potentially disjoint) subsets of data
  - ML/DL community: “sub-manifold” often means a coherent/smooth subset



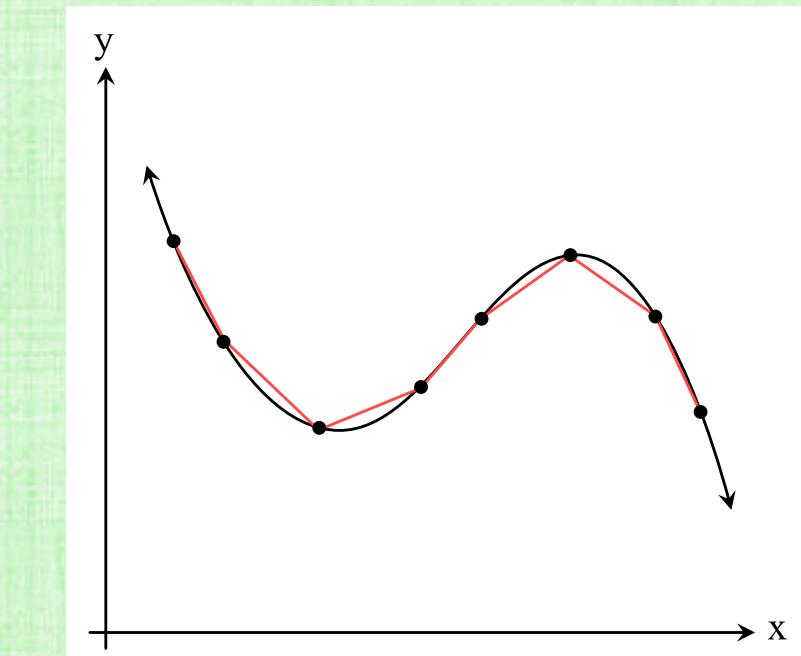
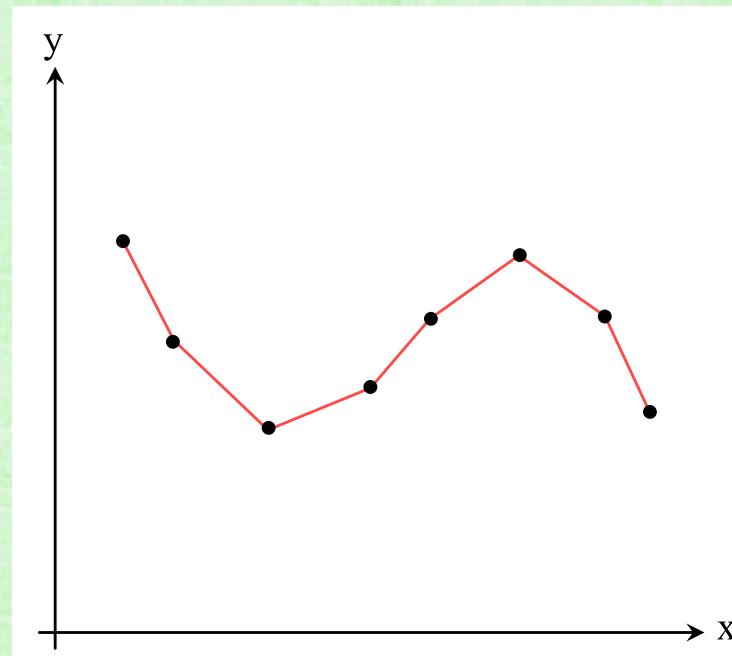
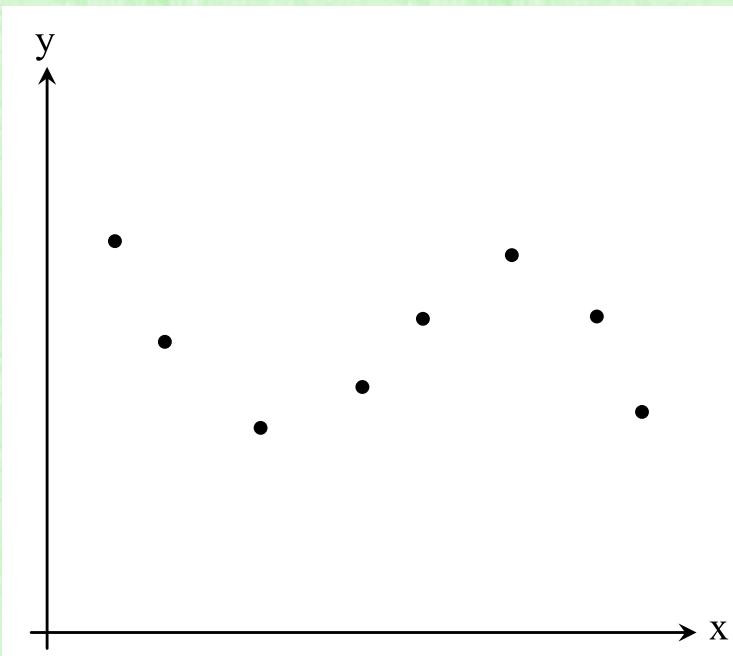
# Piecewise Constant Interpolation

- Use the first term in the Taylor expansion (only):  $f(x + h) \approx f(x)$
- Errors are  $O(h)$ , since  $f(x + h) = f(x) + O(h)$
- Recall: nearest neighbor is also piecewise constant



# Piecewise Linear Interpolation

- Use the first two terms in the Taylor expansion:  $f(x + h) \approx f(x) + hf'(x)$
- Errors are  $O(h^2)$ , since  $f(x + h) = f(x) + hf'(x) + O(h^2)$

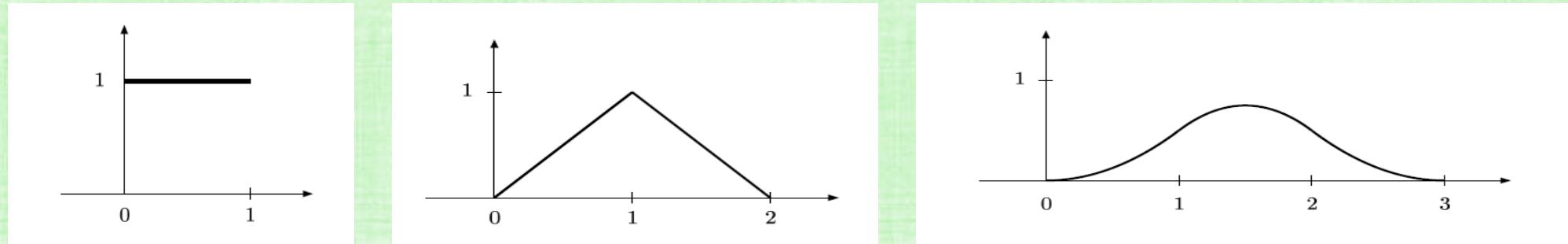


# Higher Order Piecewise Interpolation

- Piecewise quadratic interpolation uses the first three terms in the Taylor expansion and has  $O(h^3)$  errors
- Piecewise cubic interpolation uses the first four terms in the Taylor expansion and has  $O(h^4)$  errors
- Recall: higher order interpolation becomes more oscillatory (i.e. overfitting)
  - These oscillations are sometimes referred to as Gibbs phenomena

# Cubic Splines

- Piecewise cubic splines are quite popular because of their ability to match derivatives across approximation boundaries
- B-splines – hierarchical family:  $\phi_i^p$  is a piecewise polynomial of degree  $p$ 
  - Piecewise constant:  $\phi_i^0(x) = 1$  for  $x \in [x_i, x_{i+1}]$  and 0 otherwise
  - A linear  $w_i^p(x) = \frac{x-x_i}{x_{i+p+1}-x_i}$  increases the polynomial degree of  $\phi^p$  to  $\phi^{p+1}$
  - Recursively:  $\phi_i^{p+1}(x) = w_i^p(x)\phi_i^p(x) + (1 - w_{i+1}^p(x))\phi_{i+1}^p(x)$
  - Piecewise linear  $\phi_i^1$ , piecewise quadratic  $\phi_i^2$ , piecewise cubic  $\phi_i^3$ , etc.



# 2D Image Segmentation

- Divide an image's pixels into separate regions representing objects or groups of objects
- Before neural networks: Methods relied on clustering in color and/or space, graph-cuts, edge detection, etc.
- More recently: The hope is that neural networks can better mimic human perception/semantics (since humans do well on this problem)
- Training examples:
  - Input: an image (all the pixel RGB values)
  - Output: labels on all the pixels, indicating what group each pixel is in

# (Example) Bool Output Labels

- Binary segmentation of an image using binary values
- E.g. true = dog, false = not dog



Input



Output

# (Example) Integer Output Labels

- Multi-object segmentation with an integer for each object
- E.g. 1=cat, 2=dog, 3=human, 4=mug, 5=couch, 6=everything else



Input



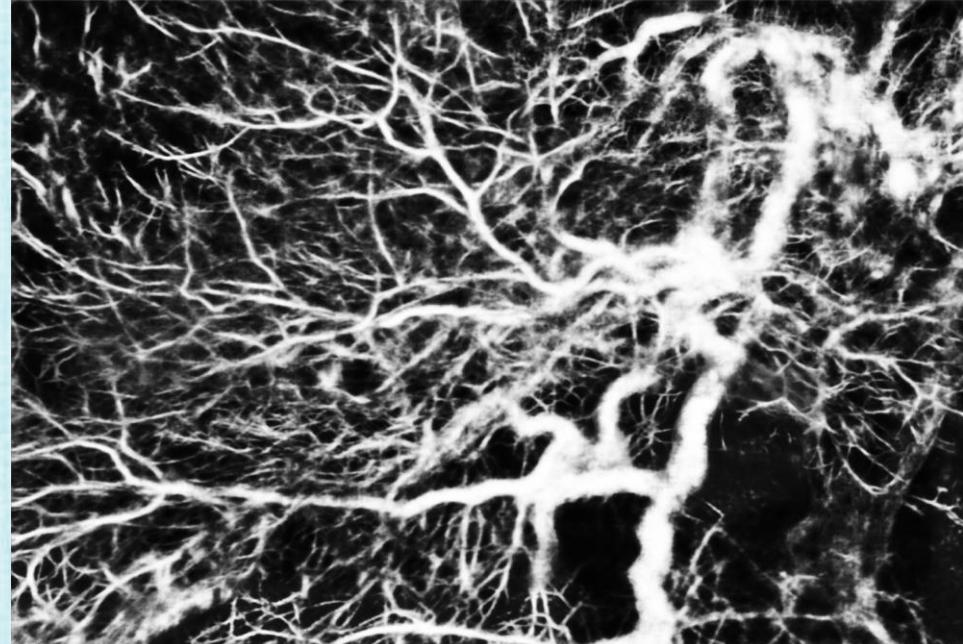
Output

# (Example) Real Number Output Labels

- Probabilistic segmentation with real number values in [0,1]
- E.g. 1=tree branch, .8=probably a branch, .2=probably not a branch, etc.



Input



Output

# Segmenting Botanical Trees

Difficult Problem:

- Trees are large-scale and geometrically-complex structures
- Branches severely occlude each other
- The images have limited pixel resolution of individual branches
- Even humans have a hard time ascertaining the correct topological structure from a single image/view
- Can we train a neural network to help?

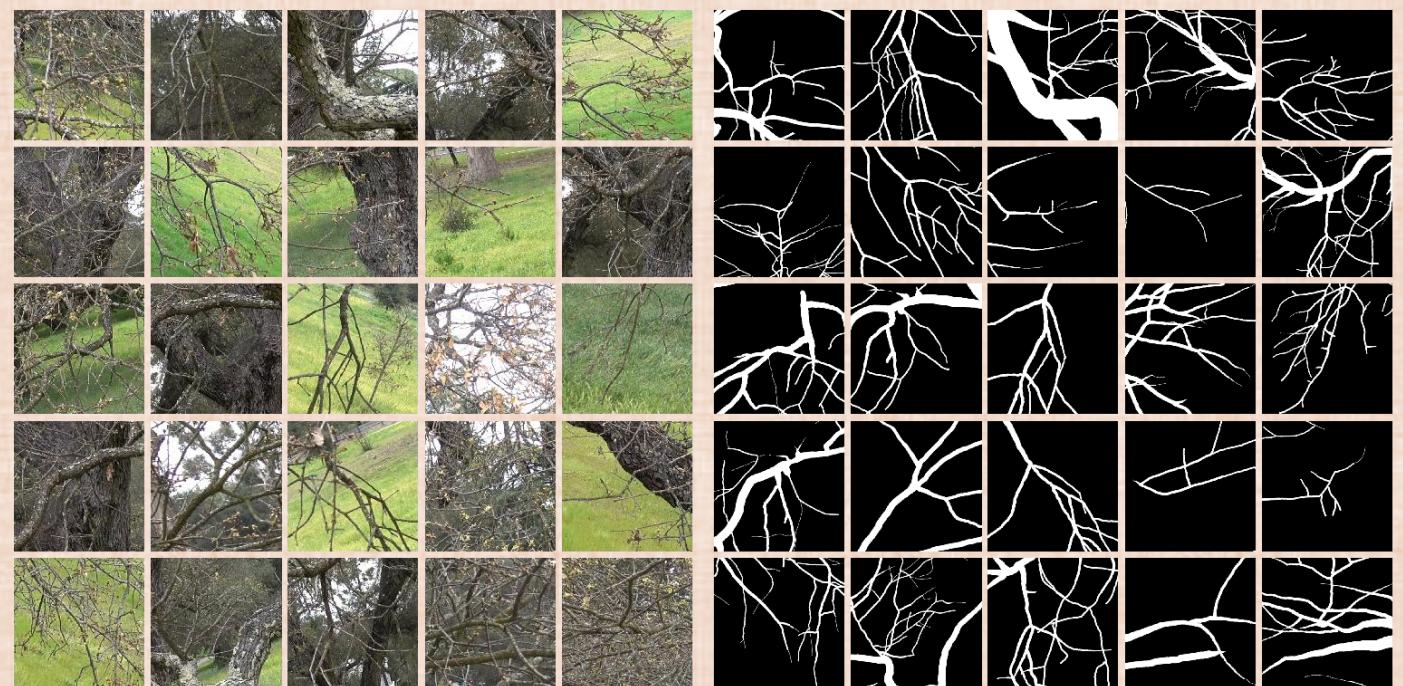
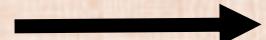
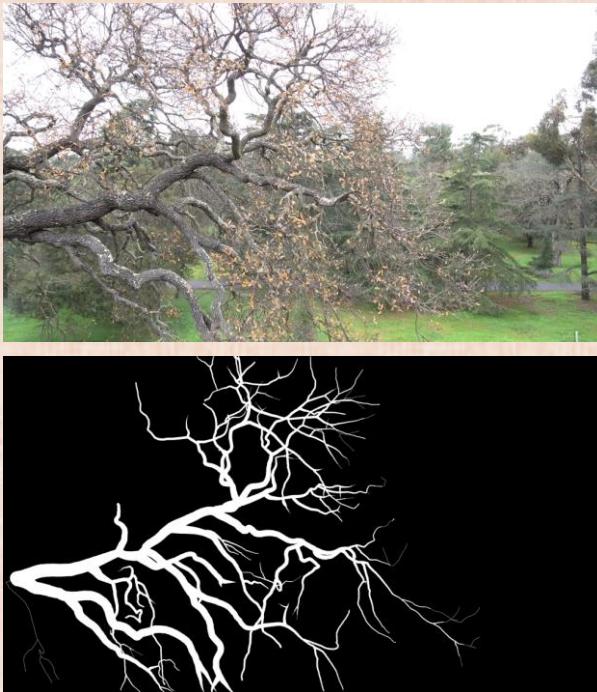
# Constructing Training Data

- Begin with a dataset of labels (tediously) created by hand
- Draw lines and thicknesses on top of branches; then, use this information to create a binary mask for the image



# Constructing (More) Training Data

- Artificially increase the amount of training data by taking various image subsets
- This also helps to avoid down-sampling (networks use low-resolution images)

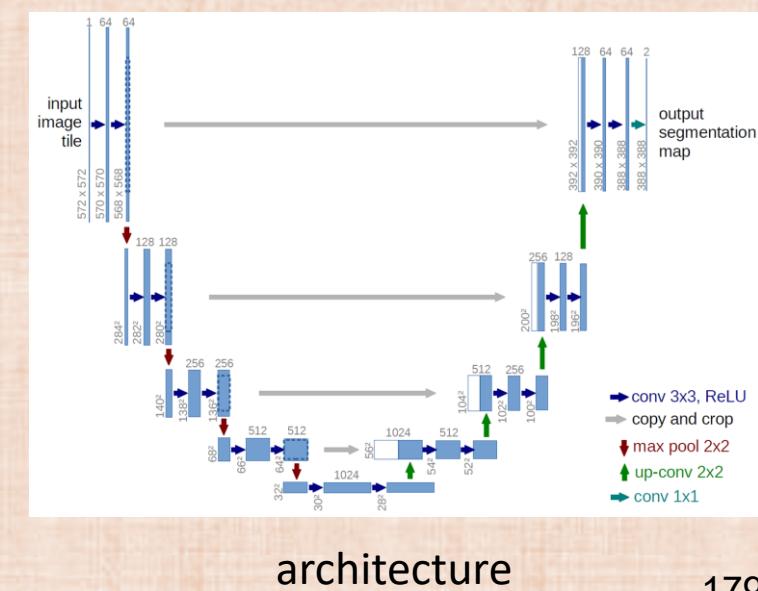


3840 pixels wide, 2160 pixels tall

512 pixels wide, 512 pixels tall

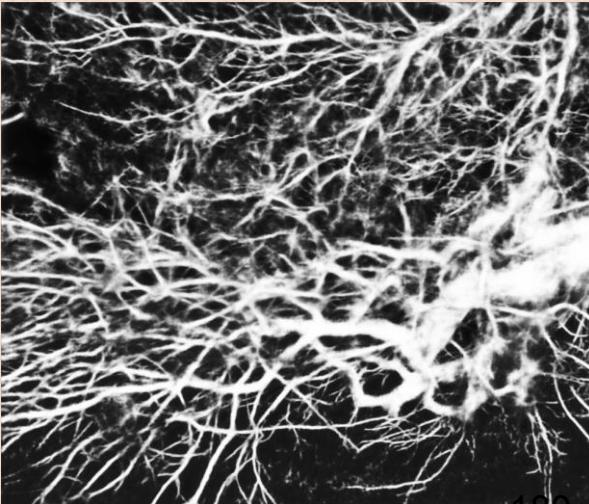
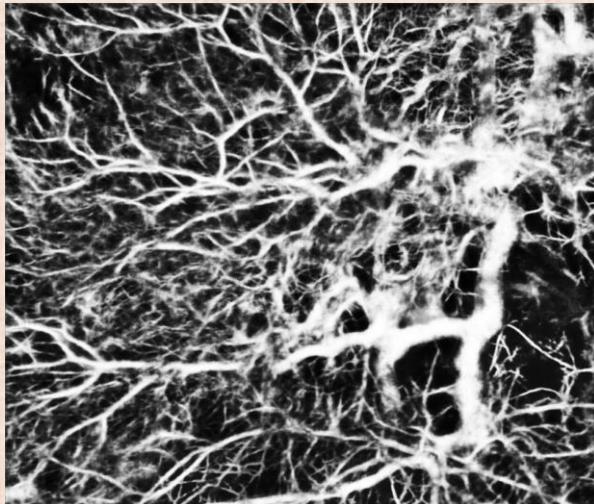
# Training the Neural Network

- Find function parameters  $c$  such that the network function  $f_c(x)$  gives minimal error on the training data (i.e. minimize network “loss”)
- The network should predict the known target labels (or close to it) from the input images



# Network Inference/Prediction

- After training, use the resulting network function  $f_{c_{trained}}(x)$  to infer/predict labels for new images (not previously hand-labeled)



# Local Approximations

- Roughly speaking, input images mostly seem to be of two different types: either (1) branches over grass or (2) clusters of branches



# Local Approximations

- Train 2 Neural Networks:
  - Divide the training data into these two disparate groups
  - Train a separate network on each group of data: separate architecture, separate trainable parameters, etc.
- Network Inference:
  - Given an input image, inference it (separately) on both networks
  - Then combine the two predictions, using the network that makes the most sense locally in each part of the image (blending predictions when appropriate)

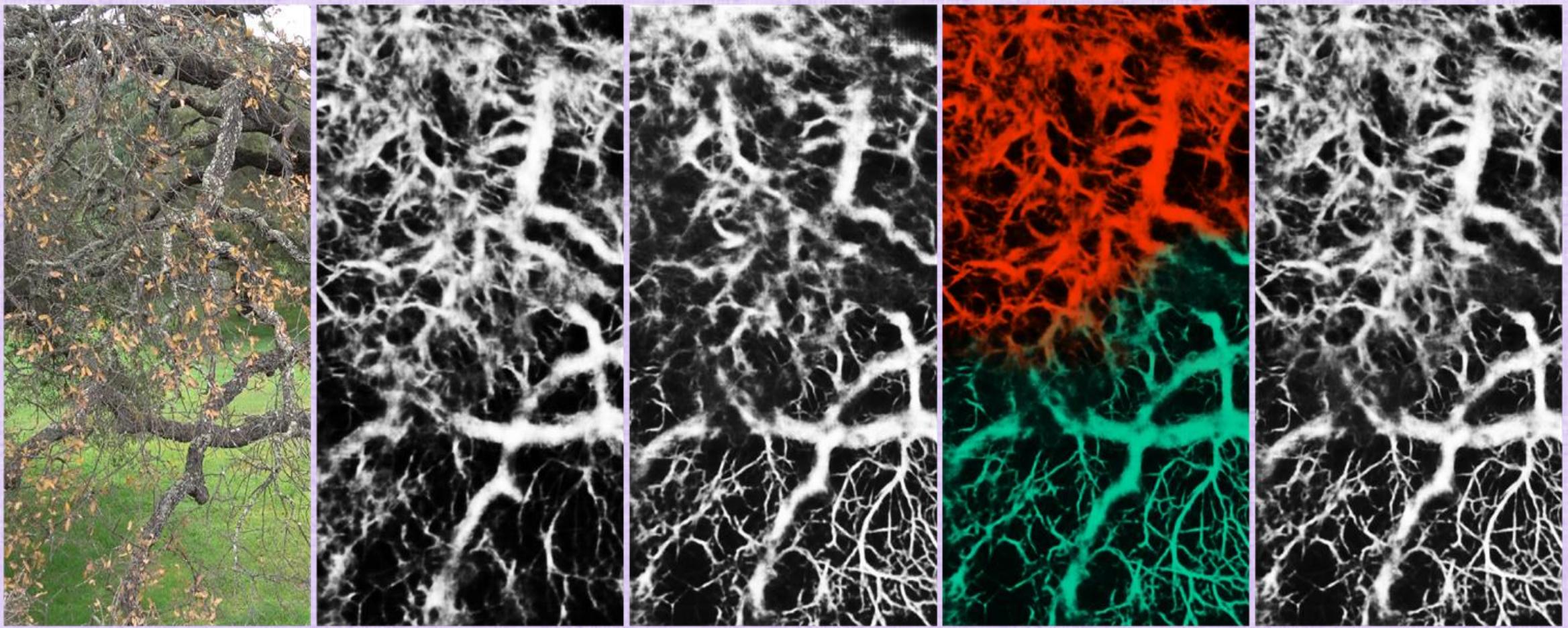
# Combining Inference Outputs

- k-means clustering on hue/saturation was used to divide the training images into 2 separate clusters; then, each cluster was used to train a network

To inference each pixel:

- Compute hue/saturation values on a small patch around the pixel
- Find the distances from the patch hue/saturation values to the 2 cluster centers
- Interpolate the outputs from the 2 networks using those distances
- The closer a pixel is to a k-means cluster, the more weight is given to that cluster's network inference/prediction

# Example



Input

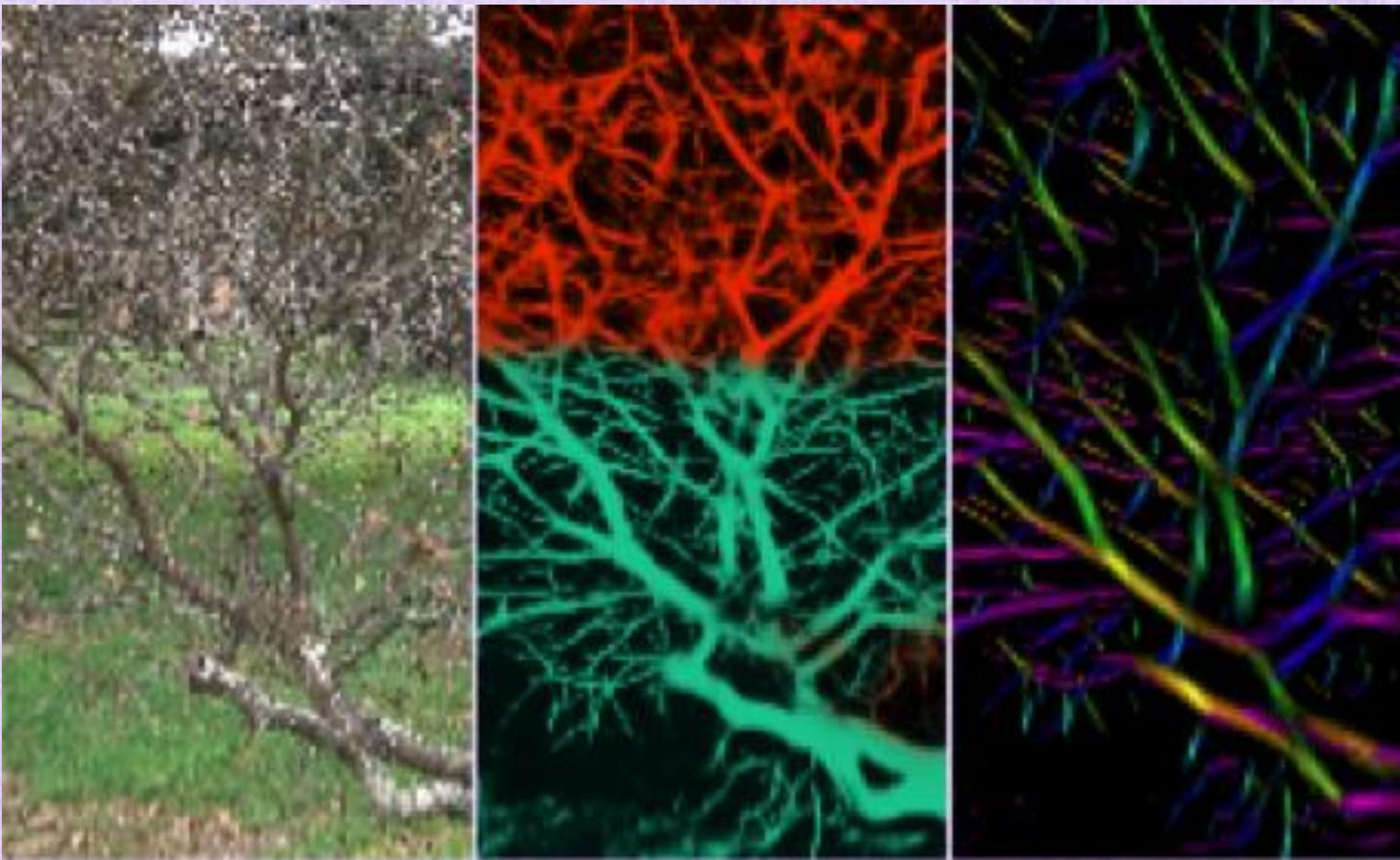
Network 1

Network 2

Combine

Final Result

# Aside: Branch Estimation





# Unit 7

# Curse of Dimensionality

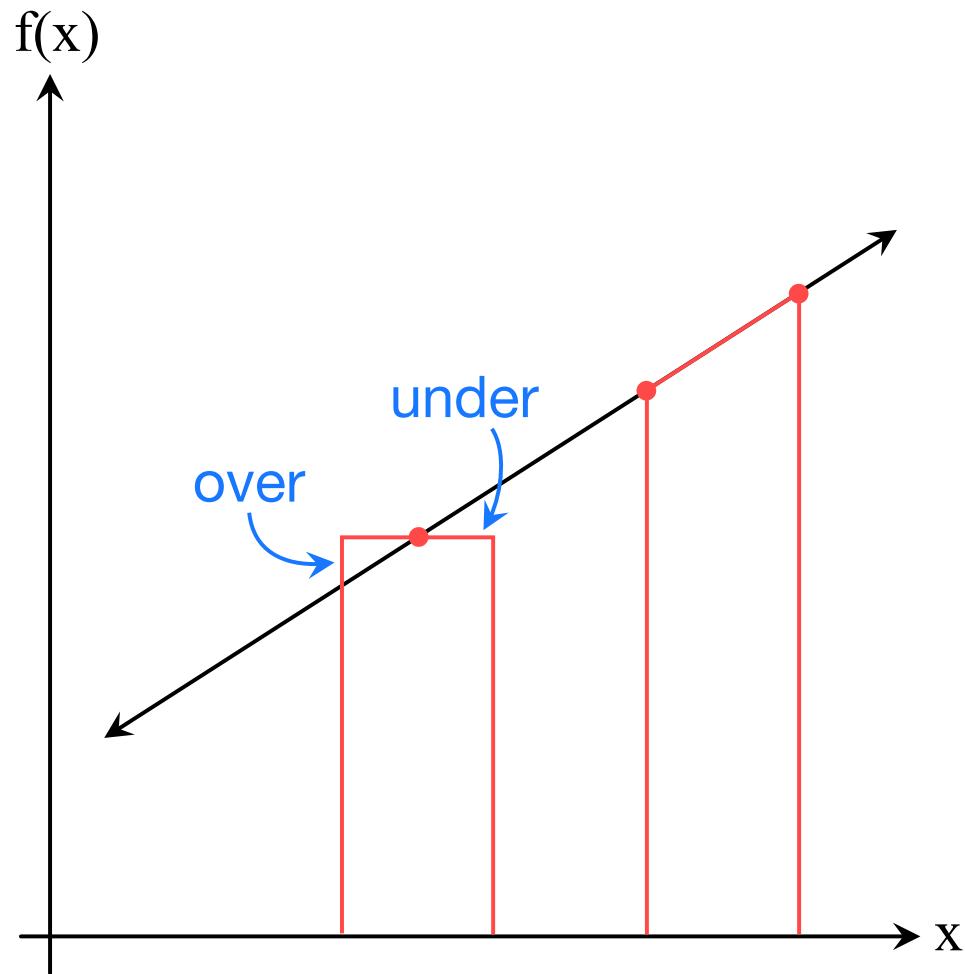
# Numerical Integration (Quadrature)

- Approximate  $\int_{x_L}^{x_R} f(x)dx$  numerically
- Break up  $[x_L, x_R]$  into subintervals, and consider each subinterval separately
- On each subinterval:
  - Reconstruct the function
  - Analytically find the area under the reconstructed curve
- These two steps can be combined in various ways (for efficiency)
- $f$  is often not explicitly known
- I.e., often only have access to output values  $f(x_i)$  given input values  $x_i$
- In addition, could be expensive to evaluate  $f(x_i)$ , especially when it requires running code

# Newton-Cotes Quadrature

- On each subinterval, choose  $p$  equally spaced points and use  $p - 1$  degree polynomial interpolation to reconstruct the function and approximate the area under the curve
- Obtains the exact solution when  $f$  is a degree  $p - 1$  polynomial (as expected)
- When the number of points  $p$  is odd, symmetric cancellation gives the exact solution on a degree  $p$  polynomial (1 degree higher than expected)

# Symmetric Cancellation



- When  $p = 2$  points, the 1<sup>st</sup> degree piecewise linear approximation integrates piecewise linear functions exactly
- When  $p = 1$  point, the 0<sup>th</sup> degree piecewise constant approximation (also) integrates piecewise linear function exactly
  - Note the cancellation of under/over approximations in the figure

# Newton-Cotes Quadrature (Examples)

- Consider a total of  $m$  intervals
- Piecewise constant approximation ( $p = 1$  point) uses a total of  $m$  points to integrate piecewise linear functions exactly
- Piecewise linear approximation ( $p = 2$  points) uses a total of  $m + 1$  points to integrate piecewise linear functions exactly
  - points on the boundary between intervals are used for both intervals
- Piecewise quadratic approximation ( $p = 3$  points) uses a total of  $2m + 1$  points to integrate piecewise cubic functions exactly
- Piecewise cubic approximation ( $p = 4$  points) uses a total of  $3m + 1$  points to integrate piecewise cubic functions exactly

# Local and Global Error

- Degree  $p$  polynomial reconstruction captures the Taylor expansion terms up to and including  $\frac{h^p}{p!} f^{(p)}(x)$ , with  $O(h^{p+1})$  errors
- This  $O(h^{p+1})$  error in the height of the function multiplied times the  $O(h)$  width of the interval gives per interval local area error of  $O(h^{p+2})$
- The total number of intervals is  $\frac{x_R - x_L}{O(h)} = O\left(\frac{1}{h}\right)$ , so the total global error is  $O\left(\frac{1}{h}\right) O(h^{p+2}) = O(h^{p+1})$
- Doubling the number of intervals halves their size leading to  $\left(\frac{1}{2}\right)^{p+1}$  as much error, which is denoted an order of accuracy of  $p + 1$

# Newton-Cotes Quadrature (Examples)

- Midpoint Rule:  $\sum_i h_i f(x_i^{mid})$ 
  - 1 point, piecewise constant, exact for piecewise linear, 2<sup>nd</sup> order accurate
- Trapezoidal Rule:  $\sum_i h_i \frac{f(x_i^{left}) + f(x_i^{right})}{2}$ 
  - 2 points, piecewise linear, exact for piecewise linear, 2<sup>nd</sup> order accurate
- Simpson's Rule:  $\sum_i h_i \frac{f(x_i^{left}) + 4f(x_i^{mid}) + f(x_i^{right})}{6}$ 
  - 3 points, piecewise quadratic, exact for piecewise cubic, 4<sup>th</sup> order accurate

# Gaussian Quadrature

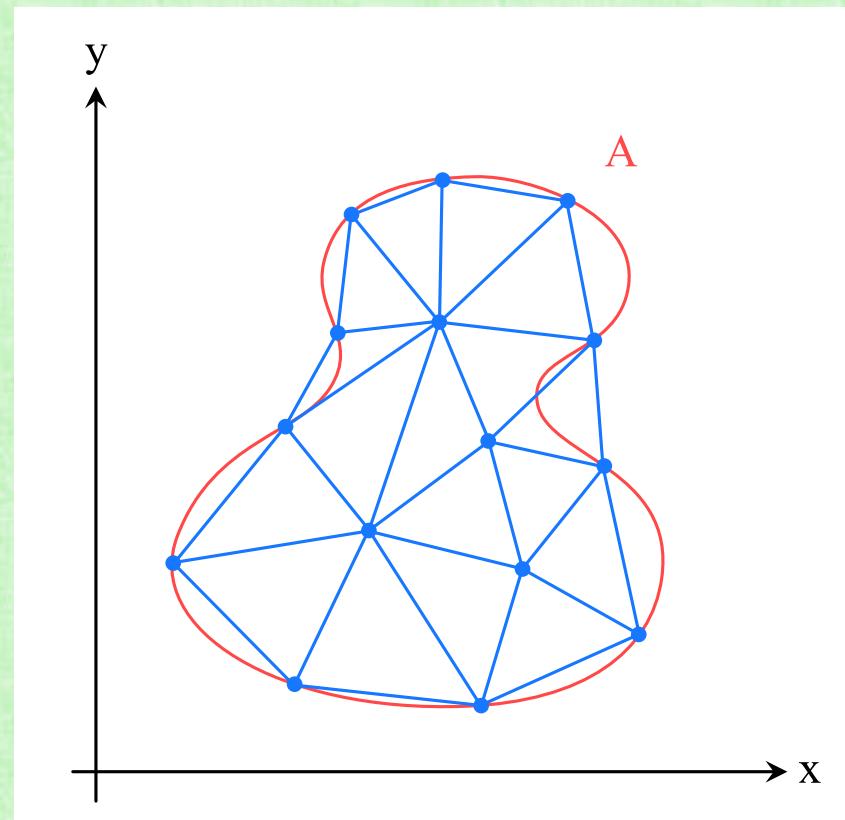
- Use  $p$  optimally chosen points to obtain a method that is exact on degree  $2p - 1$  polynomials, and thus has an order of accuracy of  $2p$
- For example:  $\sum_i h_i \frac{f\left(x_i^{mid} - \frac{h_i}{2\sqrt{3}}\right) + f\left(x_i^{mid} + \frac{h_i}{2\sqrt{3}}\right)}{2}$
- 2 points, piecewise cubic, exact for piecewise cubic, 4<sup>th</sup> order accurate
- Same accuracy as Simpson's 3 point rule
  - Simpson has 1 point on shared boundaries, so only  $2m + 1$  total points are required
  - That is, Gaussian quadrature only saves 1 point in total ( $2m$  total points)

# Two Dimensions

- $\iint_A f(x, y)dA$  where sub-regions  $dA$  of area  $A$  are considered separately
- When  $A$  is rectangular, it can be broken into sub-rectangles and addressed dimension-by-dimension using 1D techniques
- When  $A$  is more interesting, triangle sub-regions can be used to approximate it
- The difference between  $A$  and its approximation leads to a new source of error not seen in 1D (where interval boundaries were merely points)

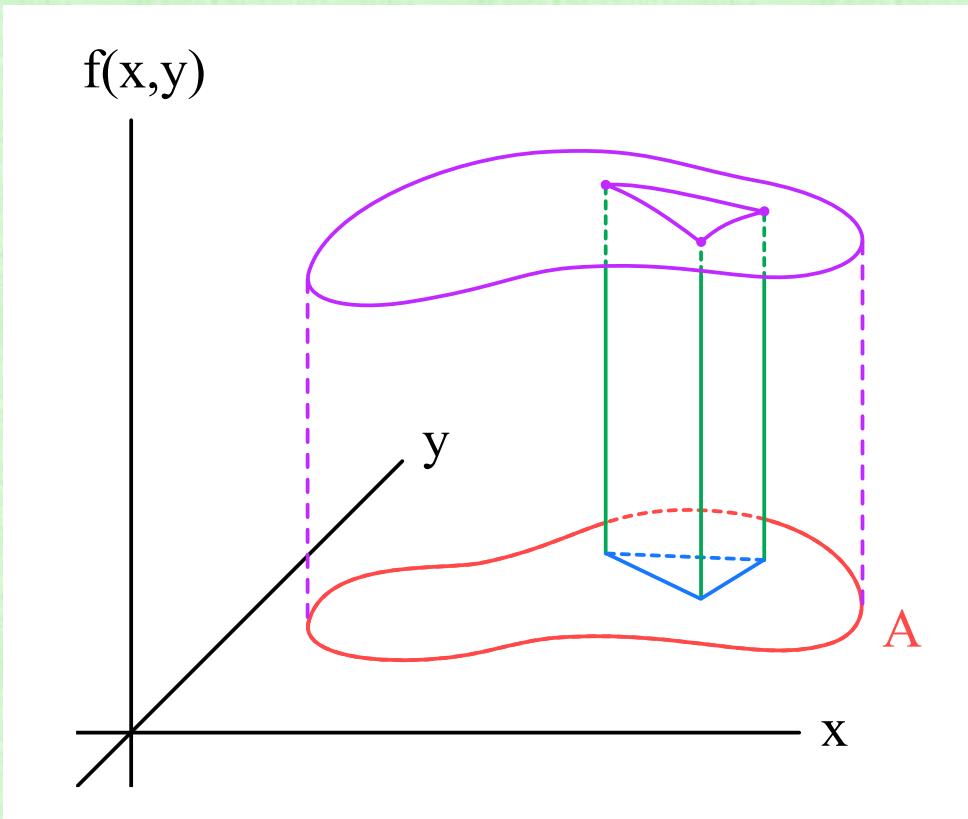
# Domain Approximation Errors

- The difference between  $A$  and its approximation (via triangles here) leads to a new source of error in the integral (missing/extraneous area)



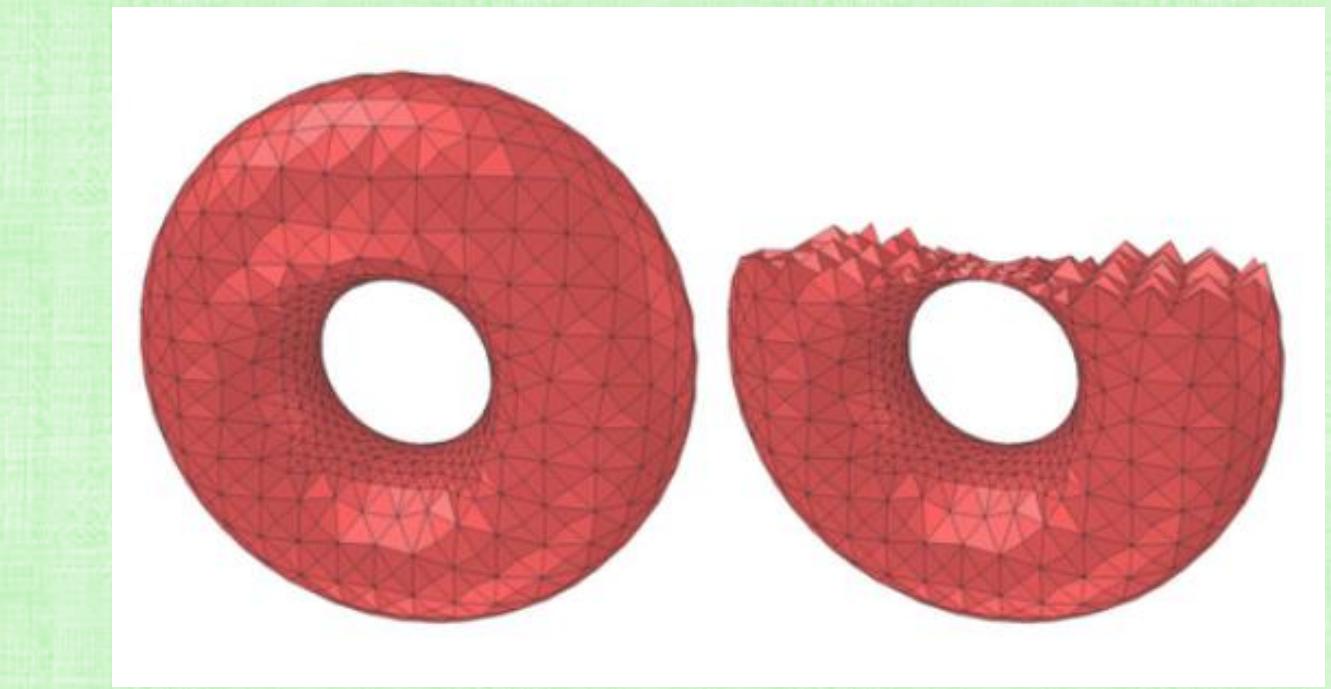
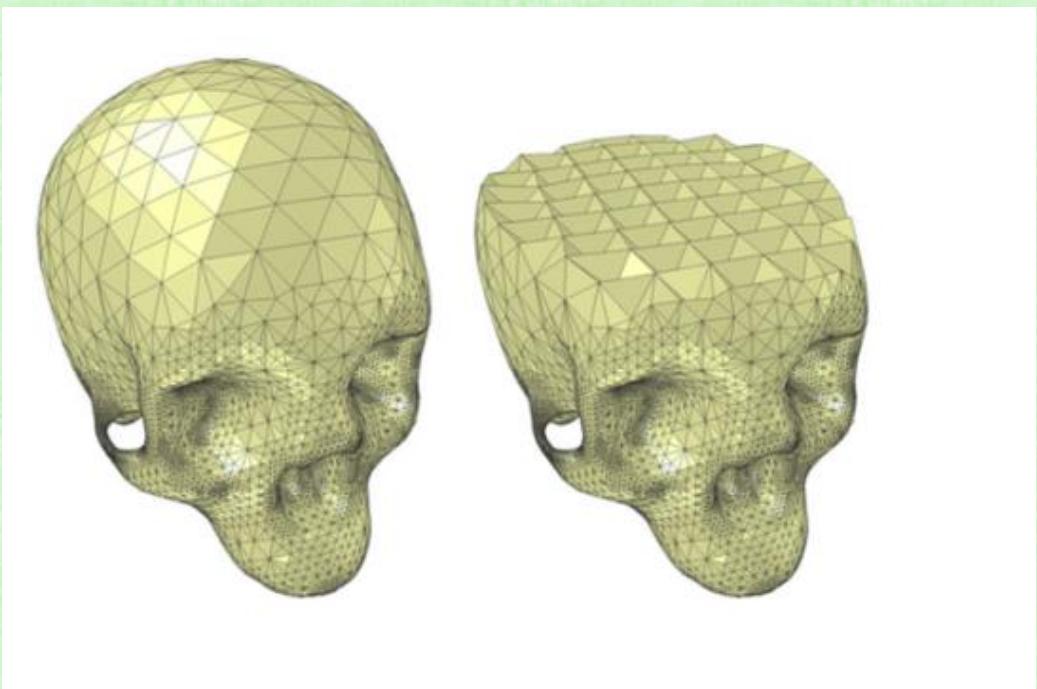
# Integrating over Sub-regions

- Each triangle sub-region utilizes optimally chosen Gaussian quadrature points to compute sub-volumes



# Three Dimensions

- $\iint_V f(x, y, z) dV$  where tetrahedral sub-regions  $dV$  of volume  $V$  are each considered separately (with Gaussian quadrature points)



# Curse of Dimensionality

- Consider a 1<sup>st</sup> order accurate method
- 1D: doubling the number of intervals cuts the error in half ( $2x$  work =  $\frac{1}{2}$  error)
- 2D: halving interval size requires 4 times the rectangles/triangles ( $4x$  work =  $\frac{1}{2}$  error)
- 3D: halving interval size requires 8 times the cubes/boxes/tets ( $8x$  work =  $\frac{1}{2}$  error)
- 4D:  $16x$  work =  $\frac{1}{2}$  error, 5D:  $32x$  work =  $\frac{1}{2}$  error, etc.
- Cutting error by a factor of 4 in 5D takes  $32^2=1024x$  work
- Cutting error by a factor of 8 in 5D takes  $32^3=32,768x$  work
- If the original code took 1 sec to run in 5D, cutting error by a factor of 8 takes 9 hours
- And cutting error by a factor of 16 takes 12 days
- And cutting the error by a factor of 32 takes over a year....

**Yep, you're cursed**

# Curse of Dimensionality

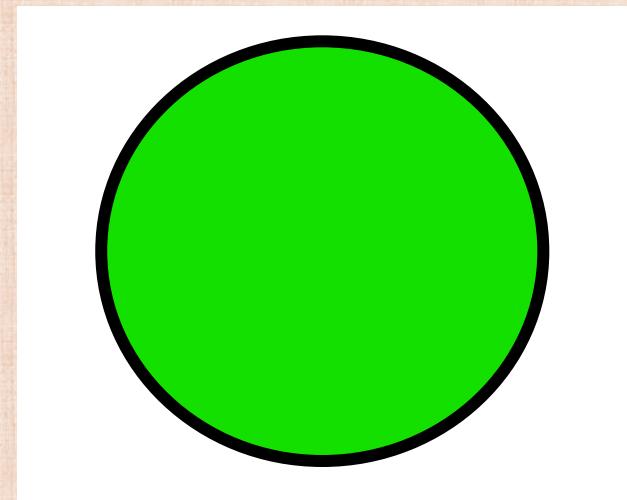
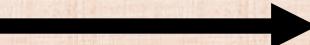
- Consider a 2<sup>nd</sup> order accurate method
- In 1D/2D/3D/4D/5D/etc. halving the interval size gives 4 times less error
- Cutting error by a factor of 4 in 5D takes 32x work
- If the original code took 1 sec to run in 5D, cutting error by a factor of 16 takes only 17 min (much faster than the 12 days for the 1<sup>st</sup> order accurate method)
- Cutting error by a factor of 1024 (just 3 decimal places more accuracy) takes over a year...
- In 10D, cutting error by a factor of 4 takes 1024x work
- Second order is better than first, but still intractable in higher dimensions
- Moreover, it's difficult/impossible to construct higher order methods in higher dimensions (and overfitting is a concern too)

# Conclusion

- Newton-Cotes style approaches are only practical for 1D/2D/3D
  - or 1D/2D/3D + time

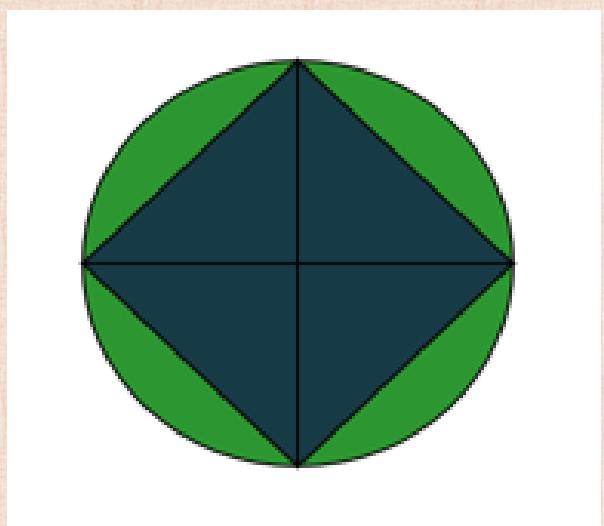
# A Simple Example

- Consider approximating  $\pi = 3.1415926535 \dots$
- Use a compass to construct a circle with radius = 1
- Since  $A = \pi r^2$ , the area of the circle is  $\pi$
- Setting  $f(x, y) = 1$  gives  $\iint_A f(x, y)dA = \pi$
- So, compute the integral...

