

Continuous Mathematical Methods, Emphasis on Machine Learning

Ron Fedkiw^{*1}, Yilin Zhu^{*23}, Winnie Lin^{*3}, Jane Wu^{*3}, Kevin Li^{*3}

^{*} Stanford University, ¹ Slide Design and Content (and Instructor), ² Slide Illustrator, ³ 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. $12+34=46$ since $1+3=4$ 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
- Add rules to deal with multi-digit numbers 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, generated procedurally, etc.

2) Model

- Derived from the Training Data in order to estimate new data points with (hopefully) minimal error
- Uses Algorithms, Statistical Reasoning, Rules, Networks, Etc.

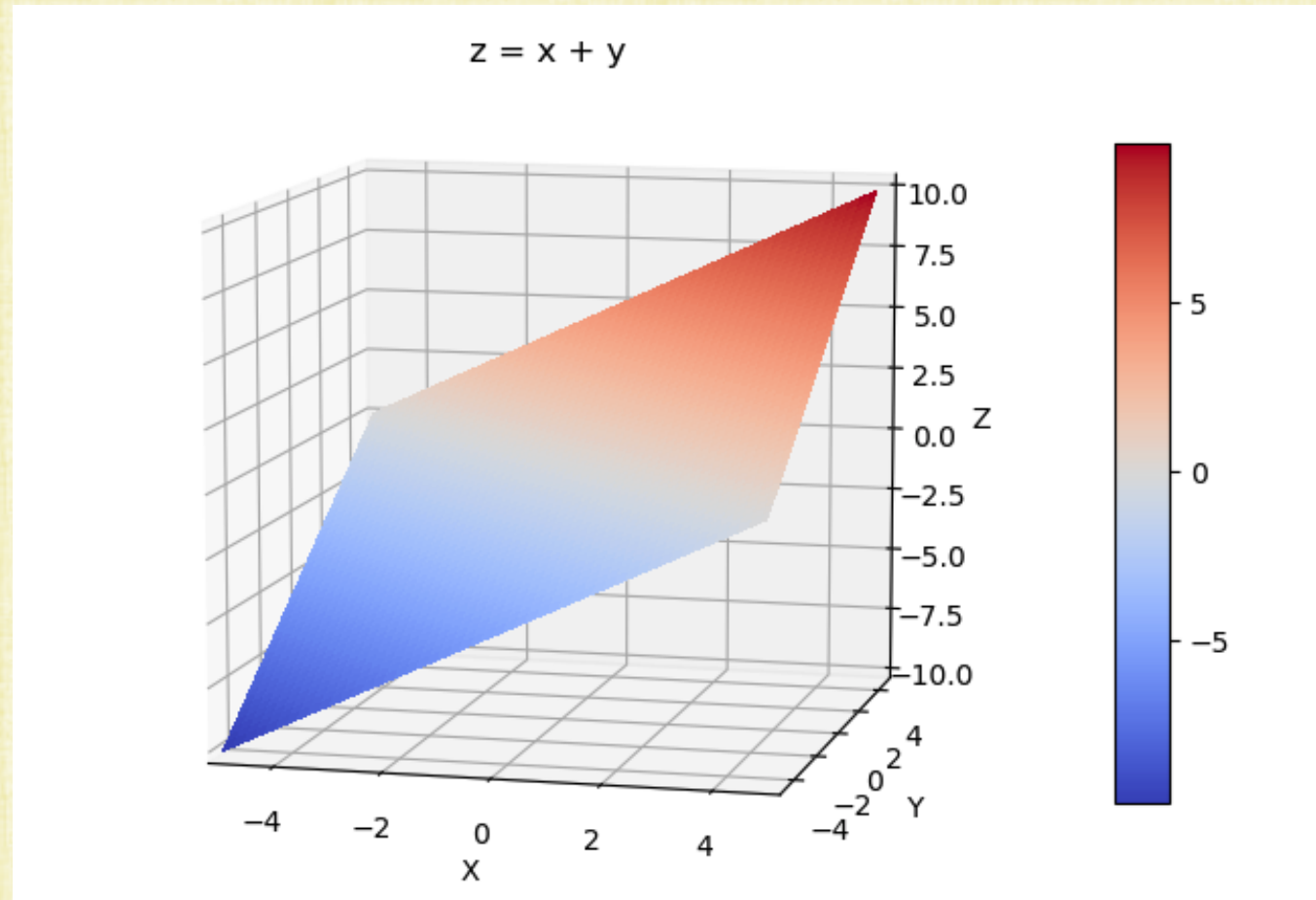
KBS vs. ML

- 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
 - Numerical algorithms are especially usefully for continuous approximations
- Since ML is derived from continuous math, it tends 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
- The plane $z = x + y$ exactly fits the training data
 - Only need 3 training points to determine a plane
- Don't need special rules for negative numbers, decimals, fractions, irrationals, 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)