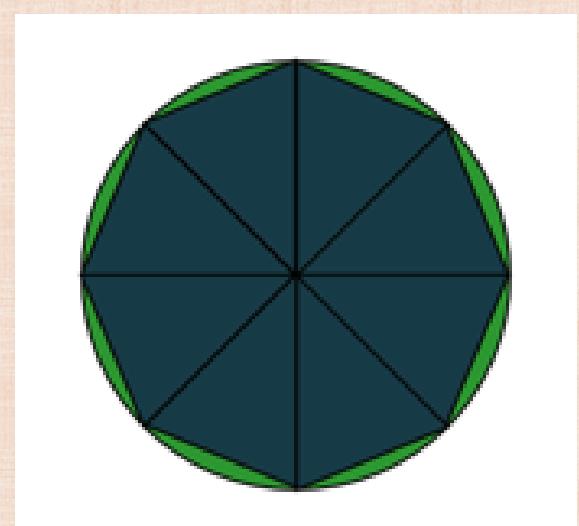


# Newton-Cotes Approach

- Inscribe triangles inside the circle
- The function  $f(x, y) = 1$  dictates computing the area of each triangle (and trivially multiplying by the height = 1)
- The difference between  $A$  and its approximation with triangles leads to errors



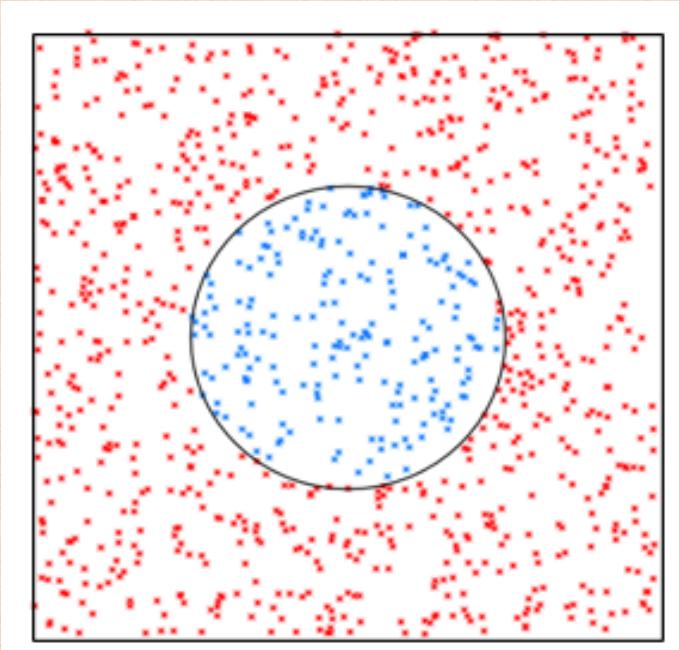
$$\pi \approx 2$$



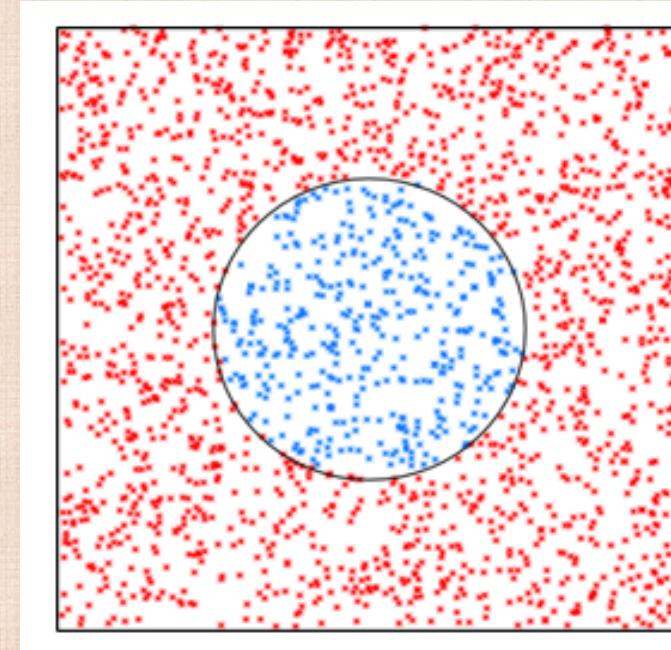
$$\pi \approx 2.8284$$

# Monte Carlo Approach

- Construct a square with side length 4 containing the circle
- Randomly generate  $N$  points in the square (color points inside the circle blue)
- Since  $\frac{A_{circle}}{A_{box}} = \frac{\pi}{16}$ , one can approximate  $\pi \approx 16 \left( \frac{N_{blue}}{N_{blue} + N_{red}} \right)$



$$\pi \approx 3.136$$



$$\pi \approx 3.1440$$

# Monte Carlo Methods

- Typically used in higher dimensions (5D or more)
- Random (pseudo-random) numbers generate sample points that are multiplied by “element size” (e.g. length, area, volume, etc.)
- Error decreases like  $\frac{1}{\sqrt{N}}$  where N is the number of samples (only  $\frac{1}{2}$  order accurate)
  - E.g. 100 times more sample points are needed to gain one more digit of accuracy
- **Very slow convergence, but independent of the number of dimensions!**
- Not competitive for lower dimensional problems (i.e., 1D, 2D, 3D), but the only tractable alternative for higher dimensional problems

# Machine Learning Implications

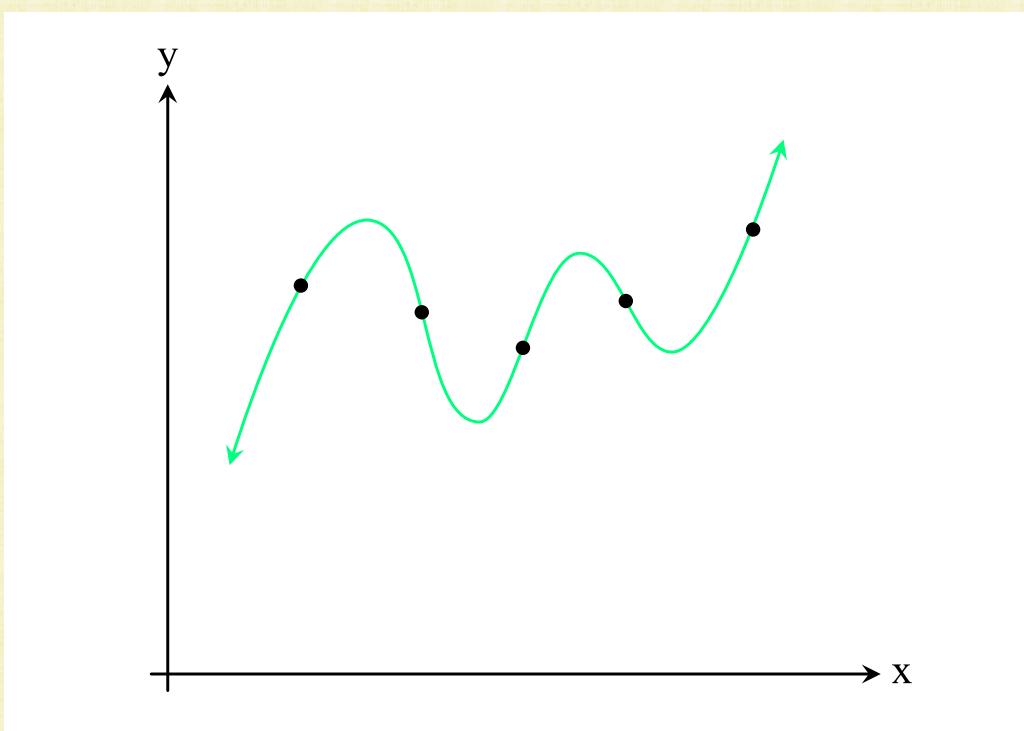
- Consider  $y = f(x)$  where  $x \in R^n$  with large  $n$
- Newton-Cotes style approaches first do polynomial interpolation, and then analytically integrate the result
- The curse of dimensionality occurs because of the sheer number of points and control volumes required to construct polynomial functions in higher dimensions
- The same sentiments hold true when constructing  $y = f(x)$  for function interpolation (e.g. inference), i.e. a higher dimensional  $x$  is intractable
- Thus, Monte Carlo approaches are far more efficient!
- This is a major reason for close collaborations between ML/DL and Statistics departments
  - as compared to classical engineering (which makes 3D models of the physical world, and as such has closer ties to Applied Mathematics)

# Unit 8

# Least Squares

# Recall: Polynomial Interpolation (Unit 1)

- Given  $m$  data points, one can (at best) draw a unique  $m - 1$  degree polynomial that goes through all of them
  - As long as they are not degenerate, like 3 points on a line

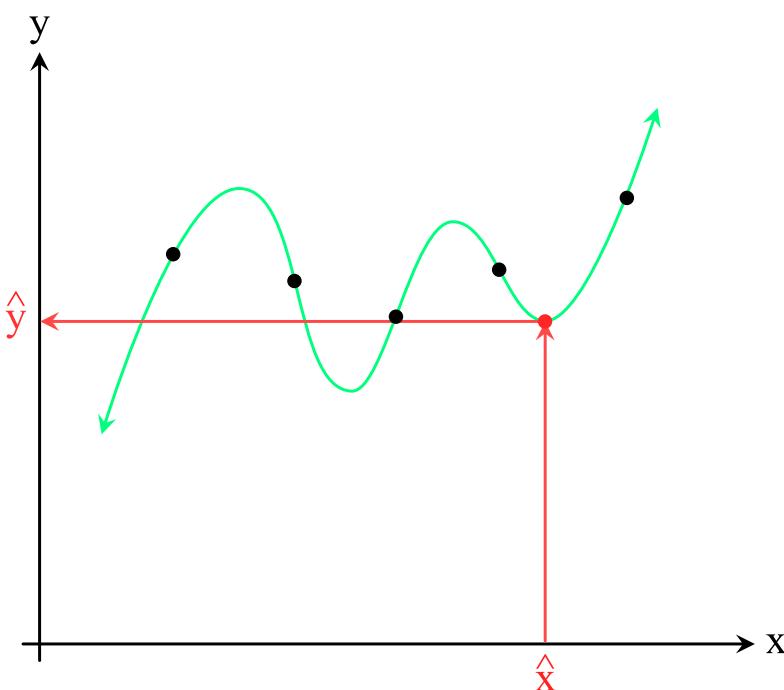


# Recall: Basis Functions (Unit 1)

- Given basis functions  $\phi$  and unknowns  $c$ :  $y = c_1\phi_1 + c_2\phi_2 + \cdots + c_n\phi_n$
- Monomial basis:  $\phi_k(x) = x^{k-1}$
- Lagrange basis:  $\phi_k(x) = \frac{\prod_{i \neq k} x - x_i}{\prod_{i \neq k} x_k - x_i}$
- Newton basis:  $\phi_k(x) = \prod_{i=1}^{k-1} x - x_i$
- Write a (linear) equation for each point, and put into matrix form:  $Ac = y$
- Monomial/Lagrange/Newton basis all give the same polynomial

# Recall: Overfitting (Unit 1)

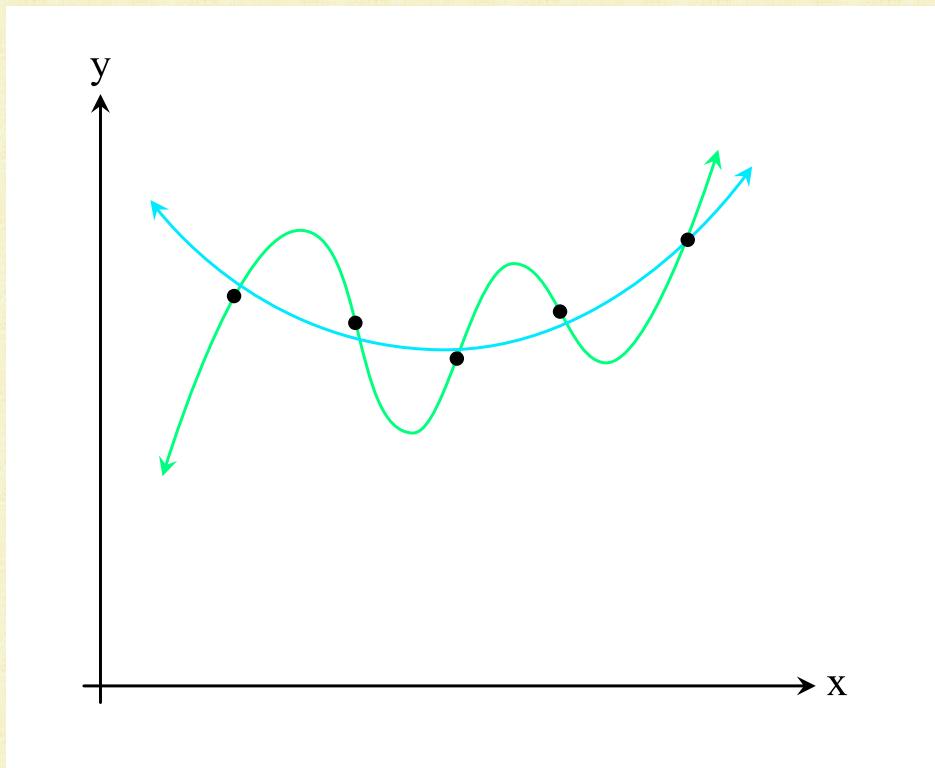
- Given a new input  $\hat{x}$ , the interpolating polynomial infers/predicts an output  $\hat{y}$  that may be far from what one may expect



- Interpolating polynomials are smooth (continuous function and derivatives)
- Thus, they wiggle/overshoot in between data points (so that they can smoothly turn back and hit the next point)
- Overly forcing polynomials to exactly hit every data point is called overfitting (overly fitting to the data)
- It results in inference/predictions that can vary wildly from the training data

# Recall: Regularization (Unit 1)

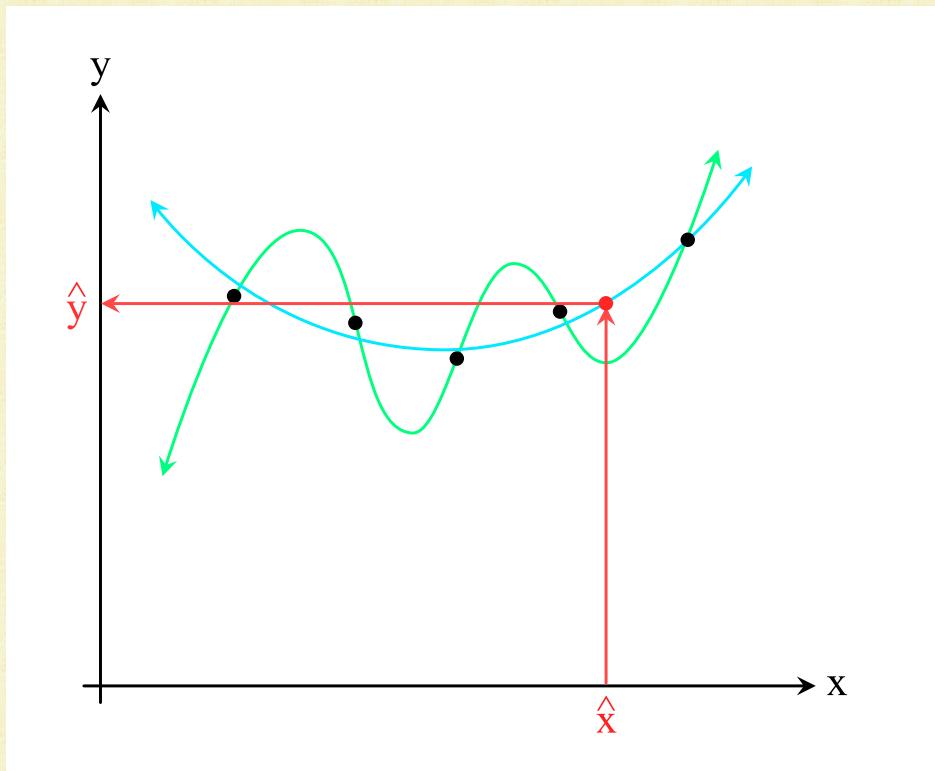
- Using a lower order polynomial that doesn't (can't) exactly fit the data points provides some degree of regularization



- A regularized interpolant contains intentional errors in the interpolation, missing some/all of the data points
- However, this hopefully makes the function more predictable/smooth in between the data points
- The data points themselves may contain noise/error, so it is not clear whether they should be interpolated exactly anyways

# Recall: Regularization (Unit 1)

- Given  $\hat{x}$ , the regularized interpolant infers/predicts a more reasonable  $\hat{y}$



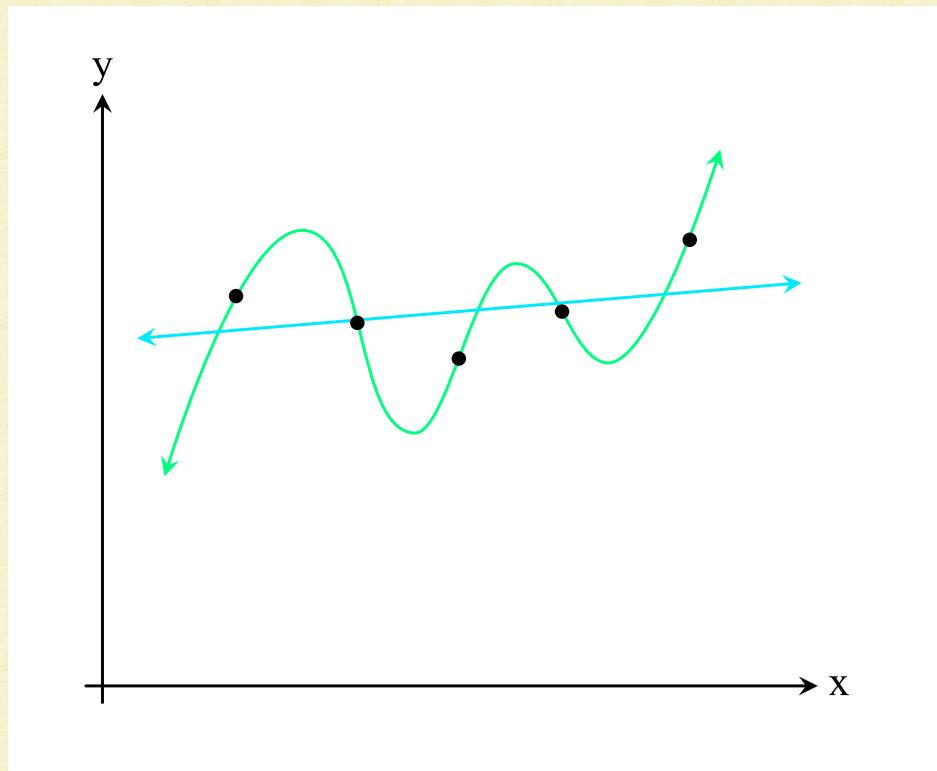
- There is a trade-off between sacrificing accuracy on fitting the original input data, and obtaining better accuracy on inference/prediction for new inputs

# Eliminating Basis Functions

- Consider  $Ac = y$ :
  - Each row of  $A$  evaluates all  $n$  basis functions  $\phi_k$  on a single data point  $x_i$
  - Each column of  $A$  evaluates all  $m$  points  $x_i$  on a single basis function  $\phi_k$
- Regularize by reducing the number of basis functions (and thus the degree of the polynomial)
- Afterwards, the process proceeds as usual
  - I.e., write a (linear) equation for each point, and put into matrix form:  $Ac = y$
- When there are more points than basis functions, there are more rows than columns (and the matrix is tall/rectangular)
- This tall matrix has full (column) rank, when the basis functions are linearly independent (and the data isn't degenerate)

# Recall: Underfitting (Unit 1)

- Using too low of an order polynomial causes one to miss the data by too much



- A linear function doesn't capture the essence of this data as well as a quadratic function does
- Choosing too simple of a model function or regularizing too much prevents one from properly representing the data

# Tall (Full Rank) Matrices

- Let  $A$  be a size  $m \times n$  tall (i.e.  $m > n$ ) matrix with full (column) rank (i.e. rank  $n$ )
- Since there are  $n$  entries in each row, the rows span at most an  $n$  dimensional space; thus, at least  $m - n$  rows are linear combinations of others
- That is,  $A$  contains (at least)  $m - n$  extra unnecessary equations (that are linear combinations of others)
- Thus,  $A$  could be reduced to  $n$  equations (and size  $n \times n$ ) without losing any information
- The SVD ( $A = U\Sigma V^T$ ) illustrates this, by the last  $m - n$  rows of  $\Sigma$  being all zeros
- The last  $m - n$  columns in  $U$  are hit by these zeros, and thus not in the range of  $A$

## Recall: Example (Unit 3)

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix} =$$

$$\begin{pmatrix} .141 & .825 & -.420 & -.351 \\ .344 & .426 & .298 & .782 \\ .547 & .028 & .644 & -.509 \\ .750 & -.371 & -.542 & .079 \end{pmatrix} \begin{pmatrix} 25.5 & 0 & 0 \\ 0 & 1.29 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} .504 & .574 & .644 \\ -.761 & -.057 & .646 \\ .408 & -.816 & .408 \end{pmatrix}$$

- Singular values are 25.5, 1.29, and 0
- Singular value of 0 indicates that the matrix is rank deficient
- The rank of a matrix is equal to its number of nonzero singular values

# Recall: Extra Dimensions (Unit 3)

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix} =$$
$$\begin{pmatrix} .141 & .825 & -.420 & \textcolor{green}{-.351} \\ .344 & .426 & .298 & \textcolor{green}{.712} \\ .547 & .028 & .644 & \textcolor{green}{-.09} \\ .750 & -.371 & -.542 & \textcolor{green}{.79} \end{pmatrix} \begin{pmatrix} 25.5 & 0 & 0 \\ 0 & 1.29 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} .504 & .574 & .644 \\ -.761 & -.057 & .646 \\ .408 & -.816 & .408 \end{pmatrix}$$

- The 3D space of vector inputs can only span a 3D subspace of  $R^4$
- The last (**green**) column of  $U$  represents the unreachable dimension, orthogonal to the range of  $A$ , and is always multiplied by 0
- One can delete this column and the associated portion of  $\Sigma$  (and still obtain a valid factorization)

# Solving Linear Systems (the right hand side)

- $Ac = b$  becomes  $U\Sigma V^T c = b$  or  $\Sigma(V^T c) = (U^T b)$  or  $\Sigma \hat{c} = \hat{b}$
- Solve  $\Sigma \hat{c} = \hat{b}$  by dividing the entries of  $\hat{b}$  by the singular values  $\sigma_k$ , then  $c = V\hat{c}$
- The last  $m - n$  equations are identically zero on the left, and would be identically zero on the right as well when a solution exists
  - E.g.  $\begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} \hat{c}_1 \\ \hat{c}_2 \end{pmatrix} = \begin{pmatrix} \hat{b}_1 \\ \hat{b}_2 \\ \hat{b}_3 \end{pmatrix}$  requires  $\hat{b}_3 = 0$  in order to have a solution
  - The last  $m - n$  columns in  $U$  are not in the range of  $A$ , so  $b$  must be in the span of the first  $n$  columns of  $U$  in order for a solution to exist

# False Statements

- Reasoning with a false statement leads to infinitely more false statements:

$$\begin{aligned} a &= b \\ a^2 &= ab \\ a^2 - b^2 &= ab - b^2 \\ (a + b)(a - b) &= b(a - b) \\ a + b &= b \\ b + b &= b \\ b(1 + 1) &= b(1) \\ 2 &= 1 \end{aligned}$$

- Don't make false statements!

# False Statements

- Reasoning with a false statement leads to infinitely more false statements:

$$\begin{aligned} Ac &= b \\ A^T Ac &= A^T b \\ c &= (A^T A)^{-1}(A^T b) \end{aligned}$$

Is it? Is it really?

- Don't make false statements!
- A mix of false/true statements makes it difficult to keep track of what is and what is not true

# False Statements

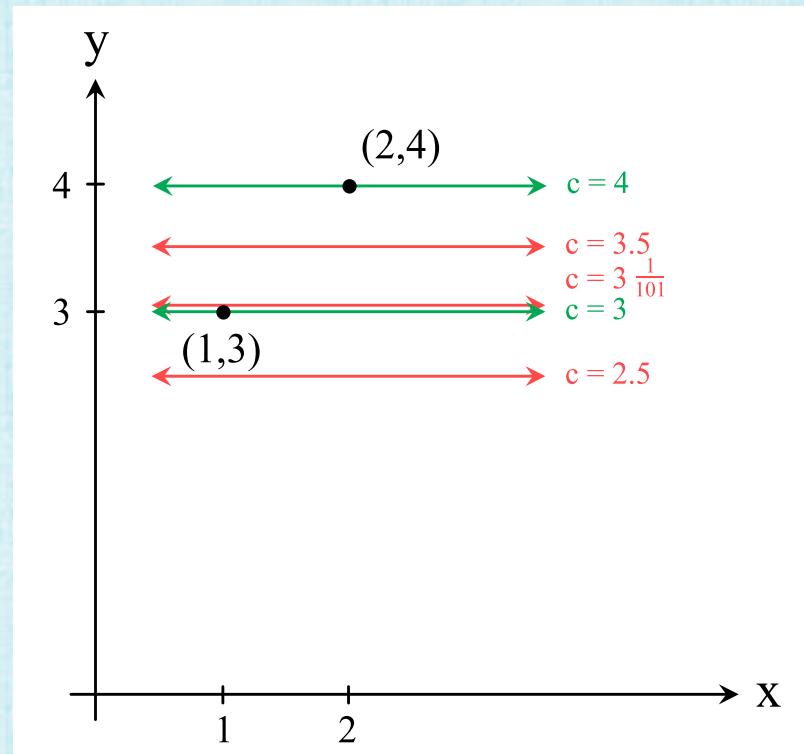
- Consider a very simple  $Ac = b$  given by:  $\begin{pmatrix} 1 \\ 1 \end{pmatrix}(c) = \begin{pmatrix} 3 \\ 4 \end{pmatrix}$
- This contains the equations  $c = 3$  and  $c = 4$ , and as such is a false statement
- Solve via  $(1 \quad 1) \begin{pmatrix} 1 \\ 1 \end{pmatrix}(c) = (1 \quad 1) \begin{pmatrix} 3 \\ 4 \end{pmatrix}$ , so  $2c = 7$  or  $c = 3.5$
- Row scale the first equation by 10 to obtain:  $\begin{pmatrix} 10 \\ 1 \end{pmatrix}(c) = \begin{pmatrix} 30 \\ 4 \end{pmatrix}$
- Solve via  $(10 \quad 1) \begin{pmatrix} 10 \\ 1 \end{pmatrix}(c) = (10 \quad 1) \begin{pmatrix} 30 \\ 4 \end{pmatrix}$ , so  $101c = 304$  or  $c = 3\frac{1}{101}$
- Perfectly valid row scaling leads to a different answer

# False Statements

- Again, starting with the same:  $\begin{pmatrix} 1 \\ 1 \end{pmatrix}(c) = \begin{pmatrix} 3 \\ 4 \end{pmatrix}$
- Subtract  $2^*(\text{row 1})$  from row 2 to obtain  $\begin{pmatrix} 1 \\ -1 \end{pmatrix}(c) = \begin{pmatrix} 3 \\ -2 \end{pmatrix}$
- Solve via  $(1 \quad -1) \begin{pmatrix} 1 \\ -1 \end{pmatrix}(c) = (1 \quad -1) \begin{pmatrix} 3 \\ -2 \end{pmatrix}$ , so  $2c = 5$  or  $c = 2.5$
- A perfectly valid row operation again leads to a different answer
- Note that  $2.5 \notin [3,4]$  either!
- Problem:  $\begin{pmatrix} 3 \\ 4 \end{pmatrix}$  is not in the range of  $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$ , so  $\begin{pmatrix} 1 \\ 1 \end{pmatrix}(c) \neq \begin{pmatrix} 3 \\ 4 \end{pmatrix}$  for  $\forall c \in \mathcal{R}$

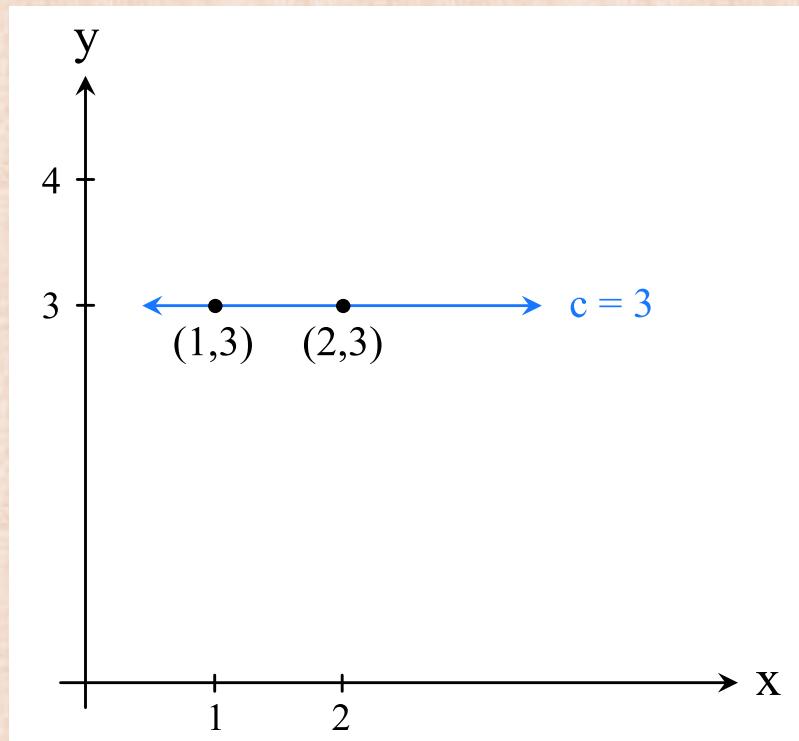
# False Statements

- Consider  $y = c_1 \phi_1$  with monomial  $\phi_1 = 1$ , and data points (1,3) and (2,4)
- This leads to the same  $\begin{pmatrix} 1 \\ 1 \end{pmatrix} (c_1) = \begin{pmatrix} 3 \\ 4 \end{pmatrix}$



# True Statements

- Consider  $y = c_1 \phi_1$  with monomial  $\phi_1 = 1$ , and data points  $(1,3)$  and  $(2,3)$
- This leads instead to  $\binom{1}{1} (c_1) = \binom{3}{3}$  which is valid and has solution  $c_1 = 3$



# True Statements

- When  $b$  is in the range of  $A$ , then  $Ac = b$  is a true statement
  - There exists at least one  $c$  (by definition) constrained by this statement
- When  $b$  is in not the range of  $A$ , then  $Ac \neq b$  is the true statement
  - In this case,  $Ac \neq b$  is true for all  $c$
- The equation for the residual  $r = b - Ac$  is always true (it's a definition)
  - When  $b$  is in the range of  $A$ , there exists a  $c$  with  $Ac = b$  and  $r = 0$
  - When  $b$  is not in the range of  $A$ , then  $Ac \neq b$  and  $r \neq 0$  for all  $c$
- The goal in both cases is to minimize the residual  $r = b - Ac$

# Norm Matters

- Consider  $y = c_1 \phi_1$  where  $\phi_1 = 1$  along with data points (1,3), (2,3), and (3,4)
- This leads to  $r = \begin{pmatrix} 3 \\ 3 \\ 4 \end{pmatrix} - \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}(c_1)$
- Setting  $c_1 = 3.5$  minimizes  $\|r\|_\infty$  with  $r = \begin{pmatrix} -.5 \\ -.5 \\ .5 \end{pmatrix}$ ,  $\|r\|_\infty = .5$ ,  $\|r\|_2 = \frac{\sqrt{3}}{2}$
- Setting  $c_1 = \frac{10}{3}$  minimizes  $\|r\|_2$  with  $r = \begin{pmatrix} -1/3 \\ -1/3 \\ 2/3 \end{pmatrix}$ ,  $\|r\|_\infty = \frac{2}{3}$ ,  $\|r\|_2 = \frac{\sqrt{6}}{3}$

# Row Operations Matter

- Given a set of equations, they can be manipulated in various ways
- These manipulations often change the answer
- Thus, one should carefully choose the residual they want minimized
- Equivalent sets of equations lead to different answers when minimizing the corresponding residuals

# Weighted Minimization

- Given  $r = b - Ac$ , some equations may be deemed more important than others
- Scaling entries in the residual (before taking the norm) changes the relative importance of various equations
- This is accomplished via: minimize  $\|Dr\|$  for a diagonal matrix  $D$  with non-zero diagonal entries
- This is equivalent to row scaling:  $Dr = Db - DAC$
- Column scaling doesn't effect the result, e.g.  $Dr = Db - DAD^{-1}(\widehat{D}c)$
- So, it can be used to preserve symmetry:  $Dr = Db - (DAD^T)(D^{-T}c)$ 
  - when  $A$  is square and symmetric

# Least Squares

- Minimizing  $\|r\|_2$  is referred to as least squares, and the resulting solution is referred to as the least squares solution
  - The least squares solution **is** the unique solution when  $\|r\|_2 = 0$
- Minimizing  $\|Dr\|_2$  is referred to as weighted least squares
- $\|r\|_2$  is minimized when  $\|r\|_2^2$  is minimized
- And  $\|r\|_2^2 = r \cdot r = (b - Ac) \cdot (b - Ac) = c^T A^T Ac - 2b^T Ac + b^T b$  is minimized when  $c^T A^T Ac - 2b^T Ac$  is minimized
- Thus, minimize  $c^T A^T Ac - 2b^T Ac$
- For weighted least squares, minimize  $c^T A^T D^2 Ac - 2b^T D^2 Ac$

# Unit 9

# Basic Optimization

# Jacobian

- The Jacobian of  $F(c) = \begin{pmatrix} F_1(c) \\ F_2(c) \\ \vdots \\ F_m(c) \end{pmatrix}$  has entries  $J_{ik} = \frac{\partial F_i}{\partial c_k}(c)$

- Thus, the Jacobian  $J(c) = F'(c) = \begin{pmatrix} \frac{\partial F_1}{\partial c_1}(c) & \frac{\partial F_1}{\partial c_2}(c) & \cdots & \frac{\partial F_1}{\partial c_n}(c) \\ \frac{\partial F_2}{\partial c_1}(c) & \frac{\partial F_2}{\partial c_2}(c) & \cdots & \frac{\partial F_2}{\partial c_n}(c) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial F_m}{\partial c_1}(c) & \frac{\partial F_m}{\partial c_2}(c) & \cdots & \frac{\partial F_m}{\partial c_n}(c) \end{pmatrix}$

# Gradient

- Consider the scalar (output) function  $f(c)$  with multi-dimensional input  $c$
- The Jacobian of  $f(c)$  is  $J(c) = \begin{pmatrix} \frac{\partial f}{\partial c_1}(c) & \frac{\partial f}{\partial c_2}(c) & \cdots & \frac{\partial f}{\partial c_n}(c) \end{pmatrix}$
- The gradient of  $f(c)$  is  $\nabla f(c) = J^T(c) = \begin{pmatrix} \frac{\partial f}{\partial c_1}(c) \\ \frac{\partial f}{\partial c_2}(c) \\ \vdots \\ \frac{\partial f}{\partial c_n}(c) \end{pmatrix}$
- In 1D, both  $J(c)$  and  $\nabla f(c) = J^T(c)$  are the usual  $f'(c)$

# Critical Points

- To identify critical points of  $f(c)$ , set the gradient to zero:  $\nabla f(c) = 0$

- This is a system of equations:

$$\begin{pmatrix} \frac{\partial f}{\partial c_1}(c) \\ \frac{\partial f}{\partial c_2}(c) \\ \vdots \\ \frac{\partial f}{\partial c_n}(c) \end{pmatrix} = 0 \quad \text{or} \quad \begin{pmatrix} \frac{\partial f}{\partial c_1}(c) = 0 \\ \frac{\partial f}{\partial c_2}(c) = 0 \\ \vdots \\ \frac{\partial f}{\partial c_n}(c) = 0 \end{pmatrix}$$

- Any  $c$  that simultaneously solves all the equations is a critical point
- In 1D, this is the usual  $f'(c) = 0$

# Jacobian of the Gradient

- Taking the **Jacobian** of the column vector **gradient** gives:

- The  $J(\nabla f(c)) = \begin{pmatrix} \frac{\partial^2 f}{\partial c_1 \partial c_1}(c) & \frac{\partial^2 f}{\partial c_2 \partial c_1}(c) & \cdots & \frac{\partial^2 f}{\partial c_n \partial c_1}(c) \\ \frac{\partial^2 f}{\partial c_1 \partial c_2}(c) & \frac{\partial^2 f}{\partial c_2 \partial c_2}(c) & \cdots & \frac{\partial^2 f}{\partial c_n \partial c_2}(c) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial c_1 \partial c_n}(c) & \frac{\partial^2 f}{\partial c_2 \partial c_n}(c) & \cdots & \frac{\partial^2 f}{\partial c_n \partial c_n}(c) \end{pmatrix}$

- Note:  $\frac{\partial^2 f}{\partial c_2 \partial c_1} = \frac{\partial}{\partial c_2} \left( \frac{\partial f}{\partial c_1} \right) = f_{c_1 c_2}$

# Hessian

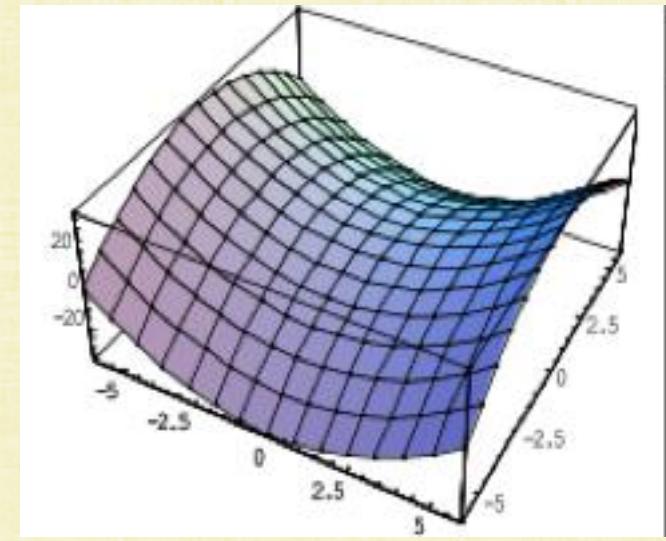
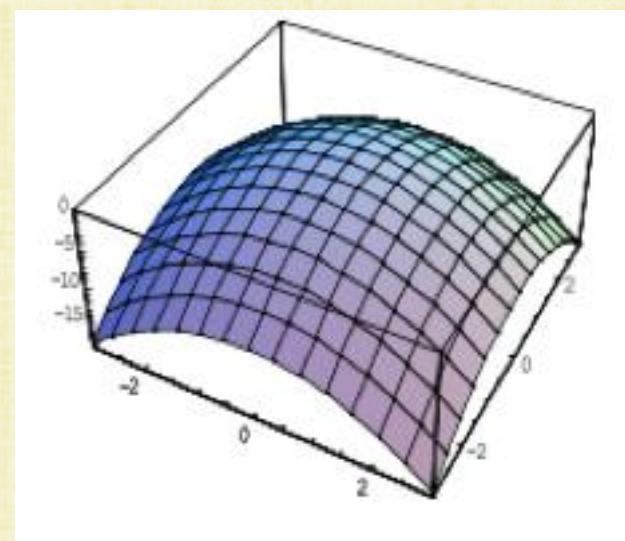
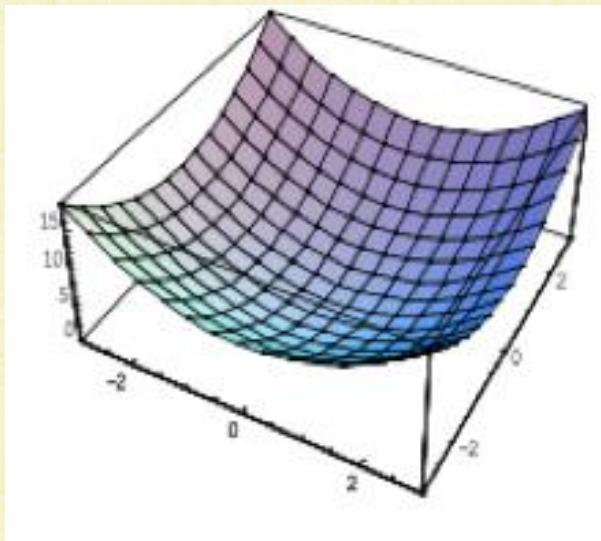
- The Hessian of  $f(c)$  is  $H(c) = J(\nabla f(c))^T$  and has entries  $H_{ik} = \frac{\partial^2 f}{\partial c_i \partial c_k}(c)$
- The Hessian is  $H(c) = \begin{pmatrix} \frac{\partial^2 f}{\partial c_1^2}(c) & \frac{\partial^2 f}{\partial c_1 \partial c_2}(c) & \cdots & \frac{\partial^2 f}{\partial c_1 \partial c_n}(c) \\ \frac{\partial^2 f}{\partial c_2 \partial c_1}(c) & \frac{\partial^2 f}{\partial c_2^2}(c) & \cdots & \frac{\partial^2 f}{\partial c_2 \partial c_n}(c) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial c_n \partial c_1}(c) & \frac{\partial^2 f}{\partial c_n \partial c_2}(c) & \cdots & \frac{\partial^2 f}{\partial c_n^2}(c) \end{pmatrix}$
- $H(c)$  is symmetric, when the order of differentiation doesn't matter
- In 1D, this is the usual  $f''(c)$

# Differential Forms

- Vector valued function:  $dF(c) = J(F(c))dc$
- Substitute  $\nabla f$  for  $F$  to get:  $d\nabla f(c) = J(\nabla f(c))dc = H^T(c)dc$
- Scalar valued function:  $df(c) = J(f(c))dc$
- Take the transpose:  $df(c) = dc^T \nabla f(c)$
- Take (another) differential:  $d^2f(c) = J(dc^T \nabla f(c))dc$
- Some hand waving:  $d^2f(c) = dc^T H^T(c)dc = dc \cdot H^T(c)dc$

# Classifying Critical Points

- Given a critical point  $c^*$ , i.e. with  $\nabla f(c^*) = 0$ , the Hessian is used to classify it
- If  $H(c^*)$  is positive definite, then  $c^*$  is a local minimum
- If  $H(c^*)$  is negative definite, then  $c^*$  is a local maximum
- Otherwise,  $H(c^*)$  is indefinite, and  $c^*$  is a saddle point



# Classifying Critical Points (in 1D)

- In 1D, given critical point  $c^*$ , i.e. with  $\nabla f(c^*) = f'(c^*) = 0$ , the Hessian is used to classify it
- In 1D,  $H(c^*) = (f''(c^*))$  is a size  $1 \times 1$  diagonal matrix with eigenvalue  $f''(c^*)$
- If  $H(c^*)$  is positive definite with eigenvalue  $f''(c^*) > 0$ , then  $c^*$  is a local minimum
  - As usual,  $f''(c^*) > 0$  implies concave up and a local min
- If  $H(c^*)$  is negative definite with eigenvalue  $f''(c^*) < 0$ , then  $c^*$  is a local maximum
  - As usual,  $f''(c^*) < 0$  implies concave down and a local max
- Otherwise,  $H(c^*)$  is indefinite with eigenvalue  $f''(c^*) = 0$ , and  $c^*$  is a saddle point
  - As usual,  $f''(c^*) = 0$  implies an inflection point (not a local extrema)

# Quadratic Form

- The quadratic form of a square matrix  $\tilde{A}$  is  $f(c) = \frac{1}{2} \mathbf{c}^T \tilde{A} \mathbf{c} - \tilde{\mathbf{b}}^T \mathbf{c} + \tilde{c}$ 
  - In 1D,  $f(c) = \frac{1}{2} \tilde{a} c^2 - \tilde{b} c + \tilde{c}$
- Minimize  $f(c)$  by (first) finding critical points where  $\nabla f(c) = 0$
- Note  $\nabla f(c) = \frac{1}{2} \tilde{A} \mathbf{c} + \frac{1}{2} \tilde{A}^T \mathbf{c} - \tilde{\mathbf{b}}$ , since  $J(\mathbf{c}^T \mathbf{v}) = J(\mathbf{v}^T \mathbf{c}) = \mathbf{v}^T$  (the gradient is  $\mathbf{v}$ )
  - Solve the symmetric system  $\frac{1}{2} (\tilde{A} + \tilde{A}^T) \mathbf{c} = \tilde{\mathbf{b}}$  to find critical points
- When  $\tilde{A}$  is symmetric,  $\nabla f(c) = \tilde{A} \mathbf{c} - \tilde{\mathbf{b}} = 0$  is satisfied when  $\tilde{A} \mathbf{c} = \tilde{\mathbf{b}}$ 
  - In 1D, the critical point is on the line of symmetry  $\tilde{c} = \frac{\tilde{b}}{\tilde{a}}$
- That is, solve  $\tilde{A} \mathbf{c} = \tilde{\mathbf{b}}$  to find the critical point

# Quadratic Form

- The Hessian of  $f(c)$  is  $H = \frac{1}{2}(\tilde{A}^T + \tilde{A})$  or just  $\tilde{A}$  when  $\tilde{A}$  is symmetric
- When  $\tilde{A}$  is SPD, the solution to  $\tilde{A}c = \tilde{b}$  is a minimum
- When  $\tilde{A}$  is symmetric negative definite, the solution to  $\tilde{A}c = \tilde{b}$  is a maximum
- When  $\tilde{A}$  is indefinite, the solution to  $\tilde{A}c = \tilde{b}$  is a saddle point
- In 1D,  $H = (\tilde{a})$  is a size  $1 \times 1$  diagonal matrix with eigenvalue  $\tilde{a}$
- As usual,  $\tilde{a} > 0$  implies concave up and a local min
- As usual,  $\tilde{a} < 0$  implies concave down and a local max
- As usual,  $\tilde{a} = 0$  implies an inflection point (not a local extrema)

# Recall: Least Squares (Unit 8)

- Minimizing  $\|r\|_2$  is referred to as least squares, and the resulting solution is referred to as the least squares solution
  - The least squares solution is the unique solution when  $\|r\|_2 = 0$
- Minimizing  $\|Dr\|_2$  is referred to as weighted least squares
- $\|r\|_2$  is minimized when  $\|r\|_2^2$  is minimized
- And  $\|r\|_2^2 = r \cdot r = (b - Ac) \cdot (b - Ac) = c^T A^T Ac - 2b^T Ac + b^T b$  is minimized when  $c^T A^T Ac - 2b^T Ac$  is minimized
- Thus, minimize  $c^T A^T Ac - 2b^T Ac$
- For weighted least squares, minimize  $c^T A^T D^2 A c - 2b^T D^2 A c$

# Normal Equations

- $c^T A^T D^2 A c - 2b^T D^2 A c$  has the same minimum as  $\frac{1}{2}c^T A^T D^2 A c - b^T D^2 A c$
- This is a quadratic form with symmetric  $\tilde{A} = A^T D^2 A$  and  $\tilde{b} = A^T D^2 b$
- The critical point is found from solving  $\tilde{A}c = \tilde{b}$  or  $A^T D^2 A c = A^T D^2 b$
- Weighted least squares defaults to ordinary least squares when  $D = I$
- For (unweighted) least squares, solve  $A^T A c = A^T b$
- These are called the normal equations

# Hessian

- Recall:  $A$  is a tall (or square) full rank matrix with size  $m \times n$  where  $m \geq n$
- The Hessian  $H = \tilde{A} = A^T A = V \Sigma^T U^T U \Sigma V^T = V \Sigma^T \Sigma V^T = V \Lambda V^T$ 
  - where  $\Lambda = \Sigma^T \Sigma$  is a size  $n \times n$  matrix of (nonzero) singular values squared
- $HV = V\Lambda$  illustrates that  $H$  has all positive eigenvalues (and so is SPD)
- That is, **the critical point is indeed a minimum** (as desired)

For weighted least squares:

- Nonzero diagonal elements in  $D$  implies that  $DAc = 0$  if and only if  $Ac = 0$ 
  - That is, a full column rank  $A$  implies a full column rank  $DA$
- Then, the SVD of  $DA$  can be used to prove that  $H = (DA)^T(DA)$  is SPD

# Unit 10

# Solving Least Squares

# Normal Equations

- Let  $\tilde{A}$  have full column rank, and be size  $m \times n$  with  $m \geq n$
- Diagonal (nonzero) weighting  $A = D\tilde{A}$  does not change the rank/size
  - but changes the answer when  $D \neq I$  and  $m \neq n$
- Minimizing  $\|r\|_2 = \|b - Ac\|_2$  leads to the normal equations  $A^T A c = A^T b$  for the critical point
- Since  $A^T A$  is SPD,  $A^T A c = A^T b$  has a unique solution obtainable via fast/efficient SPD solvers
- When  $b$  is in the range of  $A$ , the unique solution to  $A^T A c = A^T b$  makes  $r = 0$ , and is thus the unique solution to  $Ac = b$ 
  - When  $A$  is square ( $m = n$ ), and full rank, then  $b$  is always in the range of  $A$

# Condition Number

- Compare  $A = U\Sigma V^T$  and  $A^T A = V\Sigma^T \Sigma V^T = V\Lambda V^T$  where  $\Lambda = \Sigma^T \Sigma$  is a diagonal size  $n \times n$  matrix of singular values squared
- Since the singular values of  $A^T A$  are the square of those in  $A$ , the condition number  $\frac{\sigma_{max}}{\sigma_{min}}$  of  $A^T A$  is also squared (compared to  $A$ )
  - Thus, solving the normal equations requires twice the precision (e.g.  $(10^7)^2 = 10^{14}$ )
- **It takes twice as much precision to get the same number of significant digits!**
- The normal equations are not the preferred approach (unless  $A$  is extremely well conditioned)
- However, (like the SVD) it is a great tool for theoretical purposes
  - Can transform any full rank matrix into an SPD system

# Understanding Least Squares

- When  $A = U\Sigma V^T$  has full column rank,  $\Sigma = \begin{pmatrix} \hat{\Sigma} \\ \mathbf{0} \end{pmatrix}$  with  $\hat{\Sigma}$  a size  $n \times n$  diagonal matrix of (strictly) positive singular values
  - The **0 submatrix** is size  $(m - n) \times n$  and doesn't exist when  $m = n$
- Note:  $A^T A = V \Sigma^T \Sigma V^T = V \hat{\Sigma}^2 V^T$  and  $(A^T A)^{-1} = V \hat{\Sigma}^{-2} V^T$
- So  $\mathbf{c} = (A^T A)^{-1} A^T b = V \hat{\Sigma}^{-2} V^T V \Sigma^T U^T b = V (\hat{\Sigma}^{-1} \quad \mathbf{0}) U^T b$
- $A\mathbf{c} = U \Sigma V^T V (\hat{\Sigma}^{-1} \quad \mathbf{0}) U^T b = U \begin{pmatrix} \hat{\Sigma} \\ \mathbf{0} \end{pmatrix} (\hat{\Sigma}^{-1} \quad \mathbf{0}) U^T b = U \begin{pmatrix} I_{n \times n} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{pmatrix} U^T b$
- $r = b - A\mathbf{c} = UI_{m \times m}U^T b - U \begin{pmatrix} I_{n \times n} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{pmatrix} U^T b = U \begin{pmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & I_{(m-n) \times (m-n)} \end{pmatrix} U^T b$

# Understanding Least Squares

- Recall: from SVD slides (unit 3):
  - The columns of  $U$  corresponding to “nonzero” singular values form an orthonormal basis for the range of  $A$
  - The remaining columns of  $U$  form an orthonormal basis for the (unattainable) space perpendicular to the range of  $A$
- Since  $A$  only has  $n$  singular values, only the first  $n$  columns of  $U$  (which has  $m$  columns) span the range of  $A$
- Writing  $\begin{pmatrix} \hat{b}_r \\ \hat{b}_z \end{pmatrix} = U^T b$  separates  $\hat{b}_r$  (which is size  $nx1$ ) in the range of  $A$  from  $\hat{b}_z$  (which is size  $(m - n)x1$ ) orthogonal to the range of  $A$
- Then (from the last slide):  $c = V\hat{\Sigma}^{-1}\hat{b}_r$ ,  $Ac = U\begin{pmatrix} \hat{b}_r \\ 0 \end{pmatrix}$ , and  $r = U\begin{pmatrix} 0 \\ \hat{b}_z \end{pmatrix}$

# Orthogonal Matrices (and the L2 norm)

- Recall (from unit 3):
  - Orthogonal matrices have orthonormal columns (an orthonormal basis), so their transpose is their inverse. They preserve inner products, and thus are rotations, reflections, and combinations thereof.
- An orthogonal  $\hat{Q}$  has  $\hat{Q}\hat{Q}^T = \hat{Q}^T\hat{Q} = I$
- So,  $\|\hat{Q}r\|_2 = \sqrt{\hat{Q}r \cdot \hat{Q}r} = \sqrt{r^T \hat{Q}^T \hat{Q}r} = \sqrt{r^T r} = \|r\|_2$
- Similarly,  $\|\hat{Q}^T r\|_2 = \sqrt{\hat{Q}^T r \cdot \hat{Q}^T r} = \sqrt{r^T \hat{Q} \hat{Q}^T r} = \sqrt{r^T r} = \|r\|_2$
- That is, orthogonal transformations preserve Euclidean distance

# Understanding Least Squares

- $r = U \begin{pmatrix} 0 \\ \hat{b}_z \end{pmatrix}$  with orthogonal  $U$  implies  $\|r\|_2 = \|\hat{b}_z\|_2$
- Consider the diagonalized SVD view of  $Ac = b$  when  $A$  has full rank:
$$U\Sigma V^T c = b \quad \text{or} \quad \begin{pmatrix} \hat{\Sigma} \\ 0 \end{pmatrix} \hat{c} = \begin{pmatrix} \hat{b}_r \\ \hat{b}_z \end{pmatrix}$$
- The first block row gives  $c = V\hat{\Sigma}^{-1}\hat{b}_r$ , which is the least squares solution
- The second block row is  $0 = \hat{b}_z$ . The norm of the residual for this block row is  $\|\hat{b}_z\|_2$ , which is identical to  $\|r\|_2$
- The SVD approach gives the same (minimum residual) least squares solution

# Recall: Gram-Schmidt (Unit 5)

- Orthogonalizes a set of vectors
- For each new vector, subtract its (weighted) dot product overlap with all prior vectors, making it orthogonal to them
- A-orthogonal Gram-Schmidt uses an A-weighted dot/inner product
- Given vector  $\bar{S}^q$ , subtract out the A-overlap with  $s^1$  to  $s^{q-1}$  so that the resulting vector  $s^q$  has  $\langle s^q, s^{\hat{q}} \rangle_A = 0$  for  $\hat{q} \in \{1, 2, \dots, q-1\}$
- That is,  $s^q = \bar{S}^q - \sum_{\hat{q}=1}^{q-1} \frac{\langle \bar{S}^q, s^{\hat{q}} \rangle_A}{\langle s^{\hat{q}}, s^{\hat{q}} \rangle_A} s^{\hat{q}}$  where the two non-normalized  $s^{\hat{q}}$  both require division by their norm (and  $\langle s^{\hat{q}}, s^{\hat{q}} \rangle_A = \|s^{\hat{q}}\|_A^2$ )
- Proof:  $\langle s^q, s^{\tilde{q}} \rangle_A = \langle \bar{S}^q, s^{\tilde{q}} \rangle_A - \frac{\langle \bar{S}^q, s^{\tilde{q}} \rangle_A}{\langle s^{\tilde{q}}, s^{\tilde{q}} \rangle_A} \langle s^{\tilde{q}}, s^{\tilde{q}} \rangle_A = 0$

# Gram-Schmidt for QR Factorization

- From  $A$ , create a full rank  $Q$  with orthonormal columns
- For each column  $a_k$ , subtract the overlap with all prior columns in  $Q$  and make the result unit length:

$$q_k = \frac{a_k - \sum_{\hat{k}=1}^{k-1} \langle a_k, q_{\hat{k}} \rangle q_{\hat{k}}}{\left\| a_k - \sum_{\hat{k}=1}^{k-1} \langle a_k, q_{\hat{k}} \rangle q_{\hat{k}} \right\|_2}$$

- Define  $r_{\hat{k}k} = \langle a_k, q_{\hat{k}} \rangle$  for  $k > \hat{k}$ , and  $r_{kk} = \left\| a_k - \sum_{\hat{k}=1}^{k-1} \langle a_k, q_{\hat{k}} \rangle q_{\hat{k}} \right\|_2$
- Then  $q_k = \frac{a_k - \sum_{\hat{k}=1}^{k-1} r_{\hat{k}k} q_{\hat{k}}}{r_{kk}}$ , and thus  $a_k = r_{kk} q_k + \sum_{\hat{k}=1}^{k-1} r_{\hat{k}k} q_{\hat{k}} = \sum_{\hat{k}=1}^k r_{\hat{k}k} q_{\hat{k}}$
- This gives  $A = QR$  where  $R$  is upper triangular and  $Q^T Q = I$

# Gram-Schmidt QR (Example)

- Example:  $A = QR$  with upper triangular  $R$

$$\begin{pmatrix} 3 & -3 & 3 \\ 2 & -1 & 1 \\ 2 & -1 & -1 \\ 2 & -3 & 3 \\ 2 & -3 & 5 \end{pmatrix} = \begin{pmatrix} 3/5 & 0 & 0 \\ 2/5 & 1/2 & 1/2 \\ 2/5 & 1/2 & -1/2 \\ 2/5 & -1/2 & -1/2 \\ 2/5 & -1/2 & 1/2 \end{pmatrix} \begin{pmatrix} 5 & -5 & 5 \\ 0 & 2 & -4 \\ 0 & 0 & 2 \end{pmatrix}$$

- Note that  $Q^T Q = I_{3 \times 3}$  since the columns of  $Q$  are orthonormal
- However,  $QQ^T \neq I_{5 \times 5}$  and  $Q$  is only a subset of an orthogonal matrix

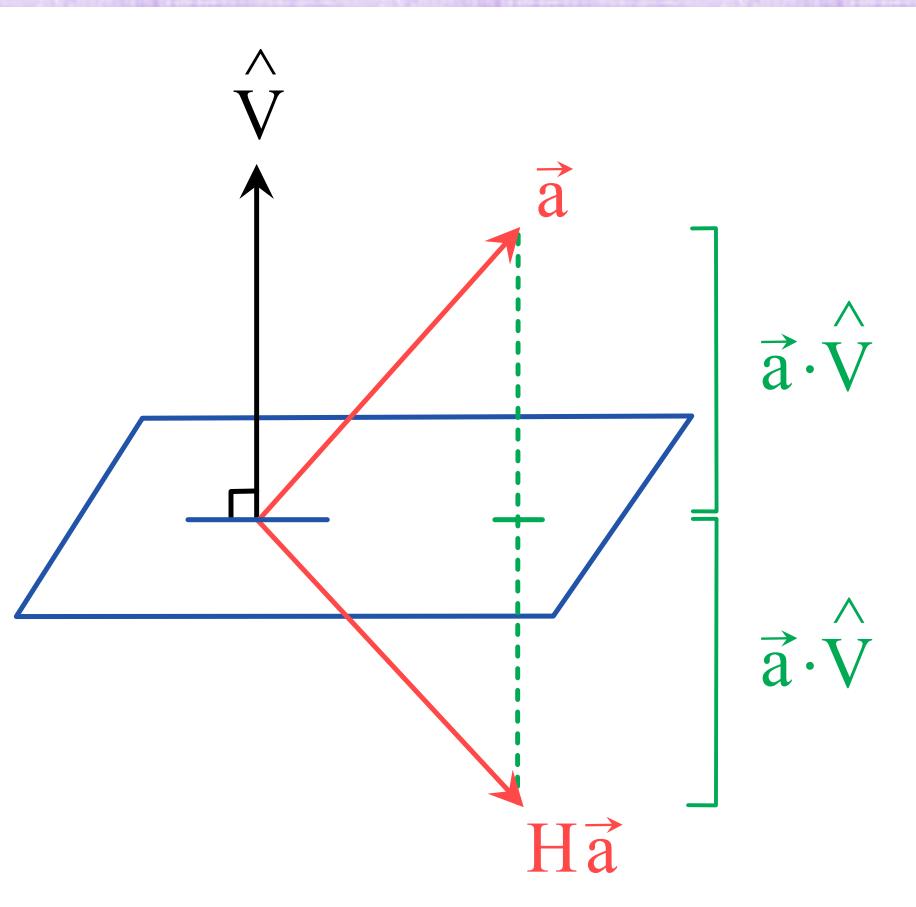
# Not Good for Large Matrices

- Gram-Schmidt has too much numerical drift for large matrices
- Don't use Gram-Schmidt to find  $A = QR$  with upper triangular  $R$  and  $Q^T Q = I$
- But it does provide a good conceptual way to think about  $A = QR$

# QR Factorization

- Consider  $A = QR$  with upper triangular  $R$  and  $Q^T Q = I$
- $Q$  is size  $m \times n$  (just like  $A$ ) with  $n$  orthonormal columns
- Let  $\tilde{Q}$  be the matrix with  $m - n$  orthonormal columns that span the space perpendicular to the range of  $Q$
- Then, the size  $m \times m$  matrix  $\hat{Q} = (Q \quad \tilde{Q})$  **is orthogonal**
- $\|r\|_2 = \|\hat{Q}^T r\|_2 = \left\| \begin{pmatrix} Q^T \\ \tilde{Q}^T \end{pmatrix} (b - QRc) \right\|_2 = \left\| \begin{pmatrix} Q^T b - Rc \\ \tilde{Q}^T b \end{pmatrix} \right\|_2$
- Only the first (block) row varies with  $c$ , so  $\|r\|_2$  is minimized by solving  $Rc = Q^T b$
- Since  $R$  is upper triangular,  $Rc = Q^T b$  can be solved **via back-substitution**

# Householder Transform



- Let the unit normal  $\hat{v}$  implicitly define a plane orthogonal to it
- Then,  $H = I - 2\hat{v}\hat{v}^T$  reflects vectors across that plane
- $Ha = a - 2(\hat{v}^T a) \hat{v}$
- $H$  is orthogonal with  $H = H^T = H^{-1}$
- Don't form the full  $H$
- Instead, apply it via the definition of  $\hat{v}$

# Householder Transform

- Choose directions  $v_k = a - Ha$  in a manner that zeroes out elements

$$\text{E.g. } v_k = \begin{pmatrix} a_1 \\ \vdots \\ a_{k-1} \\ a_k \\ a_{k+1} \\ \vdots \\ a_n \end{pmatrix} - \begin{pmatrix} a_1 \\ \vdots \\ a_{k-1} \\ \gamma \\ 0 \\ \vdots \\ 0 \end{pmatrix} = \hat{a}_k - \gamma \hat{e}_k \text{ where } \hat{a}_k = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ a_k \\ a_{k+1} \\ \vdots \\ a_n \end{pmatrix}$$

- $Ha$  should be the same length as  $a$  (i.e. a reflection), so  $\gamma = \pm \|\hat{a}_k\|_2$
- Then,  $v_k = \hat{a}_k \pm \|\hat{a}_k\|_2 \hat{e}_k$ , which is subsequently normalized to  $\hat{v}_k = \frac{v_k}{\|v_k\|_2}$
- For robustness,  $v_k = \hat{a}_k + S(a_k) \|\hat{a}_k\|_2 \hat{e}_k$  where  $S(a_k) = \pm 1$  is the sign function

# Householder Transform (Example)

- Consider  $a = \begin{pmatrix} 2 \\ 1 \\ 2 \end{pmatrix}$  in the formulation  $v_k = \hat{a}_k + S(a_k) \|\hat{a}_k\|_2 \hat{e}_k$
- Here  $\hat{a}_1 = \begin{pmatrix} 2 \\ 1 \\ 2 \end{pmatrix}$ ,  $v_1 = \begin{pmatrix} 2 \\ 1 \\ 2 \end{pmatrix} + S(2)\sqrt{9}\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 5 \\ 1 \\ 2 \end{pmatrix}$ ,  $\hat{v}_1 = \frac{1}{\sqrt{30}}\begin{pmatrix} 5 \\ 1 \\ 2 \end{pmatrix}$
- Then,  $H_1 a = a - 2(\hat{v}_1^T a) \hat{v}_1 = \begin{pmatrix} 2 \\ 1 \\ 2 \end{pmatrix} - 2 \frac{15}{\sqrt{30}} \frac{1}{\sqrt{30}} \begin{pmatrix} 5 \\ 1 \\ 2 \end{pmatrix} = \begin{pmatrix} -3 \\ 0 \\ 0 \end{pmatrix}$

# Householder Transform for QR

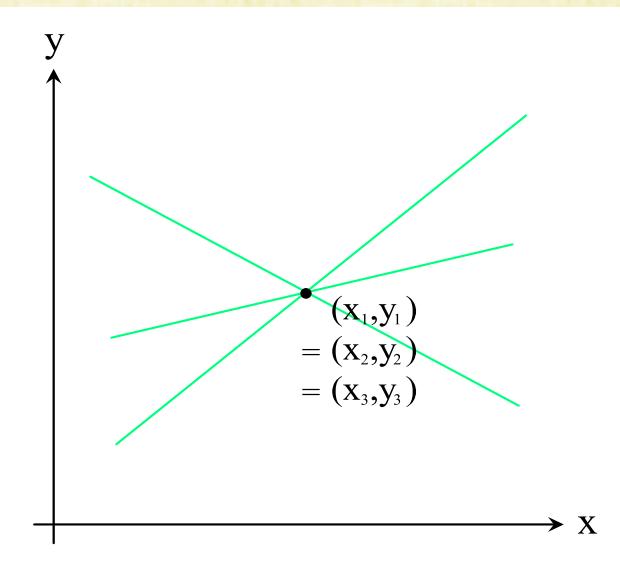
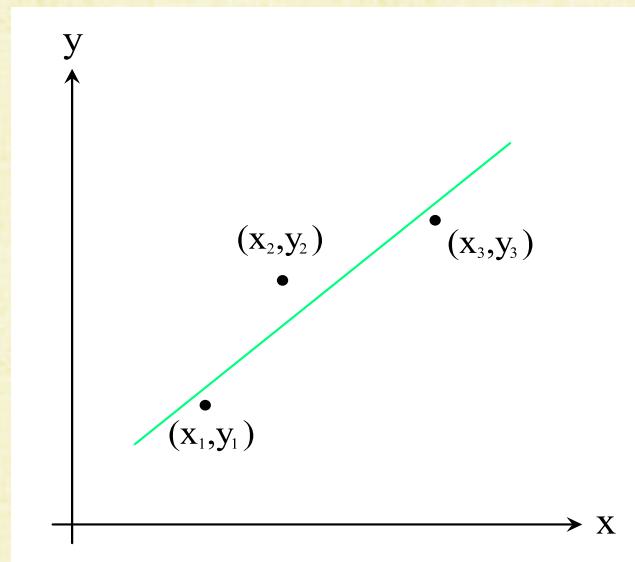
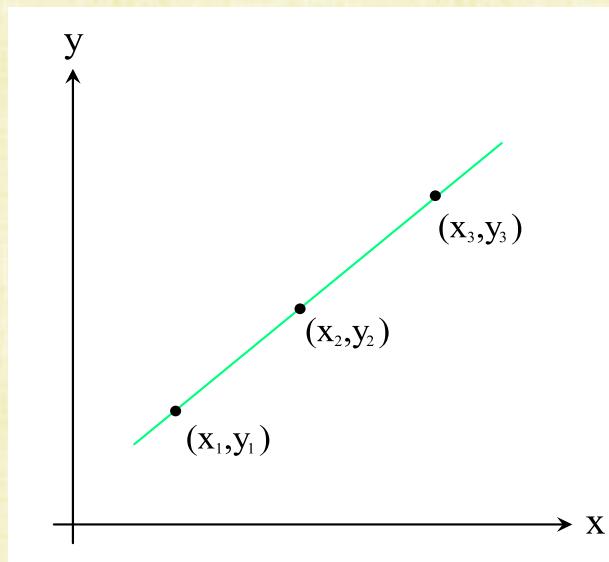
- For each column of  $A$ , construct the Householder transform that zeroes out entries below the diagonal
- Then  $H_n H_{n-1} \cdots H_2 H_1 A = \begin{pmatrix} R \\ 0 \end{pmatrix}$  and  $H_n H_{n-1} \cdots H_2 H_1 b = \begin{pmatrix} \tilde{b}_r \\ \tilde{b}_z \end{pmatrix}$
- Apply  $H_k$  efficiently using  $\hat{v}_k$ , and remember to apply it to all columns  $\geq k$ 
  - It doesn't affect columns  $< k$  (because of all the zeros at the top of  $\hat{v}_k$ )
- Note:  $H_n$  is required to get zeroes at the bottom of the last column
- Letting  $\hat{Q}^T = H_n H_{n-1} \cdots H_2 H_1$  gives  $\hat{Q}^T A = \begin{pmatrix} R \\ 0 \end{pmatrix}$  or  $A = \hat{Q} \begin{pmatrix} R \\ 0 \end{pmatrix}$
- $\|r\|_2 = \|\hat{Q}^T r\|_2 = \left\| \hat{Q}^T \left( b - \hat{Q} \begin{pmatrix} R \\ 0 \end{pmatrix} c \right) \right\|_2 = \left\| \begin{pmatrix} \tilde{b}_r \\ \tilde{b}_z \end{pmatrix} - \begin{pmatrix} Rc \\ 0 \end{pmatrix} \right\|_2$
- Solve  $Rc = \tilde{b}_r$  via back-substitution to minimize  $\|r\|_2$  to a value of  $\|\tilde{b}_z\|_2$

# Unit 11

# Zero Singular Values

# Underdetermined Systems

- Consider drawing a line  $y = c_1 + c_2x$  through 3 data points
- When the points are colinear, there is a unique solution
- When the points are not colinear, there is a least squares solution
- When the points are co-located (i.e. identical), there are infinite solutions



# Underdetermined Systems

- The Vandermonde matrix equation is  $\begin{pmatrix} 1 & x_1 \\ 1 & x_2 \\ 1 & x_3 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$
- When  $x_1 = x_2 = x_3$ , the columns are multiples of each other (matrix is rank 1)
- When  $y_1 = y_2 = y_3$ , the right hand side is in the range of the columns, and there are infinite solutions
- Otherwise, the right hand side is not in the range of the columns, and there are no solutions

# Misleading Labels

- Consider  $\begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 0 \end{pmatrix}$
- The first two rows,  $c_1 = 1$  and  $c_1 = 2$ , overdetermine  $c_1$
- The third row,  $c_2 = 3$ , uniquely determines  $c_2$
- The last row,  $0c_3 = 0$ , leaves  $c_3$  underdetermined with infinite possibilities
- It's misleading to classify an entire system as either having a unique solution, no solution, or infinite solutions
- Rather, one should do the best they can with what has been given
  - E.g. Shouldn't skip dinner because of uncertainties about what time the sun will go down

# SVD (most general framework)

- Transform  $Ac = b$  into  $\Sigma\hat{c} = \hat{b}$  (as usual)
- For each  $\sigma_k \neq 0$ , compute  $\hat{c}_k = \frac{\hat{b}_k}{\sigma_k}$  (as usual)
- When  $\sigma_k = 0$ ,  $\hat{c}_k$  is undefined (moreover, division by a small  $\sigma_k$  is dubious)
- Tall matrices have extra rows with  $0 = \hat{b}_k$  ( $\sigma_k = 0$  rows contribute to this too), and nonzero  $\hat{b}_k$  implies a nonzero residual
- A wide matrix has extra columns of zeros that leave some  $\hat{c}_k$  undetermined (the same as a  $\sigma_k = 0$  column)

# SVD (most general framework)

- Can write tall (and square) matrices as  $A = U \begin{pmatrix} \hat{\Sigma} \\ 0 \end{pmatrix} V^T$  and wide matrices as  $A = U(\hat{\Sigma} \quad 0)V^T$ , where  $\hat{\Sigma}$  may contain zeros on the diagonal
- Alternatively, can write  $A = U \begin{pmatrix} \hat{\Sigma} & 0 \\ 0 & 0 \end{pmatrix} V^T$  with  $\hat{\Sigma}$  diagonal and full rank
- Then  $\Sigma \hat{c} = \hat{b}$  has the form  $\begin{pmatrix} \hat{\Sigma} & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} \hat{c}_r \\ \hat{c}_z \end{pmatrix} = \begin{pmatrix} \hat{b}_r \\ \hat{b}_z \end{pmatrix}$
- $\|r\|_2 = \|U^T(b - Ac)\|_2 = \left\| \begin{pmatrix} \hat{b}_r \\ \hat{b}_z \end{pmatrix} - \begin{pmatrix} \hat{\Sigma} & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} \hat{c}_r \\ \hat{c}_z \end{pmatrix} \right\|_2 = \left\| \begin{pmatrix} \hat{b}_r \\ \hat{b}_z \end{pmatrix} - \begin{pmatrix} \hat{\Sigma} \hat{c}_r \\ 0 \end{pmatrix} \right\|_2$
- So  $\hat{c}_r$  determined via  $\hat{\Sigma} \hat{c}_r = \hat{b}_r$  minimizes the residual to  $\|r\|_2 = \|\hat{b}_z\|_2$
- Meanwhile, any values are acceptable for  $\hat{c}_z$

# Minimum Norm Solution

- Since any values are acceptable for  $\hat{c}_z$ , set  $\hat{c}_z = 0$  in order to indicate that these parameters have no bearing on the solution
- This is more sensible than setting  $\hat{c}_z$  to some nonzero value as if those values mattered
- Example:
  - Consider a variable related to how a hat is worn while driving, which could matter when the hat blocks the sun or keeps hair away from the eyes
  - Someone with short hair driving at night would likely have no driving dependence on the hat; in this case, reporting information on the hat is unimportant/misleading

- Thus,  $\textcolor{red}{c} = V\hat{c} = V \begin{pmatrix} \hat{c}_r \\ \hat{c}_z \end{pmatrix} = V \begin{pmatrix} \hat{\Sigma}^{-1} \hat{b}_r \\ 0 \end{pmatrix} = \sum_{\sigma_k \neq 0} v_k \frac{\hat{b}_k}{\sigma_k} = \sum_{\sigma_k \neq 0} v_k \frac{\textcolor{red}{u_k^T b}}{\sigma_k}$

# Pseudo-Inverse

- The minimum norm solution is  $c = \left( \sum_{\sigma_k \neq 0} \frac{v_k u_k^T}{\sigma_k} \right) b$
- That is,  $c = A^+ b$  with pseudo-inverse  $A^+ = \sum_{\sigma_k \neq 0} \frac{v_k u_k^T}{\sigma_k}$
- When  $A$  is square and full rank  $A^+ = A^{-1}$
- Each term is an outer product between corresponding columns of  $U$  and  $V$  weighted by one over their corresponding singular value
- Each term is a matrix of size  $n \times m$ , so this a sum of matrices

# Sum of Rank One Matrices

- $Ac = U \begin{pmatrix} \hat{\Sigma} & 0 \\ 0 & 0 \end{pmatrix} V^T c = U \begin{pmatrix} \hat{\Sigma} & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} \hat{c}_r \\ \hat{c}_z \end{pmatrix} = U \begin{pmatrix} \hat{\Sigma} \hat{c}_r \\ 0 \end{pmatrix} = \sum_{\sigma_k \neq 0} u_k \sigma_k \hat{c}_k = \sum_{\sigma_k \neq 0} u_k \sigma_k v_k^T c = (\sum_{\sigma_k \neq 0} \sigma_k u_k v_k^T) c$
- Thus,  $A = \sum_{\sigma_k \neq 0} \sigma_k u_k v_k^T$
- Each term is an outer product between corresponding columns of  $U$  and  $V$  weighted by their corresponding singular value
- Each term is a matrix of size  $m \times n$  (the same size as  $A$ )
- Each term is rank 1, since every column in the matrix is a multiple of  $u_k$

# Recall: Understanding $Ac$ (unit 3)

$$\begin{aligned}
 Ac &= \begin{pmatrix} .141 & .825 & -.420 & -.351 \\ .344 & .426 & .298 & .782 \\ .547 & .028 & .644 & -.509 \\ .750 & -.371 & -.542 & .079 \end{pmatrix} \begin{pmatrix} 25.5 & 0 & 0 \\ 0 & 1.29 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} .504 & .574 & .644 \\ -.761 & -.057 & .646 \\ .408 & -.816 & .408 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} \\
 &= \begin{pmatrix} .141 & .825 & -.420 & -.351 \\ .344 & .426 & .298 & .782 \\ .547 & .028 & .644 & -.509 \\ .750 & -.371 & -.542 & .079 \end{pmatrix} \begin{pmatrix} 25.5 & 0 & 0 \\ 0 & 1.29 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} v_1^T c \\ v_2^T c \\ v_3^T c \end{pmatrix} \\
 &= \begin{pmatrix} .141 & .825 & -.420 & -.351 \\ .344 & .426 & .298 & .782 \\ .547 & .028 & .644 & -.509 \\ .750 & -.371 & -.542 & .079 \end{pmatrix} \begin{pmatrix} \sigma_1 v_1^T c \\ \sigma_2 v_2^T c \\ \sigma_3 v_3^T c \\ 0 \end{pmatrix} \\
 &= u_1 \sigma_1 v_1^T c + u_2 \sigma_2 v_2^T c + u_3 \sigma_3 v_3^T c + u_4 0
 \end{aligned}$$

- $Ac$  projects  $c$  onto the basis vectors in  $V$ , scales by the associated singular values, and uses those results as weights on the basis vectors in  $U$

# Matrix Approximation

- Use the  $p$  largest singular values:  $A \approx \sum_{k=1}^p \sigma_k u_k v_k^T$
- The pseudo-inverse is approximated similarly:  $A^+ \approx \sum_{k=1}^p \frac{v_k u_k^T}{\sigma_k}$
- This is the best rank  $p$  approximation to  $A$ , and the main idea behind principle component analysis (PCA)
  - Often, thousands/millions of terms can be thrown away keeping only 10 to 100 terms
- Drop small singular values:  $A \approx \sum_{\sigma_k > \epsilon} \sigma_k u_k v_k^T$
- This makes the pseudo-inverse better conditioned:  $A^+ \approx \sum_{\sigma_k > \epsilon} \frac{v_k u_k^T}{\sigma_k}$ 
  - This relies on a good choice of  $\epsilon > 0$

# Recall: Approximating $A$ (unit 3)

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix} \approx$$
$$\begin{pmatrix} .141 & .325 & - .20 & .51 \\ .344 & .26 & .98 & .72 \\ .547 & .928 & .64 & -.09 \\ .750 & -.371 & -.542 & .79 \end{pmatrix} \begin{pmatrix} 25.5 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} .504 & .574 & .644 \\ -.761 & .057 & .646 \\ .408 & -.816 & .408 \end{pmatrix}$$

- The first singular value is much bigger than the second, and so represents the vast majority of what  $A$  does (note, the vectors in  $U$  and  $V$  are unit length)
- Thus, one could approximate  $A$  quite well by only using the terms associated with the largest singular value
- This is not a valid factorization, but an approximation (and the idea behind PCA)

# Rank One Updates

- For real time applications (e.g. real time decision making), iteratively add one term at a time (slowly improving the estimate)
- $c = A^+b \approx \frac{u_1^T b}{\sigma_1} v_1 + \frac{u_2^T b}{\sigma_2} v_2 + \frac{u_3^T b}{\sigma_3} v_3 + \dots$
- Note the efficient order of operations:
  - $u_k^T b$  is  $m$  multiplies and the result times  $v_k$  is  $n$  multiplies for a total of  $m + n$
  - Instead, multiplying the size  $mxn$  matrix  $v_k u_k^T$  times  $b$  is  $m \cdot n$  multiplies

# Computing the SVD

- $A^T A = V \Sigma^T \Sigma V^T$  so  $(A^T A)V = V(\Sigma^T \Sigma)$
- $AA^T = U \Sigma \Sigma^T U^T$  so  $(AA^T)U = U(\Sigma \Sigma^T)$
- If  $\sigma_k \neq 0$ , then  $\sigma_k^2$  is an eigenvalue of both  $A^T A$  and  $AA^T$  (with eigenvectors  $v_k$  and  $u_k$  respectively)
- Easier to work with the smaller of  $A^T A$  and  $AA^T$  (which are both SP(S)D) to find each eigenvalue  $\sigma_k^2$
- Then  $\sigma_k^2$  can be used in both  $A^T A$  and  $AA^T$  to find the corresponding eigenvectors

# Condition Number of Eigenproblems

- The condition number for finding an eigenvalue is different than the condition number for solving a linear system
- The condition number for finding an eigenvalue/eigenvector pair is  $\frac{1}{v_L^T v_R}$  where  $v_L$  and  $v_R$  are the normalized left/right eigenvectors
- Symmetric (Hermitian) matrices have left/right eigenvectors that are identical, so  $v_L^T v_R = 1$  and the condition number is 1

# Characteristic Polynomial

- The eigenvalue problem is typically written as  $\hat{A}\nu = \lambda\nu$
- Alternatively,  $(\hat{A} - \lambda I)\nu = 0$
- This is true when  $\det(\hat{A} - \lambda I) = 0$ , which leads to a degree  $n$  characteristic polynomial equation in  $\lambda$  (for a size  $n \times n$  matrix  $\hat{A}$ )
- Finding the roots of this equation can be quite difficult
  - Recall how difficult it was to find roots for a mere cubic equation
- **Finding roots for  $n > 3$  is undesirable**

# Similarity Transforms

- Similarity transforms,  $T^{-1}\hat{A}T$ , preserve the eigenstructure
  - $T^{-1}\hat{A}T\nu = \lambda\nu$  or  $\hat{A}(T\nu) = \lambda(T\nu)$  still has eigenvalue  $\lambda$  with modified eigenvector  $T\nu$
- When  $\hat{A}$  is real/symmetric (complex/Hermitian), an orthogonal (unitary)  $T$  exists to make  $T^{-1}\hat{A}T$  diagonal with real eigenvalues
  - e.g.  $T = V$  for  $A^TA = V\Sigma^T\Sigma V^T$  and  $T = U$  for  $AA^T = U\Sigma\Sigma^TU^T$
- When  $\hat{A}$  has distinct eigenvalues, a  $T$  exists to make  $T^{-1}\hat{A}T$  diagonal
- For any (square) matrix, a unitary  $T$  exists to make  $T^{-1}\hat{A}T$  upper triangular (Schur form) with eigenvalues on the diagonal
- Any (square) matrix can be put into Jordan form where the eigenvalues are on the diagonal, and off diagonal elements only occur on the band above the diagonal and only for defective eigenvalues (repeated eigenvalues that don't possess a full set of eigenvectors)

# QR Iteration (for eigenvalues)

- Starting with  $\hat{A}^0 = \hat{A}$
- Compute the factorization  $\hat{A}^q = Q^q R^q$  with orthogonal  $Q^q$
- Then define  $\hat{A}^{q+1} = R^q Q^q$
- Note:  $R^q Q^q = (Q^q)^T Q^q R^q Q^q = (Q^q)^T \hat{A}^q Q^q$  is a similarity transform of  $\hat{A}^q$
- If the eigenvalues are distinct,  $\hat{A}^q$  converges to a triangular matrix
- If  $\hat{A}$  is symmetric,  $\hat{A}^q$  converges to a diagonal matrix

# Power Method

- Computes the largest eigenvalue/eigenvector (great for rank 1 updates)
- Starting from  $c^0 \neq 0$ , iterate  $c^{q+1} = \hat{A}c^q$
- Suppose  $c^0$  is a linear combination of eigenvectors:  $c^0 = \sum_k \alpha_k v_k$
- Then  $c^q = \hat{A}^q c^0 = \sum_k \alpha_k \hat{A}^q v_k = \sum_k \alpha_k \lambda_k^q v_k = \lambda_{\max}^q \sum_k \alpha_k \left(\frac{\lambda_k}{\lambda_{\max}}\right)^q v_k$
- As  $q \rightarrow \infty$ ,  $\left(\frac{\lambda_k}{\lambda_{\max}}\right)^q \rightarrow 0$  for  $\lambda_k < \lambda_{\max}$  and  $c^q \rightarrow \lambda_{\max}^q \alpha_{\max} v_{\max}$
- Thus, as  $q \rightarrow \infty$ ,  $\frac{(c^{q+1})_i}{(c^q)_i} \rightarrow \lambda_{\max}$  for each of the  $i$  components of  $c$
- Deflation removes an eigenvalue from  $\hat{A}$  by subtracting off its rank 1 update
  - For example, the deflated  $A^T A - \sigma_k^2 v_k v_k^T$  or  $AA^T - \sigma_k^2 u_k u_k^T$  can be used to compute the next largest eigenvalue, etc.

# Power Method

- If  $c^0 = \sum_k \alpha_k v_k$  happens to have  $\alpha_{max} = 0$ , the method might fail (but roundoff errors can help)
- When  $c^0$  and  $\hat{A}$  are real valued, cannot obtain complex numbers
- When the largest eigenvalue is repeated, the final vector may be a linear combination of the multiple eigenvectors
- $c^q$  needs to be periodically renormalized to stop it from growing too large
- Inverse Iteration can be used to find the smallest eigenvalue of  $\hat{A}$ , since the largest eigenvalue of  $\hat{A}^{-1}$  is the smallest eigenvalue of  $\hat{A}$ 
  - $c^{q+1} = \hat{A}^{-1}c^q$  is updated by solving  $\hat{A}c^{q+1} = c^q$  to find  $c^{q+1}$
  - Useful for finding the condition number  $\frac{\sigma_{max}}{\sigma_{min}}$

# Unit 12

# Regularization

# Adding the Identity

- Add  $Ic = 0$  to drive components related to small/zero singular values to zero
  - Motivated by minimal norm solution
- Combine with the original system  $\begin{pmatrix} A \\ I \end{pmatrix} c = \begin{pmatrix} b \\ 0 \end{pmatrix}$  so that  $\begin{pmatrix} A \\ I \end{pmatrix}$  has full column rank
  - Can be solved with Householder, etc.
- The normal equations are  $(A^T \quad I) \begin{pmatrix} A \\ I \end{pmatrix} c = (A^T \quad I) \begin{pmatrix} b \\ 0 \end{pmatrix}$  or  $(A^T A + I)c = A^T b$
- Using  $A = U\Sigma V^T$ , this is  $(V\Sigma^T \Sigma V^T + I)c = V\Sigma^T \hat{b}$  or  $(\Sigma^T \Sigma + I)\hat{c} = \Sigma^T \hat{b}$
- The augmented  $\sigma_k^2 + 1$  drive  $\hat{c}_k$  with  $\sigma_k = 0$  to zero (as desired)
- However, nonzero  $\sigma_k$  have their (unique or least squares) solution perturbed as well (since  $Ic = 0$  interferes with  $Ac = b$ )

# Perturbation

- Recall the most general  $\Sigma = \begin{pmatrix} \hat{\Sigma} & 0 \\ 0 & 0 \end{pmatrix}$  with  $\hat{\Sigma}$  diagonal and full rank
- $(\Sigma^T \Sigma + I)\hat{c} = \Sigma^T \hat{b}$  becomes  $\left( \begin{pmatrix} \hat{\Sigma}^T & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} \hat{\Sigma} & 0 \\ 0 & 0 \end{pmatrix} + I \right) \begin{pmatrix} \hat{c}_r \\ \hat{c}_z \end{pmatrix} = \begin{pmatrix} \hat{\Sigma}^T & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} \hat{b}_r \\ \hat{b}_z \end{pmatrix}$
- Or  $\left( \begin{pmatrix} \hat{\Sigma}^2 & 0 \\ 0 & 0 \end{pmatrix} + I \right) \begin{pmatrix} \hat{c}_r \\ \hat{c}_z \end{pmatrix} = \begin{pmatrix} \hat{\Sigma} \hat{b}_r \\ 0 \end{pmatrix}$  where  $\hat{c}_z = 0$  as desired
- However, the  $\hat{c}_r$  terms have  $\hat{c}_k = \frac{\sigma_k}{\sigma_k^2 + 1} \hat{b}_k$  instead of  $\hat{c}_k = \frac{1}{\sigma_k} \hat{b}_k$
- This perturbs  $\hat{c}_k$  away from the correct (unique or least squares) solution

# Regularization

- For larger  $\sigma_k \gg 1$ ,  $\frac{\sigma_k}{\sigma_k^2 + 1} \approx \frac{1}{\sigma_k}$  and the perturbation of the (unique or least squares) solution is negligible
- But for  $\sigma_k \approx 1$ , the perturbation is quite large
- And for  $\sigma_k \ll 1$ ,  $\frac{\sigma_k}{\sigma_k^2 + 1} \approx 0$  drives  $\hat{c}_k$  towards zero
- Adding  $\epsilon I c = 0$  (with  $\epsilon > 0$ ) instead of  $I c = 0$  leads to  $\hat{c}_k = \frac{\sigma_k}{\sigma_k^2 + \epsilon^2} \hat{b}_k$
- This has limited effect on  $\sigma_k \gg \epsilon$ , but helps stabilize/regularize the solution for  $\sigma_k$  near and smaller than  $\epsilon$

# Initial Guess

- Can view setting  $Ic = 0$  as a solution guess of  $c = 0$
- Instead, suppose one had an initial guess of  $c = c^*$
- Add  $Ic = c^*$  to the equations (instead of  $Ic = 0$ ) to get:  $\begin{pmatrix} A \\ I \end{pmatrix} c = \begin{pmatrix} b \\ c^* \end{pmatrix}$
- Normal equations are  $(A^T A + I)c = A^T b + c^*$
- This leads to  $(\Sigma^T \Sigma + I)\hat{c} = \Sigma^T \hat{b} + V^T c^* = \Sigma^T \hat{b} + \hat{c}^*$
- This gives  $\hat{c}_k = \frac{\sigma_k}{\sigma_k^2 + 1} \hat{b}_k + \frac{1}{\sigma_k^2 + 1} \hat{c}_k^*$  which tends towards  $\hat{b}_k$  for larger  $\sigma_k$  as desired but tends towards  $\hat{c}_k^*$  for smaller  $\sigma_k$  (with  $\hat{c}_k = \hat{c}_k^*$  when  $\sigma_k = 0$ )
- Adding  $\epsilon Ic = \epsilon c^*$  gives  $\hat{c}_k = \frac{\sigma_k}{\sigma_k^2 + \epsilon^2} \hat{b}_k + \frac{\epsilon^2}{\sigma_k^2 + \epsilon^2} \hat{c}_k^*$

# Initial Guess

- Rewrite this as  $\hat{c}_k = \left(\frac{\sigma_k^2}{\sigma_k^2 + \epsilon^2}\right) \frac{\hat{b}_k}{\sigma_k} + \left(\frac{\epsilon^2}{\sigma_k^2 + \epsilon^2}\right) \hat{c}_k^*$ 
  - Note the convex weights:  $\left(\frac{\sigma_k^2}{\sigma_k^2 + \epsilon^2}\right) + \left(\frac{\epsilon^2}{\sigma_k^2 + \epsilon^2}\right) = 1$
- This is a convex combination of the usual solution  $\frac{\hat{b}_k}{\sigma_k}$  and the initial guess  $\hat{c}_k^*$ 
  - Also valid for an initial guess of  $\hat{c}_k^* = 0$
- Large  $\sigma_k$  as compared to  $\epsilon$  tend toward the usual solution:  $\hat{c}_k = \frac{\hat{b}_k}{\sigma_k}$
- Small  $\sigma_k$  as compared to  $\epsilon$  tend toward the initial guess:  $\hat{c}_k = \hat{c}_k^*$

# Iterate

- First, solve with  $\epsilon I c = 0$  to get  $\hat{c}_k = \left( \frac{\sigma_k^2}{\sigma_k^2 + \epsilon^2} \right) \frac{\hat{b}_k}{\sigma_k}$
- Then, use this solution as an initial guess and solve again to get:

$$\hat{c}_k = \left( \frac{\sigma_k^2}{\sigma_k^2 + \epsilon^2} \right) \frac{\hat{b}_k}{\sigma_k} + \left( \frac{\epsilon^2}{\sigma_k^2 + \epsilon^2} \right) \left( \frac{\sigma_k^2}{\sigma_k^2 + \epsilon^2} \right) \frac{\hat{b}_k}{\sigma_k} = \left( 1 + \left( \frac{\epsilon^2}{\sigma_k^2 + \epsilon^2} \right) \right) \left( \frac{\sigma_k^2}{\sigma_k^2 + \epsilon^2} \right) \frac{\hat{b}_k}{\sigma_k}$$

- Then, use this solution as an initial guess and solve again to get:

$$\begin{aligned}\hat{c}_k &= \left( \frac{\sigma_k^2}{\sigma_k^2 + \epsilon^2} \right) \frac{\hat{b}_k}{\sigma_k} + \left( \frac{\epsilon^2}{\sigma_k^2 + \epsilon^2} \right) \left( 1 + \left( \frac{\epsilon^2}{\sigma_k^2 + \epsilon^2} \right) \right) \left( \frac{\sigma_k^2}{\sigma_k^2 + \epsilon^2} \right) \frac{\hat{b}_k}{\sigma_k} \\ &= \left( 1 + \left( \frac{\epsilon^2}{\sigma_k^2 + \epsilon^2} \right) + \left( \frac{\epsilon^2}{\sigma_k^2 + \epsilon^2} \right)^2 \right) \left( \frac{\sigma_k^2}{\sigma_k^2 + \epsilon^2} \right) \frac{\hat{b}_k}{\sigma_k}\end{aligned}$$

# Continue Iterating

- Continuing leads to  $\hat{c}_k = \left( 1 + \left( \frac{\epsilon^2}{\sigma_k^2 + \epsilon^2} \right) + \left( \frac{\epsilon^2}{\sigma_k^2 + \epsilon^2} \right)^2 + \left( \frac{\epsilon^2}{\sigma_k^2 + \epsilon^2} \right)^3 + \dots \right) \left( \frac{\sigma_k^2}{\sigma_k^2 + \epsilon^2} \right) \frac{\hat{b}_k}{\sigma_k}$
- The geometric series in parenthesis has  $r = \frac{\epsilon^2}{\sigma_k^2 + \epsilon^2}$
- It converges to  $\frac{1}{1-r} = \frac{\sigma_k^2 + \epsilon^2}{\sigma_k^2}$  giving  $\hat{c}_k = \frac{\hat{b}_k}{\sigma_k}$  in the limit (as desired)
- When  $\sigma_k = 0$ , the convex weights are **0** and **1**, so  $\hat{c}_k = 0$  identically at every step
  - This is the desired Minimum Norm solution for these  $\sigma_k$
- As  $\sigma_k \rightarrow 0$ ,  $\hat{c}_k \rightarrow (1 + 1 + \dots + 1) \left( \frac{\sigma_k}{\epsilon^2} \right) \hat{b}_k \rightarrow 0$  for a finite number of steps
  - Small  $\sigma_k$  are regularized, sending their associated  $\hat{c}_k \rightarrow 0$

# Convergence Rate

- After  $q$  iterations, the geometric series sum is  $\frac{1-r^q}{1-r} = \frac{\sigma_k^2 + \epsilon^2}{\sigma_k^2} \left(1 - \left(\frac{\epsilon^2}{\sigma_k^2 + \epsilon^2}\right)^q\right)$
- This gives  $\hat{c}_k = \left(1 - \left(\frac{\epsilon^2}{\sigma_k^2 + \epsilon^2}\right)^q\right) \frac{\hat{b}_k}{\sigma_k}$  implying monotonic convergence to  $\hat{c}_k = \frac{\hat{b}_k}{\sigma_k}$ 
  - Because  $r = \left(\frac{\epsilon^2}{\sigma_k^2 + \epsilon^2}\right) < 1$  implies  $r^q \rightarrow 0$  monotonically as  $q \rightarrow \infty$
- The convergence is quick for large  $\sigma_k$  (as desired)
- Smaller  $\sigma_k$  have  $\frac{\epsilon^2}{\sigma_k^2 + \epsilon^2}$  closer to 1, so their  $\hat{c}_k$  increase more slowly from zero towards  $\frac{\hat{b}_k}{\sigma_k}$ 
  - Smaller  $\sigma_k$  are thus regularized

# Comparison with PCA

- After  $q$  iterations, PCA incorporates the  $q$  largest  $\sigma_k$  components into the solution
- PCA does not include any contribution (at all) for the other components
  - Smaller  $\sigma_k$  components are Heaviside thresholded to be identically zero
- After  $q$  iterations, the aforementioned iterative approach **does not include the full contribution** of the  $q$  largest  $\sigma_k$  components
  - It only includes  $(1 - r_k^q)$  times those components, but  $(1 - r_k^q)$  is close to 1 when  $\sigma_k$  is large
- The iterative approach includes contributions from **all** components
  - The contribution from smaller  $\sigma_k$  components is smaller, since their  $(1 - r_k^q)$  is not as close to 1 when  $\sigma_k$  is small
  - The aforementioned iterative method has a significantly smoother fall-off for increasing  $\sigma_k$

# Aside

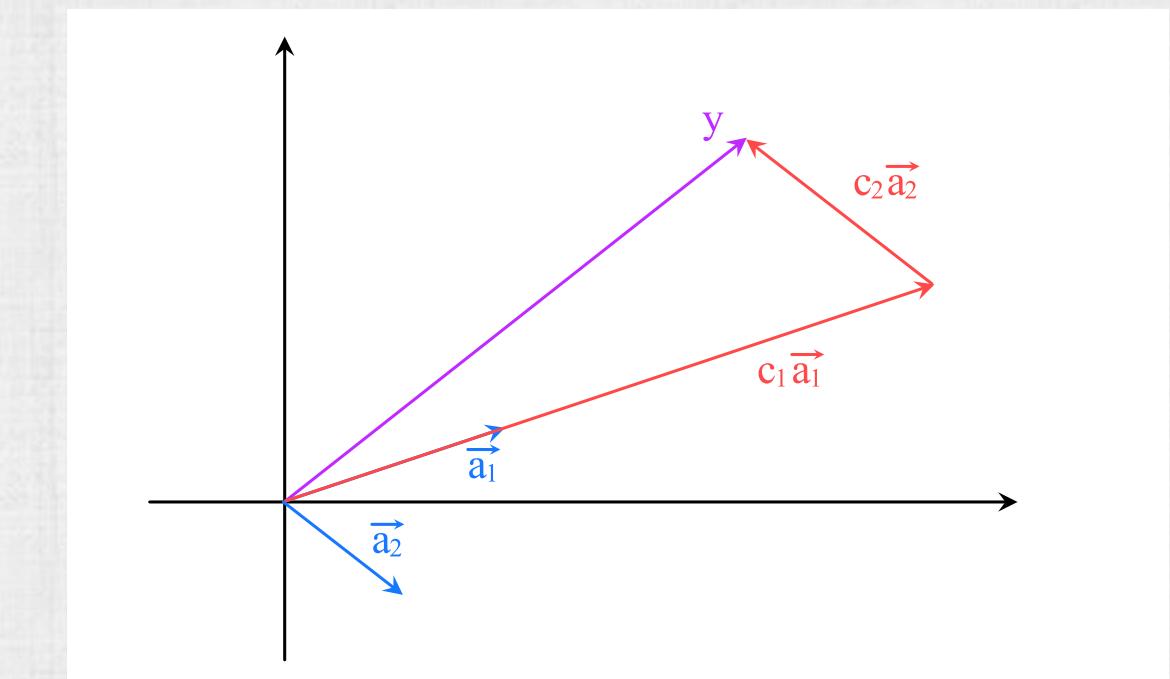
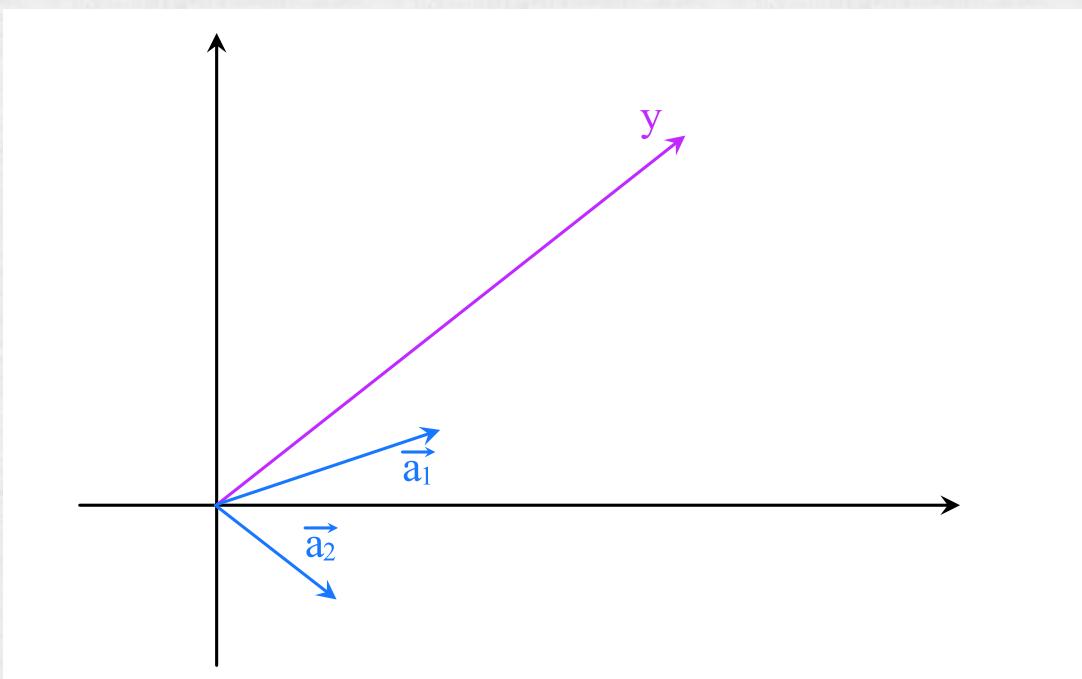
- The aforementioned iterative method and the analysis via a geometric series (slides 7-10) were derived (by the instructor) in preparation for the Winter 2019 offering of this course
- The non-iterative version of the method is a version of Levenberg-Marquardt
- Hyde, D., Bao, M., and Fedkiw, R., "On Obtaining Sparse Semantic Solutions for Inverse Problems, Control, and Neural Network Training", J. Comp. Phys. 443, 110498 (2021).