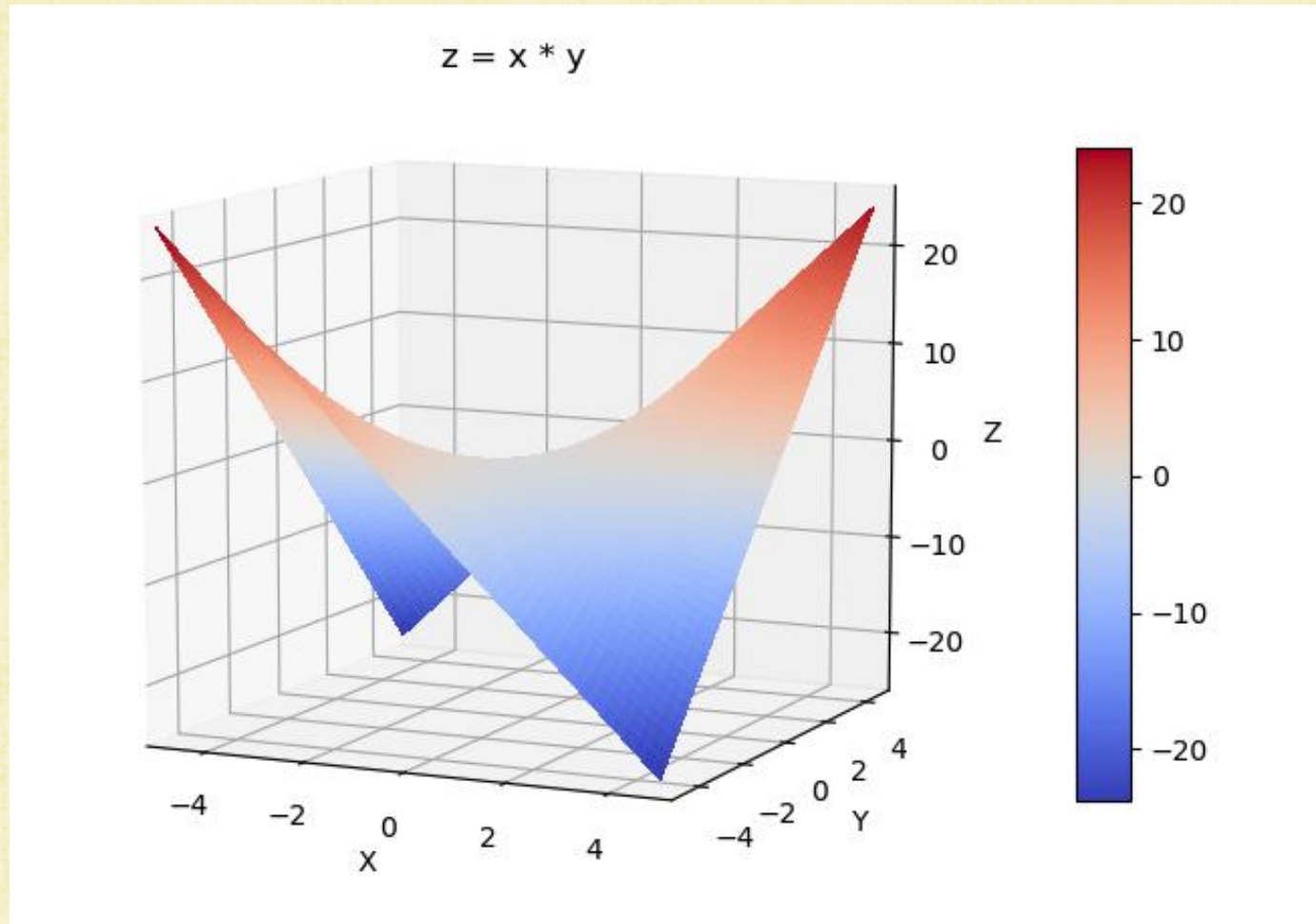
ML Approach to Addition



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 nonlinear model function $z = x * y$ can be found to fit the training data
 - Note: 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, one might be able to discover $z_i = x_i\#y_i$ for some (x_i, y_i) pairs
- 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

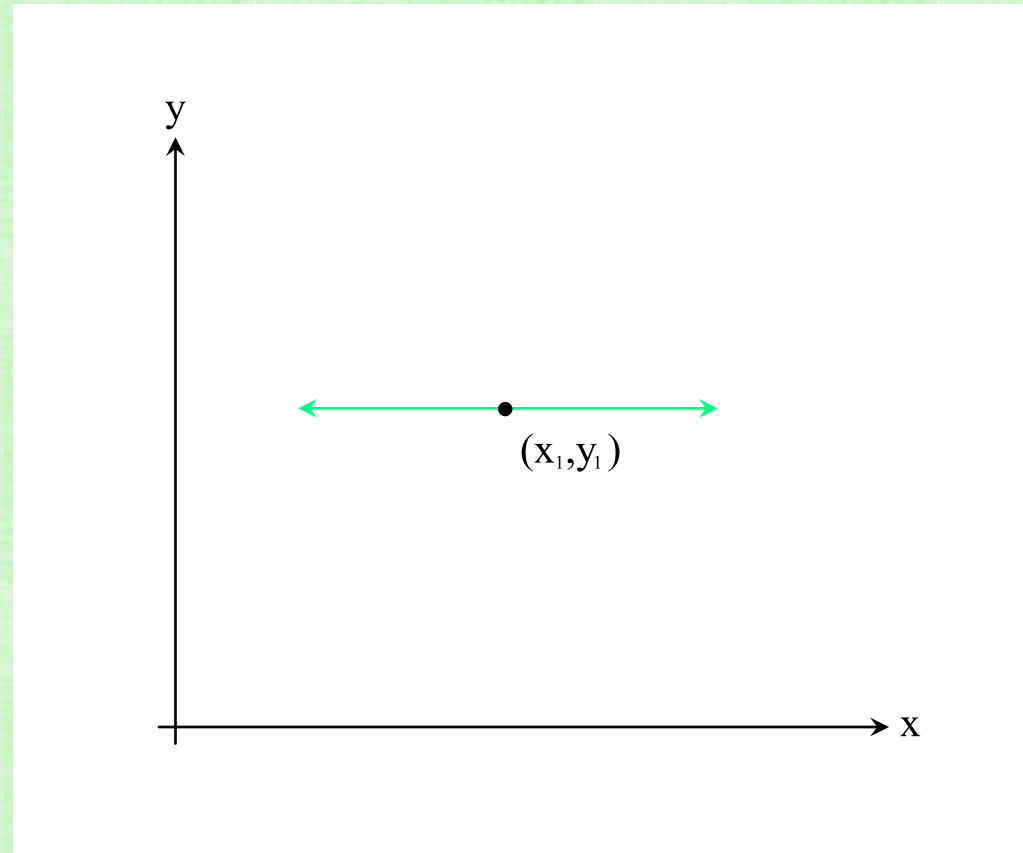
- If asked to multiply 51.023 times 298.5, one might quickly estimate that 50 times 300 is 15,000
- This is a nearest neighbor algorithm, relying on nearby data where the answer is known, better known, or easier to determine
- Given (\hat{x}, \hat{y}) , find the closest (Euclidean distance) training data (x_i, y_i) and return its 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 data interpolation in more detail
- Let's begin with a very simple case with $1D$ inputs and $1D$ outputs, i.e. $y = f(x)$