# Adding a Diagonal Matrix

- Adding  $Dc = 0$  to obtain:  $\begin{pmatrix} A \\ D \end{pmatrix} c = \begin{pmatrix} b \\ 0 \end{pmatrix}$  weights some variables more strongly towards zero than others
- The normal equations are  $(A^T A + D^2)c = A^T b$
- Equivalently  $(V\Sigma^T \Sigma V^T + D^2)c = V\Sigma^T \hat{b}$  or  $(\Sigma^T \Sigma + V^T D^2 V)\hat{c} = \Sigma^T \hat{b}$
- These normal equations can also be derived starting from  $\begin{pmatrix} \Sigma \\ DV \end{pmatrix} \hat{c} = \begin{pmatrix} \hat{b} \\ 0 \end{pmatrix}$ 
  - Unfortunately,  $D$  shears the vectors in  $V$  creating issues
- This motivates column scaling the equations to  $\begin{pmatrix} AD^{-1} \\ I \end{pmatrix} Dc = \begin{pmatrix} b \\ 0 \end{pmatrix}$  so that the resulting  $\begin{pmatrix} \tilde{A} \\ I \end{pmatrix} \tilde{c} = \begin{pmatrix} b \\ 0 \end{pmatrix}$  can be treated in the usual way with  $I\tilde{c} = 0$

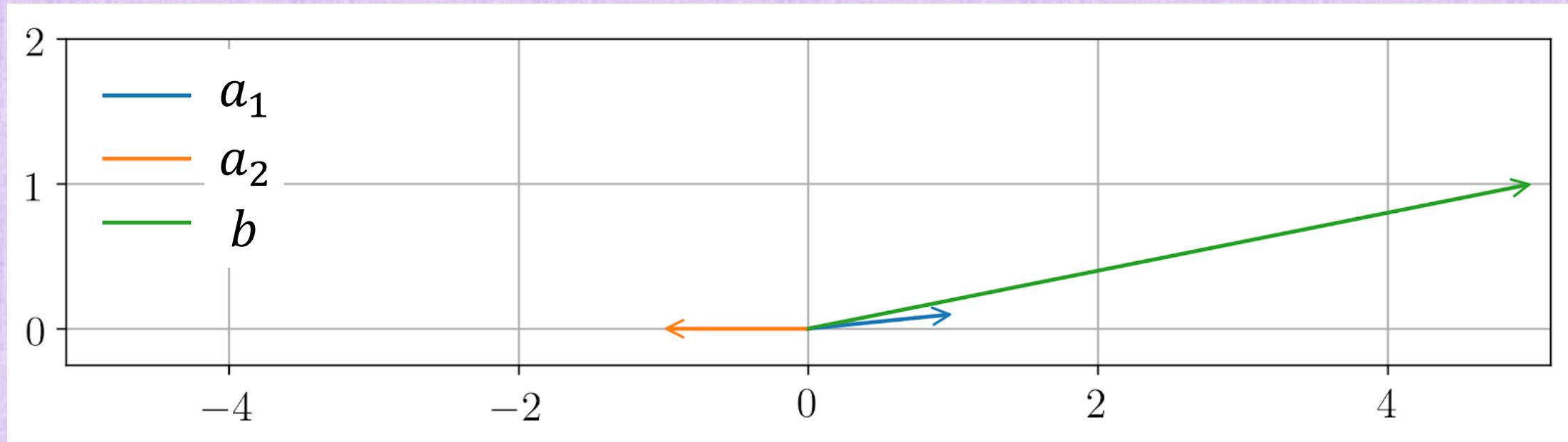
# Recall: Matrix Columns as Vectors (unit 1)

- Let the  $k$ -th column of  $A$  be vector  $a_k$ , so  $Ac = y$  is equivalent to  $\sum_k c_k a_k = y$
- That is, find a linear combination of the columns of  $A$  that gives the right hand side vector  $y$



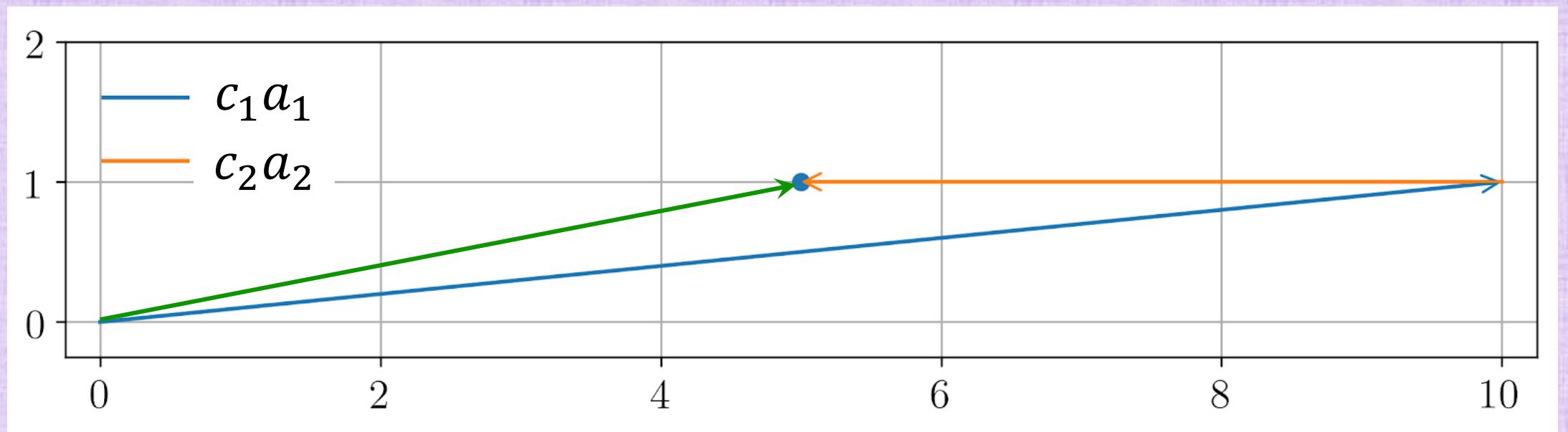
# Example

- Determine  $c_1$  and  $c_2$  such that  $c_1a_1 + c_2a_2 = b$  or  $Ac = b$



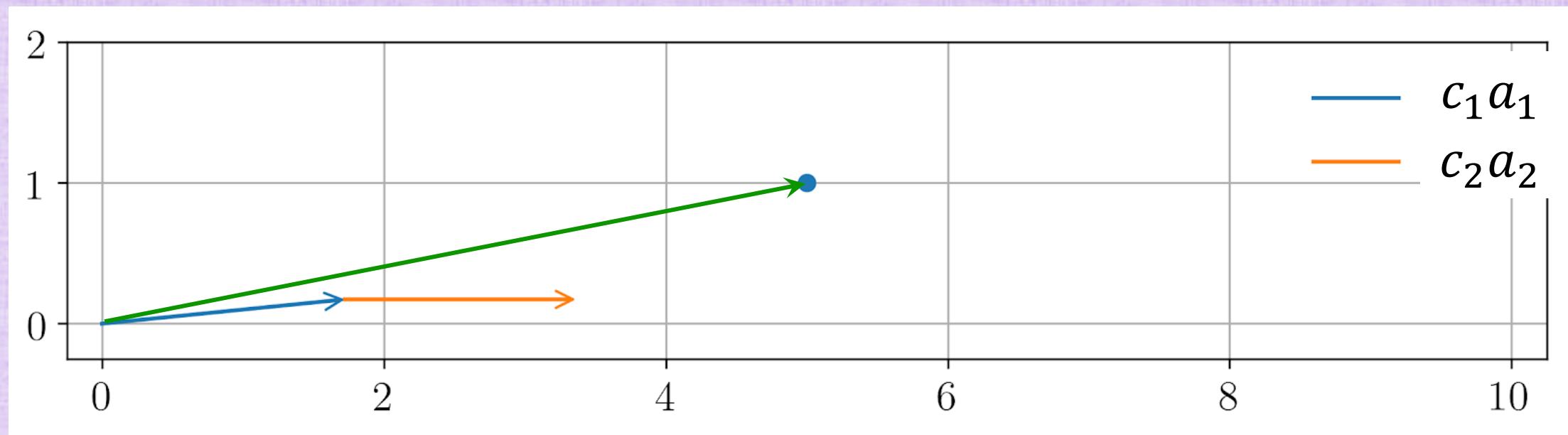
# Example (overshooting)

- Since  $a_1$  and  $a_2$  are not parallel, there is a unique solution
- However, this solution overshoots  $b$  by quite a bit, and then backtracks



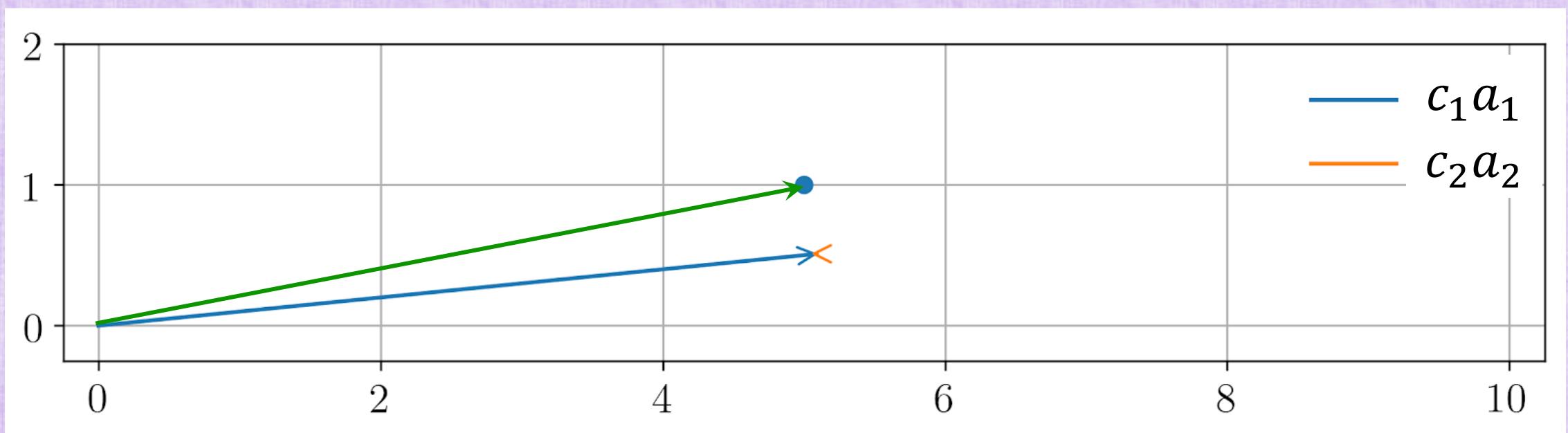
# Example (regularization/damping)

- Adding regularization of  $Ic = 0$  damps both components of the solution



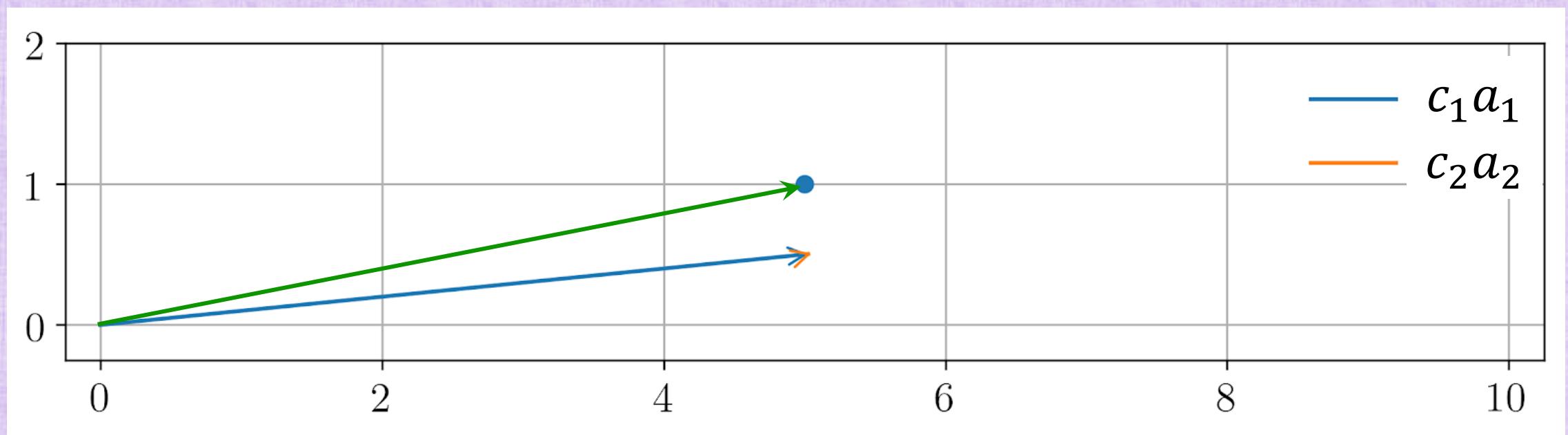
# Example (smarter regularization)

- Adding regularization of  $\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} c = 0$  only damps  $c_2$  and allows  $c_1 a_1$  to estimate  $b$  unhindered



# Example (coordinate descent)

- Coordinate Descent looks at one vector at a time
- After making good progress with  $a_1$ , there is little advantage to using  $a_2$



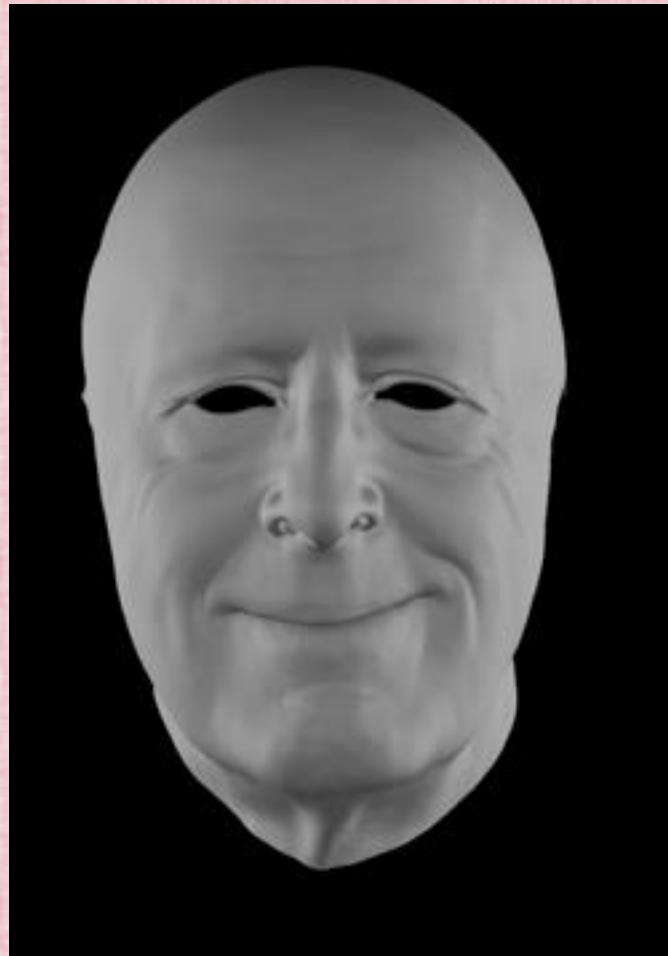
# Geometric Approach (Advantages)

- Thinking geometrically avoids issues with the rank of  $A$
- Other concerns may be more important:
  - Use as few columns as possible - Setting many  $c_k$  to zero gives a sparser solution (which is easier to glean semantic information from)
  - Correlation - Columns more parallel to  $b$  may be more relevant than those more perpendicular
  - Gains - Columns that have a large dot product with  $b$ 's direction make more progress towards  $b$  with smaller  $c_k$  values (more minimal solution norm)

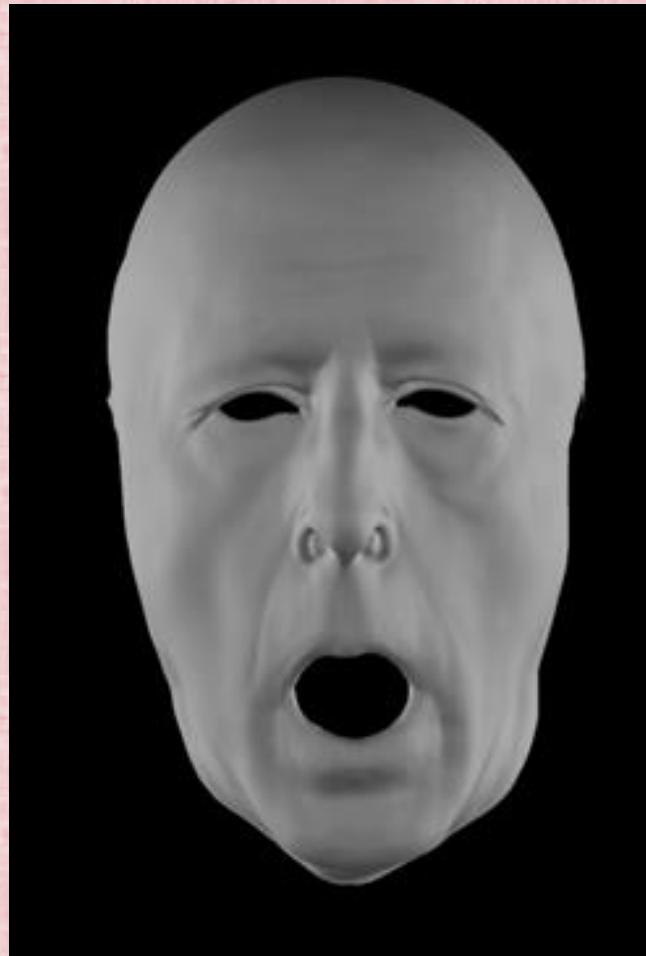
# Correlation vs. Gains

- Consider  $a_k \cdot b = \|a_k\|_2 \|b\|_2 \cos \Theta$  where  $\Theta$  measures how parallel  $a_k$  and  $b$  are
- Correlation preference uses the columns  $a_k$  with smaller  $\Theta$ , i.e. columns that point more closely in the same direction as  $b$
- When the  $c_k$  represent actions, the goal of minimizing action (gains) leads to a preference for smaller  $c_k$ 
  - similar in spirit to  $Ic = 0$  or minimum norm solutions
- Then, columns that make more progress in the direction of  $b$  might be preferable
- Progress in the direction of  $b$  is measured via  $a_k \cdot \frac{b}{\|b\|_2}$  or  $\|a_k\|_2 \cos \Theta$

# Facial Animation



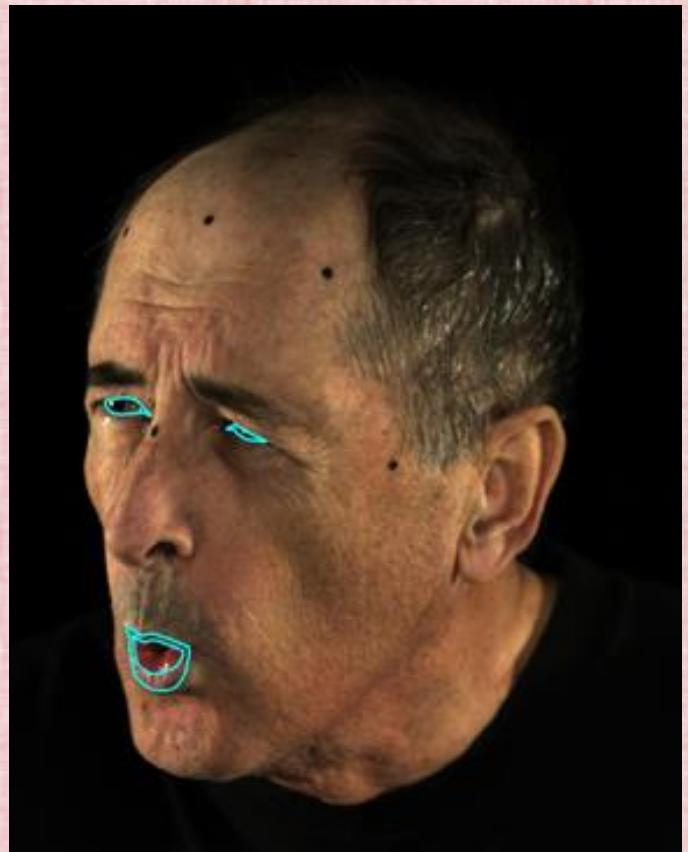
$\varphi(\theta_1)$



$\varphi(\theta_2)$

- Create a procedural skinning model of a face, where (input) animation parameters  $\theta$  lead to a 3D position (output) for every vertex of the face mesh  $\varphi(\theta)$
- E.g. in blend shape systems, each component of  $\theta$  corresponds to a different expression (or sub-expression), and setting multiple components to be nonzero mixes expressions

# Facial Tracking



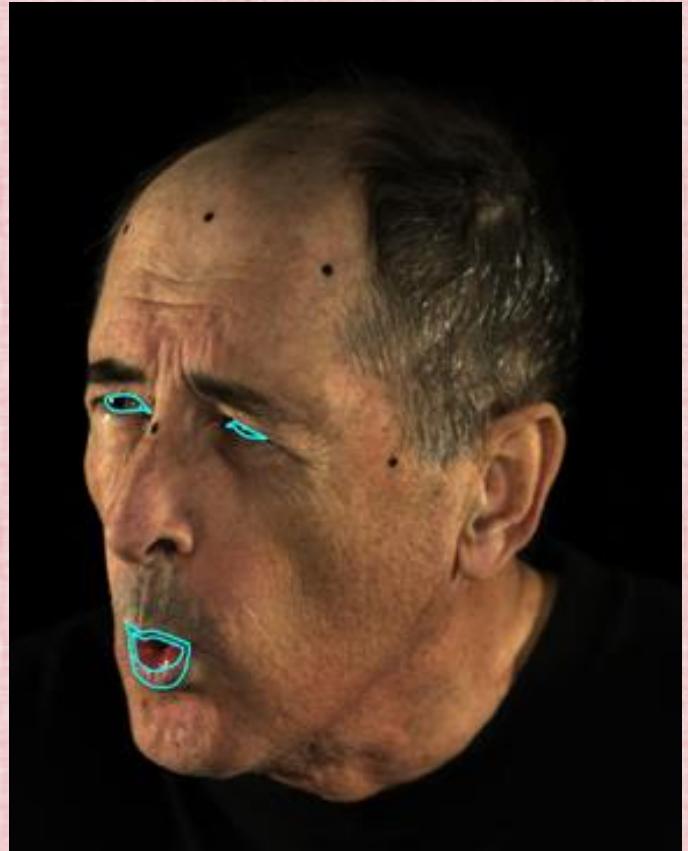
2D RGB Image



3D model

- On the 3D model, embed (red) curves around the eyes/mouth that move with the 3D surface as it deforms
- Draw similar (blue) curves on a 2D RGB image of the actual face
- Goal: projection of the red curves (onto the image plane) should overlap the blue curves (giving an estimate of  $\theta$  from the 2D RGB image)

# Facial Tracking



2D RGB Image



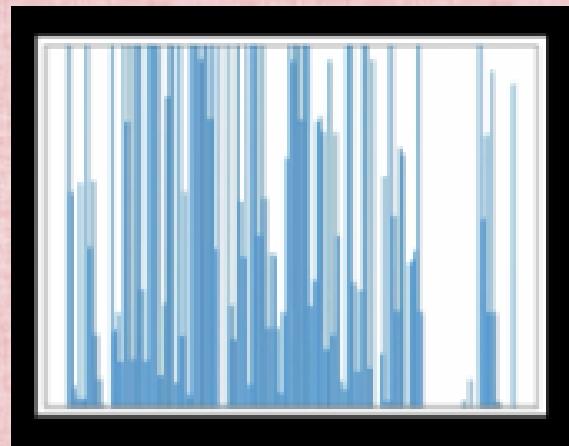
3D model

- The blue curves are data  $C^*$
- The red curves  $C$  are a function of the 3D geometry  $\varphi$ , which in turn is a function of the animation parameters  $\theta$ , i.e.  $C(\varphi(\theta))$
- Determine  $\theta$  that minimizes the difference  $\|C(\varphi(\theta)) - C^*\|$  between the curves

# Solving for the Animation Parameters



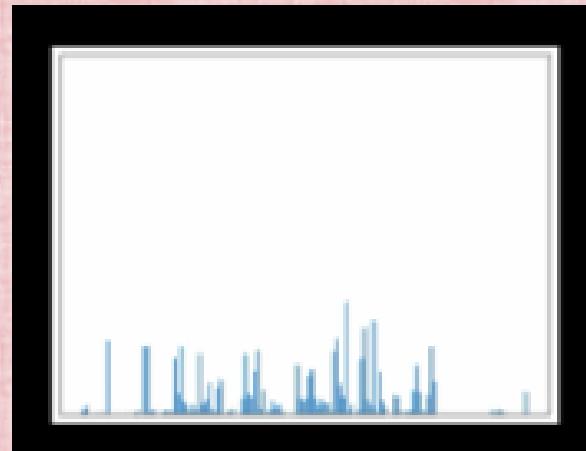
- This nonlinear problem can be solved via optimization
- At every step of optimization, the problem is linearized
- The linear problem  $Ac = b$  gives a search direction that is used to make progress towards the solution



- The optimization performs poorly without regularization
- The resulting  $\theta$  values are wild and arbitrary (as seen in the figure)
- The curves provide **too little data** for the optimization to work well

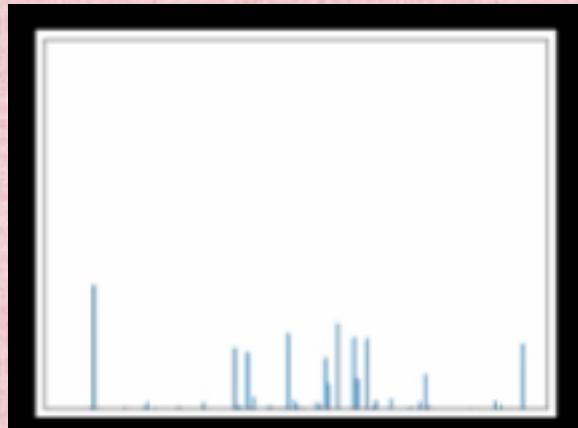
# L2 Regularization

- Adding  $I\theta = 0$  to the linearized problem at every iteration has the expected result:
  - The regularized problem is much more solvable, and the results are less noisy
  - However,  $\theta$  is overly damped (as seen in the figure)
    - Also, a large number of animation parameters  $\theta$  are nonzero, even for this is relatively simple expression
    - This hinders the interpretability (semantics) of  $\theta$



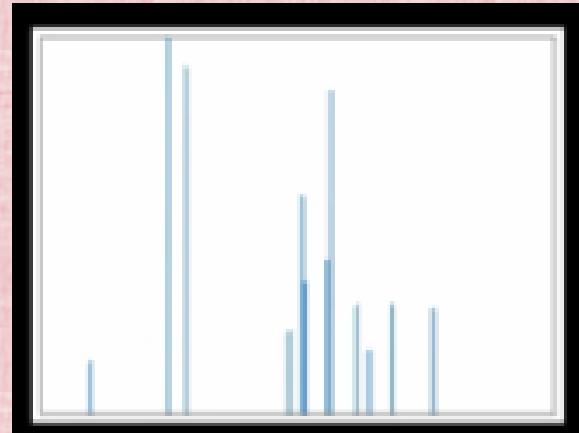
# “Soft L1” Regularization

- There are many options for regularization
- In particular, “soft L1” typically produces a sparser set of solution parameters than L2 regularization (see figure)
- A sparser solution allows one to better ascertain semantic meaning from the animation parameters  $\theta$
- Although,  $\theta$  is still overly damped



# (Geometric) Column Space Search

- The column space search gives a sparse set of solution parameters with significantly less damping
- This allows one to better ascertain semantic meaning from the animation parameters  $\theta$



# Unit 13

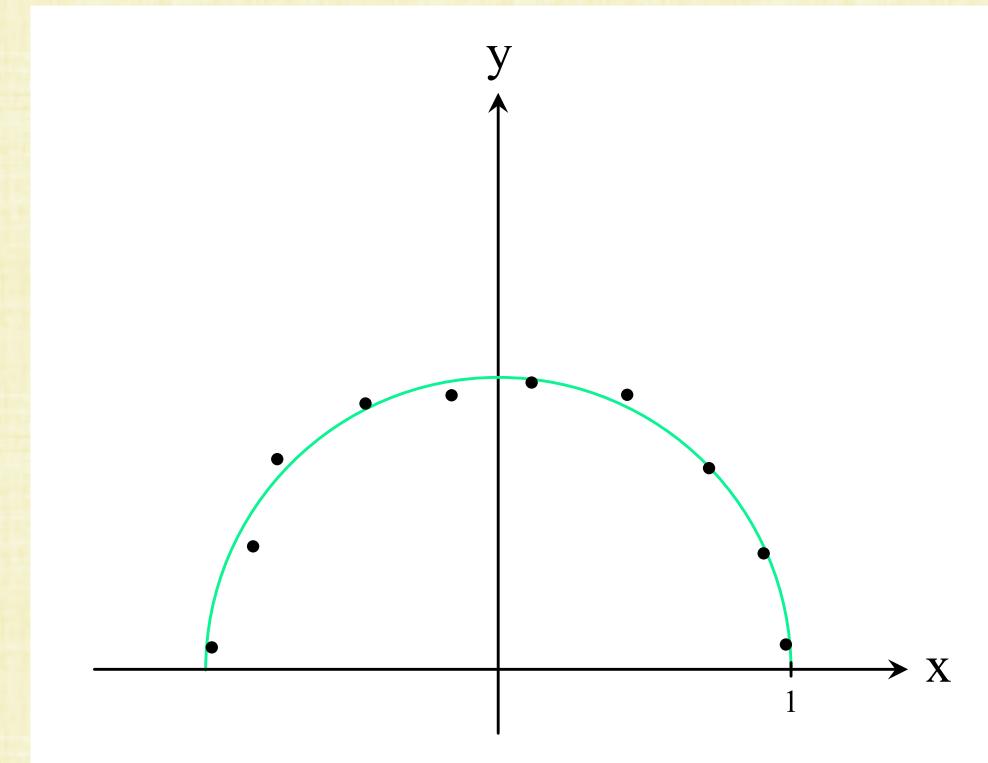
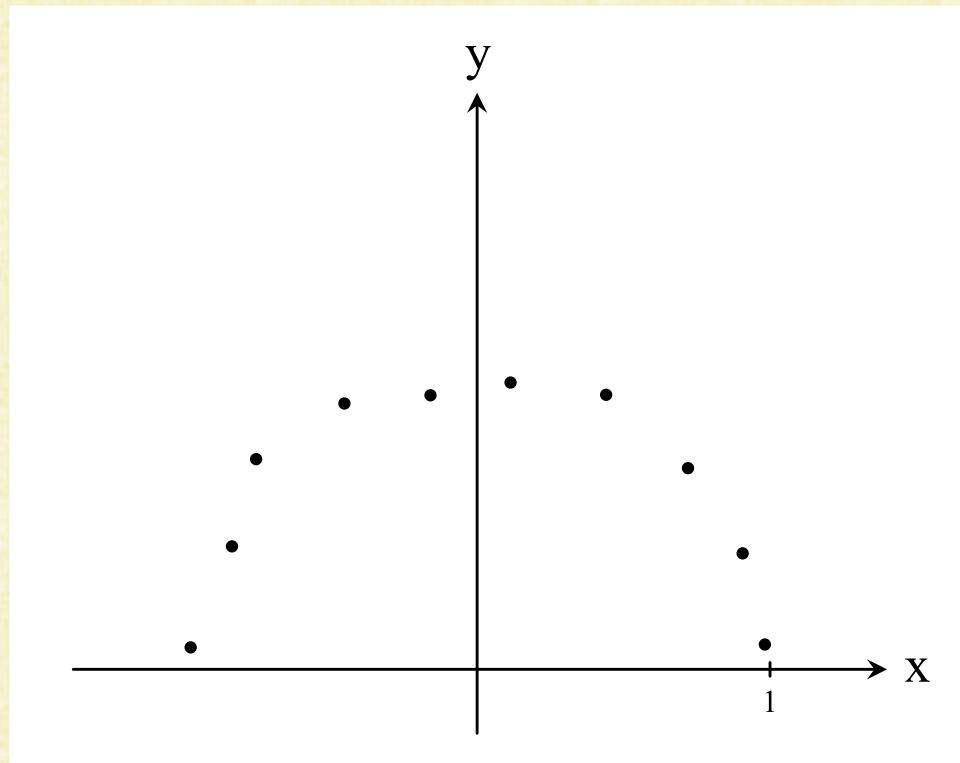
# Optimization

# Part II Roadmap

- Part I – Linear Algebra (units 1-12)  $Ac = b$ 
    - linearize
    - line search
  - Part II – Optimization (units 13-20)
    - (units 13-16) Optimization -> Nonlinear Equations -> 1D roots/minima
    - (units 17-18) Computing/Avoiding Derivatives
    - (unit 19) Hack 1.0: “I give up”  $H = I$  and  $J$  is mostly 0 (descent methods)
    - (unit 20) Hack 2.0: “It’s an ODE!?” (adaptive learning rate and momentum)
- 
- ```
graph TD; A["Part I – Linear Algebra (units 1-12)  $Ac = b$ "] -- "linearize" --> B["(units 13-16) Optimization -> Nonlinear Equations -> 1D roots/minima"]; A -- "line search" --> C["(units 17-18) Computing/Avoiding Derivatives"]; D["(unit 19) Hack 1.0: ‘I give up’  $H = I$  and  $J$  is mostly 0 (descent methods)"]; E["(unit 20) Hack 2.0: ‘It’s an ODE!?’ (adaptive learning rate and momentum)"]; B --- Theory["Theory"]; Methods["Methods"]
```

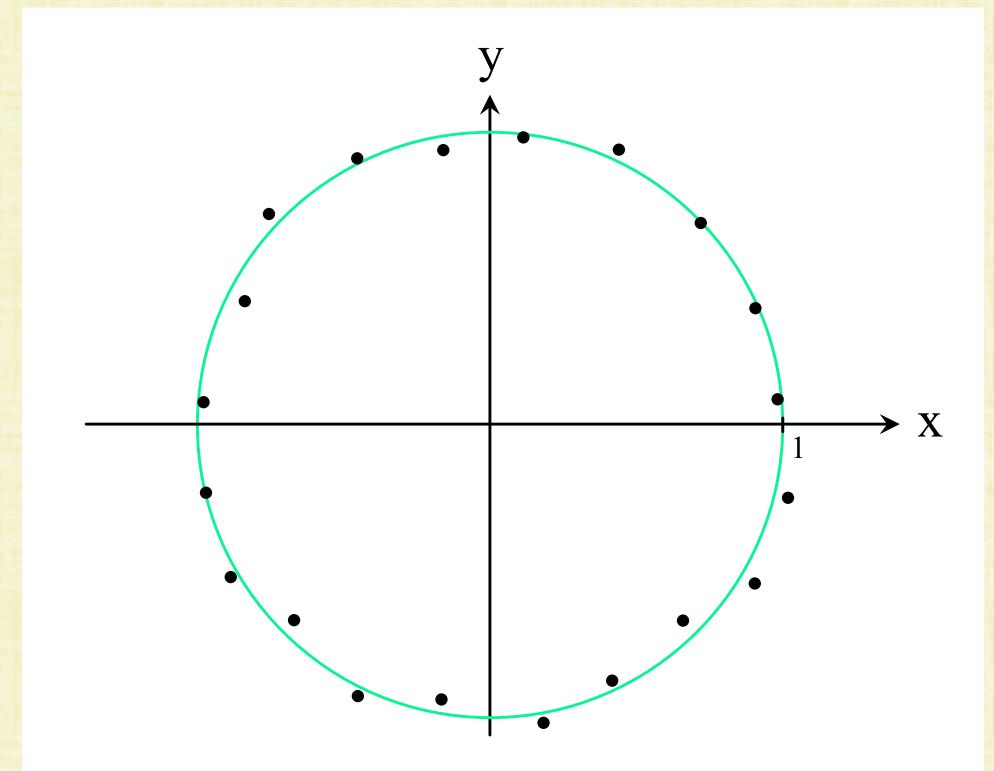
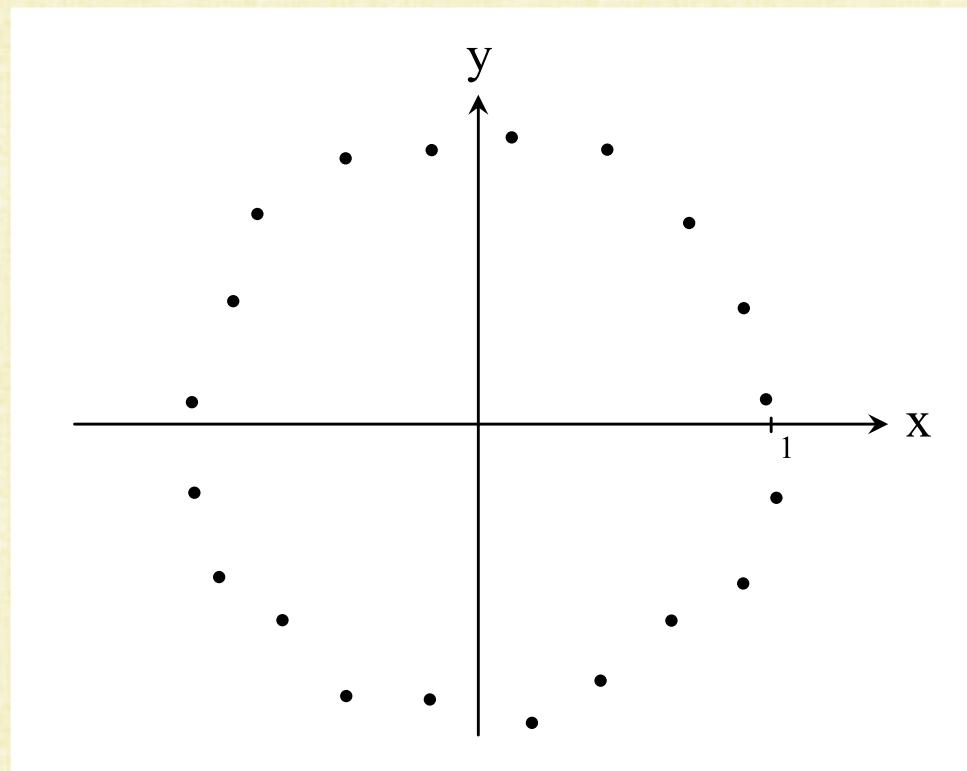
# Function Approximation

- Consider data  $(x_i, y_i)$  as shown below
- Here,  $y = \sqrt{1 - x^2}$  looks like a good approximation



# Function Approximation

- Consider data  $(x_i, y_i)$  as shown below
- Here,  $x^2 + y^2 = 1$  looks like a good approximation (but it's not a function)



# Function Approximation

- A function need not be explicit in  $y$ , just a general relationship between  $x$  and  $y$ , i.e.  $f(x, y) = 0$
- Since it is difficult to consider all possible functions at the same time, one typically chooses a parametric family of possible functions  $f$  (a model for  $f$ )
  - E.g.,  $f$  could be all possible circles  $(x - c_1)^2 + (y - c_2)^2 - c_3^2 = 0$  where the center  $(c_1, c_2)$  and radius  $c_3$  are chosen to best fit the data
- $f(x, y, c) = 0$  could be a family of circles, or polynomials, or a network architecture, etc.
- Determine parameters  $c$  that make  $f(x, y, c) = 0$  best fit the training data, i.e. that make  $\|f(x_i, y_i, c)\|$  close to zero for all  $i$ 
  - Don't forget to be careful about overfitting/underfitting

# Choosing a Norm

- $f(x, y, c)$  may have scalar or vector output; in the latter case, a norm needs to be chosen for  $\|f(x_i, y_i, c)\|$ , e.g.  $L^1$ ,  $L^2$ ,  $L^\infty$ , “soft”  $L^1$ , etc.
  - E.g.,  $\|f(x_i, y_i, c)\|_2 = \sqrt{f(x_i, y_i, c)^T f(x_i, y_i, c)}$
- There is an  $f(x_i, y_i, c)$  for each ordered pair  $(x_i, y_i)$ , so a norm needs to be chosen to combine all of these together as well
  - E.g.,  $\sqrt{\sum_i \|f(x_i, y_i, c)\|_2^2} = \sqrt{\sum_i f(x_i, y_i, c)^T f(x_i, y_i, c)}$
- Find  $c$  that minimizes  $\sqrt{\sum_i f(x_i, y_i, c)^T f(x_i, y_i, c)}$ , or equivalently that minimizes  $\sum_i f(x_i, y_i, c)^T f(x_i, y_i, c)$
- Since all the  $(x_i, y_i)$  are known, the cost function is only a function of  $c$ 
  - Minimize  $\hat{f}(c) = \sum_i f(x_i, y_i, c)^T f(x_i, y_i, c)$ , which is Nonlinear Least Squares

# Optimization

- Minimize cost function  $\hat{f}(c)$ , possibly subject to some constraints
- The constraints can be equations or inequalities (e.g.  $c_k > 0$  for all  $k$ )
- Constraints can often be folded into the cost function, if one is willing to accept the consequences (more on this later)
- When constraints are present, it is called constrained optimization; otherwise, it is called unconstrained optimization
- Since maximizing  $\hat{f}(c)$  is equivalent to minimizing  $-\hat{f}(c)$ , optimization is typically (always) approached as a minimization problem
- Optimization algorithms often get stuck in and/or only guarantee the ability to find local minima (presumably one might prefer global minima)
  - Sometimes finding lots of local minima, and then choosing the smallest of those, is a good strategy

# Conditioning

- Recall: Minimizing the residual  $r = b - Ac$  led to normal equations  $A^T Ac = A^T b$  that square the condition number
- This is an issue for optimization as well:
  - Optimization considers critical points where  $\frac{\partial \hat{f}}{\partial c_k}(c) = 0$  simultaneously for all  $k$
  - Having all partial derivatives approach zero near critical points makes the function locally flat, and thus algorithms struggle to find robust downhill search directions
- The condition number for minimizing  $\hat{f}(c)$  is typically the square of that for solving  $\hat{f}(c) = 0$ 
  - Can only expect **half as many significant digits of accuracy**
  - If an error tolerance of  $\epsilon$  would be used for solving  $\hat{f}(c) = 0$ , then a weaker (larger)  $\sqrt{\epsilon}$  is more appropriate for minimizing  $\hat{f}(c)$

# Nonlinear Systems of Equations

- The critical points are the points where  $\frac{\partial \hat{f}}{\partial c_k}(c) = 0$  simultaneously for all  $k$
- Stacking all the (potentially) nonlinear functions  $\frac{\partial \hat{f}}{\partial c_k}(c)$  into a single vector valued function, the critical points are solutions to:  $F(c) = \begin{pmatrix} \frac{\partial \hat{f}}{\partial c_1}(c) \\ \frac{\partial \hat{f}}{\partial c_2}(c) \\ \vdots \\ \frac{\partial \hat{f}}{\partial c_n}(c) \end{pmatrix} = 0$
- $F(c) = J_{\hat{f}}^T(c) = \nabla \hat{f}(c) = 0$  is a nonlinear system of equations
  - It may have **no solution**, **any finite number of solutions**, or **infinite solutions**

# (Equality) Constrained Optimization

- Constraints can be equalities, e.g.  $\hat{g}(c) = 0$ , or inequalities (see unit 17)
- Given a diagonal matrix  $D$  of (positive) weights indicating the relative importance of various constraints, one can add a penalty term of the form  $\hat{g}^T(c)D\hat{g}(c) \geq 0$  to the cost function and proceed with unconstrained optimization
  - I.e., minimize  $\hat{f}(c) + \hat{g}^T(c)D\hat{g}(c)$  via unconstrained optimization
- Various other options also exist
- An alternative approach uses Lagrange multipliers  $\eta$  as new variables, and minimizes  $\hat{f}(c) + \eta^T \hat{g}(c)$

# Lagrange Multipliers

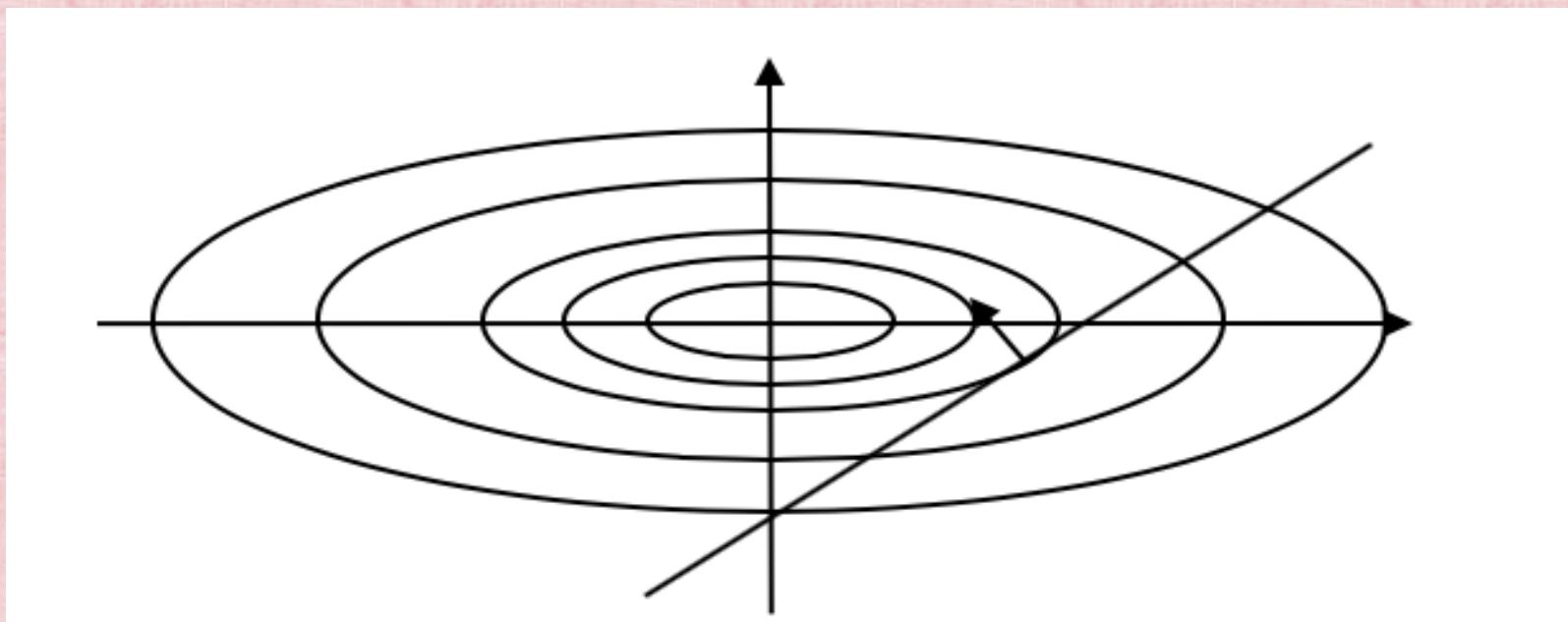
- Minimize  $\hat{f}(c) + \eta^T \hat{g}(c)$
- Critical Points:  $\nabla(\hat{f}(c) + \eta^T \hat{g}(c)) = \begin{pmatrix} J_{\hat{f}}^T(c) + (\eta^T J_{\hat{g}}(c))^T \\ \hat{g}(c) \end{pmatrix} = 0$ 
  - Note: the constraints  $\hat{g}(c) = 0$  are automatically satisfied at critical points
- Critical points satisfy  $J_{\hat{f}}^T(c) = -J_{\hat{g}}^T(c)\eta$  instead of the usual  $J_{\hat{f}}^T(c) = 0$
- In the simple case when  $\hat{g}(c)$  is linear in  $c$ , the Hessian is  $\begin{pmatrix} H_{\hat{f}}(c) & J_{\hat{g}}^T \\ J_{\hat{g}} & 0 \end{pmatrix}$  which is symmetric but not positive definite
  - However, positive definiteness is only required on the tangent space to the constraint surface (i.e., on the null space of  $J_{\hat{g}}$ )

# Lagrange Multipliers (Example)

- Minimize  $\hat{f}(c) = .5c_1^2 + 2.5c_2^2$  subject to  $\hat{g}(c) = c_1 - c_2 - 1 = 0$
- So, minimize  $.5c_1^2 + 2.5c_2^2 + \eta_1(c_1 - c_2 - 1)$
- Critical Points:  $\begin{pmatrix} c_1 \\ 5c_2 \\ c_1 - c_2 - 1 \end{pmatrix} + \begin{pmatrix} 1 \\ -1 \\ 0 \end{pmatrix} \eta_1 = \begin{pmatrix} c_1 + \eta_1 \\ 5c_2 - \eta_1 \\ c_1 - c_2 - 1 \end{pmatrix} = 0$
- $\begin{pmatrix} 1 & 0 & 1 \\ 0 & 5 & -1 \\ 1 & -1 & 0 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \eta_1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$  or  $\begin{pmatrix} c_1 \\ c_2 \\ \eta_1 \end{pmatrix} = \begin{pmatrix} 5/6 \\ -1/6 \\ -5/6 \end{pmatrix}$
- The Hessian is  $\begin{pmatrix} (1 & 0) & (1) \\ (0 & 5) & (-1) \\ (1 & -1) & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 5 & -1 \\ 1 & -1 & 0 \end{pmatrix}$

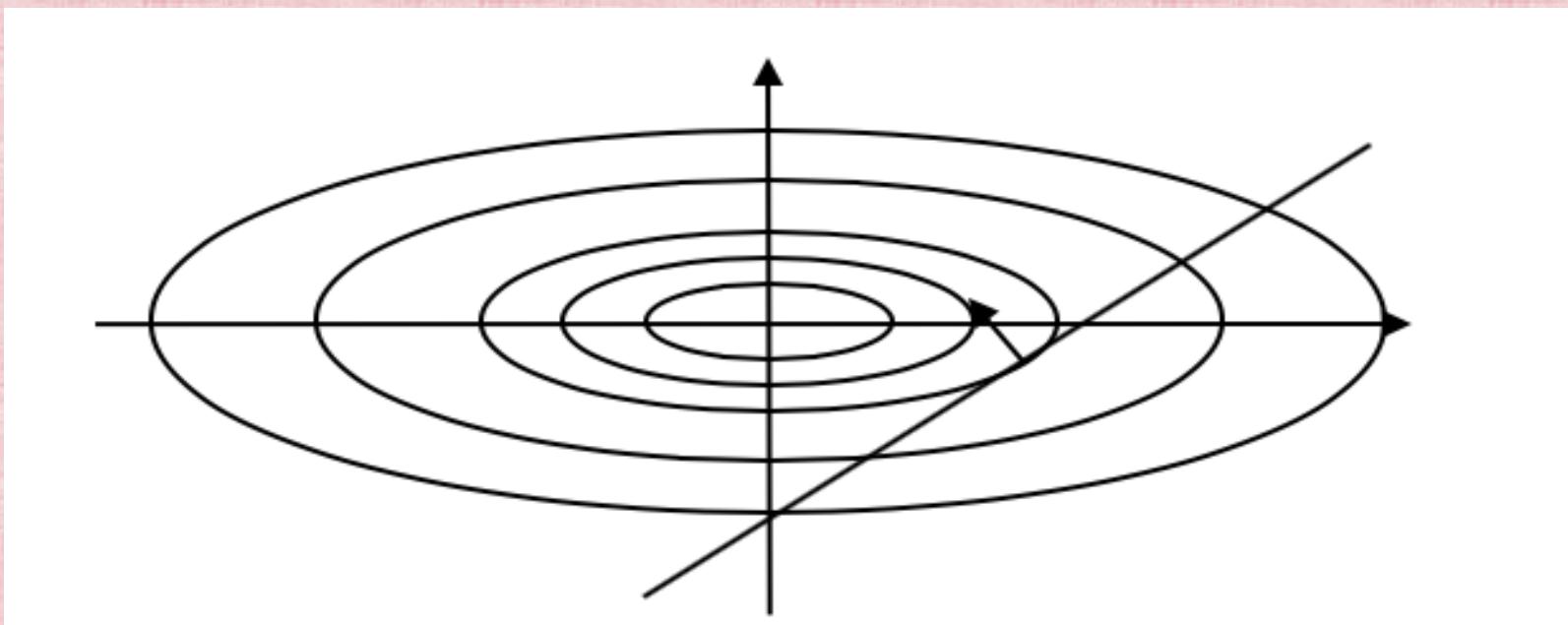
# Lagrange Multipliers (Example)

- Isocontours of  $\hat{f}(c)$  are ellipses, and the constraint is the line  $c_2 = c_1 - 1$
- At critical point  $\left(\frac{5}{6}, -\frac{1}{6}\right)$ , the steepest descent direction  $-\nabla \hat{f} = \left(-\frac{5}{6}, \frac{5}{6}\right)$  is perpendicular to the constraint surface that has direction  $(1,1)$



# Lagrange Multipliers (Example)

- Plugging  $c_2 = c_1 - 1$  into  $\hat{f}(c)$  gives  $.5c_1^2 + 2.5(c_1 - 1)^2 = 3c_1^2 - 5c_1 + 2.5$  which is a parabola with minimum at  $c_1 = \frac{5}{6}$  (as expected)



# Unit 14

# Nonlinear Systems

# Part II Roadmap

- Part I – Linear Algebra (units 1-12)  $Ac = b$ 
    - Part II – Optimization (units 13-20)
      - (units 13-16) Optimization -> Nonlinear Equations -> 1D roots/minima
      - (units 17-18) Computing/Avoiding Derivatives
      - (unit 19) Hack 1.0: “I give up”  $H = I$  and  $J$  is mostly 0 (descent methods)
      - (unit 20) Hack 2.0: “It’s an ODE!?” (adaptive learning rate and momentum)
- 
- ```
graph TD; A["Part I – Linear Algebra (units 1-12)  $Ac = b$ "] -- "linearize" --> B["Nonlinear Equations"]; A -- "line search" --> C["1D roots/minima"]; D["Part II – Optimization (units 13-20)"] --- E["(units 13-16) Optimization -> Nonlinear Equations -> 1D roots/minima"]; D --- F["(units 17-18) Computing/Avoiding Derivatives"]; D --- G["(unit 19) Hack 1.0: ‘I give up’  $H = I$  and  $J$  is mostly 0 (descent methods)"]; D --- H["(unit 20) Hack 2.0: ‘It’s an ODE!?’ (adaptive learning rate and momentum)"]; bracket[{"Theory", "Methods"}] --- C
```

# Recall: Jacobian (Unit 9)

- The Jacobian of  $F(c) = \begin{pmatrix} F_1(c) \\ F_2(c) \\ \vdots \\ F_m(c) \end{pmatrix}$  has entries  $J_{ik} = \frac{\partial F_i}{\partial c_k}(c)$

- Thus, the Jacobian  $J(c) = F'(c) = \begin{pmatrix} \frac{\partial F_1}{\partial c_1}(c) & \frac{\partial F_1}{\partial c_2}(c) & \cdots & \frac{\partial F_1}{\partial c_n}(c) \\ \frac{\partial F_2}{\partial c_1}(c) & \frac{\partial F_2}{\partial c_2}(c) & \cdots & \frac{\partial F_2}{\partial c_n}(c) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial F_m}{\partial c_1}(c) & \frac{\partial F_m}{\partial c_2}(c) & \cdots & \frac{\partial F_m}{\partial c_n}(c) \end{pmatrix}$

# Linearization

- Solving a nonlinear system  $F(c) = 0$  is difficult
- Linearize via the multidimensional version of the Taylor expansion:

$$F(c) \approx F(c^*) + F'(c^*) (c - c^*) = F(c^*) + F'(c^*)\Delta c$$

- More valid when  $\Delta c$  is small (i.e. for  $c$  close enough to  $c^*$ )
- Alternatively written as  $F(c) - F(c^*) \approx F'(c^*)\Delta c$
- The chain rule  $\frac{dF(c)}{dt} = F'(c) \frac{dc}{dt}$  works for any variable  $t$ , and can be written in differential form:  $dF(c) = F'(c)dc$ 
  - Often referred to as the total derivative
  - Using finite size differentials leads to the approximation:  $\Delta F(c) \approx F'(c)\Delta c$
  - In 1D,  $df = f'(c)dc$  and  $\Delta f \approx f'(c)\Delta c$  are the usual  $\frac{df}{dc} = f'(c)$  and  $\frac{\Delta f}{\Delta c} \approx f'(c)$

# Newton's Method

- Iteratively, starting with  $c^0$ , recursively find:  $c^1, c^2, c^3, \dots$
- Newton's Method uses  $\Delta F(c) \approx F'(c)\Delta c$  to write  $F(c^{q+1}) - F(c^q) = F'(c^q)\Delta c^q$  where  $\Delta c^q = c^{q+1} - c^q$ 
  - Aiming for  $F(c) = 0$  motivates setting  $F(c^{q+1}) = 0$
  - Alternatively, set  $F(c^{q+1}) = \beta F(c^q)$  where  $0 \leq \beta < 1$  aims to more slowly shrink  $F(c^q)$  towards zero
  - Then,  $F'(c^q)\Delta c^q = (\beta - 1)F(c^q)$  with  $\beta$  often set to 0
- The linear system  $F'(c^q)\Delta c^q = (\beta - 1)F(c^q)$  is solved for  $\Delta c^q$ , which is used to update  $c^{q+1} = c^q + \Delta c^q$

# Newton's Method

- Requires **repeatedly solving a linear system**, making robustness and efficiency for linear system solvers quite important
  - Need to consider size, rank, conditioning, symmetry, etc. of  $F'(c^q)$
- $F'(c^q)$  may be difficult to compute, since it requires every first derivative
  - Newton's Method contains linearization errors, so approximations of  $F'(c^q)$  are often valid/worthwhile (e.g. symmetrization, etc.)
  - More on this in units 17/18 Computing/Avoiding Derivatives
- Generally speaking, there are no guarantees on convergence
  - May converge to any one of many roots when multiple roots exist, or not converge at all

# Solving Linear Systems (Review)

- Theory, all matrices: **SVD** (units 3, 9, 11)
- Square, full rank, dense:
  - LU factorization with pivoting (unit 2)
  - Symmetric: **Cholesky** factorization (unit 4), **Symmetric approximation** (unit 4)
- Square, full rank, sparse (iterative solvers) (unit 5):
  - SPD (sometimes SPSD): **Conjugate Gradients**
  - Nonsymmetric/Indefinite: GMRES, MINRES, BiCGSTAB (not steepest descent)
- Tall, full rank (least squares to minimize residual) (unit 8):
  - normal equations (units 9, 10), **QR**, Gram-Schmidt, **Householder** (unit 10)
- Any size/rank (minimum norm solution) (unit 11):
  - Pseudo-Inverse, **PCA approximation**, **Power Method** (unit 11)
  - **Levenberg-Marquardt** (iteration too), **Column Space Geometric Approach** (unit 12)

# Line Search

- Given the linearization error in  $F'(c^q)\Delta c^q = -F(c^q)$ , the resulting  $\Delta c^q$  can lead to a poor estimate for  $c^{q+1}$  via  $c^{q+1} = c^q + \Delta c^q$
- Thus,  $\Delta c^q$  is often (instead) used as a search direction, i.e.  $c^{q+1} = c^q + \alpha^q \Delta c^q$
- The parameterized line  $c^{q+1}(\alpha) = c^q + \alpha \Delta c^q$  is used as a 1D (input) domain
- Find  $\alpha^q$  such that  $F(c^{q+1}(\alpha^q)) = 0$  simultaneously for all equations
- Safe Set methods restrict  $\alpha$  in various ways, e.g.  $0 \leq \alpha \leq 1$

# Line Search

- Since  $F$  is vector valued, consider  $g(\alpha) = F(c^{q+1}(\alpha))^T F(c^{q+1}(\alpha)) = 0$
- Since  $g(\alpha) \geq 0$ , solutions to  $F(c^{q+1}(\alpha)) = 0$  are minima of  $g(\alpha)$
- $g(\alpha)$  might be strictly positive (with no  $g(\alpha) = 0$ ), but minimizing  $g(\alpha)$  might still help to make progress towards a solution
- Option 1: find simultaneous roots of the vector valued  $F(c^{q+1}(\alpha)) = 0$
- Option 2: find roots of or minimize  $g(\alpha) = \frac{1}{2} F^T(c^{q+1}(\alpha)) F(c^{q+1}(\alpha))$

# Optimization Problems

- Minimize the scalar cost function  $\hat{f}(c)$  by finding the critical points where  $\nabla \hat{f}(c) = J_{\hat{f}}^T(c) = F(c) = 0$
- $F'(c^q)\Delta c^q = -F(c^q)$  gives the search direction, where  $F'(c) = J_F(c) = H_{\hat{f}}^T(c)$
- That is, solve  $H_{\hat{f}}^T(c^q)\Delta c^q = -J_{\hat{f}}^T(c^q)$  to find the search direction  $\Delta c^q$
- Option 1: find simultaneous roots of the vector valued  $J_{\hat{f}}^T(c^{q+1}(\alpha)) = 0$ , which are critical points of  $\hat{f}(c)$
- Option 2: find roots of or minimize  $g(\alpha) = \frac{1}{2}J_{\hat{f}}(c^{q+1}(\alpha))J_{\hat{f}}^T(c^{q+1}(\alpha))$  to find or make progress toward critical points of  $\hat{f}(c)$
- Option 3: minimize  $\hat{f}(c^{q+1}(\alpha))$  directly

# Unit 15

# 1D Root Finding

# Part II Roadmap

- Part I – Linear Algebra (units 1-12)  $Ac = b$ 
    - linearize
    - line search
  - Part II – Optimization (units 13-20)
    - (units 13-16) Optimization -> Nonlinear Equations -> 1D roots/minima
    - (units 17-18) Computing/Avoiding Derivatives
    - (unit 19) Hack 1.0: “I give up”  $H = I$  and  $J$  is mostly 0 (descent methods)
    - (unit 20) Hack 2.0: “It’s an ODE!?” (adaptive learning rate and momentum)
- 
- ```
graph TD; A["Part I – Linear Algebra (units 1-12)  $Ac = b$ "]; B["Part II – Optimization (units 13-20)"]; C["(units 13-16) Optimization -> Nonlinear Equations -> 1D roots/minima"]; D["(units 17-18) Computing/Avoiding Derivatives"]; E["(unit 19) Hack 1.0: ‘I give up’  $H = I$  and  $J$  is mostly 0 (descent methods)"]; F["(unit 20) Hack 2.0: ‘It’s an ODE!?’ (adaptive learning rate and momentum)"]; G["Theory"]; H["Methods"]; A -- "linearize" --> C; A -- "line search" --> C; C --> D; C --> E; C --> F; C --> G; C --> H;
```

# Fixed Point Iteration

- Find roots of  $g(t)$ , i.e. where  $g(t) = 0$
- Let  $\hat{g}(t) = g(t) + t$  and iterate  $t^{q+1} = \hat{g}(t^q)$  until convergence
- The converged  $t^*$  satisfies  $t^* = \hat{g}(t^*) = g(t^*) + t^*$  implying that  $g(t^*) = 0$
- Converges when:  $|g'(t^*)| < 1$ , the initial guess is close enough, and  $g$  is sufficiently smooth
- $e^{q+1} = t^{q+1} - t^* = \hat{g}(t^q) - \hat{g}(t^*) = g'(\hat{t})(t^q - t^*) = g'(\hat{t})e^q$  for some  $\hat{t}$  between  $t^{q+1}$  and  $t^*$  (by the Mean Value Theorem)
- When all  $g'(\hat{t})$  have  $|g'(\hat{t})| \leq C < 1$ , then  $|e^q| \leq C^q |e^0|$  proves convergence

# Convergence Rate

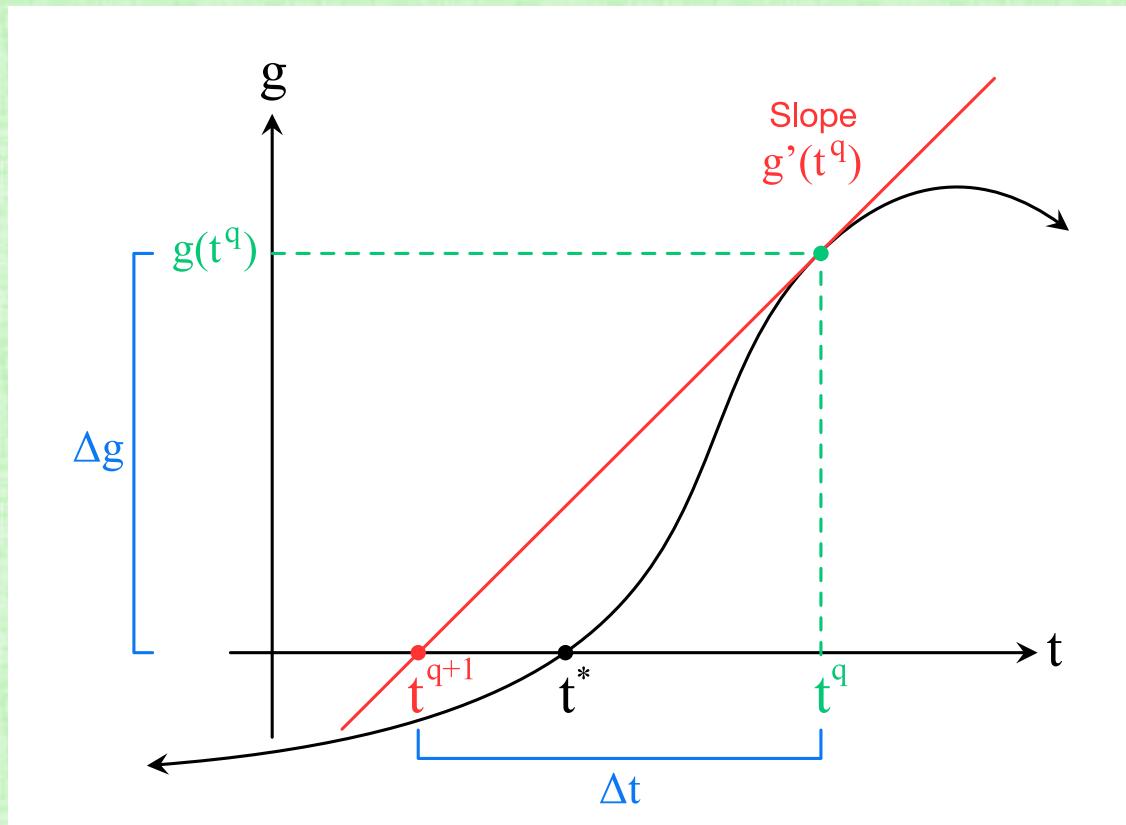
- Consider  $\|e^{q+1}\| \leq C\|e^q\|^p$  as  $q \rightarrow \infty$  where  $C \geq 0$ 
  - When  $p = 1$ ,  $C < 1$  is required, and the convergence rate is linear
  - When  $p > 1$ , the convergence rate is superlinear
  - When  $p = 2$ , the convergence rate is quadratic
- Statements only apply asymptotically (once convergence is happening)
- No guarantee of converging to a desired root (when other roots are present)
- Recall,  $g(t) = 0$  may contain approximations, so it's not clear how accurate the root finder needs to be anyways

# 1D Newton's Method

- Solve  $g'(t^q)\Delta t = -g(t^q)$  and update  $t^{q+1} = t^q + \Delta t = t^q - \frac{g(t^q)}{g'(t^q)}$
- Stop when  $|g(t^q)| < \epsilon$ , which implies  $|t^{q+1} - t^q| < \frac{\epsilon}{|g'(t^q)|}$ 
  - Thus, poorly conditioned when  $g'(t^*)$  is small
  - Especially problematic for repeated roots where  $g'(t^*) = 0$
- Quadratic convergence rate ( $p = 2$ )
- Requires computing  $g$  and  $g'$  every iteration, and computing derivatives isn't always straightforward/cheap (see units 17/18 Computing/Avoiding Derivatives)

# 1D Newton's Method

- $t^{q+1} = t^q - \frac{g(t^q)}{g'(t^q)}$  or alternatively  $g'(t^q) = \frac{\Delta g}{\Delta t} = \frac{g(t^q) - 0}{t^q - t^{q+1}}$

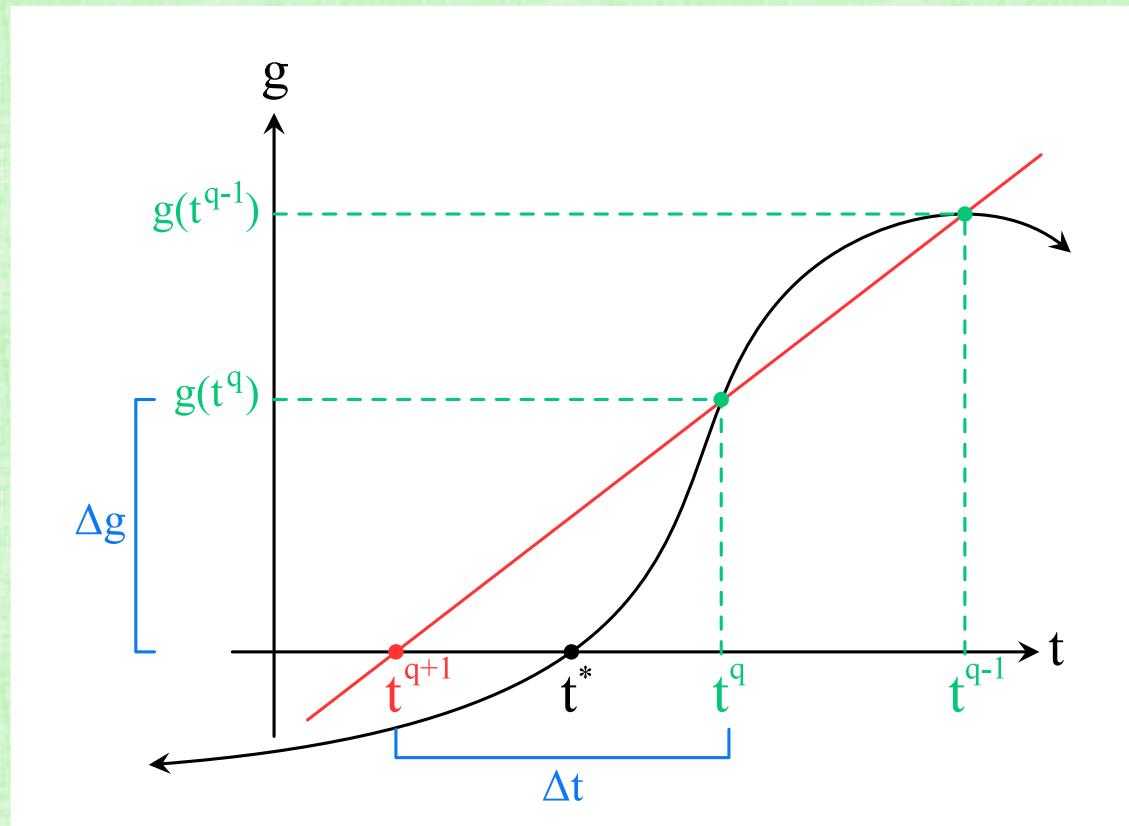


# Secant Method

- Replace  $g'(t^q)$  in Newton's method with an estimate (a few choices for this)
- Standard technique/method draws a line through previous iterates
- Estimate  $g'(t^q) \approx \frac{g(t^q) - g(t^{q-1})}{t^q - t^{q-1}}$
- Then  $t^{q+1} = t^q - g(t^q) \frac{t^q - t^{q-1}}{g(t^q) - g(t^{q-1})}$
- Superlinear convergence rate with  $p \approx 1.618$
- Often/typically faster than Newton, since only  $g$  (not  $g'$ ) is needed while only a few extra iterations are required for the same accuracy

# Secant Method

- $t^{q+1} = t^q - g(t^q) \frac{t^q - t^{q-1}}{g(t^q) - g(t^{q-1})}$  based on  $g'(t^q) \approx \frac{g(t^q) - g(t^{q-1})}{t^q - t^{q-1}}$

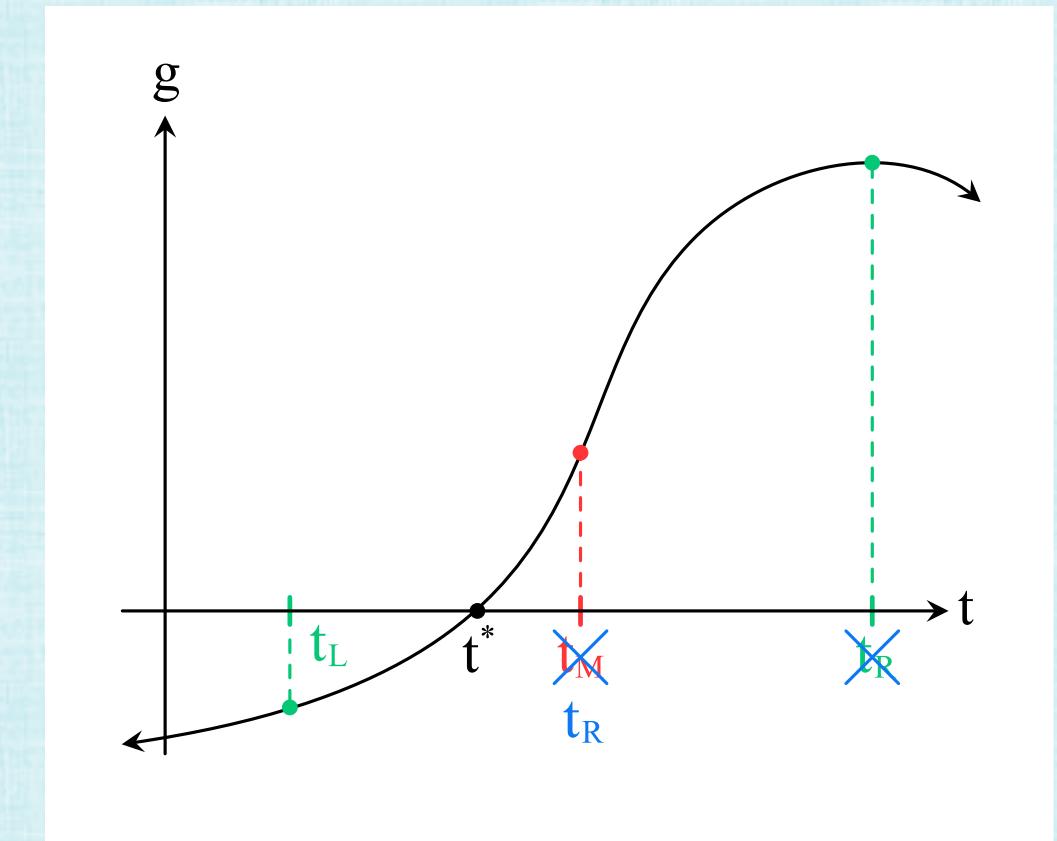
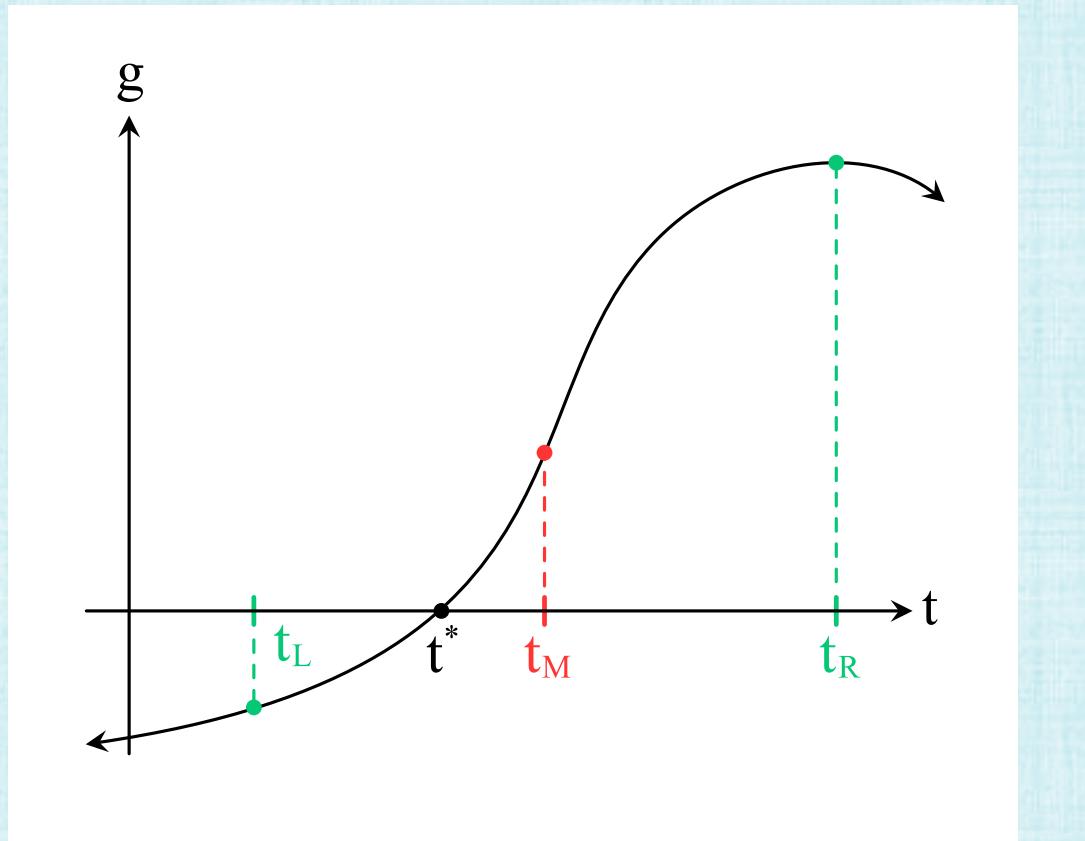


# Bisection Method

- If  $g(t_L)g(t_R) < 0$ , then (assuming continuity) the sign change indicates a root in the interval  $[t_L, t_R]$
- Let  $t_M = \frac{t_L + t_R}{2}$ , and if  $g(t_L)g(t_M) < 0$ , set  $t_R = t_M$ 
  - Otherwise, set  $t_L = t_M$  knowing that  $g(t_M)g(t_R) < 0$  is true
- Iterate until  $t_R - t_L < \epsilon$
- Guaranteed to converge to a root in the interval (unlike Newton/Secant)
- The interval shrinks in size by a factor of two each iteration
- So, linear convergence rate ( $p = 1$ ) with  $C = \frac{1}{2}$

# Bisection Method

- If  $g(t_L)g(t_M) < 0$ , set  $t_R = t_M$ ; otherwise, set  $t_L = t_M$



# Mixed Methods

- Given an interval with a root indicated by  $g(t_L)g(t_R) < 0$
- Iterate with Newton/Secant as long as the iterates stay inside the interval
  - When iteration attempts to leave the interval, use prior iterates to shrink the interval as much as possible (while still guaranteeing a root)
- Bisection can be used to continue to shrink the interval, whenever Newton/Secant would fail to stay inside the current interval
- Leverages the speed of Newton/Secant, while still guaranteeing convergence via Bisection
- Many/various strategies exist

# Function/Derivative Requirements

- All methods require evaluation of the function  $g$
- Newton also requires the derivative  $g'$  (as do mixed methods using Newton)

# Useful Derivatives

- $\frac{\partial}{\partial t} c^{q+1}(t) = \Delta c^q$ , since  $c^{q+1}(t) = c^q + t\Delta c^q$
- $\frac{\partial}{\partial t} F(c^{q+1}(t)) = J_F(c^{q+1}(t))\Delta c^q$  and  $\frac{\partial}{\partial t} F^T(c^{q+1}(t)) = (\Delta c^q)^T J_F^T(c^{q+1}(t))$ 
  - $\frac{\partial}{\partial t} F_i(c^{q+1}(t)) = (J_F)_i(c^{q+1}(t)) \Delta c^q$  where  $F_i(c^{q+1}(t))$  are the scalar row components of  $F(c^{q+1}(t))$
- Scalar  $\hat{f}(c^{q+1}(t))$  has system  $J_{\hat{f}}^T(c^{q+1}(t)) = 0$  for critical points
- $\frac{\partial}{\partial t} J_{\hat{f}}^T(c^{q+1}(t)) = H_{\hat{f}}^T(c^{q+1}(t))\Delta c^q$  and  $\frac{\partial}{\partial t} J_{\hat{f}}(c^{q+1}(t)) = (\Delta c^q)^T H_{\hat{f}}(c^{q+1}(t))$ 
  - $\frac{\partial}{\partial t} (J_{\hat{f}}^T)_i(c^{q+1}(t)) = (H_{\hat{f}}^T)_i(c^{q+1}(t))\Delta c^q$

# Recall: Line Search (Unit 14)

- Given the linearization error in  $F'(c^q)\Delta c^q = -F(c^q)$ , the resulting  $\Delta c^q$  can lead to a poor estimate for  $c^{q+1}$  via  $c^{q+1} = c^q + \Delta c^q$
- Thus,  $\Delta c^q$  is often (instead) used as a search direction, i.e.  $c^{q+1} = c^q + \alpha^q \Delta c^q$
- The parameterized line  $c^{q+1}(\alpha) = c^q + \alpha \Delta c^q$  is used as a 1D (input) domain
- Find  $\alpha^q$  such that  $F(c^{q+1}(\alpha^q)) = 0$  simultaneously for all equations
- Safe Set methods restrict  $\alpha$  in various ways, e.g.  $0 \leq \alpha \leq 1$

# Recall: Line Search (Unit 14)

- Since  $F$  is vector valued, consider  $g(\alpha) = F(c^{q+1}(\alpha))^T F(c^{q+1}(\alpha)) = 0$
- Since  $g(\alpha) \geq 0$ , solutions to  $F(c^{q+1}(\alpha)) = 0$  are minima of  $g(\alpha)$
- $g(\alpha)$  might be strictly positive (with no  $g(\alpha) = 0$ ), but minimizing  $g(\alpha)$  might still help to make progress towards a solution
- Option 1: find simultaneous roots of the vector valued  $F(c^{q+1}(\alpha)) = 0$
- Option 2: find roots of or minimize  $g(\alpha) = \frac{1}{2} F^T(c^{q+1}(\alpha)) F(c^{q+1}(\alpha))$

# Nonlinear Systems Problems

- Solve  $J_F(c^q)\Delta c^q = -F(c^q)$  for  $\Delta c^q$  and use  $c^{q+1}(t) = c^q + t\Delta c^q$  in  $F(c^{q+1}(t)) = 0$
- Option 1: For vector valued  $F(c^{q+1}(t))$ , find simultaneous (for all  $i$ ) **roots** for all the  $g_i(t) = F_i(c^{q+1}(t)) = 0$ 
  - Here,  $g'_i(t) = (J_F)_i(c^{q+1}(t))\Delta c^q$
- Option 2: Find **roots** of  $g(t) = \frac{1}{2}F^T(c^{q+1}(t))F(c^{q+1}(t)) = 0$ 
  - Here,  $g'(t) = \frac{1}{2}F^T(c^{q+1}(t))J_F(c^{q+1}(t))\Delta c^q + \frac{1}{2}(\Delta c^q)^T J_F^T(c^{q+1}(t))F(c^{q+1}(t))$
  - Both terms are scalars, so  $g'(t) = F^T(c^{q+1}(t))J_F(c^{q+1}(t))\Delta c^q$

# Recall: Optimization Problems (Unit 14)

- Minimize the scalar cost function  $\hat{f}(c)$  by finding the critical points where  $\nabla \hat{f}(c) = J_{\hat{f}}^T(c) = F(c) = 0$
- $F'(c^q)\Delta c^q = -F(c^q)$  gives the search direction, where  $F'(c) = J_F(c) = H_{\hat{f}}^T(c)$
- That is, solve  $H_{\hat{f}}^T(c^q)\Delta c^q = -J_{\hat{f}}^T(c^q)$  to find the search direction  $\Delta c^q$
- Option 1: find simultaneous roots of the vector valued  $J_{\hat{f}}^T(c^{q+1}(\alpha)) = 0$ , which are critical points of  $\hat{f}(c)$
- Option 2: find roots of or minimize  $g(\alpha) = \frac{1}{2}J_{\hat{f}}(c^{q+1}(\alpha))J_{\hat{f}}^T(c^{q+1}(\alpha))$  to find or make progress toward critical points of  $\hat{f}(c)$
- Option 3: minimize  $\hat{f}(c^{q+1}(\alpha))$  directly

# Optimization Problems

- Solve  $H_{\hat{f}}^T(c^q)\Delta c^q = -J_{\hat{f}}^T(c^q)$  for  $\Delta c^q$  and use  $c^{q+1}(t) = c^q + t\Delta c^q$  in  $J_{\hat{f}}^T(c^{q+1}(t)) = 0$
- Option 1: For vector valued  $J_{\hat{f}}^T(c^{q+1}(t))$ , find simultaneous (for all  $i$ ) **roots** for all the  $g_i(t) = (J_{\hat{f}}^T)_i(c^{q+1}(t)) = 0$  to find the critical points of  $\hat{f}(c)$ 
  - Here,  $g'_i(t) = (H_{\hat{f}}^T)_i(c^{q+1}(t))\Delta c^q$
- Option 2: Find **roots** of  $g(t) = \frac{1}{2}J_{\hat{f}}^T(c^{q+1}(t))J_{\hat{f}}(c^{q+1}(t)) = 0$  to find or make progress toward critical points of  $\hat{f}(c)$ 
  - Here,  $g'(t) = \frac{1}{2}J_{\hat{f}}^T(c^{q+1}(t))H_{\hat{f}}^T(c^{q+1}(t))\Delta c^q + \frac{1}{2}(\Delta c^q)^T H_{\hat{f}}(c^{q+1}(t))J_{\hat{f}}^T(c^{q+1}(t))$
  - Both terms are scalars, so  $g'(t) = J_{\hat{f}}^T(c^{q+1}(t))H_{\hat{f}}^T(c^{q+1}(t))\Delta c^q$
- Option 3: **Minimize**  $\hat{f}(c^{q+1}(t))$  directly (see **unit 16**)

# Unit 16

# 1D Optimization

# Part II Roadmap

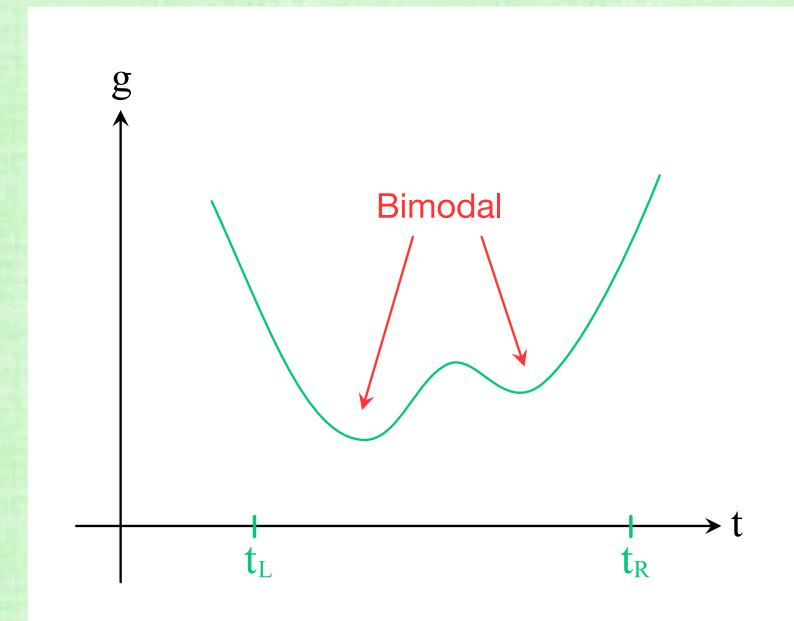
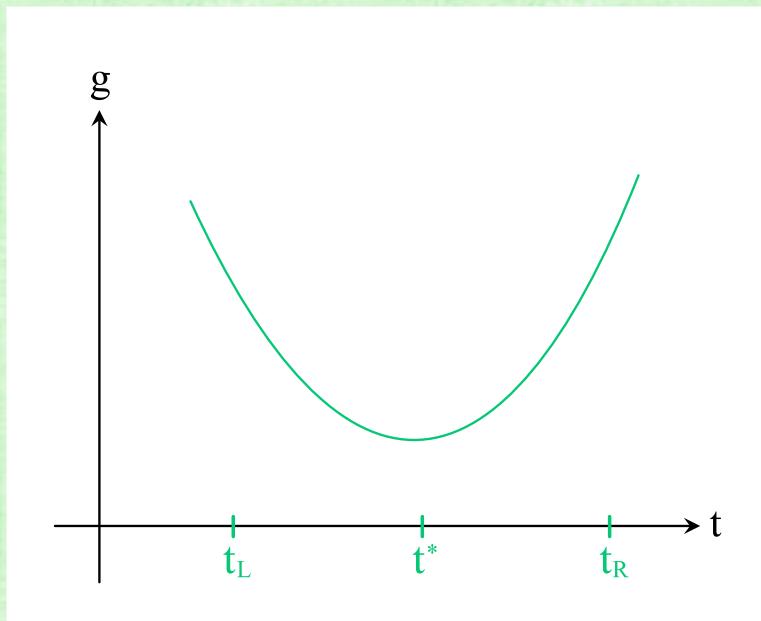
- Part I – Linear Algebra (units 1-12)  $Ac = b$ 
    - linearize
    - line search
  - Part II – Optimization (units 13-20)
    - (units 13-16) Optimization -> Nonlinear Equations -> 1D roots/minima
    - (units 17-18) Computing/Avoiding Derivatives
    - (unit 19) Hack 1.0: “I give up”  $H = I$  and  $J$  is mostly 0 (descent methods)
    - (unit 20) Hack 2.0: “It’s an ODE!?” (adaptive learning rate and momentum)
- 
- ```
graph TD; A["Part I – Linear Algebra (units 1-12)  $Ac = b$ "] -- "linearize" --> B["Part II – Optimization (units 13-20)"]; A -- "line search" --> B; C["(units 13-16) Optimization -> Nonlinear Equations -> 1D roots/minima"] --- D["(units 17-18) Computing/Avoiding Derivatives"]; D --- E["(unit 19) Hack 1.0: ‘I give up’  $H = I$  and  $J$  is mostly 0 (descent methods)"]; E --- F["(unit 20) Hack 2.0: ‘It’s an ODE!?’ (adaptive learning rate and momentum)"]; style C fill:#00f,color:#00f; style D fill:#00f,color:#00f; style E fill:#00f,color:#00f; style F fill:#00f,color:#00f; style B fill:#00f,color:#00f;
```

# Leveraging Root Finding (from unit 15)

- Relative extrema of  $g(t)$  occur at critical points where  $g'(t) = 0$
- Thus, can use root finding on  $g'$  to identify relative extrema
- Newton:  $t^{q+1} = t^q - \frac{g'(t^q)}{g''(t^q)}$  (dividing by  $g''$  is even worse than dividing by  $g'$ )
- Secant:  $t^{q+1} = t^q - g'(t^q) \frac{t^q - t^{q-1}}{g'(t^q) - g'(t^{q-1})}$  (could replace  $g'$  with approximations too)
- Bisection:  $g'(t_L)g'(t_R) < 0$  is the new condition
- Mixed Methods: mixing the above (as per unit 15)

# Unimodal

- Unimodal means one mode (bimodal means two modes)
- In 1D optimization, this means that the function has one relative minimum
- $g(t)$  is unimodal in  $[t_L, t_R]$  if and only if  $g$  is monotonically decreasing in  $[t_L, t^*]$  and monotonically increasing in  $[t^*, t_R]$

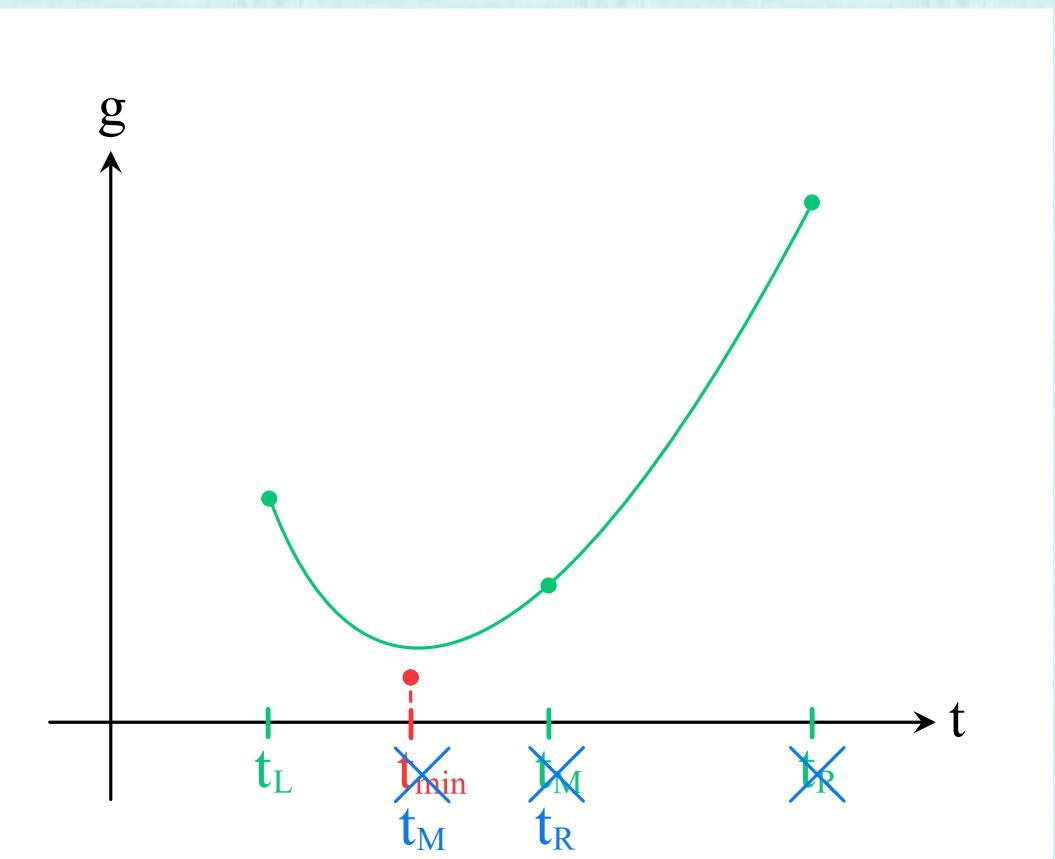
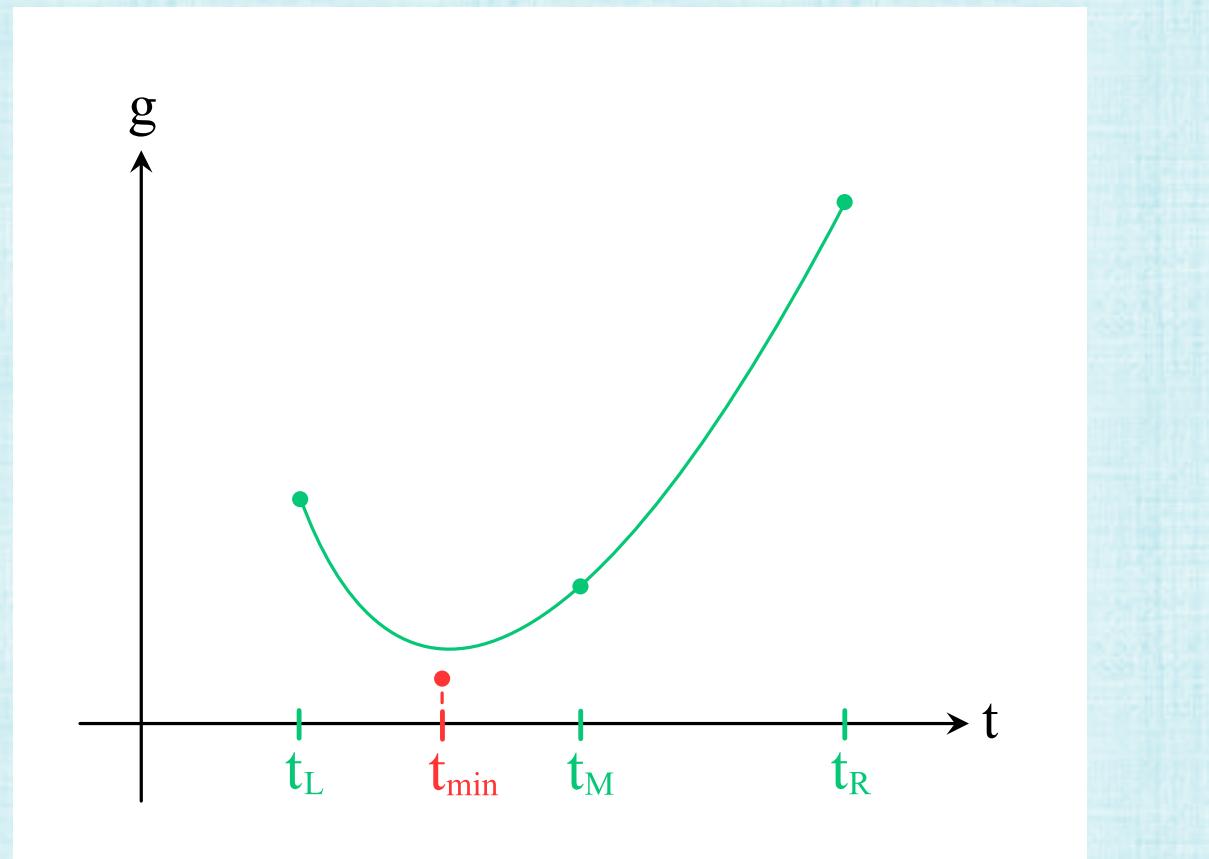