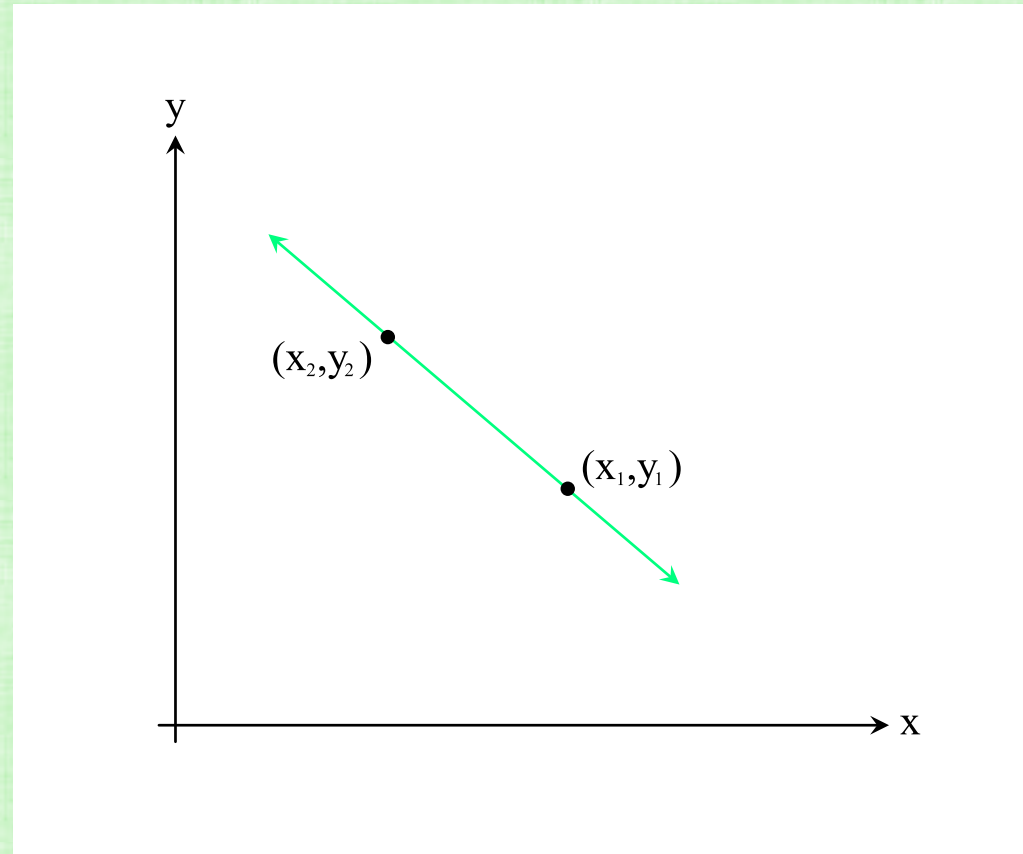
Polynomial Interpolation

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



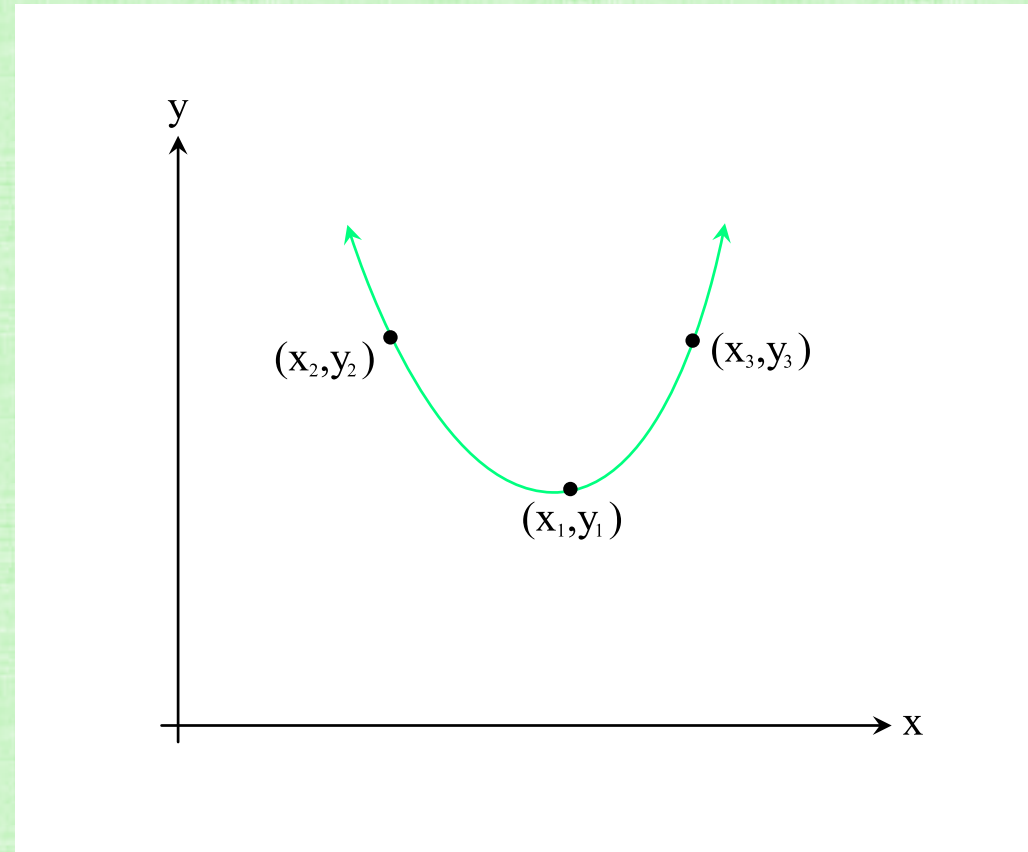
Polynomial Interpolation

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



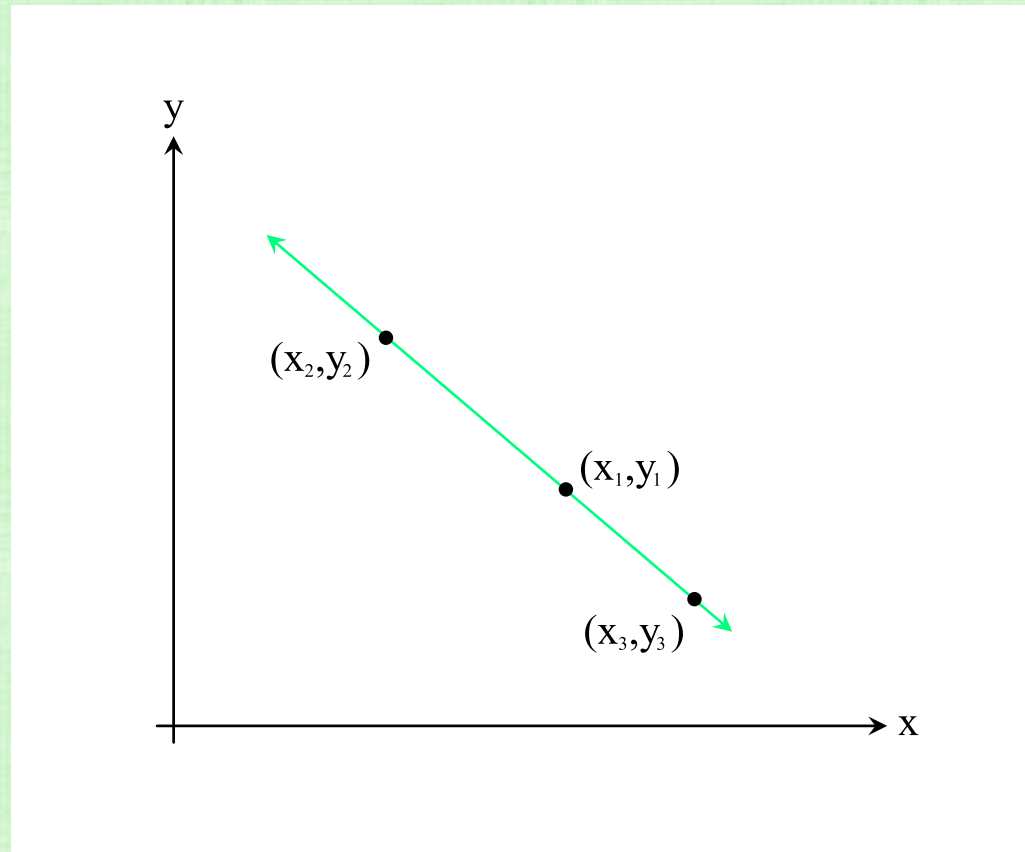
Polynomial Interpolation

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



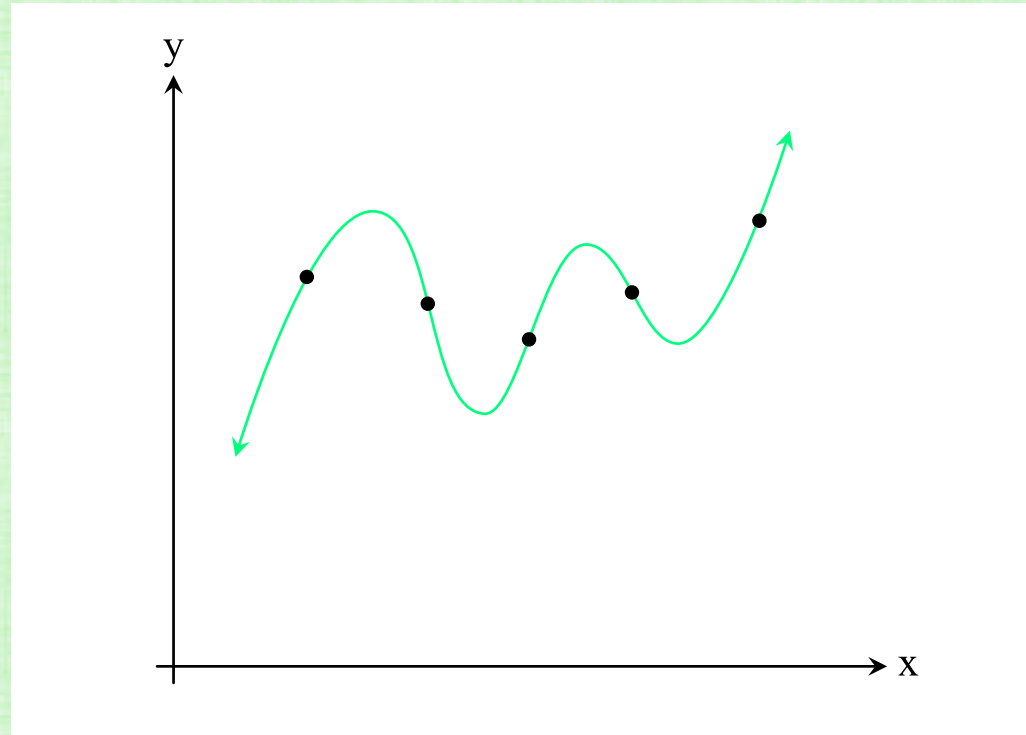
Polynomial Interpolation