# Successive Parabolic Interpolation

- Motivated by Newton/Secant (which use lines to find candidates for roots), use parabolas to find candidates for minima
- Given interval  $[t_L, t_R]$  with midpoint  $t_M = \frac{t_L+t_R}{2}$ , create the unique parabola through  $t_L$ ,  $t_R$ , and  $t_M$ 
  - A unimodal  $g$  in  $[t_L, t_R]$  makes this parabola concave up
  - Let  $t_{min}$  be the point where the parabola takes on its minimum value
- Assume  $t_{min} < t_M$  (otherwise, simply rename them)
- If  $g(t_{min}) \leq g(t_M)$ , discard  $[t_M, t_R]$  which cannot contain the minimum
  - Then, set  $t_R = t_M$  and  $t_M = t_{min}$
- If  $g(t_{min}) \geq g(t_M)$ , discard  $[t_L, t_{min}]$  which cannot contain the minimum
  - Then, set  $t_L = t_{min}$  and  $t_M = t_M$  (no change)
- Superlinear convergence rate with  $p \approx 1.325$

# Successive Parabolic Interpolation

- When  $g(t_{min}) \leq g(t_M)$ , discard  $[t_M, t_R]$  and set  $t_R = t_M$  and  $t_M = t_{min}$



# Golden Section Search

- Unlike bisection (for root finding), 3 points is not enough to discard an interval during 1D minimization
- Successive parabolic interpolation demonstrated that 4 points is enough
- Let interval  $[t_L, t_R]$  have intermediate points with  $t_L < t_{M1} < t_{M2} < t_R$ 
  - If  $g$  is unimodal in  $[t_L, t_R]$ , one can safely discard either  $[t_L, t_{M1}]$  or  $[t_{M2}, t_R]$
- If  $g(t_{M1}) \leq g(t_{M2})$ , discard  $[t_{M2}, t_R]$  which cannot contain the minimum
- If  $g(t_{M1}) \geq g(t_{M2})$ , discard  $[t_L, t_{M1}]$  which cannot contain the minimum

# Golden Section Search (magic number)

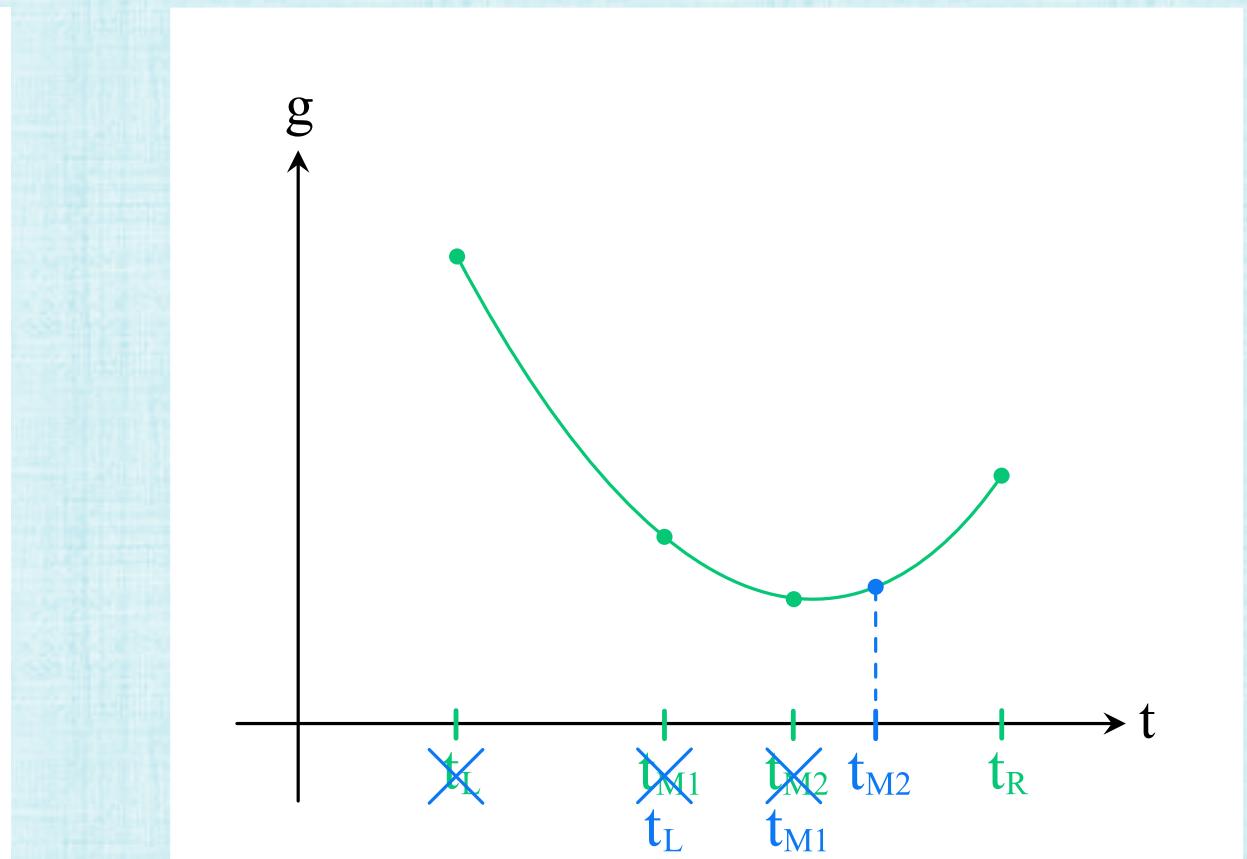
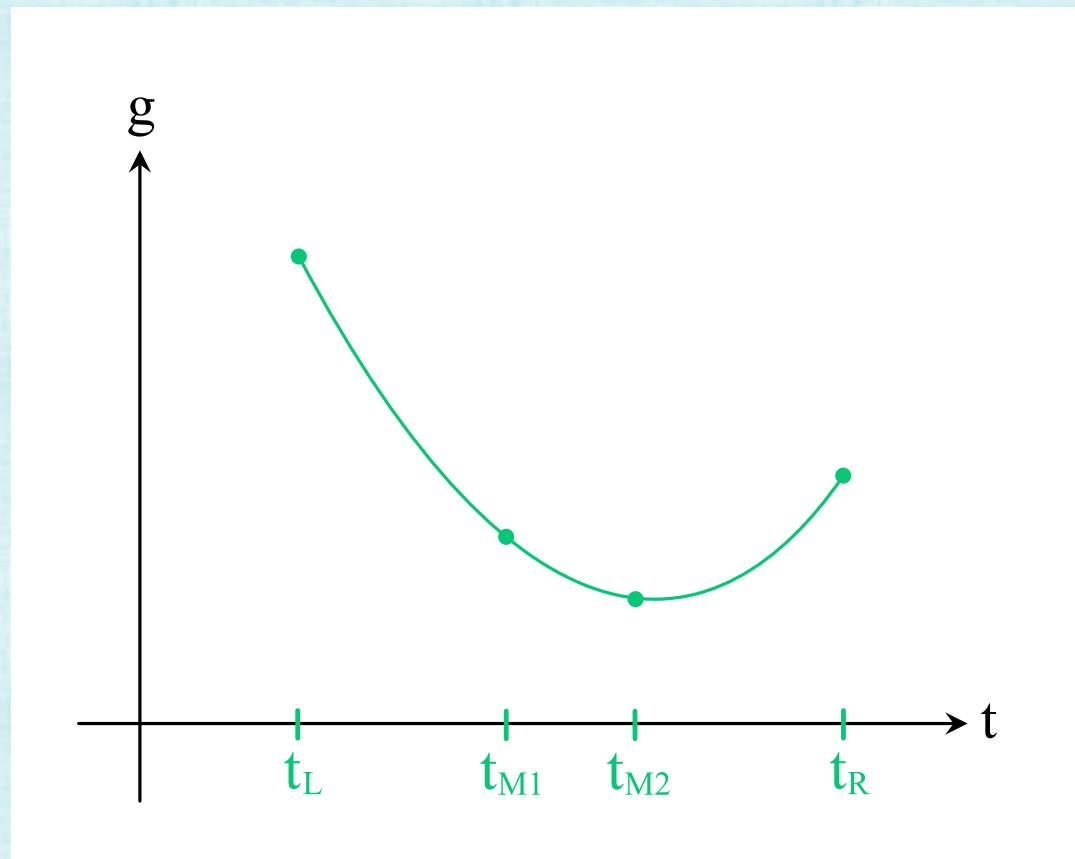
- After discarding an interval, either  $t_{M1}$  or  $t_{M2}$  becomes an endpoint, and keeping the other as an interior point (efficiently) reduces evaluations of  $g$
- Let  $\delta = t_R - t_L$  be the interval size and  $\lambda \in (0, .5)$  be the fraction inward of  $t_{M1}$
- Then  $t_{M1} = t_L + \lambda\delta$ , and symmetric placement gives  $t_{M2} = (t_L + \delta) - \lambda\delta$
- Discard the left interval (discarding the right gives the same math) to obtain  $t_L^{new} = t_{M1}$  and  $\delta^{new} = (1 - \lambda)\delta$
- Then  $t_{M2} = (t_L^{new} - \lambda\delta + \delta) - \lambda\delta = t_L^{new} + (1 - 2\lambda)\frac{\delta^{new}}{1-\lambda}$  can be designated as either  $t_{M1}^{new}$  or  $t_{M2}^{new}$  if one has  $\frac{1-2\lambda}{1-\lambda}$  equal to either  $\lambda$  or  $1 - \lambda$
- Of the four solutions, only one has  $\lambda \in (0, .5)$ :  $\lambda = \frac{3-\sqrt{5}}{2}$  with  $t_{M2}$  becoming  $t_{M1}^{new}$

# Golden Section Search

- Rewrite:  $t_{M1} = (1 - \lambda)t_L + \lambda t_R$  and  $t_{M2} = \lambda t_L + (1 - \lambda)t_R$
- Switch the parameter to the more typical  $\tau = 1 - \lambda = \frac{\sqrt{5}-1}{2}$  (since  $\lambda = \frac{3-\sqrt{5}}{2}$ )
- Then,  $t_{M1} = \tau t_L + (1 - \tau)t_R$  and  $t_{M2} = (1 - \tau)t_L + \tau t_R$
- If  $g(t_{M1}) \leq g(t_{M2})$ , discard  $[t_{M2}, t_R]$ , set  $t_R = t_{M2}$ ,  $t_{M2} = t_{M1}$ , and recompute  $t_{M1}$
- If  $g(t_{M1}) \geq g(t_{M2})$ , discard  $[t_L, t_{M1}]$ , set  $t_L = t_{M1}$ ,  $t_{M1} = t_{M2}$ , and recompute  $t_{M2}$
- Stop when the interval size is small (as usual)
- Linear convergence rate ( $p = 1$ ) with  $C = \frac{1-\lambda}{1} = \tau \approx .618$

# Golden Section Search

- If  $g(t_{M1}) \geq g(t_{M2})$ , discard  $[t_L, t_{M1}]$ , set  $t_L = t_{M1}$ ,  $t_{M1} = t_{M2}$ , recompute  $t_{M2}$



# Mixed Methods

- Given a unimodal interval  $[t_L, t_R]$
- Iterate with Successive Parabolic Interpolation as long as the iterates stay inside the interval
  - When iteration attempts to leave the interval, use prior iterates to shrink the interval as much as possible (while still containing the minima)
- Golden Section Search can be used to continue to shrink the interval, whenever Successive Parabolic Interpolation would fail to stay inside the current interval
- Leverages the speed of Successive Parabolic Interpolation, while still guaranteeing convergence via Golden Section Search
- Many/various strategies exist

# Function/Derivative Requirements

- All methods require function evaluation  $g$
- Root finding approaches differentiate  $g$  and solve  $g'(t) = 0$  to identify critical points
  - All root finding methods require function evaluation, which is  $g'$  here
  - **Newton** (and mixed methods using Newton) requires the derivative of the function, which is  $g''$  here (since the function is  $g'$ )

# Recall: Useful Derivatives (unit 15)

- $\frac{\partial}{\partial t} c^{q+1}(t) = \Delta c^q$ , since  $c^{q+1}(t) = c^q + t\Delta c^q$
- $\frac{\partial}{\partial t} F(c^{q+1}(t)) = J_F(c^{q+1}(t))\Delta c^q$  and  $\frac{\partial}{\partial t} F^T(c^{q+1}(t)) = (\Delta c^q)^T J_F^T(c^{q+1}(t))$ 
  - $\frac{\partial}{\partial t} F_i(c^{q+1}(t)) = (J_F)_i(c^{q+1}(t)) \Delta c^q$  where  $F_i(c^{q+1}(t))$  are the scalar row components of  $F(c^{q+1}(t))$
- Scalar  $\hat{f}(c^{q+1}(t))$  has system  $J_{\hat{f}}^T(c^{q+1}(t)) = 0$  for critical points
- $\frac{\partial}{\partial t} J_{\hat{f}}^T(c^{q+1}(t)) = H_{\hat{f}}^T(c^{q+1}(t))\Delta c^q$  and  $\frac{\partial}{\partial t} J_{\hat{f}}(c^{q+1}(t)) = (\Delta c^q)^T H_{\hat{f}}(c^{q+1}(t))$ 
  - $\frac{\partial}{\partial t} (J_{\hat{f}}^T)_i(c^{q+1}(t)) = (H_{\hat{f}}^T)_i(c^{q+1}(t))\Delta c^q$

# Useful Derivatives (Continued)

- $\frac{\partial}{\partial t} J_F(c^{q+1}(t)) = (\Delta c^q)^T H_F(c^{q+1}(t))$ 
  - $H_F$  is a rank 3 tensor of all 2<sup>nd</sup> derivatives of  $F$
  - $\frac{\partial}{\partial t} (J_F)_i(c^{q+1}(t)) = (\Delta c^q)^T (H_F)_i(c^{q+1}(t))$
- $\frac{\partial}{\partial t} H_{\hat{f}}^T(c^{q+1}(t)) = (\Delta c^q)^T OMG_{\hat{f}}^T(c^{q+1}(t))$ 
  - $OMG_{\hat{f}}^T$  is a rank 3 tensor of all 3<sup>rd</sup> derivatives of  $\hat{f}$
  - $\frac{\partial}{\partial t} \left( H_{\hat{f}}^T \right)_i(c^{q+1}(t)) = (\Delta c^q)^T \left( OMG_{\hat{f}}^T \right)_i(c^{q+1}(t))$

# Recall: Nonlinear Systems Problems (unit 15)

- Solve  $J_F(c^q)\Delta c^q = -F(c^q)$  for  $\Delta c^q$  and use  $c^{q+1}(t) = c^q + t\Delta c^q$  in  $F(c^{q+1}(t)) = 0$
- Option 1: For vector valued  $F(c^{q+1}(t))$ , find simultaneous (for all  $i$ ) **roots** for all the  $g_i(t) = F_i(c^{q+1}(t)) = 0$ 
  - Here,  $g'_i(t) = (J_F)_i(c^{q+1}(t))\Delta c^q$
- Option 2: Find **roots** of  $g(t) = \frac{1}{2}F^T(c^{q+1}(t))F(c^{q+1}(t)) = 0$ 
  - Here,  $g'(t) = \frac{1}{2}F^T(c^{q+1}(t))J_F(c^{q+1}(t))\Delta c^q + \frac{1}{2}(\Delta c^q)^T J_F^T(c^{q+1}(t))F(c^{q+1}(t))$
  - Both terms are scalars, so  $g'(t) = F^T(c^{q+1}(t))J_F(c^{q+1}(t))\Delta c^q$

# Nonlinear Systems Problems

- Solve  $J_F(c^q)\Delta c^q = -F(c^q)$  for  $\Delta c^q$  and use  $c^{q+1}(t) = c^q + t\Delta c^q$  in  $F(c^{q+1}(t)) = 0$
- Option 1: For vector valued  $F(c^{q+1}(t))$ , find simultaneous (for all  $i$ ) **minima** for all the  $g_i(t) = F_i(c^{q+1}(t))$  aiming for roots where all  $F_i(c^{q+1}(t)) = 0$ 
  - Here,  $g'_i(t) = (J_F)_i(c^{q+1}(t))\Delta c^q$  and  $g''_i(t) = (\Delta c^q)^T (H_F)_i(c^{q+1}(t)) \Delta c^q$
- Option 2: **Minimize**  $g(t) = \frac{1}{2} F^T(c^{q+1}(t)) F(c^{q+1}(t))$  aiming for its roots
  - Here,  $g'(t) = F^T(c^{q+1}(t)) J_F(c^{q+1}(t)) \Delta c^q$
  - $g''(t) = F^T(c^{q+1}(t)) (\Delta c^q)^T H_F(c^{q+1}(t)) \Delta c^q + (\Delta c^q)^T J_F^T(c^{q+1}(t)) J_F(c^{q+1}(t)) \Delta c^q$

# Recall: Optimization Problems (unit 15)

- Solve  $H_{\hat{f}}^T(c^q)\Delta c^q = -J_{\hat{f}}^T(c^q)$  for  $\Delta c^q$  and use  $c^{q+1}(t) = c^q + t\Delta c^q$  in  $J_{\hat{f}}^T(c^{q+1}(t)) = 0$
- Option 1: For vector valued  $J_{\hat{f}}^T(c^{q+1}(t))$ , find simultaneous (for all  $i$ ) **roots** for all the  $g_i(t) = (J_{\hat{f}}^T)_i(c^{q+1}(t)) = 0$  to find the critical points of  $\hat{f}(c)$ 
  - Here,  $g'_i(t) = (H_{\hat{f}}^T)_i(c^{q+1}(t))\Delta c^q$
- Option 2: Find **roots** of  $g(t) = \frac{1}{2}J_{\hat{f}}^T(c^{q+1}(t))J_{\hat{f}}(c^{q+1}(t)) = 0$  to find or make progress toward critical points of  $\hat{f}(c)$ 
  - Here,  $g'(t) = \frac{1}{2}J_{\hat{f}}^T(c^{q+1}(t))H_{\hat{f}}^T(c^{q+1}(t))\Delta c^q + \frac{1}{2}(\Delta c^q)^T H_{\hat{f}}(c^{q+1}(t))J_{\hat{f}}^T(c^{q+1}(t))$
  - Both terms are scalars, so  $g'(t) = J_{\hat{f}}^T(c^{q+1}(t))H_{\hat{f}}^T(c^{q+1}(t))\Delta c^q$
- Option 3: **Minimize**  $\hat{f}(c^{q+1}(t))$  directly (see **unit 16**)

# Optimization Problems

- Solve  $H_{\hat{f}}^T(c^q)\Delta c^q = -J_{\hat{f}}^T(c^q)$  for  $\Delta c^q$  and use  $c^{q+1}(t) = c^q + t\Delta c^q$  in  $J_{\hat{f}}^T(c^{q+1}(t)) = 0$
- Option 1: For vector valued  $J_{\hat{f}}^T(c^{q+1}(t))$ , find simultaneous (for all  $i$ ) **minima** for all the  $g_i(t) = (J_{\hat{f}}^T)_i(c^{q+1}(t))$  aiming for the roots which are critical points of  $\hat{f}(c)$ 
  - Here,  $g'_i(t) = (H_{\hat{f}}^T)_i(c^{q+1}(t))\Delta c^q$  and  $g''_i(t) = (\Delta c^q)^T \left( OMG_{\hat{f}}^T \right)_i (c^{q+1}(t)) \Delta c^q$
- Option 2: **Minimize**  $g(t) = \frac{1}{2} J_{\hat{f}}^T(c^{q+1}(t)) J_{\hat{f}}(c^{q+1}(t))$  aiming for the roots which are critical points of  $\hat{f}(c)$ 
  - Here,  $g'(t) = J_{\hat{f}}(c^{q+1}(t)) H_{\hat{f}}^T(c^{q+1}(t)) \Delta c^q$
  - $g''(t) = J_{\hat{f}}(c^{q+1}(t)) (\Delta c^q)^T OMG_{\hat{f}}^T(c^{q+1}(t)) \Delta c^q + (\Delta c^q)^T H_{\hat{f}}(c^{q+1}(t)) H_{\hat{f}}^T(c^{q+1}(t)) \Delta c^q$
- Option 3: **Minimize**  $\hat{f}(c^{q+1}(t))$  directly
  - $g'(t) = J_{\hat{f}}(c^{q+1}(t)) \Delta c^q$  and  $g''(t) = (\Delta c^q)^T H_{\hat{f}}(c^{q+1}(t)) \Delta c^q$

# Unit 17

# Computing Derivatives

# Part II Roadmap

- Part I – Linear Algebra (units 1-12)  $Ac = b$ 
    - linearize
    - line search
  - Part II – Optimization (units 13-20)
    - (units 13-16) Optimization -> Nonlinear Equations -> 1D roots/minima
    - (units 17-18) Computing/Avoiding Derivatives
    - (unit 19) Hack 1.0: “I give up”  $H = I$  and  $J$  is mostly 0 (descent methods)
    - (unit 20) Hack 2.0: “It’s an ODE!?” (adaptive learning rate and momentum)
- 
- ```
graph TD; A["Part I – Linear Algebra (units 1-12)  $Ac = b$ "] -- "red arrow" --> B["linearize"]; A -- "red arrow" --> C["line search"]; D["Part II – Optimization (units 13-20)"] --> E["(units 13-16) Optimization -> Nonlinear Equations -> 1D roots/minima"]; D --> F["(units 17-18) Computing/Avoiding Derivatives"]; D --> G["(unit 19) Hack 1.0: ‘I give up’  $H = I$  and  $J$  is mostly 0 (descent methods)"]; D --> H["(unit 20) Hack 2.0: ‘It’s an ODE!?’ (adaptive learning rate and momentum)"]
```

# Smoothness

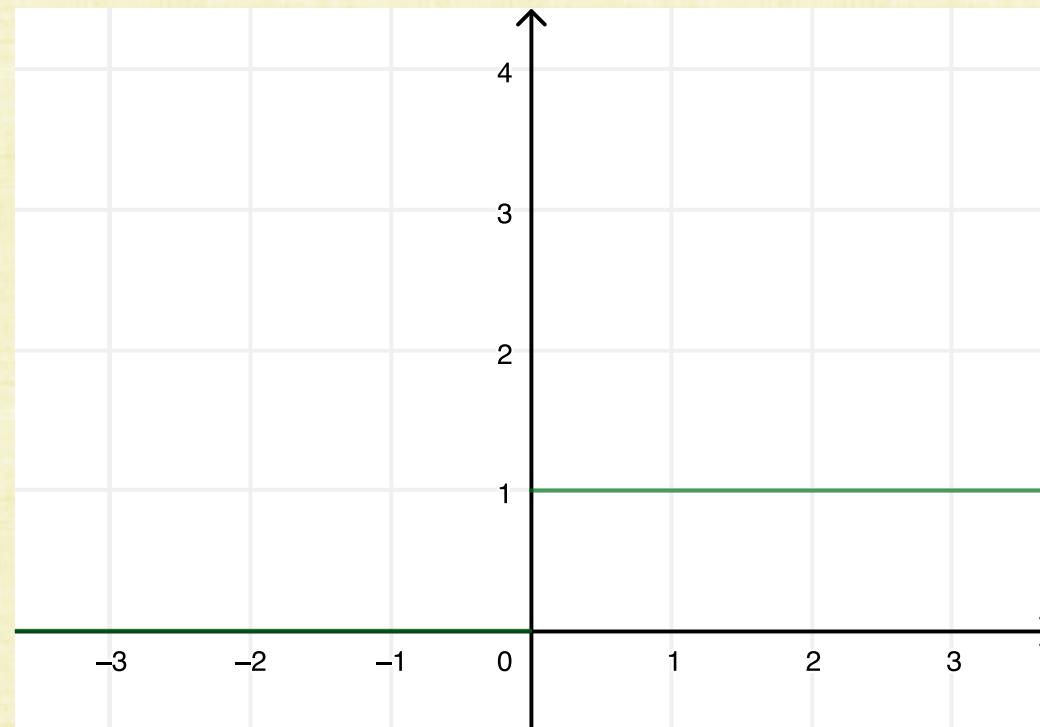
- Discontinuous functions cannot be differentiated
  - Even methods that don't require derivatives struggle when functions are discontinuous
- Continuous functions may have kinks (discontinuities in derivatives)
  - Discontinuous derivatives can cause methods that depend on derivatives to fail, since function behavior cannot be adequately predicted from one side of the kink to the other
- Typically, functions need to be “smooth enough”, which has varying meaning depending on the approach
- Specialty approaches exist for special classes of functions, e.g. linear algebra, linear programming, convex optimization, second order cone program (SOCP), etc.
  - Nonlinear Systems/Optimization are more difficult, and best practices/techniques often do not exist

# Biological Neurons (towards “real” AI)

- Aim to mimic human biological neural networks and learning
- Biological neurons are “all or none”, which motivates similar strategies in artificial neural networks
  - This leads to a discontinuous function with an identically zero derivative everywhere else
  - Disastrous for optimization!
- Biological neurons fire with increased frequency for stronger signals
  - This leads to a piecewise constant and discontinuous derivative
  - Problematic for optimization!
- Smoothing allows optimization to work, i.e. to minimize the loss to find the parameters/coefficients (for the network architecture)

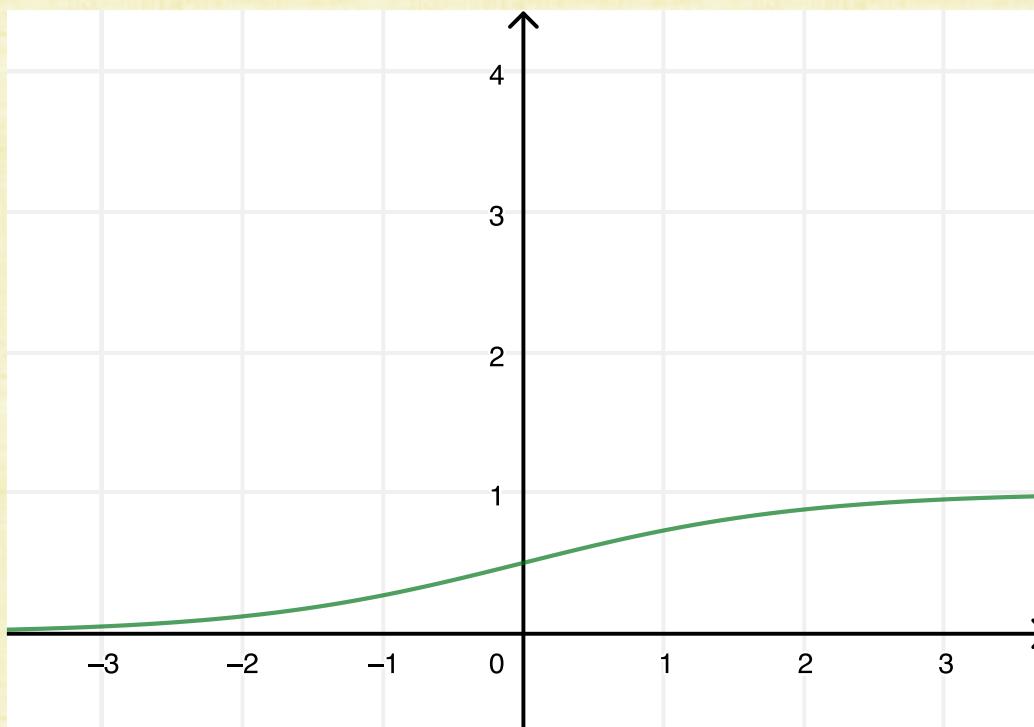
# Heaviside Function

- $H(x) = 1$  for  $x \geq 0$ , and  $H(x) = 0$  for  $x < 0$
- Motivated by biological neurons being “all or none”, but has a discontinuity at 0 and an identically zero derivative everywhere else



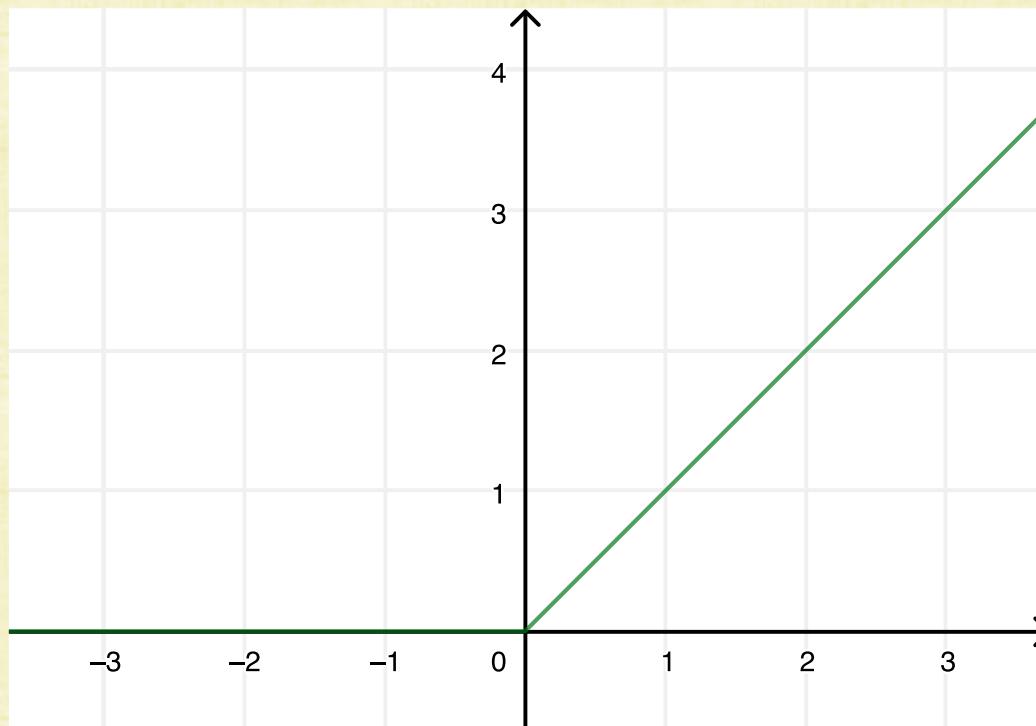
# Sigmoid Function (an example)

- Any smoothed Heaviside function, e.g.  $S(x) = \frac{1}{1+e^{-x}}$  (there are many options)
- Continuous and monotonically increasing, although the derivative is close to zero away from  $x = 0$



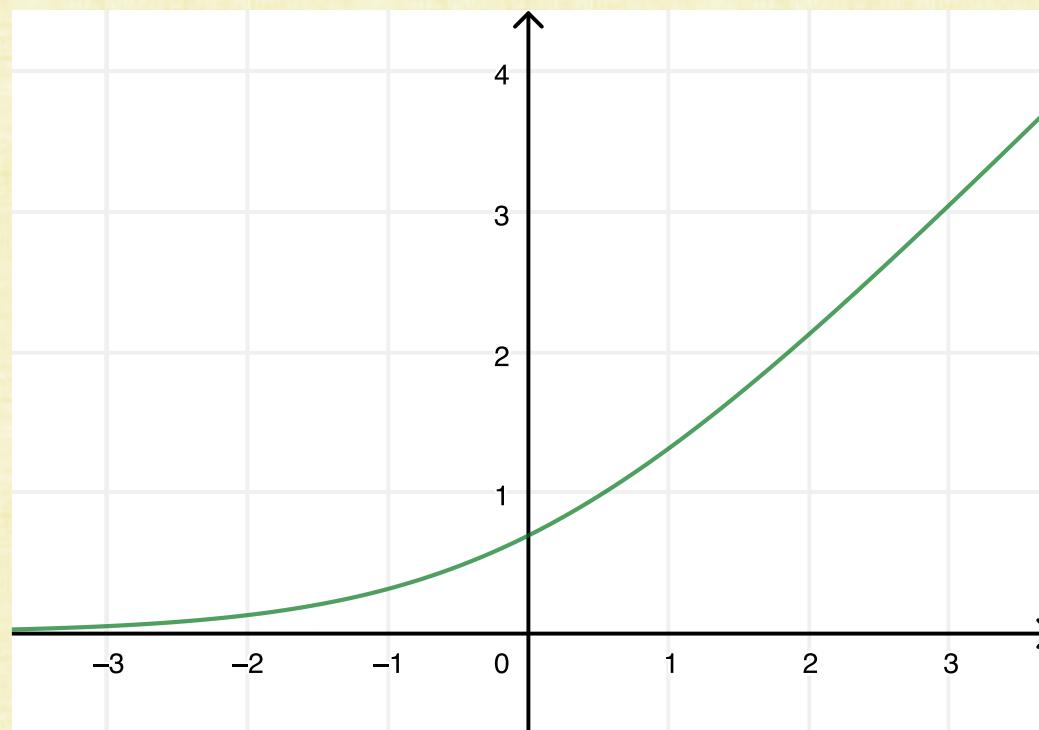
# Rectifier Functions (an example)

- $R(x) = \max(x, 0)$  or similar functions which are continuous and have increasing values
- Motivated by biological neurons firing with increased frequency for stronger signals
- Piecewise constant and discontinuous derivative causes issues with optimization



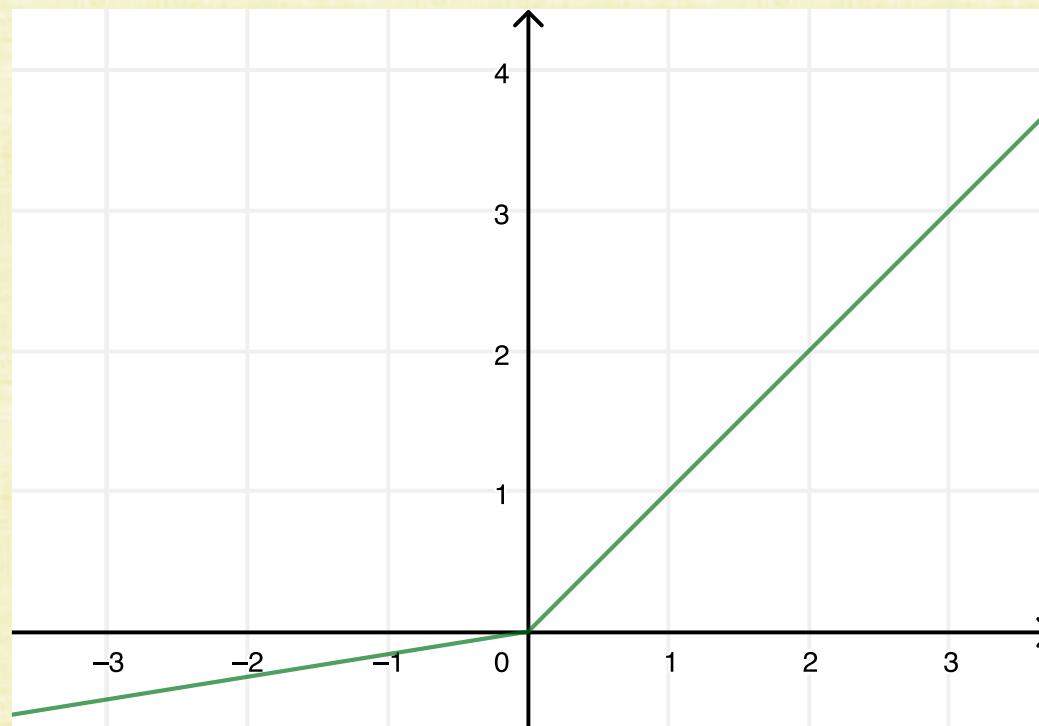
# Softplus Function

- Softplus function  $SP(x) = \log(1 + e^x)$  smooths the discontinuous derivative typical of rectifier functions



# Leaky Rectifier Function

- Modifies the negative part of a rectifier function to also have a positive slope instead of being set to zero
- Can be smoothed (as well)



# Arg/Soft Max

- Arg Max returns 1 for the largest argument and 0 for the other arguments
- E.g. (.99,1) → (0,1), (1,.99) → (1,0), etc.
- Highly discontinuous!
- Soft Max is a smoothed out version, e.g.  $(x_1, x_2) \rightarrow \left( \frac{e^{x_1}}{e^{x_1} + e^{x_2}}, \frac{e^{x_2}}{e^{x_1} + e^{x_2}} \right)$
- This is a smooth function of the arguments, differentiable, etc.
- Variants/weightings exist to make it closer/further from Arg Max (while preserving differentiability)

# Example: Binary Classification

- Training data  $(x_i, y_i)$  where the  $y_i = \pm 1$  are binary class labels
- Find plane  $n^T(x - x_o) = 0$  that separates the data between the two class labels ( $n$  is the unit normal and  $x_o$  is a point on the plane)
- The closest  $x_i$  on each side of the plane are called support vectors
- If the plane is equidistant between the support vectors, then they lie on parallel planes:  $n^T(x - x_o) = \pm\epsilon$  (where  $\epsilon$  is the margin)
- Dividing by  $\epsilon$  to normalize gives  $c^T(x - x_o) = \pm 1$  where  $c$  is in the normal direction (not unit length); then, maximizing the margin  $\epsilon$  is equivalent to minimizing  $\|c\|_2$

# Example: Binary Classification

- Minimize  $\hat{f}(c) = \frac{1}{2} c^T c$  subject to inequality constraints:
  - $c^T(x_i - x_o) \geq 1$  when  $y_i = 1$ , and  $c^T(x_i - x_o) \leq -1$  when  $y_i = -1$
  - Can combine into  $y_i c^T(x_i - x_o) \geq 1$  for every data point
  - Alternatively,  $y_i(c^T x_i - b) \geq 1$  with scalar unknown  $b = c^T x_o$
- New data will be inferred/classified based on the sign of  $c^T x_{new} - b$
- When approached via unconstrained optimization, the Heaviside function can be used to incorporate the constraints into the cost function
  - Subsequently smoothing the Heaviside function is called soft-margin

# (Inequality) Constrained Optimization

- Minimize  $\hat{f}(c)$  subject to  $\hat{g}(c) \geq 0$  (or  $\hat{g}(c) > 0$ )
- Create a penalty term  $-H(-\hat{g}_i(c))\hat{g}_i(c)$  which is nonzero only when  $\hat{g}_i(c) < 0$ 
  - This penalty term is minimized by forcing negative  $\hat{g}_i(c)$  towards zero (as desired)
- Given a diagonal matrix  $D$  of (positive) weights indicating the relative importance of various constraints, unconstrained optimization can be used to minimize
$$\hat{f}(c) - \sum_i H(-\hat{e}_i^T D \hat{g}(c)) \hat{e}_i^T D \hat{g}(c)$$
  - This requires differentiating the non-smooth Heaviside function
  - Smoothing the Heaviside function makes the modified cost function differentiable

# Symbolic Differentiation

- When a function is known in closed form, it can be differentiated by hand
- Software packages such as Mathematica can aid in symbolic differentiation (and subsequent simplification)
- Some benefits of knowing the closed form derivative:
  - Provides a better understanding of the underlying problem
  - Enables well thought out smoothing/regularization
  - Allows one to implement more efficient code
  - Subsequently allows access to more accurate higher derivatives
  - Some of the aforementioned benefits enable the use of better solvers
  - Helps to write/maintain code with less bugs
  - Etc.

# Example

- Suppose a code has the following functions:
  - $f(t) = t^2 - 4$  with  $f'(t) = 2t$ , and  $g(t) = t - 2$  with  $g'(t) = 1$
- Suppose another part of the code combines these functions:
  - $h(t) = \frac{f(t)}{g(t)}$  with  $h'(t) = \frac{g(t)f'(t)-f(t)g'(t)}{(g(t))^2}$
- Then  $h(2) = \frac{f(2)}{g(2)} = \frac{0}{0}$  and  $h'(2) = \frac{g(2)f'(2)-f(2)g'(2)}{(g(2))^2} = \frac{0 \cdot 4 - 0 \cdot 1}{0^2}$
- Adding a small  $\epsilon > 0$  to the denominators to avoid division by zero gives  $h(2) = 0$  and  $h'(2) = 0$
- Adding a small  $\epsilon > 0$  to denominators is often done whenever they are small, making  $h(t) \approx 0$  and  $h'(t) \approx 0$  for  $t \approx 2$  as well
- Of course,  $h(t) = t + 2$  is a straight line with  $h(2) = 4$  and  $h'(t) = 1$  everywhere

# Symbolic Differentiation of Code

- Sometimes a function is not analytically known and/or merely represents the output of some source code
- But, **parts** of the code may have known derivatives, and those known derivatives can be utilized/leveraged via the mathematical rules for differentiation
- Moreover, when parts of the code are always used consecutively, they can be merged; subsequently, merged code with known derivatives in each part can often have the derivative treatment simplified for accuracy/robustness/efficiency

# Differentiate the Right Thing

- Consider an iterative solver (e.g. CG, Minres, etc.) that solves  $Ac = b$  to find  $c$  given  $b$
- Sometimes the code is enormous, complicated, confusing, a black box, etc. (basically impenetrable)
- It is tempting to consider some of the code bases that claim to differentiate such chunks of code
  - Sometimes these approaches work, and the answers are reasonable
  - But, it is often difficult to know whether or not computational inaccuracies (as discussed in this class) are having an adverse effect on such a black box approach
- Alternatively, when invertible:  $c = A^{-1}b$  and  $\frac{\partial c_k}{\partial b_i} = \tilde{a}_{ki}$  where  $\tilde{a}_{ki}$  is an entry in  $A^{-1}$ 
  - A similar approach can be taken for  $A^+$ , which can be estimated robustly via PCA, the Power Method, etc.
- The derivative is independent of the iterative solver (CG, Minres, etc.) and the errors that might accumulate within the iterative solver due to poor conditioning
  - More recently, this sort of approach is being referred to as an **implicit layer**

# Used Car Sales

- Beware of the claim: it is good to be able to use something without understanding it
- The claim is often true, and many of us enjoy driving our cars without understanding much of what is under the hood
- However, those who design cars, manufacture cars, repair cars, etc. benefit greatly from understanding as much as possible about them, and the rest of us benefit enormously from their expertise
- Though, admittedly, there are those in the car business, such as those who sell used cars, who legitimately don't require any real knowledge/expertise
- The question is: **what kind of computer scientist do you want to be?**

# Oversimplified Thinking

- Beware of claims that drastically oversimplify
- E.g., some say that code is very simple and merely consists of simple operations like add/subtract/multiply/divide that are easily differentiated
- However, in reality, even the simple  $z = x + y$  has subtleties that can matter
  - E.g. the computer actually executes  $z = \text{round}(x + y)$
- Too many claim that issues they have not carefully considered don't matter in practice; meanwhile, many state-of-the-art practices in ML/DL are not well understood (leaving one to question this claim)

# Finite Differences

- Derivatives can be approximated by various formulas, similar to how the Secant method was derived from Newton's method
- Given a small perturbation  $h > 0$ , Taylor expansions can be manipulated to write:
  - Forward Difference:  $g'(t) = \frac{g(t+h)-g(t)}{h} + O(h)$ , 1<sup>st</sup> order accurate
  - Backward Difference:  $g'(t) = \frac{g(t)-g(t-h)}{h} + O(h)$ , 1<sup>st</sup> order accurate
  - Central Difference:  $g'(t) = \frac{g(t+h)-g(t-h)}{2h} + O(h^2)$ , 2<sup>nd</sup> order accurate
  - Second Derivative:  $g''(t) = \frac{g(t+h)-2g(t)+g(t-h)}{h^2} + O(h^2)$ , 2<sup>nd</sup> order accurate
- These approximations can be evaluated even when  $g(t)$  is not known precisely, but merely represents the output of some code with input  $t$

# Finite Differences (Drawbacks)

- Finite Differences only give an approximation to the derivative, and contain truncation errors related to the perturbation size  $h$
- One has to reason about the effects that truncation error (and the size of  $h$ ) have on other aspects of the code
- If the code is very long and complex, the overall effects of truncation errors may be unclear
- Still, finite difference methods have had a broad positive impact in computational science!