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



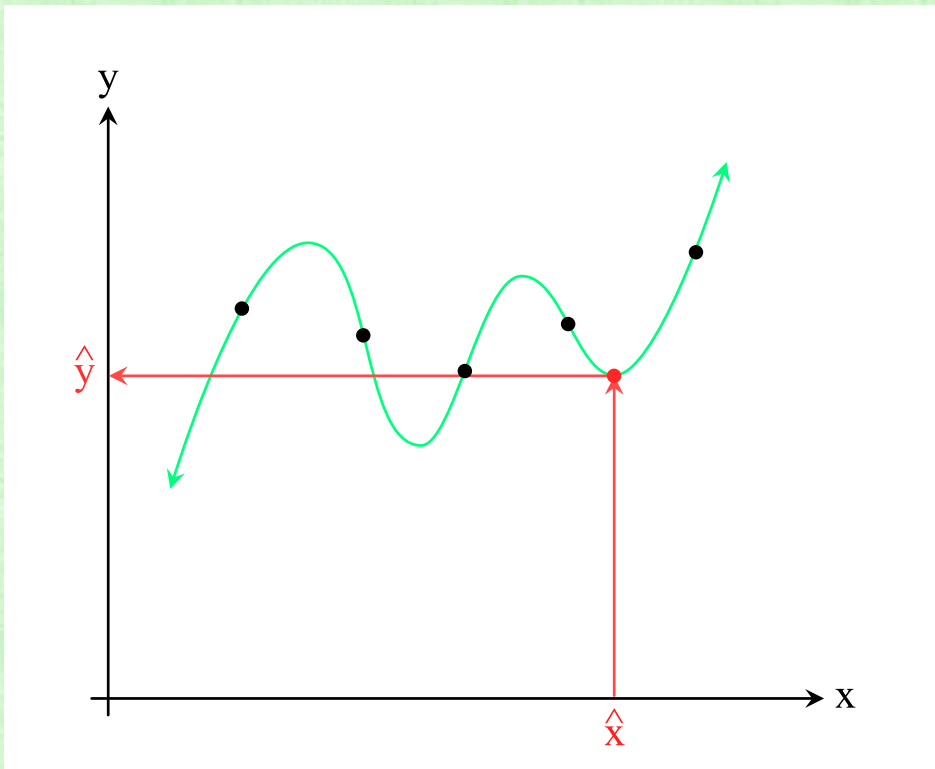
Polynomial Interpolation

- 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're not degenerate (e.g. 3 points on a line)



Overfitting

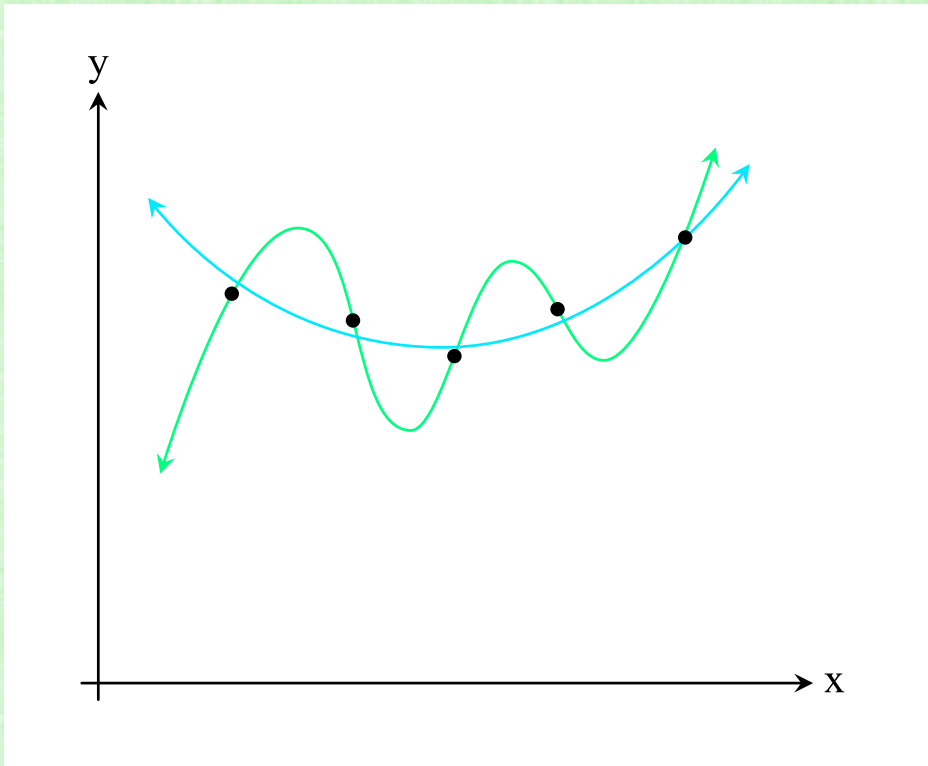
- Given a new input \hat{x} , the interpolating polynomial infers/predicts an output \hat{y} that may be far from what one may expect