# Automatic Differentiation

- In machine learning, this is often referred to as Back Propagation
- For every (potentially vector valued) function  $F(c_{input})$  written into the code, an analytically correct companion function for the Jacobian matrix  $\frac{\partial F}{\partial c}(c_{input})$  is also written
- Then when evaluating  $F(c_{input})$ , one can also evaluate  $\frac{\partial F}{\partial c}(c_{input})$ 
  - Of course,  $\frac{\partial F}{\partial c}(c_{input})$  contains roundoff errors based on machine precision (and conditioning, etc.)
  - But it does not contain the much larger truncation errors present in finite differencing
- Code can be considered in chunks, which combine together various functions via arithmetic/compositional rules
  - Analytic differentiation has its own set of rules (linearity, product rule, quotient rule, chain rule, etc.) that can be used to assemble the derivative (evaluated at  $c_{input}$ ) for the code chunk
- Roundoff errors will accumulate, of course, and the resulting error has the potential to be catastrophic (this is typically even worse for the much larger truncation errors)

# Second Derivatives

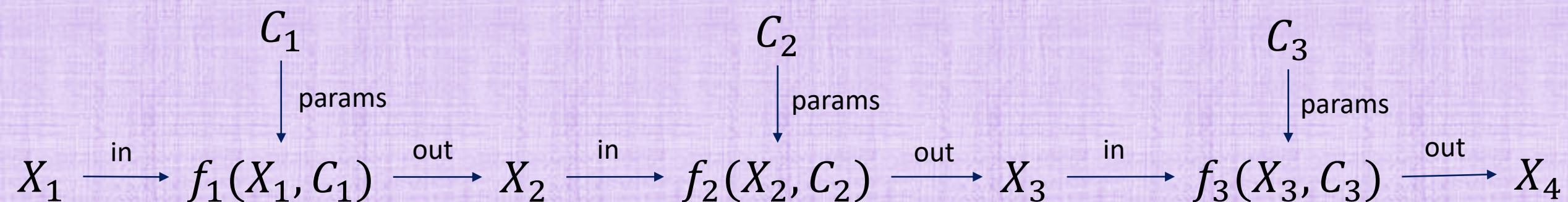
- If  $c_{input}$  is size  $n$  and  $F(c_{input})$  is size  $m$ , the Jacobian matrix  $\frac{\partial F}{\partial c}(c_{input})$  is size  $m \times n$
- The Hessian of second derivatives is size  $m \times n \times n$ 
  - Recall:  $m = 1$  for optimization, i.e. for  $\hat{f}(c_{input})$
- Writing automatic differentiation functions for all possible second derivatives can be difficult/tedious
- Storing Hessians for all second derivatives can be unwieldy/intractable
- Roundoff error accumulation can be an even bigger problem for second derivatives, and the resulting errors are typically even more likely to lead to adverse effects
- Additional smoothness is required for second derivatives
- Some of these issues are problems for any method that considers second derivatives (not specific to an automatic differentiation approach)

# Dropout

- One way to combat overfitting is to train several different network architectures on the same data, inference them all, and average the result
  - This is costly, especially if there are many networks
- Dropout is a “hacky” approach to achieving a network function averaged over multiple network architectures
  - Though Google did patent it
- The idea is to simply ignore parts of the code with some probability when training the network, mimicking a perturbed network architecture
- Although this can be seen as computing correct derivatives on perturbed functions/architectures, it can also equivalently be seen as adding uncertainty to the derivative computation
- That is, instead of regularization via model averaging, it can be seen as creating a network robust to errors in derivative estimation

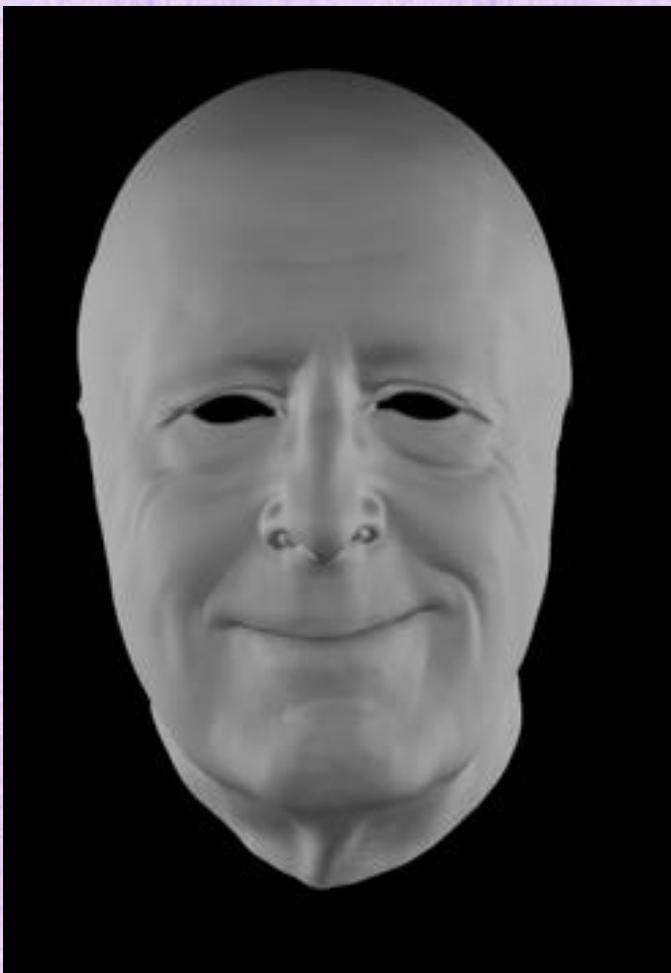
# Function Layers

- Many complex processes work in a pipeline with many function layers
- Each layer completes a tasks on its inputs  $X_j$  to create outputs  $X_{j+1}$
- Each layer may depend on parameters  $C_j$
- There may be a known/desired output  $X_{target}$  to compare the final result to



$$\hat{f}(X_4) = \|X_4 - X_{target}\|$$

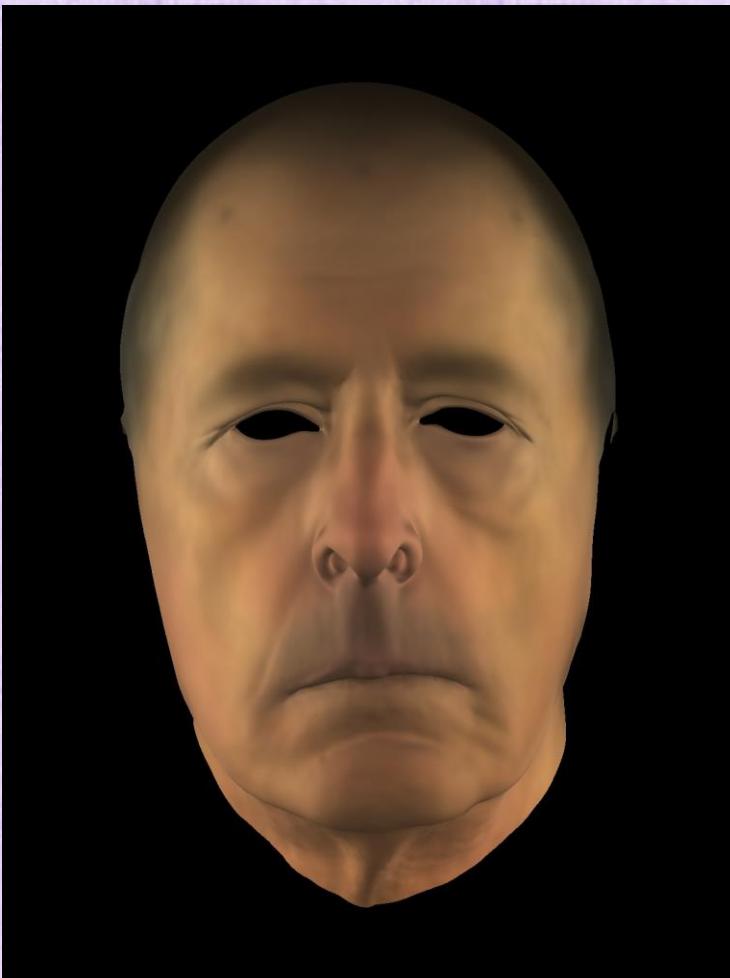
# Function Layers (Example)



## LAYER 1

- Input: animation controls
- Function: linear blend shapes, nonlinear skinning, quasistatic physics simulation, etc. to deform a face
- Parameters: lots of hand tuned or known parameters including shape libraries, etc.
- Output: 3D vertex positions of a triangle mesh

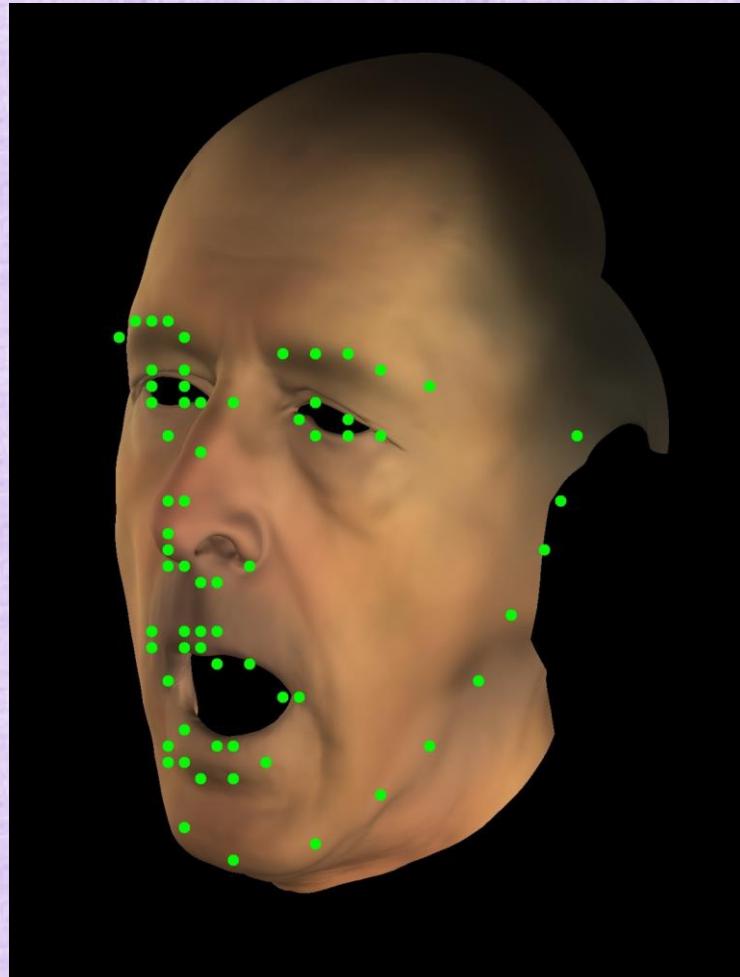
# Function Layers (Example)



## LAYER 2

- Input: 3D vertex positions of a triangle mesh
- Function: scanline renderer or ray tracer
- Parameters: lots of hand tuned or known parameters for material models, lighting and shading, textures, etc.
- Output: RGB colors for pixels (a 2D image)

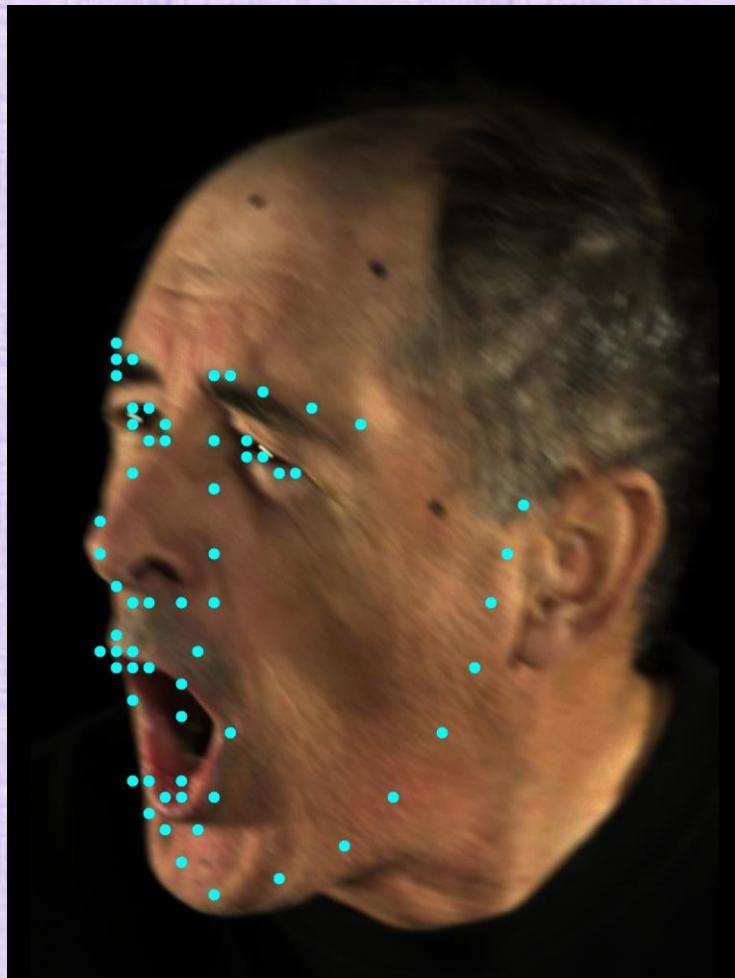
# Function Layers (Example)



## LAYER 3

- Input: RGB colors for pixels (a 2D image)
- Function: facial landmark detector
- Parameters: parameters for the neural network architecture, determined by training the network to match hand labeled data
- Output: 2D locations of landmarks on the image

# Function Layers (Example)

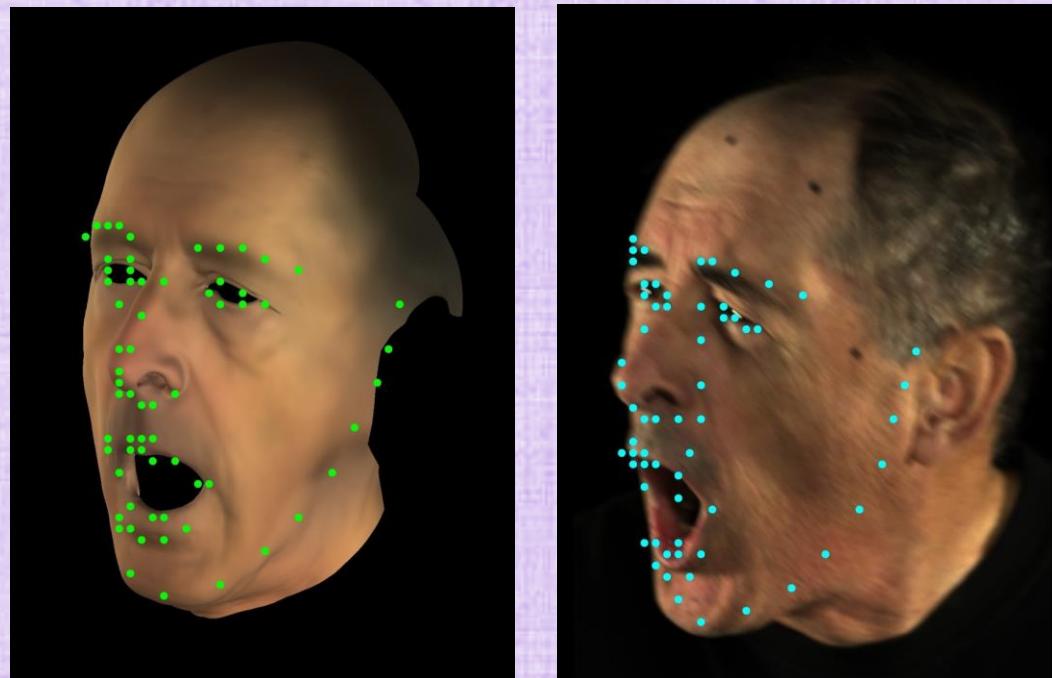


## TARGET OUTPUT

- Run a landmark detector on a photograph of the individual to obtain 2D landmark locations
- The goal is to have the 2D landmarks output from the complex multi-layered function match the 2D landmarks on the photograph

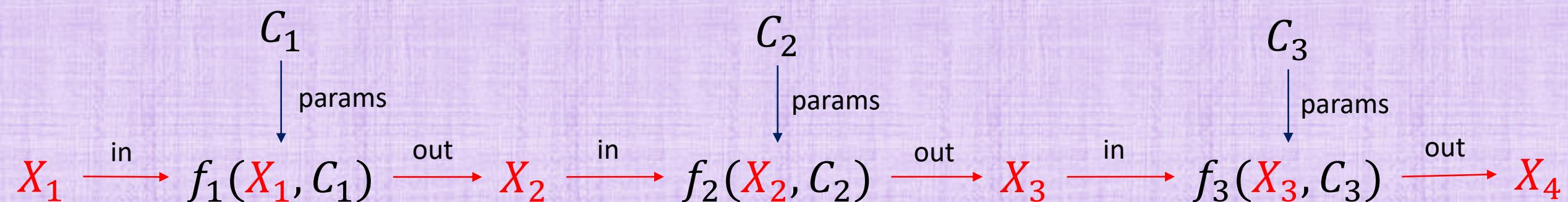
# Function Layers (Example)

- Modifying animation controls changes the triangulated surface which changes the rendered pixels in the image which changes the network's determination of landmarks
- When the two sets of landmarks agree, the animation controls give some indication of what the person in the photograph was doing



# Classical Optimization

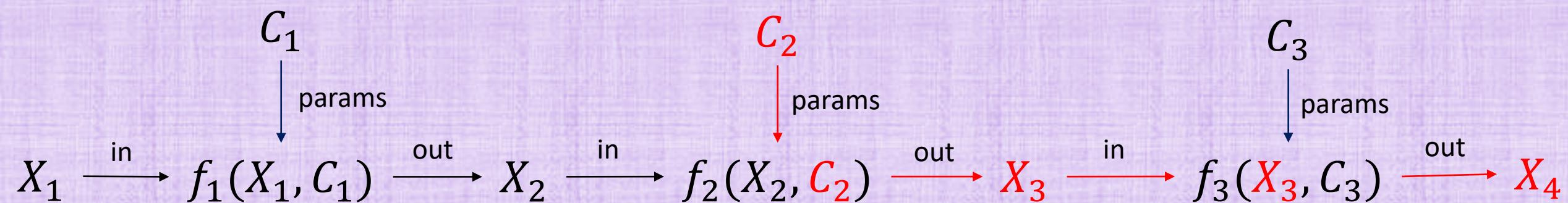
- Find the input  $X_1$  that minimizes  $\hat{f}(X_4)$
- Chain rule:  $\frac{\partial \hat{f}(X_4)}{\partial X_1} = \frac{\partial \hat{f}(X_4)}{\partial X_4} \frac{\partial X_4}{\partial X_3} \frac{\partial X_3}{\partial X_2} \frac{\partial X_2}{\partial X_1} = \frac{\partial \hat{f}(X_4)}{\partial X_4} \frac{\partial f_3(X_3, C_3)}{\partial X_3} \frac{\partial f_2(X_2, C_2)}{\partial X_2} \frac{\partial f_1(X_1, C_1)}{\partial X_1}$
- Parameters are considered fixed/constant



$$\hat{f}(X_4) = \|X_4 - X_{target}\|$$

# Network Training

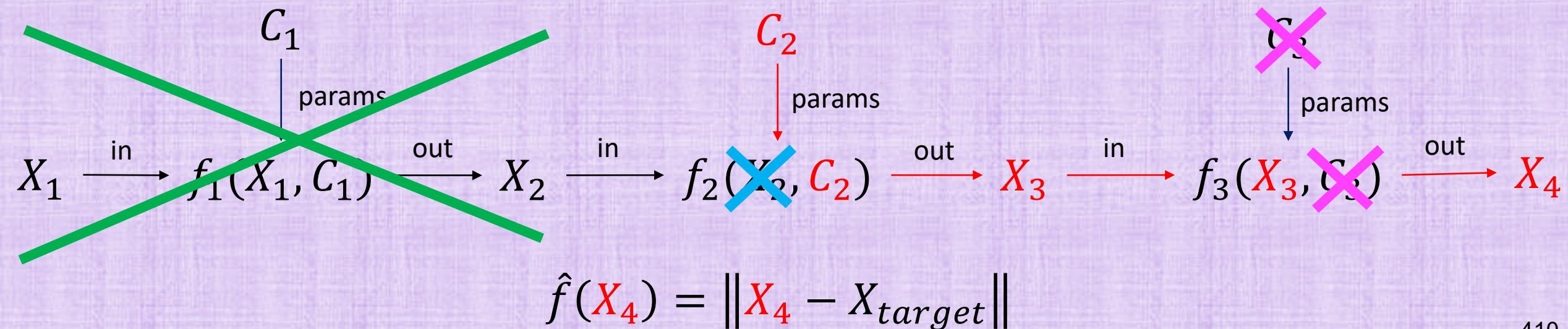
- Train network  $f_2$  by finding parameters  $C_2$  that minimize  $\hat{f}(X_4)$
- Chain rule:  $\frac{\partial \hat{f}(X_4)}{\partial C_2} = \frac{\partial \hat{f}(X_4)}{\partial X_4} \frac{\partial X_4}{\partial X_3} \frac{\partial X_3}{\partial C_2} = \frac{\partial \hat{f}(X_4)}{\partial X_4} \frac{\partial f_3(X_3, C_3)}{\partial X_3} \frac{\partial f_2(X_2, C_2)}{\partial C_2}$



$$\hat{f}(X_4) = \|X_4 - X_{target}\|$$

# Network Training

- Any preprocess to the network does not require differentiability
- The network itself only requires differentiability with respect to its parameters
- Any postprocess to the network requires input/output differentiability, but does not require differentiability with respect to its parameters



# Unit 18

# Avoiding Derivatives

# Part II Roadmap

- Part I – Linear Algebra (units 1-12)  $Ac = b$ 
    - linearize
    - line search
  - Part II – Optimization (units 13-20)
    - (units 13-16) Optimization -> Nonlinear Equations -> 1D roots/minima
    - (units 17-18) Computing/Avoiding Derivatives
    - (unit 19) Hack 1.0: “I give up”  $H = I$  and  $J$  is mostly 0 (descent methods)
    - (unit 20) Hack 2.0: “It’s an ODE!?” (adaptive learning rate and momentum)
- 
- ```
graph TD; A["Part I – Linear Algebra (units 1-12)  $Ac = b$ "] -- "linearize" --> B["(units 13-16) Optimization -> Nonlinear Equations -> 1D roots/minima"]; A -- "line search" --> C["(units 17-18) Computing/Avoiding Derivatives"]; A -- "line search" --> D["(unit 19) Hack 1.0: “I give up”  $H = I$  and  $J$  is mostly 0 (descent methods)"]; A -- "line search" --> E["(unit 20) Hack 2.0: “It’s an ODE!?” (adaptive learning rate and momentum)"]; B --- F["Theory"]; C --- F; D --- G["Methods"]; E --- G;
```

# 1D Root Finding (see Unit 15)

- Newton's method requires  $g'$ , as do mixed methods using Newton
- Secant method replaces  $g'$  with a secant line though two prior iterates
- Finite differencing (unit 17) may be used to approximate this derivative as well, although one needs to determine the size of the perturbation  $h$
- Automatic differentiation (unit 17) may be used to find the value of  $g'$  at a particular point, if/when “backprop” code exists, even when  $g$  and  $g'$  are not known in closed form
- Convergence is only guaranteed under certain conditions, emphasizing the importance of safe set methods (such as mixed methods with bisection)
- Safe set methods also help to guard against errors in derivative approximations

# 1D Optimization (see Unit 16)

- Root finding approaches search for critical points as the roots of  $g'$ 
  - All root finding methods use the function itself ( $g'$  here)
  - Newton (and mixed methods using Newton) require the derivative of the function ( $g''$  here)
- Can use secant lines for  $g'$  and interpolating parabolas for  $g''$ , using either prior iterates (unit 16) or finite differences (unit 17)
- Automatic differentiation (unit 17) may be leveraged as well
  - Although, not (typically) for approaches that require  $g''$
- Safe set methods (such as mixed methods with bisection or golden section search) help to guard against errors in the approximation of various derivatives

# Nonlinear Systems (see Unit 14)

- $J_F(c^q)\Delta c^q = -F(c^q)$  is solved to find the search direction  $\Delta c^q$ 
  - Then, line search utilizes various 1D approaches (unit 15/16)
- The Jacobian matrix of first derivatives  $J_F(c^q)$  needs to be evaluated (given  $c^q$ )
- Each entry  $\frac{\partial F_i}{\partial c_k}(c^q)$  can be approximated via finite differences (unit 17) or automatic differentiation (unit 17)
- Quasi-Newton approaches make various aggressive approximations to the Jacobian  $J_F(c^q)$
- Quasi-Newton can wildly perturb the search direction, so **robust/safe set approaches to the 1D line search become quite important to making “progress” towards solutions**

# Broyden's Method

- An initial guess for the Jacobian is repeatedly corrected with rank one updates, similar in spirit to a secant approach
- Let  $J^0 = I$
- Solve  $J^q \Delta c^q = -F(c^q)$  to find search direction  $\Delta c^q$ 
  - Use 1D line search to find  $c^{q+1}$  and thus  $F(c^{q+1})$ ; then, update  $\Delta c^q = c^{q+1} - c^q$
- Update  $J^{q+1} = J^q + \frac{1}{(\Delta c^q)^T \Delta c^q} (F(c^{q+1}) - F(c^q) - J^q \Delta c^q)(\Delta c^q)^T$
- Note:  $J^{q+1}(c^{q+1} - c^q) = F(c^{q+1}) - F(c^q)$
- That is,  $J^{q+1}$  satisfies a secant type equation  $J \Delta c = \Delta F$

# Optimization (see Unit 13)

- Scalar cost function  $\hat{f}(c)$  has critical points where  $J_{\hat{f}}^T(c) = 0$  (unit 13)
- $H_{\hat{f}}^T(c^q)\Delta c^q = -J_{\hat{f}}^T(c^q)$  is solved to find a search direction  $\Delta c^q$  (unit 14)
- Then, line search utilizes various 1D approaches (unit 15/16)
- The Hessian matrix of second derivatives  $H_{\hat{f}}^T(c^q)$  and the Jacobian vector of first derivatives  $J_{\hat{f}}^T(c^q)$  both need to be evaluated (given  $c^q$ )
- The various entries can be evaluated via finite differences (unit 17) or automatic differentiation (unit 17)
- These approaches can struggle on the Hessian matrix of second partial derivatives
- This makes Quasi-Newton approaches quite popular for optimization
  - When  $c$  is large, the  $O(n^2)$  Hessian  $H_{\hat{f}}^T$  is unwieldy/intractable, so some approaches instead approximate the action of  $H_{\hat{f}}^{-T}$  on a vector (i.e., on the right hand side)

# Broyden's Method (for Optimization)

- Same formulation as for nonlinear systems
- Solve for the search direction, and use 1D line search to find  $c^{q+1}$  and  $J_{\hat{f}}^T(c^{q+1})$
- Update  $\Delta c^q = c^{q+1} - c^q$  and  $\Delta J_{\hat{f}}^T = J_{\hat{f}}^T(c^{q+1}) - J_{\hat{f}}^T(c^q)$
- Update  $(H_{\hat{f}}^T)^{q+1} = (H_{\hat{f}}^T)^q + \frac{1}{(\Delta c^q)^T \Delta c^q} \left( \Delta J_{\hat{f}}^T - (H_{\hat{f}}^T)^q \Delta c^q \right) (\Delta c^q)^T$
- So that  $(H_{\hat{f}}^T)^{q+1} \Delta c^q = \Delta J_{\hat{f}}^T$

# Broyden's Method (for Optimization)

- For the inverse, using  $\Delta c^q = c^{q+1} - c^q$  and  $\Delta J_{\hat{f}}^T = J_{\hat{f}}^T(c^{q+1}) - J_{\hat{f}}^T(c^q)$
- Update  $(H_{\hat{f}}^{-T})^{q+1} = (H_{\hat{f}}^{-T})^q + \frac{(\Delta c^q - (H_{\hat{f}}^{-T})^q \Delta J_{\hat{f}}^T) (\Delta c^q)^T (H_{\hat{f}}^{-T})^q}{(\Delta c^q)^T (H_{\hat{f}}^{-T})^q \Delta J_{\hat{f}}^T}$
- So that  $(H_{\hat{f}}^{-T})^{q+1} \Delta J_{\hat{f}}^T = \Delta c^q$
- Solving  $H_{\hat{f}}^T(c^{q+1}) \Delta c^{q+1} = -J_{\hat{f}}^T(c^{q+1})$  is replaced with defining the search direction by  $\Delta c^{q+1} = -(H_{\hat{f}}^{-T})^{q+1} J_{\hat{f}}^T(c^{q+1})$

# SR1 (Symmetric Rank 1)

- For the inverse, using  $\Delta c^q = c^{q+1} - c^q$  and  $\Delta J_{\hat{f}}^T = J_{\hat{f}}^T(c^{q+1}) - J_{\hat{f}}^T(c^q)$
- Update  $(H_{\hat{f}}^{-T})^{q+1} = (H_{\hat{f}}^{-T})^q + \frac{(\Delta c^q - (H_{\hat{f}}^{-T})^q \Delta J_{\hat{f}}^T) (\Delta c^q - (H_{\hat{f}}^{-T})^q \Delta J_{\hat{f}}^T)^T}{(\Delta c^q - (H_{\hat{f}}^{-T})^q \Delta J_{\hat{f}}^T)^T \Delta J_{\hat{f}}^T}$
- So that  $(H_{\hat{f}}^{-T})^{q+1} \Delta J_{\hat{f}}^T = \Delta c^q$
- Solving  $H_{\hat{f}}^T(c^{q+1}) \Delta c^{q+1} = -J_{\hat{f}}^T(c^{q+1})$  is replaced with defining the search direction by  $\Delta c^{q+1} = -(H_{\hat{f}}^{-T})^{q+1} J_{\hat{f}}^T(c^{q+1})$

# DFP (Davidon-Fletcher-Powell)

- For the inverse, using  $\Delta c^q = c^{q+1} - c^q$  and  $\Delta J_{\hat{f}}^T = J_{\hat{f}}^T(c^{q+1}) - J_{\hat{f}}^T(c^q)$
- Update  $(H_{\hat{f}}^{-T})^{q+1} = (H_{\hat{f}}^{-T})^q - \frac{(H_{\hat{f}}^{-T})^q \Delta J_{\hat{f}}^T \Delta J_{\hat{f}} (H_{\hat{f}}^{-T})^q}{\Delta J_{\hat{f}} (H_{\hat{f}}^{-T})^q \Delta J_{\hat{f}}^T} + \frac{\Delta c^q (\Delta c^q)^T}{(\Delta c^q)^T \Delta J_{\hat{f}}^T}$
- So that  $(H_{\hat{f}}^{-T})^{q+1} \Delta J_{\hat{f}}^T = \Delta c^q$
- Solving  $H_{\hat{f}}^T(c^{q+1}) \Delta c^{q+1} = -J_{\hat{f}}^T(c^{q+1})$  is replaced with defining the search direction by  $\Delta c^{q+1} = -(H_{\hat{f}}^{-T})^{q+1} J_{\hat{f}}^T(c^{q+1})$

# BFGS (Broyden-Fletcher-Goldfarb-Shanno)

- For the inverse, using  $\Delta c^q = c^{q+1} - c^q$  and  $\Delta J_{\hat{f}}^T = J_{\hat{f}}^T(c^{q+1}) - J_{\hat{f}}^T(c^q)$
- Update  $(H_{\hat{f}}^{-T})^{q+1} = \left( I - \frac{\Delta c^q \Delta J_{\hat{f}}}{(\Delta c^q)^T \Delta J_{\hat{f}}^T} \right) (H_{\hat{f}}^{-T})^q \left( I - \frac{\Delta J_{\hat{f}}^T (\Delta c^q)^T}{(\Delta c^q)^T \Delta J_{\hat{f}}^T} \right) + \frac{\Delta c^q (\Delta c^q)^T}{(\Delta c^q)^T \Delta J_{\hat{f}}^T}$
- So that  $(H_{\hat{f}}^{-T})^{q+1} \Delta J_{\hat{f}}^T = \Delta c^q$
- Solving  $H_{\hat{f}}^T(c^{q+1}) \Delta c^{q+1} = -J_{\hat{f}}^T(c^{q+1})$  is replaced with defining the search direction by  $\Delta c^{q+1} = -(H_{\hat{f}}^{-T})^{q+1} J_{\hat{f}}^T(c^{q+1})$

# L-BFGS (Limited Memory BFGS)

- These methods store an  $n \times n$  approximation to the inverse Hessian
  - This can become unwieldy for large problems
  - Smarter storage can be accomplished by storing the vectors that describe the outer products; however, the number of vectors grows with  $q$
- L-BFGS estimates the inverse Hessian using only a few of the prior vectors
  - often less than 10 vectors (**vectors, vector spaces, not matrices**)
- This makes it quite popular for machine learning
- On optimization methods for deep learning, Andrew Ng et al., ICML 2011
  - “we show that more sophisticated off-the-shelf optimization methods such as Limited memory BFGS (L-BFGS) and Conjugate gradient (CG) with line search can significantly simplify and speed up the process of pretraining deep algorithms”

# Gradient/Steepest Descent

- Approximate  $H_{\hat{f}}^T$  very crudely with the identity matrix
  - which is the **first step** of all the aforementioned methods
- That is,  $H_{\hat{f}}^T(c^q)\Delta c^q = -J_{\hat{f}}^T(c^q)$  becomes  $I\Delta c^q = -J_{\hat{f}}^T(c^q)$
- So, the search direction is  $\Delta c^q = -J_{\hat{f}}^T(c^q) = -\nabla \hat{f}(c^q)$ 
  - This is the steepest descent direction
- See unit 19

# Coordinate Descent

- Coordinate Descent ignores  $H_{\hat{f}}^T(c^q)\Delta c^q = -J_{\hat{f}}^T(c^q)$  completely
- Instead,  $\Delta c^q$  is set to the various coordinate directions  $\hat{e}_k$

# Nonlinear Least Squares (ML relevancy)

- Minimize a cost function of the form:  $\hat{f}(c) = \frac{1}{2} \tilde{f}^T(c) \tilde{f}(c)$
- Recall from Unit 13:
  - Determine parameters  $c$  that make  $f(x, y, c) = 0$  best fit the training data, i.e. that make  $\|f(x_i, y_i, c)\|_2^2 = f(x_i, y_i, c)^T f(x_i, y_i, c)$  close to zero for all  $i$
  - Combining all  $(x_i, y_i)$ , minimize  $\hat{f}(c) = \frac{1}{2} \sum_i f(x_i, y_i, c)^T f(x_i, y_i, c)$
- Let  $m$  be the number of data points and  $\hat{m}$  be the output size of  $f(x, y, c)$
- Define  $\tilde{f}(c)$  by stacking the  $\hat{m}$  outputs of  $f(x, y, c)$  consecutively  $m$  times, so that the vector valued output of  $\tilde{f}(c)$  is length  $m * \hat{m}$
- Then,  $\hat{f}(c) = \frac{1}{2} \sum_i f(x_i, y_i, c)^T f(x_i, y_i, c) = \frac{1}{2} \tilde{f}^T(c) \tilde{f}(c)$

# Nonlinear Least Squares (Critical Points)

- Minimize  $\hat{f}(c) = \frac{1}{2} \tilde{f}^T(c) \tilde{f}(c)$
- Jacobian matrix of  $\tilde{f}$  is  $J_{\tilde{f}}(c) = \begin{pmatrix} \frac{\partial \tilde{f}}{\partial c_1}(c) & \frac{\partial \tilde{f}}{\partial c_2}(c) & \dots & \frac{\partial \tilde{f}}{\partial c_n}(c) \end{pmatrix}$
- Critical points of  $\hat{f}(c)$  have  $J_{\hat{f}}^T(c) = \begin{pmatrix} \tilde{f}^T(c) \frac{\partial \tilde{f}}{\partial c_1}(c) \\ \tilde{f}^T(c) \frac{\partial \tilde{f}}{\partial c_2}(c) \\ \vdots \\ \tilde{f}^T(c) \frac{\partial \tilde{f}}{\partial c_n}(c) \end{pmatrix} = J_{\tilde{f}}^T(c) \tilde{f}(c) = 0$

# Gauss Newton

- $J_{\tilde{f}}^T(c)\tilde{f}(c) = 0$  becomes  $J_{\tilde{f}}^T(c)(\tilde{f}(c^q) + J_{\tilde{f}}(c^q)\Delta c^q + \dots) = 0$ 
  - Using the Taylor series:  $\tilde{f}(c) = \tilde{f}(c^q) + J_{\tilde{f}}(c^q)\Delta c^q + \dots$
- Eliminating high order terms:  $J_{\tilde{f}}^T(c)(\tilde{f}(c^q) + J_{\tilde{f}}(c^q)\Delta c^q) \approx 0$
- Evaluating  $J_{\tilde{f}}^T$  at  $c^q$  gives  $J_{\tilde{f}}^T(c^q)J_{\tilde{f}}(c^q)\Delta c^q \approx -J_{\tilde{f}}^T(c^q)\tilde{f}(c^q)$
- Compare to  $H_{\hat{f}}^T(c^q)\Delta c^q = -J_{\hat{f}}^T(c^q)$  and note that  $J_{\hat{f}}^T(c) = J_{\tilde{f}}^T(c)\tilde{f}(c)$
- Thus, Gauss Newton uses the estimate:  $H_{\hat{f}}^T(c^q) \approx J_{\tilde{f}}^T(c^q)J_{\tilde{f}}(c^q)$

# Gauss Newton ( $QR$ approach)

- The Gauss Newton equations  $J_{\tilde{f}}^T(c^q)J_{\tilde{f}}(c^q)\Delta c^q = -J_{\tilde{f}}^T(c^q)\tilde{f}(c^q)$  are the normal equations for  $J_{\tilde{f}}(c^q)\Delta c^q = -\tilde{f}(c^q)$
- Thus, (instead) solve  $J_{\tilde{f}}(c^q)\Delta c^q = -\tilde{f}(c^q)$  via any least squares ( $QR$ ) and minimum norm approach
- Note: setting the second factor in  $J_{\tilde{f}}^T(c)(\tilde{f}(c^q) + J_{\tilde{f}}(c^q)\Delta c^q) \approx 0$  to zero also leads to  $J_{\tilde{f}}(c^q)\Delta c^q = -\tilde{f}(c^q)$
- This is a linearization of the nonlinear system  $\tilde{f}(c) = 0$ , aiming to minimize  $\hat{f}(c) = \frac{1}{2}\tilde{f}^T(c)\tilde{f}(c)$

# Weighted Gauss Newton

- Given a diagonal matrix  $D$  indicating the importance of various equations:

$$DJ_{\tilde{f}}(c^q)\Delta c^q = -D\tilde{f}(c^q)$$
$$J_{\tilde{f}}^T(c^q)D^2J_{\tilde{f}}(c^q)\Delta c^q = -J_{\tilde{f}}^T(c^q)D^2\tilde{f}(c^q)$$

- Recall: Row scaling changes the importance of the equations
- Recall: Thus, it also changes the (unique) least squares solution for any overdetermined degrees of freedom

# Regularized Gauss Newton

- When concerned about small singular values in  $J_{\tilde{f}}(c^q)\Delta c^q = -\tilde{f}(c^q)$ , one can add  $\epsilon I = 0$  as extra equations (unit 12 regularization)
- This results in  $(J_{\tilde{f}}^T(c^q)J_{\tilde{f}}(c^q) + \epsilon^2 I)\Delta c^q = -J_{\tilde{f}}^T(c^q)\tilde{f}(c^q)$
- This is often called Levenberg-Marquardt or Damped (Nonlinear) Least Squares

# Unit 19

# Descent Methods

# Part II Roadmap

- Part I – Linear Algebra (units 1-12)  $Ac = b$ 
    - linearize
    - line search
  - Part II – Optimization (units 13-20)
    - (units 13-16) Optimization -> Nonlinear Equations -> 1D roots/minima
    - (units 17-18) Computing/Avoiding Derivatives
    - (unit 19) Hack 1.0: “I give up”  $H = I$  and  $J$  is mostly 0 (descent methods)
    - (unit 20) Hack 2.0: “It’s an ODE!?” (adaptive learning rate and momentum)
- 
- ```
graph TD; A["Part I – Linear Algebra (units 1-12)  $Ac = b$ "] -- "red arrow" --> B["linearize"]; A -- "red arrow" --> C["line search"]; D["Part II – Optimization (units 13-20)"] --> E["(units 13-16) Optimization -> Nonlinear Equations -> 1D roots/minima"]; D --> F["(units 17-18) Computing/Avoiding Derivatives"]; D --> G["(unit 19) Hack 1.0: “I give up”  $H = I$  and  $J$  is mostly 0 (descent methods)"]; D --> H["(unit 20) Hack 2.0: “It’s an ODE!?” (adaptive learning rate and momentum)"]; G --> I["Theory"]; H --> J["Methods"]
```

# Recall: Gradient (Unit 9)

- Consider the scalar (output) function  $f(c)$  with multi-dimensional input  $c$
- The Jacobian of  $f(c)$  is  $J(c) = \begin{pmatrix} \frac{\partial f}{\partial c_1}(c) & \frac{\partial f}{\partial c_2}(c) & \cdots & \frac{\partial f}{\partial c_n}(c) \end{pmatrix}$
- The gradient of  $f(c)$  is  $\nabla f(c) = J^T(c) = \begin{pmatrix} \frac{\partial f}{\partial c_1}(c) \\ \frac{\partial f}{\partial c_2}(c) \\ \vdots \\ \frac{\partial f}{\partial c_n}(c) \end{pmatrix}$
- In 1D, both  $J(c)$  and  $\nabla f(c) = J^T(c)$  are the usual  $f'(c)$

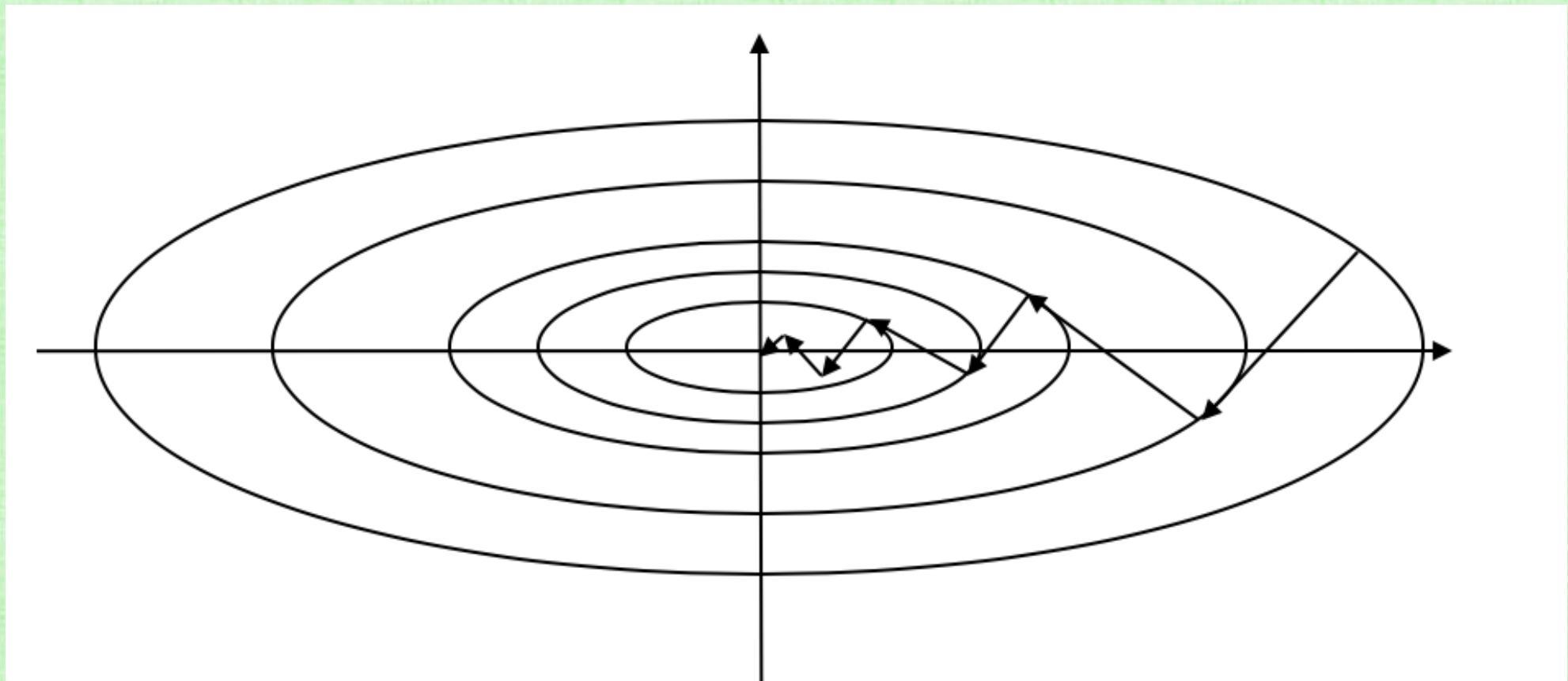
# Gradient/Steepest Descent

- Given a cost function  $\hat{f}(c)$ 
  - $\nabla \hat{f}(c)$  is the direction in which  $\hat{f}(c)$  increases the fastest
  - $-\nabla \hat{f}(c)$  is the direction in which  $\hat{f}(c)$  decreases the fastest
- Thus,  $-\nabla \hat{f}(c)$  is considered the direction of steepest descent
- Using  $-\nabla \hat{f}(c)$  as the search direction is known as steepest descent
  - This can be thought of as always “walking in the steepest downhill direction”
  - However, never going uphill can lead to local minima
- Methods that use  $-\nabla \hat{f}(c)$  in various ways are known as gradient descent methods
- Recall (Unit 18) approximating  $H_{\hat{f}}^T \approx I$  in  $H_{\hat{f}}^T(c^q)\Delta c^q = -J_{\hat{f}}^T(c^q)$  leads to steepest descent:  $\Delta c^q = -J_{\hat{f}}^T(c^q) = -\nabla \hat{f}(c^q)$

# Steepest Descent for Quadratic Forms

- Recall (Unit 9):
  - Quadratic Form of SPD  $\tilde{A}$  is  $f(c) = \frac{1}{2}c^T \tilde{A}c - \tilde{b}^T c + \tilde{c}$
  - Minimize  $f(c)$  by finding critical points where  $\nabla f(c) = \tilde{A}c - \tilde{b} = 0$
  - That is, solve  $\tilde{A}c = \tilde{b}$  to find the critical point
- Recall (Unit 5):
  - Steepest descent search direction:  $-\nabla f(c) = \tilde{b} - \tilde{A}c = r$
  - $r^q = b - Ac^q$ ,  $\alpha^q = \frac{r^q \cdot r^q}{r^q \cdot Ar^q}$ ,  $c^{q+1} = c^q + \alpha^q r^q$  is iterated until  $r^q$  is small enough
  - The main drawback to steepest descent is that it repeatedly searches in the same directions too often, especially for higher condition number matrices
  - Because it takes a long time to converge, we instead advocated for Conjugate Gradients

# Steepest Descent for Quadratic Forms



CG would (instead) solve this in 2 steps

# Recall: Nonlinear Least Squares (Unit 18)

- Minimize a cost function of the form:  $\hat{f}(c) = \frac{1}{2} \tilde{f}^T(c) \tilde{f}(c)$
- Recall from Unit 13:
  - Determine parameters  $c$  that make  $f(x, y, c) = 0$  best fit the training data, i.e. that make  $\|f(x_i, y_i, c)\|_2^2 = f(x_i, y_i, c)^T f(x_i, y_i, c)$  close to zero for all  $i$
  - Combining all  $(x_i, y_i)$ , minimize  $\hat{f}(c) = \frac{1}{2} \sum_i f(x_i, y_i, c)^T f(x_i, y_i, c)$
- Let  $m$  be the number of data points and  $\hat{m}$  be the output size of  $f(x, y, c)$
- Define  $\tilde{f}(c)$  by stacking the  $\hat{m}$  outputs of  $f(x, y, c)$  consecutively  $m$  times, so that the vector valued output of  $\tilde{f}(c)$  is length  $m * \hat{m}$
- Then,  $\hat{f}(c) = \frac{1}{2} \sum_i f(x_i, y_i, c)^T f(x_i, y_i, c) = \frac{1}{2} \tilde{f}^T(c) \tilde{f}(c)$

# Recall: Nonlinear Least Squares (Unit 18)

- Minimize  $\hat{f}(c) = \frac{1}{2} \tilde{f}^T(c) \tilde{f}(c)$
- Jacobian matrix of  $\tilde{f}$  is  $J_{\tilde{f}}(c) = \begin{pmatrix} \frac{\partial \tilde{f}}{\partial c_1}(c) & \frac{\partial \tilde{f}}{\partial c_2}(c) & \dots & \frac{\partial \tilde{f}}{\partial c_n}(c) \end{pmatrix}$
- Critical points of  $\hat{f}(c)$  have  $J_{\hat{f}}^T(c) = \begin{pmatrix} \tilde{f}^T(c) \frac{\partial \tilde{f}}{\partial c_1}(c) \\ \tilde{f}^T(c) \frac{\partial \tilde{f}}{\partial c_2}(c) \\ \vdots \\ \tilde{f}^T(c) \frac{\partial \tilde{f}}{\partial c_n}(c) \end{pmatrix} = J_{\tilde{f}}^T(c) \tilde{f}(c) = 0$

# Steepest Descent for Nonlinear Least Squares

- Search direction  $-\nabla \hat{f}(c) = -J_{\tilde{f}}^T(c) = -J_{\tilde{f}}^T(c)\tilde{f}(c) = \begin{pmatrix} -\tilde{f}^T(c) \frac{\partial \tilde{f}}{\partial c_1}(c) \\ -\tilde{f}^T(c) \frac{\partial \tilde{f}}{\partial c_2}(c) \\ \vdots \\ -\tilde{f}^T(c) \frac{\partial \tilde{f}}{\partial c_n}(c) \end{pmatrix}$
- Recall that  $\tilde{f}(c)$  is the  $\hat{m}$  outputs of  $f(x_i, y_i, c)$  stacked consecutively  $m$  times, once for each data point  $(x_i, y_i)$
- Thus, each of the  $n$  terms of the form  $-\tilde{f}^T(c) \frac{\partial \tilde{f}}{\partial c_k}(c)$  is a (potentially expensive) sum through  $m * \hat{m}$  terms (recall:  $m$  is the amount of training data)

# Gradient Descent for Nonlinear Least Squares

- When there is a lot of data,  $m$  can be extremely large
  - This is exacerbated when the  $\frac{\partial \tilde{f}}{\partial c_k}$  are expensive to compute
- Using all the data is called Batch Gradient Descent
- When only a (small) subset of the data is used to compute the search direction (ignoring the rest of the data), this is called Mini-Batch Gradient Descent
- When only a single data point is used to compute the search direction (chosen randomly/sequentially), this is called Stochastic Gradient Descent (SGD)

# Unit 20

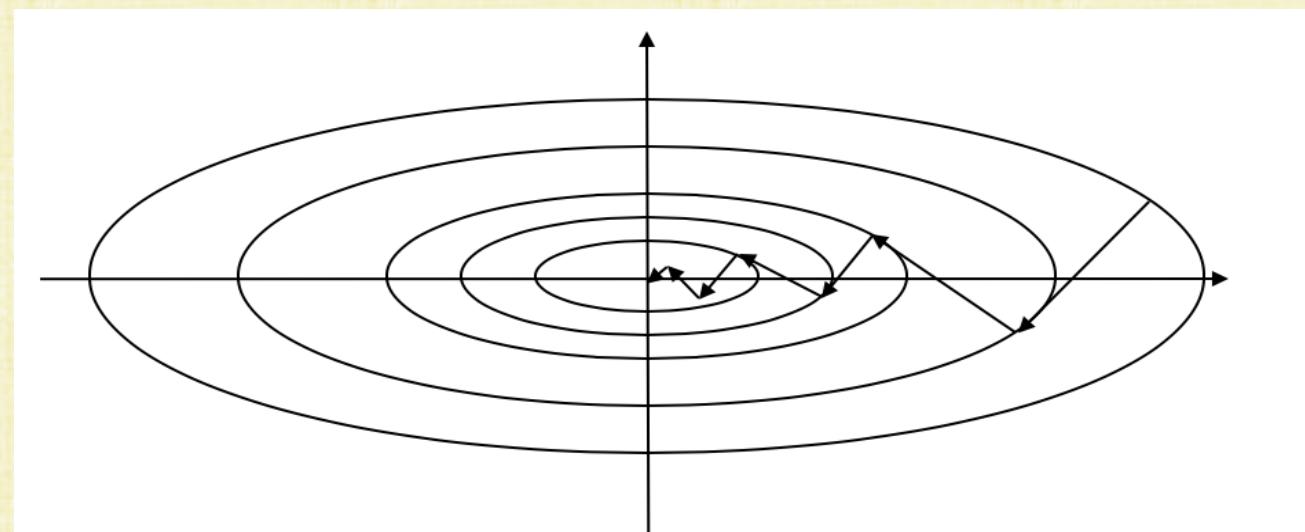
# Momentum Methods

# Part II Roadmap

- Part I – Linear Algebra (units 1-12)  $Ac = b$ 
    - linearize
    - line search
  - Part II – Optimization (units 13-20)
    - (units 13-16) Optimization -> Nonlinear Equations -> 1D roots/minima
    - (units 17-18) Computing/Avoiding Derivatives
    - (unit 19) Hack 1.0: “I give up”  $H = I$  and  $J$  is mostly 0 (descent methods)
    - (unit 20) Hack 2.0: “It’s an ODE!?” (adaptive learning rate and momentum)
- 
- ```
graph TD; A["Part I – Linear Algebra (units 1-12)  $Ac = b$ "] -- "linearize" --> B[" $Ac = b$ "]; A -- "line search" --> B; C["Part II – Optimization (units 13-20)"] --> D["(units 13-16) Optimization -> Nonlinear Equations -> 1D roots/minima"]; C --> E["(units 17-18) Computing/Avoiding Derivatives"]; C --> F["(unit 19) Hack 1.0: ‘I give up’  $H = I$  and  $J$  is mostly 0 (descent methods)"]; C --> G["(unit 20) Hack 2.0: ‘It’s an ODE!?’ (adaptive learning rate and momentum)"]; D -- Theory --> H["Theory"]; F -- Methods --> I["Methods"]
```

# Path through Parameter Space

- Optimization solvers iteratively update the state variable  $c$  at each iteration
- For difficult problems (such as neural network training), this is typically done via a 1D line search at each iteration
- The union of all such line searches can be thought of as a path through parameter space



# Continuous Path vs Discrete Path

- Each iteration is a discrete jump from one point to another, and connecting them with a 1D line segment is merely a visualization
- In the limit as the size of the segments goes to zero (and the number of iterations goes to infinity), one obtains a continuous path
- Can parameterize this path/curve with a scalar  $t$  (typically called time)
- Then  $c(t)$  is a continuous path in parameter space ( $c(t)$  is a position)
- Changing the value of  $t$  moves the position  $c(t)$  along the path
- Differentiating the continuous path gives a time varying velocity:  $\frac{dc}{dt}(t)$  or  $c'(t)$