- 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)
- Overfitting leads to inference/predictions that can vary wildly from the training data

Regularization

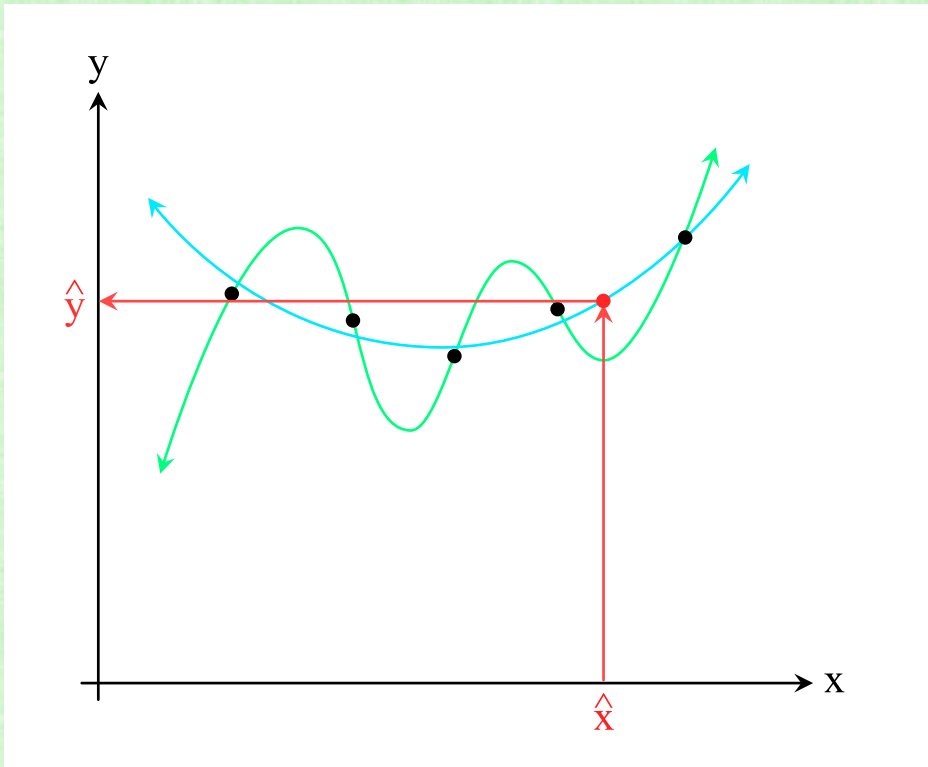
- Using a lower order polynomial that doesn't (can't) exactly fit the data points provides regularization



- A regularized interpolant contains intentional error, 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

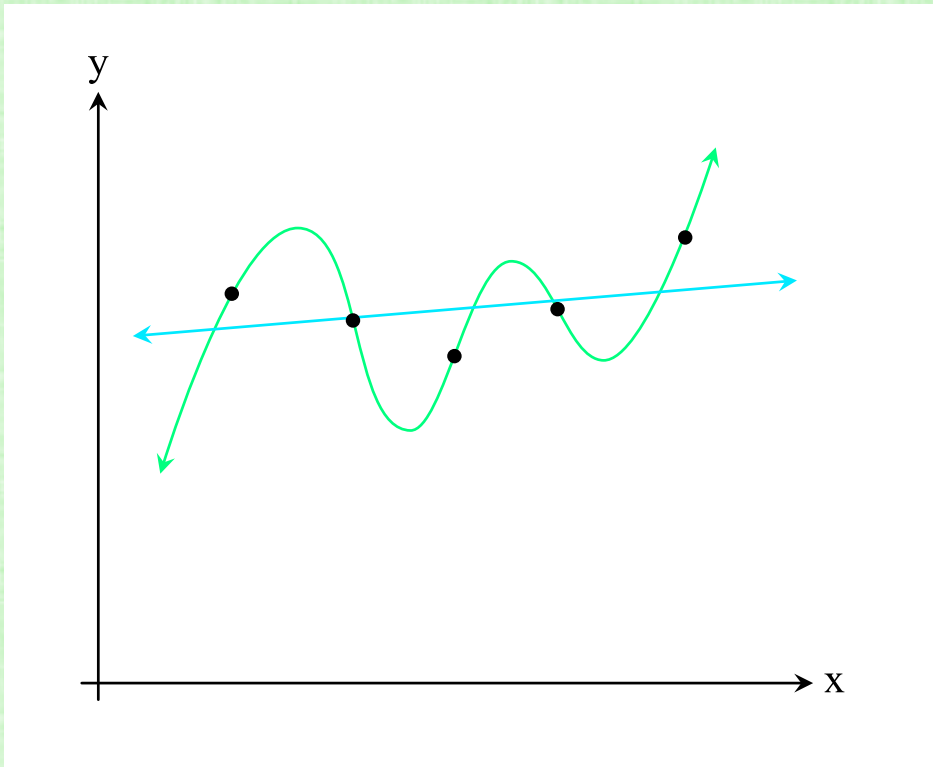
- Given \hat{x} , the regularized interpolant infers/predicts a more reasonable \hat{y}



- 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

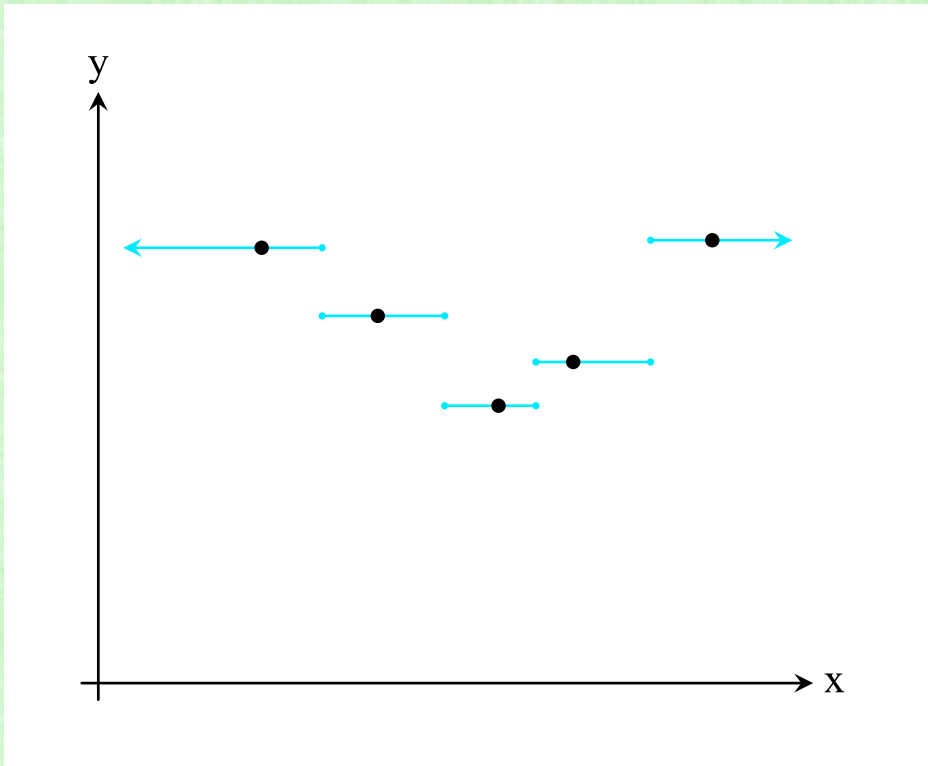
- Using too low of an order polynomial causes it to miss the data by too much



- 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

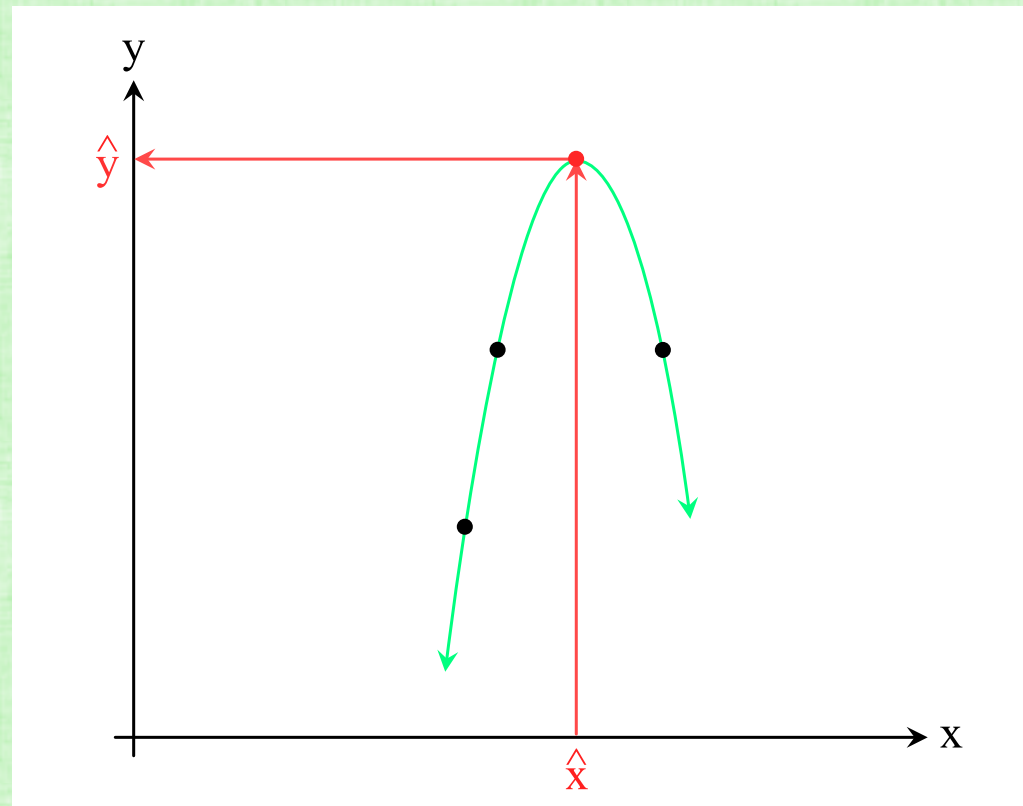
- Piecewise-constant interpolation (equivalent to nearest neighbor)