# Ordinary Differential Equations (ODEs)

- ODEs are equations that describe rates of change
- E.g.,  $\frac{dc}{dt}(t) = f(t, c(t))$  states that the parameter space velocity is  $f(t, c(t))$
- “Solving” an ODE means finding a function with rates of change described by the ODE
- Given the velocity along the curve  $\frac{dc}{dt}(t) = f(t, c(t))$ , find the curve  $c(t)$  itself
- Consider a (greedy) steepest decent path which always follows the steepest downhill direction for a cost function  $\hat{f}(c)$ 
  - A suitable velocity is any (positive) scalar multiple of  $-\nabla \hat{f}(c(t))$
  - This leads to an ODE:  $\frac{dc}{dt}(t) = -\nabla \hat{f}(c(t))$

# Gradient Flow

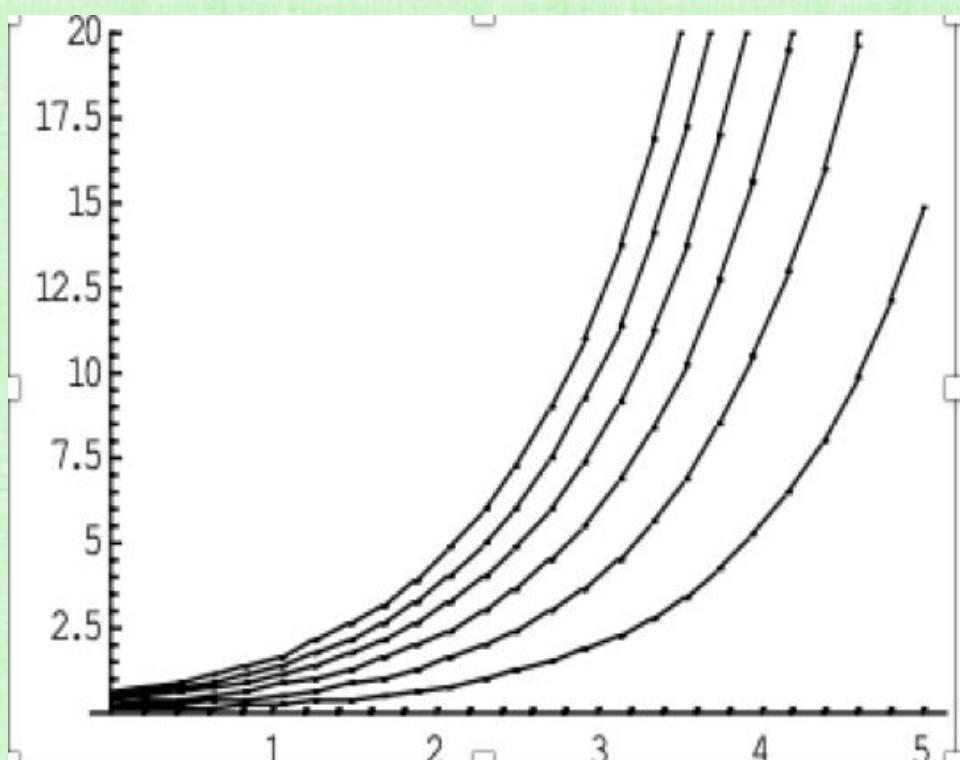
- The ODE for gradient flow is:  $\frac{dc}{dt}(t) = -\nabla \hat{f}(c(t))$

- Or (in more detail): 
$$\begin{pmatrix} \frac{dc_1}{dt}(t) \\ \frac{dc_2}{dt}(t) \\ \vdots \\ \frac{dc_n}{dt}(t) \end{pmatrix} = \begin{pmatrix} -\frac{\partial \hat{f}}{\partial c_1}(c(t)) \\ -\frac{\partial \hat{f}}{\partial c_2}(c(t)) \\ \vdots \\ -\frac{\partial \hat{f}}{\partial c_n}(c(t)) \end{pmatrix}$$

- $c(t)$  is a function of time  $t$  that evolves/changes based on the local gradient of the cost function,  $-\nabla \hat{f}(c(t))$
- This path follows the direction of steepest descent

# Families of Solutions

- ODEs are initial value problems: the solution depends on the initial (starting) condition



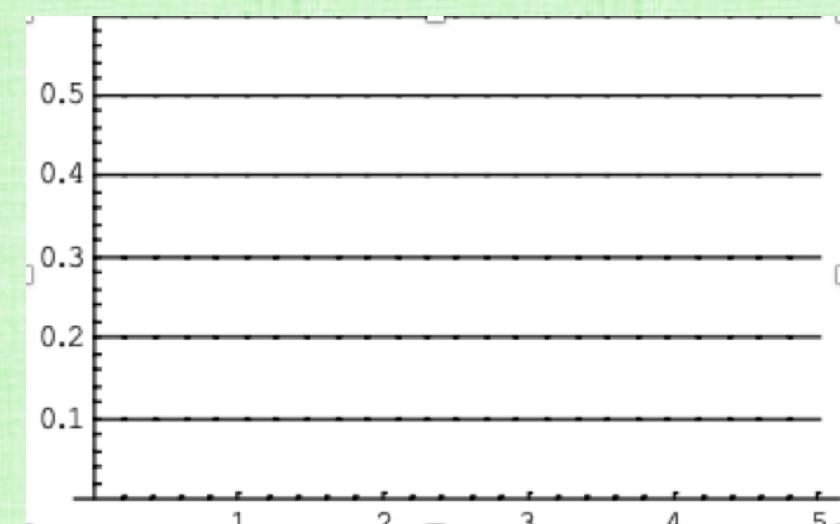
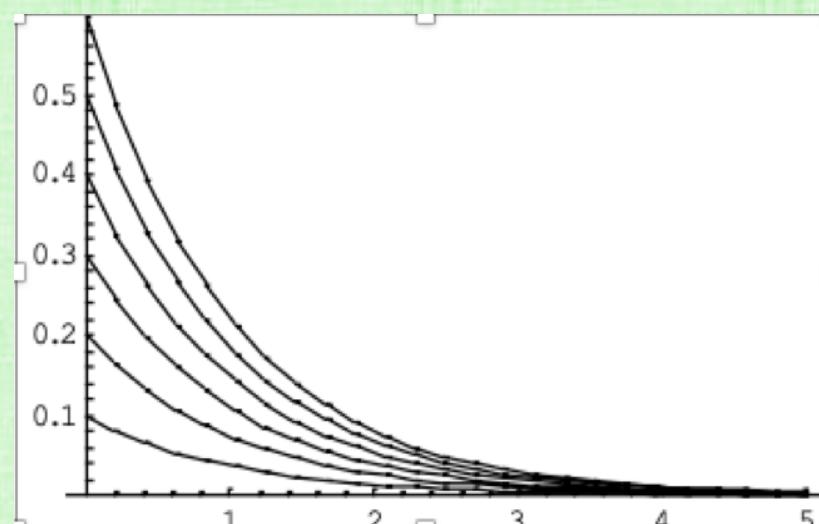
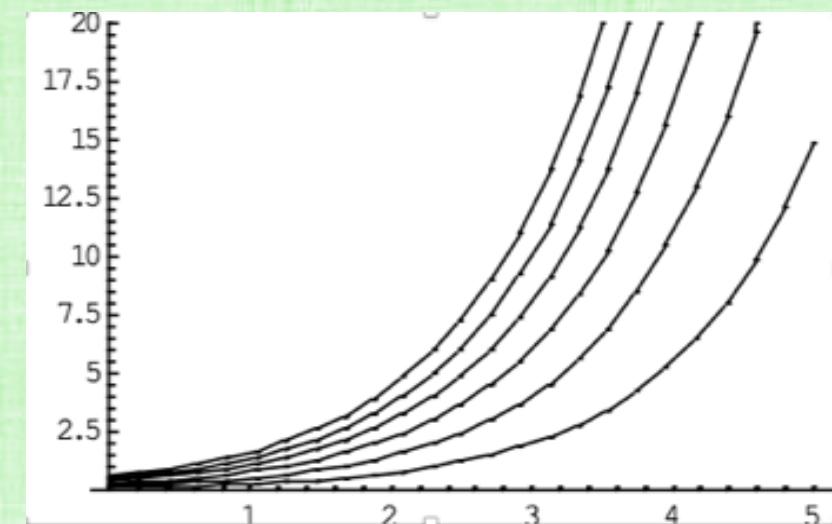
- E.g.  $c' = c$  or  $\frac{dc}{dt} = c$  or  $\frac{dc}{c} = dt$
- $\int_{c_o}^c \frac{1}{c} dc = \int_{t_o}^t dt$  or  $\ln c - \ln c_o = t - t_o$
- $\ln \frac{c}{c_o} = t - t_o$  or  $\frac{c}{c_o} = e^{t-t_o}$  or  $c = c_o e^{t-t_o}$
- Solution  $c(t) = c_o e^{t-t_o}$  depends on the initial condition  $c(t_o) = c_o$
- The figure shows solutions for various values of  $c_o$  at  $t_o = 0$

# Gradient Flow

- We ansatz that following the solution trajectory in gradient flow leads to a preferred minimum of  $\hat{f}(c)$
- Numerical errors will cause perturbations away from this desired trajectory
- Hopefully, the perturbed trajectories stay nearby the desired trajectory
- Hopefully, the perturbed trajectories lead to the same minima
- Sometimes, there are bifurcations of solution trajectories
- In such regions, perturbations can lead to very different (presumably less preferred) minima

# Posedness

- Consider  $c' = \lambda c$  with solution family  $c(t) = c_o e^{\lambda(t-t_o)}$



- $\lambda > 0$ , exponential growth, ill-posed
- Small changes in initial conditions (and/or small solver errors) result in large changes to the trajectory
- $\lambda < 0$ , exponential decay, well-posed
- Small changes in initial conditions (and/or small solver errors) are damped by converging trajectories
- $\lambda = 0$ , constant solution, linearly stable, mildly ill-posed
- Small changes in initial conditions (and/or small solver errors) result in (slow but cumulative) trajectory drift

# Posedness

- Consider a system of ODEs  $c' = F(t, c)$  with Jacobian matrix  $J_F(t, c) = \frac{\partial F}{\partial c}(t, c)$
- Since  $c(t)$  is time varying, so is  $J_F(t, c(t))$
- Whenever an eigenvalue of  $J_F(t, c(t))$  is positive, the associated part of the solution becomes ill-posed and trajectories can (wildly) diverge
  - This typically pollutes the entire solution vector
- Thus, all eigenvalues of  $J_F(t, c(t))$  must be non-positive for all  $t$  for the problem to be considered well-posed
- Moreover, eigenvalues close to zero may be suspect due to numerical errors
- Ill-posedness can rapidly lead to solution family bifurcation and thus minima far from what one might expect

# Stability and Accuracy

- For well-posed ODEs, a numerical approach is considered stable if it does not overflow and produce NaNs (i.e. shoot off to  $\infty$  in parameter space)
- Stability guarantees typically result in a restriction on the size of the time step  $\Delta t$ 
  - Otherwise, the method may go unstable
- For well posed ODEs, a stable numerical approach can be analyzed for accuracy to see how well it matches known solutions
- Hopefully, stability and reasonable accuracy keep the numerical solution of the ODE close to an ideal trajectory (leading to the preferred minimum)

# Forward Euler Method

- Approximate  $c' = f(t, c)$  with  $\frac{c^{q+1} - c^q}{\Delta t} = f(t^q, c^q)$
- Or recursively:  $c^{q+1} = c^q + \Delta t f(t^q, c^q)$
- Recall: Taylor series  $c^{q+1} = c^q + \Delta t f(t^q, c^q) + O(\Delta t^2)$
- So, there is an  $O(\Delta t^2)$  local truncation error each time step (or iteration)
- Overall,  $\frac{t_f - t_0}{\Delta t} = O\left(\frac{1}{\Delta t}\right)$  time steps are taken
- So, the total error or global truncation error is  $O(\Delta t^2)O\left(\frac{1}{\Delta t}\right) = O(\Delta t)$
- This makes the method 1<sup>st</sup> order accurate
  - Recall comments on accuracy and Newton-Cotes approaches in Unit 7 Curse of Dimensionality

# Runge-Kutta (RK) Methods

- More accurate methods can be constructed similarly, i.e. by considering Taylor series:
- **1<sup>st</sup> order:**  $\frac{c^{q+1} - c^q}{\Delta t} = f(t^q, c^q)$  which is forward Euler
- **2<sup>nd</sup> order:**  $\frac{c^{q+1} - c^q}{\Delta t} = \frac{1}{2} k_1 + \frac{1}{2} k_2$  where  $k_1 = f(t^q, c^q)$  is used in a forward Euler (predictor) update in order to compute  $k_2 = f(t^{q+1}, c^q + \Delta t k_1)$
- **4<sup>th</sup> order:**  $\frac{c^{q+1} - c^q}{\Delta t} = \frac{1}{6} k_1 + \frac{1}{3} k_2 + \frac{1}{3} k_3 + \frac{1}{6} k_4$  where  $k_1 = f(t^q, c^q)$ ,  $k_2 = f\left(t^{q+\frac{1}{2}}, c^q + \frac{\Delta t}{2} k_1\right)$ ,  $k_3 = f\left(t^{q+\frac{1}{2}}, c^q + \frac{\Delta t}{2} k_2\right)$ ,  $k_4 = f(t^{q+1}, c^q + \Delta t k_3)$ 
  - Again, each term builds on the prior in a predictor style fashion

# TVD Runge-Kutta Methods

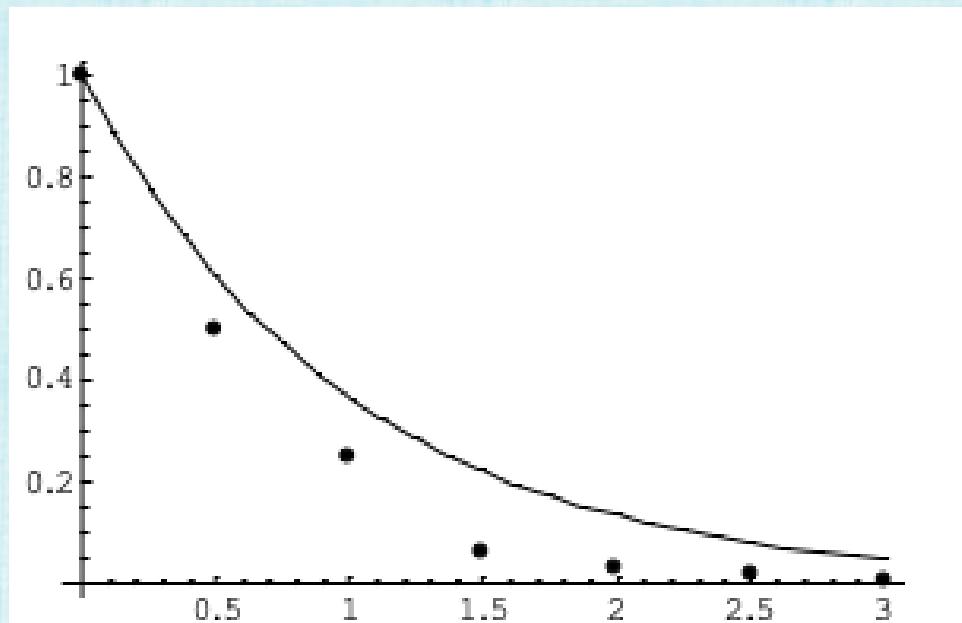
- Combinations of well-behaved forward Euler and well-behaved averaging
- 1<sup>st</sup> order: same as standard RK and forward Euler
- 2<sup>nd</sup> order: same as standard 2<sup>nd</sup> order RK (also called the midpoint rule, the modified Euler method, and Heun's predictor-corrector method)
  - Take two forward Euler steps,  $\frac{\hat{c}^{q+1} - c^q}{\Delta t} = f(t^q, c^q)$  and  $\frac{\hat{c}^{q+2} - \hat{c}^{q+1}}{\Delta t} = f(t^{q+1}, \hat{c}^{q+1})$ , and average the initial and final state,  $c^{q+1} = \frac{1}{2}c^q + \frac{1}{2}\hat{c}^{q+2}$
- 3<sup>rd</sup> order: Take two Euler steps, but average differently  $\hat{c}^{q+\frac{1}{2}} = \frac{3}{4}c^q + \frac{1}{4}\hat{c}^{q+2}$ ; then, take another forward Euler step,  $\frac{\hat{c}^{q+\frac{3}{2}} - \hat{c}^{q+\frac{1}{2}}}{\Delta t} = f(t^{q+\frac{1}{2}}, \hat{c}^{q+\frac{1}{2}})$ , and average again,  $c^{q+1} = \frac{1}{3}c^q + \frac{2}{3}\hat{c}^{q+\frac{3}{2}}$

# Stability Analysis

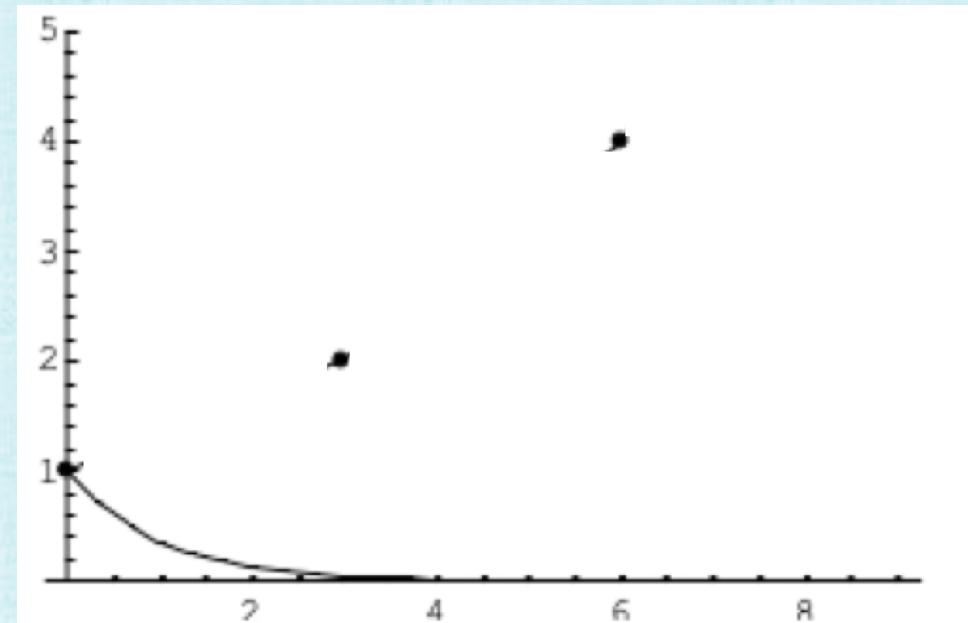
- Consider the model equation  $c' = \lambda c$  with a well-posed  $\lambda < 0$
- This models how an eigenvalue of a Jacobian matrix might behave
- Forward Euler gives  $c^{q+1} = c^q + \Delta t \lambda c^q = (1 + \Delta t \lambda) c^q = (1 + \Delta t \lambda)^{q+1} c^0$
- The error shrinks and the solution decays (as it should for  $\lambda < 0$ ) as long as  $|1 + \Delta t \lambda| < 1$
- This leads to  $-1 < 1 + \Delta t \lambda < 1$  or  $-2 < \Delta t \lambda < 0$  or  $-\frac{2}{\lambda} > \Delta t > 0$
- Since  $\lambda < 0$  and  $\Delta t > 0$ , one needs  $\Delta t < \frac{2}{-\lambda}$  for stability
- This is called a time step restriction

# Stability (Example)

- Consider  $c' = -c$  with  $c(0) = 1$ , where  $\lambda = -1$  implies  $\Delta t < 2$  for stability



- Here,  $\Delta t = .5$  is stable
- Iterates (dots) track the solution (curve)



- Here,  $\Delta t = 3$  is unstable
- Iterates (dots) grow exponentially
- The actual solution (curve) is shown decaying

# Gradient Flow

- Using forward Euler on the gradient flow ODE gives:  $c^{q+1} = c^q - \Delta t \nabla \hat{f}(c^q)$
- This is the exact same formula utilized for 1D line search  $c^{q+1} = c^q + \Delta t \Delta c^q$  when using the (greedy) steepest descent search direction  $\Delta c^q = -\nabla \hat{f}(c^q)$
- When line search is used, a 1D root/minimization approach is used to determine the next iterate
- This forward Euler interpretation suggests that one may instead choose  $\Delta t$  according to various ODE (or other similar) considerations

# Adaptive Time Stepping

- ODEs utilize either a fixed size  $\Delta t$  or time varying  $\Delta t^q$ , and the latter case is referred to as adaptive time stepping
- The ML community refer to  $\Delta t$  as the learning rate, and time steps as epochs
- When sub-iterations use only partially valid approximations of  $-\nabla \hat{f}(c^q)$ , e.g. mini-batch or SGD (unit 19), an epoch refers to one pass through the entire set of training data
  - i.e. each epoch allows  $-\nabla \hat{f}(c^q)$  estimates to see all the data

# Adaptive Learning Rates

- Adagrad maintains a separate adaptive learning rate for each parameter, and modifies them based on past gradients computed for that parameter
  - Moving more/less in certain directions (because of per-parameter learning rates) changes the search direction
- Since the learning rates are based on a time history, the method is less localized and hopefully more robust (better behaved)
- Unfortunately, the learning rates monotonically decrease and often go to zero (stalling out the algorithm)
- Adadelta and RMSprop decrease the effect of prior gradients so that the learning rate is not monotonically driven to zero

# Implicit Methods

- Used to take larger time steps (compared to forward Euler and RK methods)
- Implicit methods have either no time step restriction or a very generous one
- However, one typically requires a nonlinear solver to advance each time step
- Sometimes, the nonlinear solver requires more computational effort than all the smaller (and simpler) time steps of forward Euler and/or RK combined (making it less efficient)
- The large time steps often lead to overly damped solutions (or unwanted oscillations)

# Backward (Implicit) Euler

- $\frac{c^{q+1} - c^q}{\Delta t} = f(t^{q+1}, c^{q+1})$  is 1<sup>st</sup> order accurate with  $O(\Delta t)$  error
- Stability:  $\frac{c^{q+1} - c^q}{\Delta t} = \lambda c^{q+1}$  implies  $c^{q+1} = \frac{1}{1-\Delta t \lambda} c^q$  where  $0 < \left| \frac{1}{1-\Delta t \lambda} \right| < 1$ 
  - Thus, unconditionally stable since the inequality holds for all  $\Delta t$  (assuming  $\lambda < 0$ )
- Typically need to solve a nonlinear equation to find  $c^{q+1}$  (can be expensive)
- As  $\Delta t \rightarrow \infty$ , the method asymptotes to  $f(t^{q+1}, c^{q+1}) = 0$ , which is the correct steady state solution
  - But, overly damping makes one get there too fast, which is especially undesirable when the higher frequencies are important
- Great for stiff problems where high frequencies don't contribute much to the solution (and thus overly damping them is fine)

# Implicit Stochastic Gradient Descent (ISGD)

- Used in Nonlinear Least Squares to overcome instabilities caused by using large time steps with forward Euler
- Forward Euler:  $c^{q+1} = c^q - \Delta t \nabla \hat{f}(c^q)$
- Backward (implicit) Euler:  $c^{q+1} = c^q - \Delta t \nabla \hat{f}(c^{q+1})$
- Since SGD only evaluates the gradient for one piece of data at a time, evaluating the gradient implicitly is a bit less unwieldy (as compared to doing so using all the data at the same time)

# Trapezoidal Rule

- $\frac{c^{q+1} - c^q}{\Delta t} = \frac{f(t^q, c^q) + f(t^{q+1}, c^{q+1})}{2}$  is 2<sup>nd</sup> order accurate with  $O(\Delta t^2)$  error
  - Averages forward Euler and backward Euler
- Stability:  $\frac{c^{q+1} - c^q}{\Delta t} = \frac{\lambda c^q + \lambda c^{q+1}}{2}$  implies  $c^{q+1} = \frac{1 + \frac{\Delta t \lambda}{2}}{1 - \frac{\Delta t \lambda}{2}} c^q$  where  $0 < \left| \frac{1 + \frac{\Delta t \lambda}{2}}{1 - \frac{\Delta t \lambda}{2}} \right| < 1$ 
  - Thus, unconditionally stable since the inequality holds for all  $\Delta t$  (assuming  $\lambda < 0$ )
- Typically need to solve a nonlinear equation to find  $c^{q+1}$  (can be expensive)
- As  $\Delta t \rightarrow \infty$ , the method asymptotes to  $f(t^{q+1}, c^{q+1}) = -f(t^q, c^q)$  which can cause unwanted oscillations
  - E.g., when  $c' = \lambda c$ , this is  $c^{q+1} = -c^q$  which is oscillatory
  - More generally for  $c' = f(t, c)$ , this is  $(c')^{q+1} = -(c')^q$  estimating the derivative as changing sign every iteration (causing oscillations)

# Momentum

- Optimization methods often struggle when they are too local
- Adaptive learning rates based on time history (as discussed above) help to address this
- Momentum methods also aim to address this
- Momentum methods derive their motivation from Newton's Second Law
- Physical objects carry a time history of past interactions via their momentum
- The forces currently being applied to an object are combined with all previous forces to obtain the current trajectory/velocity

# Newton's Second Law

- Kinematics describe position  $X(t)$ , velocity  $V(t)$ , acceleration  $A(t)$  as functions of time  $t$  via  $\frac{dX}{dt}(t) = V(t)$  and  $\frac{dV}{dt}(t) = A(t)$ 
  - Gradient flow  $\frac{dc}{dt}(t) = -\nabla \hat{f}(c(t))$  is a kinematic equation
- Dynamics describe responses to external forces
  - Newton's second law  $F(t) = MA(t)$  is a dynamics equation
  - $V'(t) = A(t) = \frac{F(t)}{M}$  or  $\frac{d^2X}{dt^2}(t) = X''(t) = \frac{F(t)}{M}$
- Combining kinematics and dynamics gives: 
$$\begin{pmatrix} X'(t) \\ V'(t) \end{pmatrix} = \begin{pmatrix} V(t) \\ \frac{F(t, X(t), V(t))}{M} \end{pmatrix}$$

# Aside: First Order Systems

- Higher order ODEs are often reduced to first order systems
  - E.g. consider:  $c'''' = f(t, c, c', c'', c''')$
  - Define new variables:  $c_1 = c, c_2 = c', c_3 = c'',$  and  $c_4 = c'''$
  - Then  $\begin{pmatrix} c_1' \\ c_2' \\ c_3' \\ c_4' \end{pmatrix} = \begin{pmatrix} c_2 \\ c_3 \\ c_4 \\ f(t, c_1, c_2, c_3, c_4) \end{pmatrix}$  is an equivalent first order system
- Newton's second law  $F = MX''$  can be written as  $\begin{pmatrix} X' \\ V' \end{pmatrix} = \begin{pmatrix} V \\ F/M \end{pmatrix}$

# Momentum Methods

- Newton's second law:  $\begin{pmatrix} X'(t) \\ MV'(t) \end{pmatrix} = \begin{pmatrix} V(t) \\ F(t, X(t), V(t)) \end{pmatrix}$ 
  - The second equation augments the momentum with the current forces
  - That momentum is used in the first equation (after dividing by mass to get a velocity)
- Interpreting this from an optimization standpoint:
  - Instead of always using the current search direction, one should still be incorporating the effects of prior search directions
- This makes the optimization method less localized, and hopefully more robust (better behaved)

# (Momentum-Style) Gradient Flow

- Split the forward Euler discretization  $c^{q+1} = c^q - \Delta t \nabla \hat{f}(c^q)$  into two parts:
$$c^{q+1} = c^q + \Delta t v^q \quad \text{and} \quad v^q = -\nabla \hat{f}(c^q)$$
- Here,  $v^q$  is a velocity in parameter space
- Instead of setting the velocity equal to the (negative) gradient, treat gradients as forces that affect the velocity:

$$v^{q+1} = v^q - \Delta t \nabla \hat{f}(c^q)$$

- This results in a forward Euler discretization of  $\begin{pmatrix} c'(t) \\ v'(t) \end{pmatrix} = \begin{pmatrix} v(t) \\ -\nabla \hat{f}(c^q) \end{pmatrix}$

# “The” ML Momentum Method

- The original momentum method is backward Euler on  $c$  and forward Euler on  $v$ , i.e.  $c^{q+1} = c^q + \Delta t v^{q+1}$  and  $v^{q+1} = v^q - \Delta t \nabla \hat{f}(c^q)$ 
  - Since the second equation can be updated first, the first equation doesn't require a special solver
- Combining these into a single equation:  $c^{q+1} = c^q + \Delta t v^q - \Delta t^2 \nabla \hat{f}(c^q)$
- Taking liberties to treat  $\Delta t$  and  $\Delta t^2$  as two separate independent parameters leads to:  $c^{q+1} = c^q + \alpha v^q - \beta \nabla \hat{f}(c^q)$
- Setting  $\beta = \Delta t$  recovers the original discretization of gradient flow augmented with a new history dependent velocity term:  $c^{q+1} = c^q + \alpha v^q - \Delta t \nabla \hat{f}(c^q)$ 
  - Writing this final equation as  $c^{q+1} = c^q + \Delta t v^{q+1}$  illustrates an inconsistent velocity update of  $v^{q+1} = \frac{\alpha}{\Delta t} v^q - \nabla \hat{f}(c^q)$

# Nesterov Momentum

- Uses a predictor-corrector approach similar to 2<sup>nd</sup> order Runge- Kutta
- First, a forward Euler predictor step is taken  $\hat{c}^{q+1} = c^q + \Delta t \hat{v}^{q+1}$  using a velocity of  $\hat{v}^{q+1} = \frac{\alpha}{\Delta t} v^q$  (instead of  $v^{q+1} = \frac{\alpha}{\Delta t} v^q - \nabla \hat{f}(c^q)$  from the last slide)
  - The current gradient information is ignored in the predictor step
  - Simplifying, the predictor step is  $\hat{c}^{q+1} = c^q + \alpha v^q$
- Then, the gradient is evaluated at this new location  $\hat{c}^{q+1}$  and used in “The” ML Momentum method:  $c^{q+1} = c^q + \Delta t v^{q+1}$  and  $v^{q+1} = \frac{\alpha}{\Delta t} v^q - \nabla \hat{f}(\hat{c}^{q+1})$ 
  - As a single equation:  $c^{q+1} = c^q + \alpha v^q - \Delta t \nabla \hat{f}(\hat{c}^{q+1})$
  - Once again, there is an inconsistent velocity update  $v^{q+1} = \frac{\alpha}{\Delta t} v^q - \nabla \hat{f}(\hat{c}^{q+1})$

# Physics/ODE Consistency

- Numerical ODE theory dictates (via consistency with the Taylor expansion) that the correct solution/path should be obtained as  $\Delta t \rightarrow 0$ 
  - $c^{q+1} = c^q + \Delta t v^{q+1}$  properly resolves  $c' = v$
  - But,  $v^{q+1} = \frac{\alpha}{\Delta t} v^q - \nabla \hat{f}(\tilde{c})$  (with  $\tilde{c}$  either  $c^q$  or  $\hat{c}^{q+1}$ ) is problematic
- Revert to where we took liberties with  $c^{q+1} = c^q + \alpha v^q - \beta \nabla \hat{f}(\tilde{c})$
- Choose  $\beta = \hat{\beta} \Delta t^2$  (instead of  $\beta = \Delta t$ ) to obtain  $v^{q+1} = \frac{\alpha}{\Delta t} v^q - \Delta t \hat{\beta} \nabla \hat{f}(\tilde{c})$
- Setting  $\alpha = \Delta t$  leads to a consistent  $v^{q+1} = v^q - \Delta t \hat{\beta} \nabla \hat{f}(\tilde{c})$  where  $\hat{\beta} > 0$  determines the strength of the steepest descent force
  - Forces (in physical systems) should be independent of  $\Delta t$ , and should accumulate to the same  $O(1)$  net effect in  $O(1)$  time (regardless of  $\Delta t$ )

# Adam

- Mixes ideas from adaptive learning rates and momentum methods:
  - Adaptive learning rate for each parameter (uses squared gradients to scale the learning rate, like RMSprop)
  - Uses a moving average of the gradient, like momentum methods
- AdaMax variant uses the  $L^\infty$  norm instead of the  $L^2$  norm
- Nadam variant uses Nesterov momentum for the moving averages
- The original Adam paper had impressive results, which were duplicated by others, and the method has been quite popular
- Some recent work states that Adam might converge quicker than SGD w/momentum, but sometimes quicker to a worse solution (and so some practitioners are going back to SGD)
  - Still a lot to do!

# Adam: A Method for Stochastic Optimization

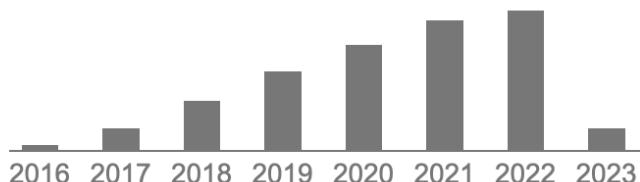
Authors Diederik P. Kingma, Jimmy Ba

Publication date 2014/12/22

Journal Proceedings of the 3rd International Conference on Learning Representations (ICLR)

Description We introduce Adam, an algorithm for first-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower-order moments. The method is straightforward to implement, is computationally efficient, has little memory requirements, is invariant to diagonal rescaling of the gradients, and is well suited for problems that are large in terms of data and/or parameters. The method is also appropriate for non-stationary objectives and problems with very noisy and/or sparse gradients. The hyper-parameters have intuitive interpretations and typically require little tuning. Some connections to related algorithms, on which Adam was inspired, are discussed. We also analyze the theoretical convergence properties of the algorithm and provide a regret bound on the convergence rate that is comparable to the best known results under the online convex optimization framework. Empirical results demonstrate that Adam works well in practice and compares favorably to other stochastic optimization methods. Finally, we discuss AdaMax, a variant of Adam based on the infinity norm.

Total citations Cited by 138431



# Constant Acceleration Equations

- Taylor expansion:  $X^{q+1} = X^q + \Delta t V^q + \frac{\Delta t^2}{2} A^q + O(\Delta t^3)$
- In order to determine  $X^{q+1}$  with  $O(\Delta t^3)$  accuracy, one only needs  $V^q$  with  $O(\Delta t^2)$  accuracy and  $A^q$  with  $O(\Delta t)$  accuracy
- In the system of equations for Newtons second law,  $V' = F/M$  requires  $O(\Delta t)$  less accuracy than  $X' = V$  requires
- The standard kinematic formulas in basic physics use:
  - piecewise constant accelerations  $A^q$
  - piecewise linear velocities  $V^{q+1} = V^q + \Delta t A^q$
  - piecewise quadratic positions  $X^{q+1} = X^q + \Delta t V^q + \frac{\Delta t^2}{2} A^q$

# Newmark Methods

- $X^{q+1} = X^q + \Delta t V^q + \frac{\Delta t^2}{2} ((1 - 2\beta)A^q + 2\beta A^{q+1})$
- $V^{q+1} = V^q + \Delta t ((1 - \gamma)A^q + \gamma A^{q+1})$
- $\beta = \gamma = 0$  constant acceleration equations (on the last slide)
- Second order accurate if and only if  $\gamma = \frac{1}{2}$ , i.e.  $V^{q+1} = V^q + \Delta t \frac{A^q + A^{q+1}}{2}$
- $\gamma = \frac{1}{2}, \beta = \frac{1}{4}$  is Trapezoidal Rule (on both  $X$  and  $V$ )
  - $X^{q+1} = X^q + \Delta t V^q + \frac{\Delta t^2}{4} (A^q + A^{q+1})$  becomes  $X^{q+1} = X^q + \Delta t V^q + \frac{\Delta t}{2} (V^{q+1} - V^q)$  or  
 $X^{q+1} = X^q + \Delta t \frac{V^q + V^{q+1}}{2}$
- $\gamma = \frac{1}{2}, \beta = 0$  is Central Differencing:  $X^{q+1} = X^q + \Delta t V^q + \frac{\Delta t^2}{2} A^q$

# Central Differentiating

- $X^{q+1} = X^q + \Delta t V^q + \frac{\Delta t^2}{2} A^q$  and  $V^{q+1} = V^q + \Delta t \frac{A^q + A^{q+1}}{2}$
- Adding  $X^{q+2} = X^{q+1} + \Delta t V^{q+1} + \frac{\Delta t^2}{2} A^{q+1}$  to  $X^{q+1} = X^q + \Delta t V^q + \frac{\Delta t^2}{2} A^q$  gives  
$$X^{q+2} - X^q = \Delta t(V^q + V^{q+1}) + \frac{\Delta t^2}{2}(A^q + A^{q+1}) = \Delta t(V^q + V^{q+1}) + \Delta t(V^{q+1} - V^q) = 2\Delta t V^{q+1}$$
- So  $V^{q+1} = \frac{X^{q+2} - X^q}{2\Delta t}$  (a second order accurate central difference)
- Subtracting (same equations) gives  $X^{q+2} - 2X^{q+1} + X^q = \Delta t(V^{q+1} - V^q) + \frac{\Delta t^2}{2}(A^{q+1} - A^q) = \frac{\Delta t^2}{2}(A^q + A^{q+1}) + \frac{\Delta t^2}{2}(A^{q+1} - A^q) = \Delta t^2 A^{q+1}$
- So  $A^{q+1} = \frac{X^{q+2} - 2X^{q+1} + X^q}{\Delta t^2}$  (a second order accurate central difference)

# Staggered Position and Velocity

- Update position with a staggered velocity  $X^{q+1} = X^q + \Delta t V^{q+\frac{1}{2}}$
- Using averaging  $V^{q+1} = \frac{V^{q+\frac{1}{2}} + V^{q+\frac{3}{2}}}{2}$  which still equals  $\frac{X^{q+2} - X^q}{2\Delta t}$  as desired
- $A^{q+1} = \frac{(X^{q+2} - X^{q+1}) - (X^{q+1} - X^q)}{\Delta t^2} = \frac{V^{q+\frac{3}{2}} - V^{q+\frac{1}{2}}}{\Delta t}$
- This last term is equal to both  $\frac{V^{q+1} - V^{q+\frac{1}{2}}}{(\Delta t/2)}$  and  $\frac{V^{q+\frac{3}{2}} - V^{q+1}}{(\Delta t/2)}$
- So  $V^{q+1} = V^{q+\frac{1}{2}} + \frac{\Delta t}{2} A^{q+1}$  and  $V^{q+\frac{3}{2}} = V^{q+1} + \frac{\Delta t}{2} A^{q+1}$
- The second equation shifted one index is  $V^{q+\frac{1}{2}} = V^q + \frac{\Delta t}{2} A^q$

# Staggered Central Differencing

- $V^{q+\frac{1}{2}} = V^q + \frac{\Delta t}{2} A(X^q, V^q)$  and  $X^{q+1} = X^q + \Delta t V^{q+\frac{1}{2}}$  are explicit
- $V^{q+1} = V^{q+\frac{1}{2}} + \frac{\Delta t}{2} A(X^{q+1}, V^{q+1})$  is explicit in  $X$  but implicit in  $V$
- Position based forces (e.g. elasticity) are typically nonlinear making them hard to invert (good that we don't have to), whereas velocity based forces (e.g. damping) are typically linear making them easier to invert (which we need to)
- Position based forces are often important for material behavior (good we don't overdamp them), whereas velocity based damping doesn't suffer much from increased damping (which we do if we switch from trapezoidal rule to backward Euler in the last step, i.e.  $V^{q+1} = V^q + \Delta t A(X^{q+1}, V^{q+1})$ )
- Position based forces don't require too stringent a time step restriction (good, because we need one), whereas velocity based forces typically require a very small time step restriction (which we can ignore with an implicit solve)

# Appendix

# Notation

# Unit 1: Intro

- $x, y, z$  are data inputs/outputs
- $A$  is a matrix ( $I$  for identity),  $b$  is the right hand side ( $y$  is used when the right hand side is the data)
- $i = 1, m$  subscript enumerates data (and thus rows of a matrix  $A$ )
- $f$  is function of the data
- $\hat{x}, \hat{y}, \hat{z}, \hat{f}, \hat{\phi}$  are inference/approximation of same variables or functions
- $c$  represents unknown parameters to characterize functions
- $k = 1, n$  subscript enumerates  $c$  (and thus columns of a matrix  $A$ )
- $a_k$  is column of  $A$
- $\Sigma_k$  is the sum over all  $k$ ,  $\Pi_{i \neq k}$  is the product over all  $i$  not equal to  $k$
- Quadratic Formula slide: uses standard notation for the quadratic formula
- $\phi$  are basis functions
- $\theta$  are pose parameters,  $\varphi$  represents all vertex positions of the cloth mesh
- $S$  are the skinned vertex positions of the body mesh,  $D$  is the displacement from the body mesh to the cloth mesh
- $u, v$  are the 2D texture space coordinate system,  $n$  is the (unit) normal direction
- $I$  is 2D RGB image data,  $\psi$  interpolates RGB values and converts them to a 3D displacement

# Unit 2: Linear Systems

- $R^n$  is an  $n$  dimensional Cartesian space (e.g.  $R^1, R^2, R^3$ )
- $a_{ik}$  is the element in row  $i$  and column  $k$  of  $A$
- $A^T$  is the transpose of matrix  $A$ , and  $A^{-1}$  is its inverse
- $\det A$  is the determinant of  $A$
- $\exists$  is "there exists", and  $\forall$  is "for all"
- $\hat{e}_i$  are the standard basis vectors, with a 1 in the  $i$ -th entry (and 0's elsewhere)
- Gaussian Elimination slides  $m_{ik}$  special column,  $M_{ik}, L_{ik}$  elimination matrices
- $I_{m \times m}$  is a size  $m \times m$  identity matrix
- $U$  upper triangular matrix,  $L$  lower triangular matrix
- $\hat{c}$  transformed version of  $c$
- $P$  permutation matrix (with its own special notation)

# Unit 3: Understanding Matrices

- $\lambda$  eigenvalue (scalar)
- $v$  eigenvector,  $u$  right eigenvector (both column vectors)
- $\alpha$  is a scalar
- $i = \sqrt{-1}$  when dealing with complex numbers
- \* superscript indicates a complex conjugate (for imaginary numbers)
- $\hat{b}, \tilde{b}, \hat{c}$  perturbed or transformed  $b, c$
- $\hat{A}^{-1}, \hat{I}$  approximate versions of  $A^{-1}, I$
- $U, V$  orthogonal (for SVD)
- $u_k, v_k$  are columns of  $U, V$
- $\Sigma$  diagonal (not necessarily square, potentially has zeros on the diagonal)
- $\sigma_k$  singular values (diagonal entries of  $\Sigma$ )

# Unit 4: Special Matrices

- $v, u$  column vectors
- $u \cdot v$  or  $\langle u, v \rangle$  is the inner product (or dot product) between  $u$  and  $v$
- $\langle u, v \rangle_A$  is the  $A$  weighted inner product
- $\Lambda$  is a diagonal matrix of eigenvalues
- $l_{ik}$  is an element of  $L$
- $\hat{A}$  is an approximation of  $A$

# Unit 5: Iterative Solvers

- $q$  superscript, integer for sequences/iterations (iterative solvers)
- $\epsilon$  small number
- $t$  time
- $X, V$  position and velocity
- $r, e$  residual and error (column vectors)
- $\hat{r}, \hat{e}$  are transformed versions of  $r, e$
- $s$  search direction
- $\alpha, \beta$  are scalars
- $\bar{S}$  column vector (potential search direction)

# Unit 6: Local Approximations

- $p$  is an integer for sequences, polynomial degree, order of accuracy
- $p!$  is  $p$  factorial
- $h$  scalar (relatively small)
- $f'$  and  $f''$  one derivative and two derivatives
- $f^{(p)}$  parenthesis (integer) indicates taking  $p$  derivatives
- $\phi$  basis functions
- $w$  weighting function

# Unit 7: Curse of Dimensionality

- $A, V$  area and volume
- $r$  radius
- $N$  integer, number of sample points
- $\vec{x}$  vector of data input to a function

# Unit 8: Least Squares

- False Statements (first slide):  $a, b$  scalars
- $D, \hat{D}$  diagonal matrices

# Unit 9: Basic Optimization

- $F$  system of functions (output is a vector not a scalar)
- $\partial$  partial derivative
- $J$  Jacobian matrix of all first partial derivatives
- $F'$  is the Jacobian of  $F$
- $\nabla f$  gradient of scalar function  $f$  (Jacobian transposed)
- $H$  matrix of all second partial derivatives of scalar function  $f$  (Jacobian of the gradient transposed)
- $c^*$  critical point (special value of  $c$ )
- $\tilde{A}$  matrix
- $\tilde{b}, \tilde{c}$  vectors

# Unit 10: Solving Least Squares

- $\hat{\Sigma}$  diagonal invertible matrix (no zeros on the diagonal)
- $I_{n \times n}$  stresses the size of the identity as  $n \times n$
- $\hat{b}_r, \hat{b}_z$  sub-vectors of  $\hat{b}$  of shorter length ( $r$  for range,  $z$  for zero)
- $\hat{Q}$  orthogonal matrix
- $Q, \tilde{Q}$  are tall matrices with orthonormal columns (subsets of an orthogonal matrix)
- $q_k$  column of  $Q$
- $R$  upper triangular matrix
- $r_{ik}$  entry of  $R$
- Householder slides:  $\hat{v}$  normal vector,  $H$  householder matrix,  $a$  column vector

# Unit 11: Zero Singular Values

- $c_r, c_z$  sub-vectors of  $\hat{c}$  of shorter length (range and zero abbreviations)
- $A^+$  pseudo-inverse of  $A$
- $T$  matrix (for similarity transforms)
- $Q^q$  is orthogonal and  $R^q$  is upper triangular
- Power Method Slides:  $A^q$  and  $\lambda^q$  are  $A$  and  $\lambda$  raised to the  $q$  power

# Unit 12: Regularization

- $\epsilon$  is a small positive number
- $c^*$  is an initial guess for  $c$
- $r$  used in its geometric series capacity (a scalar)
- $D$  is a diagonal matrix with all positive diagonal entries
- $a_k$  is a column of  $A$
- $\Theta$  is the angle between two vectors
- $\theta$  are pose parameters,  $\varphi$  represents all vertex positions of the face mesh
- $C^*$  are 2D curves (vertices connected by line segments) drawn on the image
- $C$  are 3D curves embedded on the 3D geometry, and subsequently projected into the 2D image space

# Unit 13: Optimization

- $f$  briefly is allowed to be either vector valued (or stay scalar valued)
- $\hat{f}$  is a (scalar) cost function for optimization
- $F$  is a system of functions (the gradient in the case of optimization)
- $\hat{g}$  is a vector valued function of constraints
- $\eta$  is a column vector of scalar Lagrange multipliers

# Unit 14: Nonlinear Systems

- $c^*$  is a point to linearize about
- $d$  is for the standard derivative
- $t$  is an arbitrary (scalar) variable
- $dc$  is a vanishingly small differential (of  $c$ )
- $\Delta$  finite size difference
- $\alpha, \beta$  are scalars with  $\beta \in [0,1)$
- $g$  scalar function (that determines the line search parameter  $\alpha$ )

# Unit 15: Root Finding

- $\hat{g}$  is a modified  $g$
- $t$  is search parameter in 1D, replacing  $\alpha$
- $t^*$  is the converged solution
- $e$  is the error
- $g'$  is the derivative of  $g$
- $\hat{t}$  is a particular  $t$
- $C \geq 0$  is a scalar
- $p$  integer (power)
- $t_L, t_R$  interval bounds
- $t_M$  interval midpoint

# Unit 16: 1D Optimization

- $t_{min}, t_{M1}, t_{M2}$  more  $t$  values
- $\delta$  scalar (interval size)
- $\lambda \in (0, .5)$  is a scalar
- $\tau \in (0,1)$  is a scalar
- $H_F$  is a 3<sup>rd</sup> order tensor of 2<sup>nd</sup> derivatives of  $F$
- $OMG_{\hat{f}}$  is a 3<sup>rd</sup> order tensor of 3<sup>rd</sup> derivatives of  $\hat{f}$

# Unit 17: Computing Derivatives

- $H$  is the Heaviside function
- $\hat{f}$  is a scalar function to be minimized
- $\hat{g}$  is a vector-valued function of constraints ( $\hat{g}_i$  is a component of  $\hat{g}$ )
- $\hat{e}_i$  is the  $i$ -th standard basis vector
- $n$  is a (possibly) high-dimensional unit normal
- $\epsilon > 0, b$  are scalars
- $e, \log$  are the usual exponential and logarithmic functions
- $C_1, C_2, C_3$  are different sets of parameters
- $f_1, f_2, f_3$  are different functions
- $X_1, X_2, X_3, X_4$  are the data as it is processed through the pipeline
- $X_{target}$  is the desired final result as the data is processed through the pipeline

# Unit 18: Avoiding Derivatives

- $\hat{m}$  is the integer length of the column vector output of  $f(x, y, c)$
- $\tilde{f}(c)$  is a column vector of size  $m * \hat{m}$  that stacks the  $\hat{m}$  outputs of  $f(x_i, y_i, c)$  for each of the  $m$  data points  $(x_i, y_i)$
- $\hat{e}_k$  is the standard basis vector

# Unit 19: Descent Methods

- (covered in other units)

# Unit 20: Momentum Methods

- $t$  is time
- $t_o, t_f$  initial and final time
- $\Delta t$  time step size
- $k_1, k_2, k_3, k_4$  intermediate function approximations in RK methods
- $\hat{c}$  intermediate states for TVD RK methods
- $\lambda$  is a scalar, and represents an eigenvalue
- $X(t), V(t), A(t), F(t), M$  position, velocity, acceleration, force, mass
- $v$  is the velocity of state  $c$  in parameter space
- $\alpha, \beta, \hat{\beta}$  are scalars