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

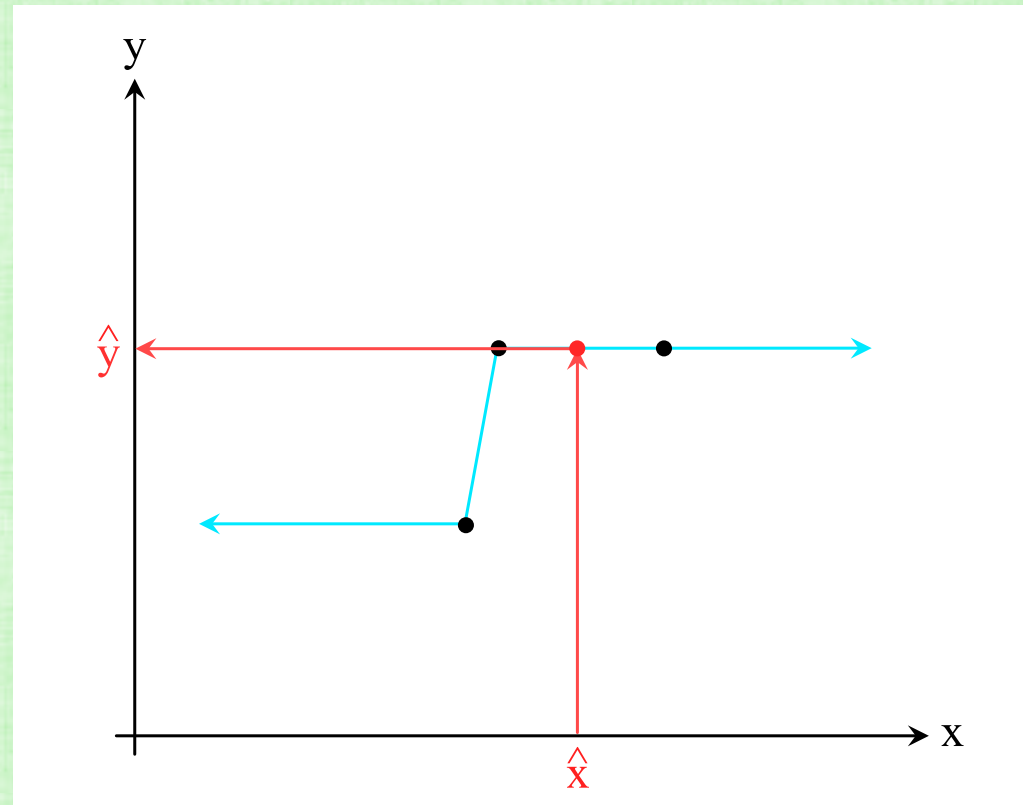
Caution: Overfitting

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



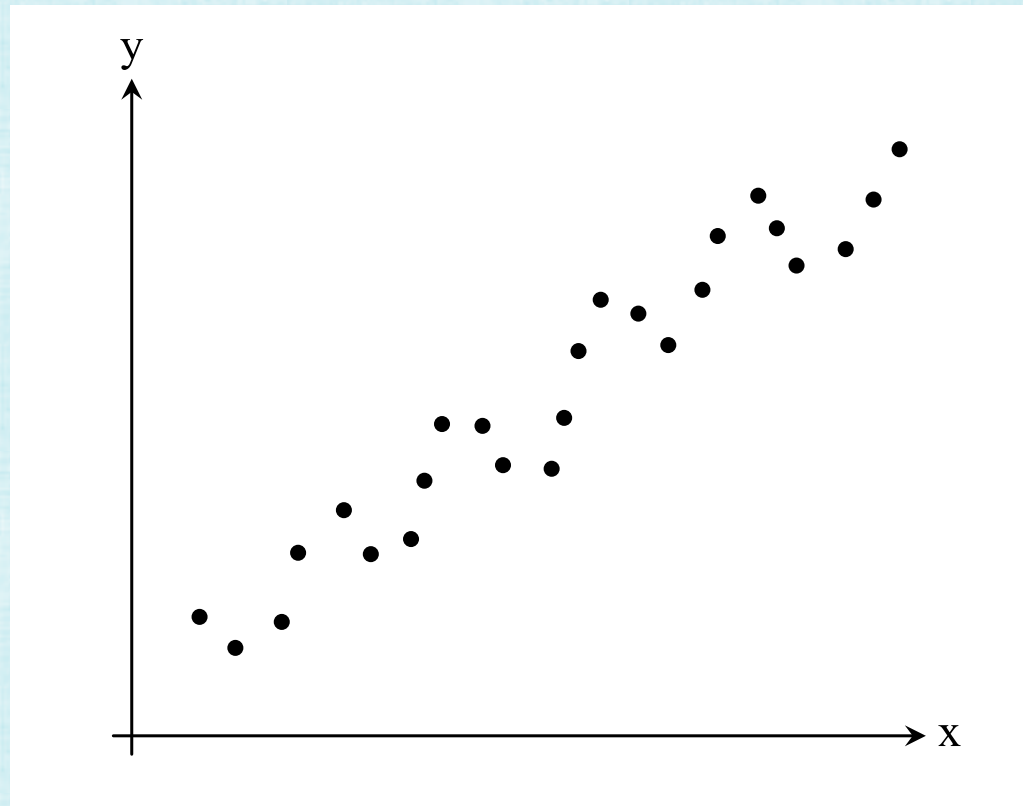
Caution: Overfitting

- A piecewise linear approach works much better on this data



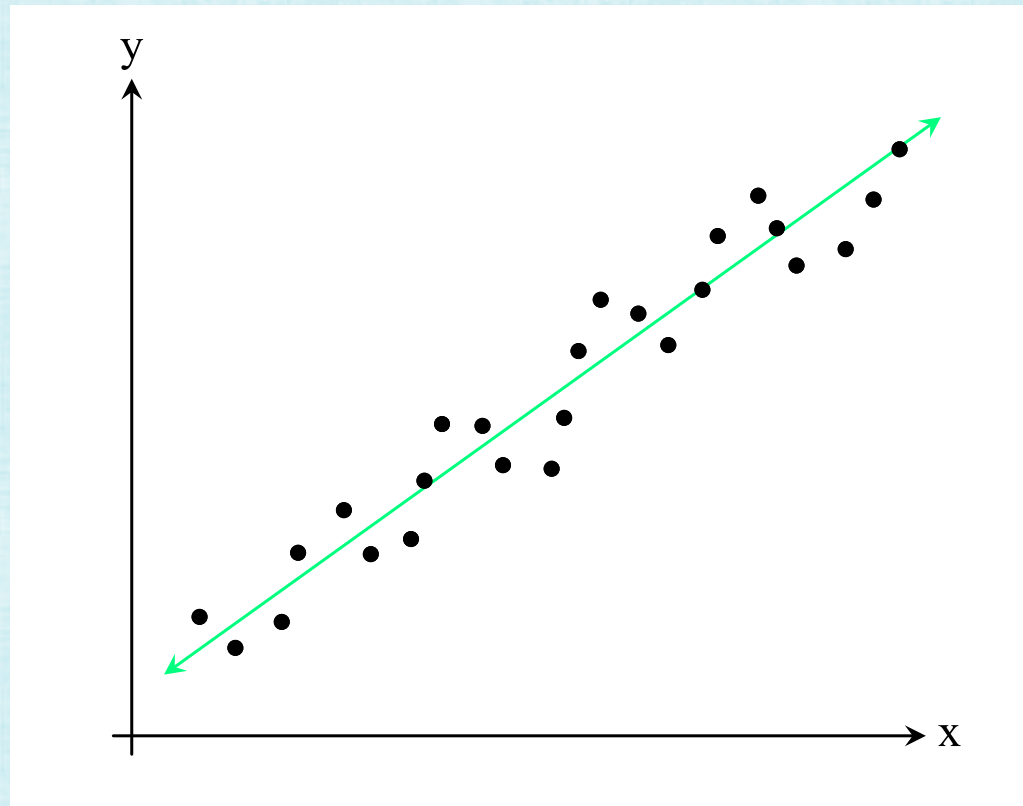
Noisy Data

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



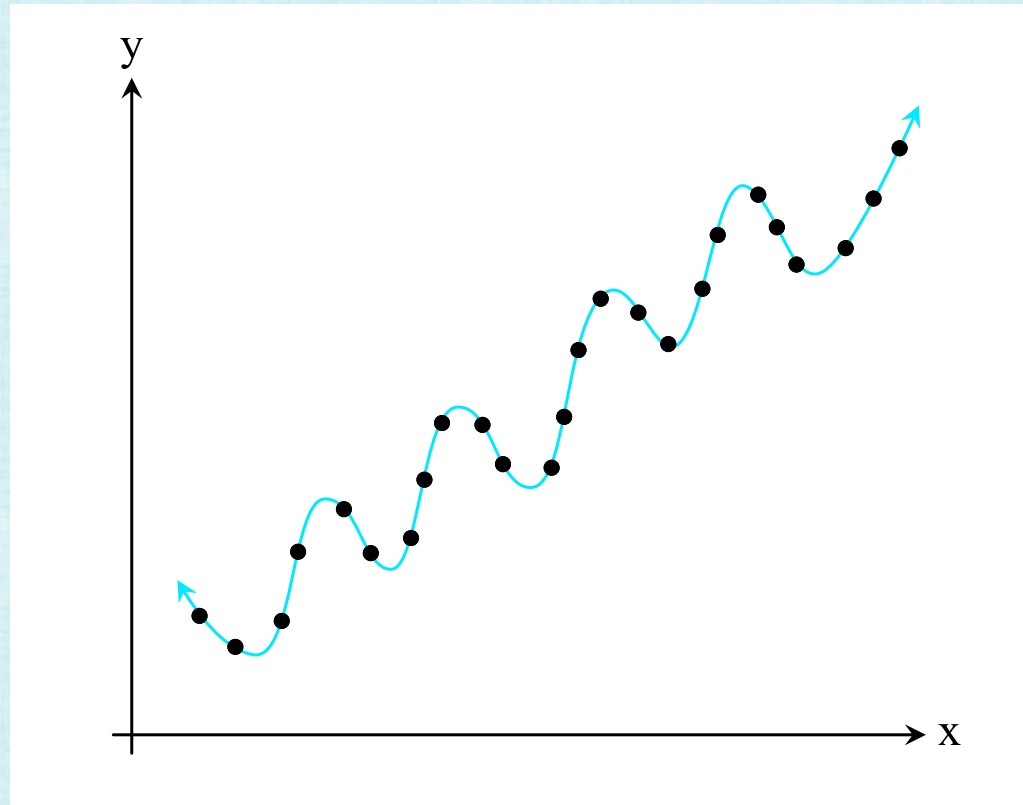
Linear Regression

- 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 does 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 the model validation data to the known answers
- When disseminating results advocating the “best” validated model, inferencing on the test data suggests how well the validated model might generalize to unseen data
 - Competitions on unseen data have become a good way to stop “cheating” on test data

Break until Thursday

Errors in Equations

- **Modeling errors** – Parts of a problem under consideration might be ignored. E.g., when simulating solids/fluids, sometimes frictional/viscous effects are not included or only approximated.
- **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. (The speed of light is 299792458 m/s exactly, so ok for double precision but not for single precision.)

Errors in Numerical Methods

- **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).

Errors in Inputs

- **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.
- **Inaccurate Measurements** – It can be difficult to accurately measure real-world phenomena, generating another source of inaccurate inputs.

A Robust Computational Approach

- **Well-Posedness**: A **problem** is ill-posed if small changes in the inputs lead to large changes in the outputs, implying that any source of error could dominate the results.
- **Condition Number**: An **algorithm** is ill-conditioned if small changes in the 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.
- **Stability**: An algorithm is stable if it can complete itself in any meaningful way. Unstable algorithms give wildly explosive results, usually leading to NaN's.
- **Accuracy**: Accuracy refers to the size of the error, or how close the answer is to the correct solution.

A Robust Computational Approach

- Only well-posed problems can be tackled computationally
- Computational Approach:
 - 1) Conditioning - formulate a well-conditioned approach, or as well-conditioned as is possible
 - 2) Stability - devise a stable algorithm; otherwise, the result could be NaNs
 - 3) Accuracy – even a well-conditioned and stable approach can result in large errors; so, make the algorithm as accurate as is warranted or practical

Being Careful: Vector Norms

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

$$\|x\|_2 = z \sqrt{\left(\frac{x_1}{z}\right)^2 + \dots + \left(\frac{x_m}{z}\right)^2}$$

find z=max(abs(x(i)))

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

Being Careful: Quadratic Formula

- Consider $.0501x^2 - 98.78x + 5.015 = 0$
 - To 10 digits of accuracy, the solutions are: $x \approx 1971.605916$ 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 the de-rationalized quadratic formula gives:

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

- The second root is fine, but the **first root is wrong** by a factor of two!
- Conclusion: use one formula for each root

Being Careful: L'Hopitals Rule

- Consider $\frac{x^2-4}{x-2}$ near $x = 2$ where it becomes $\frac{0}{0}$
- Adding a small number to the denominator $\frac{x^2-4}{x-2+\epsilon}$ incorrectly gives 0 near $x = 2$
- Using $\frac{x^2-4}{x-2} = x + 2$ leads to correct values near 4 when x is near 2
- Similar issues occur for $\frac{\sin x}{x}$ near $x = 0$
- Similar issues occur for 0 times ∞ and other cases where L'Hopitals rule is needed to address removable singularities

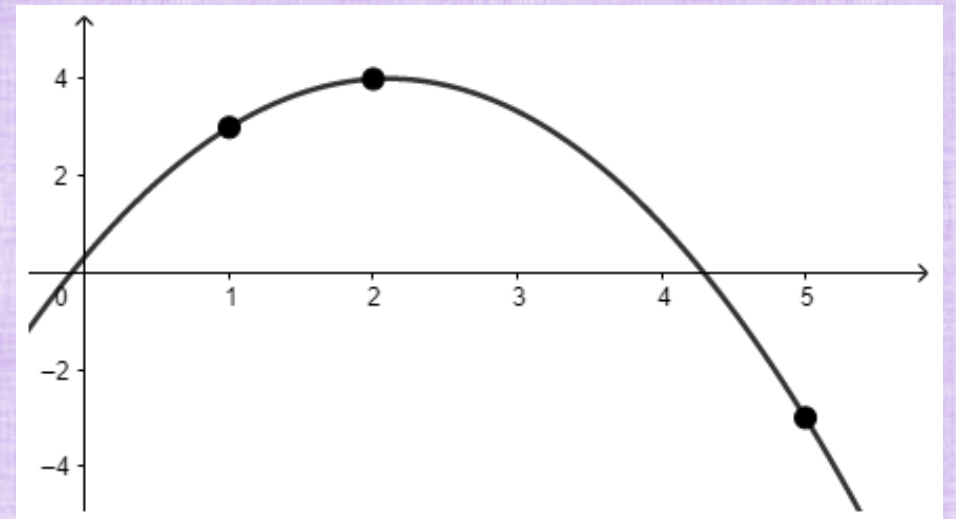
Did you know about these issues?

- Imagine debugging code with the correct quadratic formula implementation and getting zero digits of accuracy on a test case!

Polynomial Interpolation

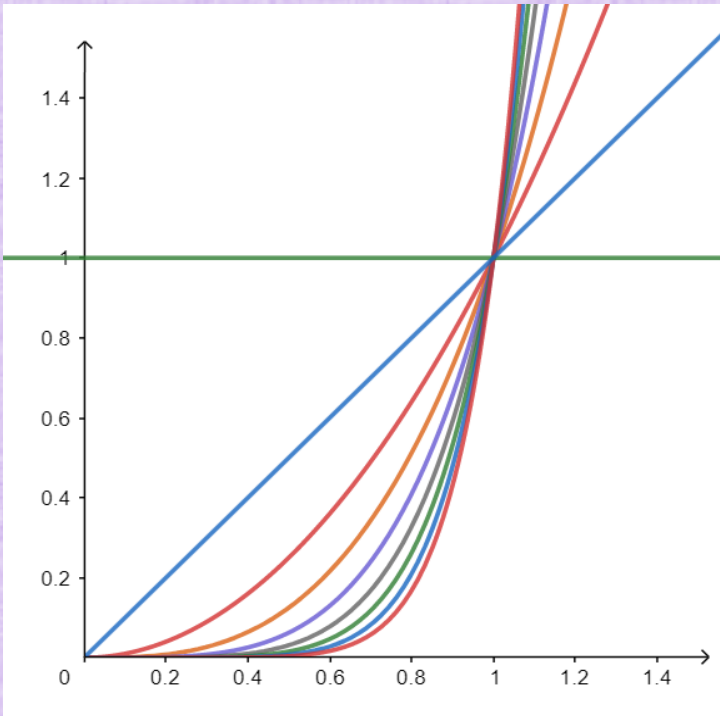
- Given m data points (x_i, y_i) , find the unique polynomial that passes through them: $y = c_1 + c_2x + c_3x^2 + \dots + c_mx^{m-1}$
- Write an equation for each data point, note that the equations are linear, and put into matrix form
- For example, consider $(1,3)$, $(2,4)$, $(5, -3)$ and a quadratic 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
$$\begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 1/3 \\ 7/2 \\ -5/6 \end{pmatrix}$$
 and $f(x) = \frac{1}{3} + \frac{7}{2}x - \frac{5}{6}x^2$



Polynomial Interpolation

- Solve $Ac = y$ where the Vandermonde matrix A has a row for each data point. Each row looks like $(1 \quad x_i \quad x_i^2 \quad \dots \quad x_i^{m-1})$

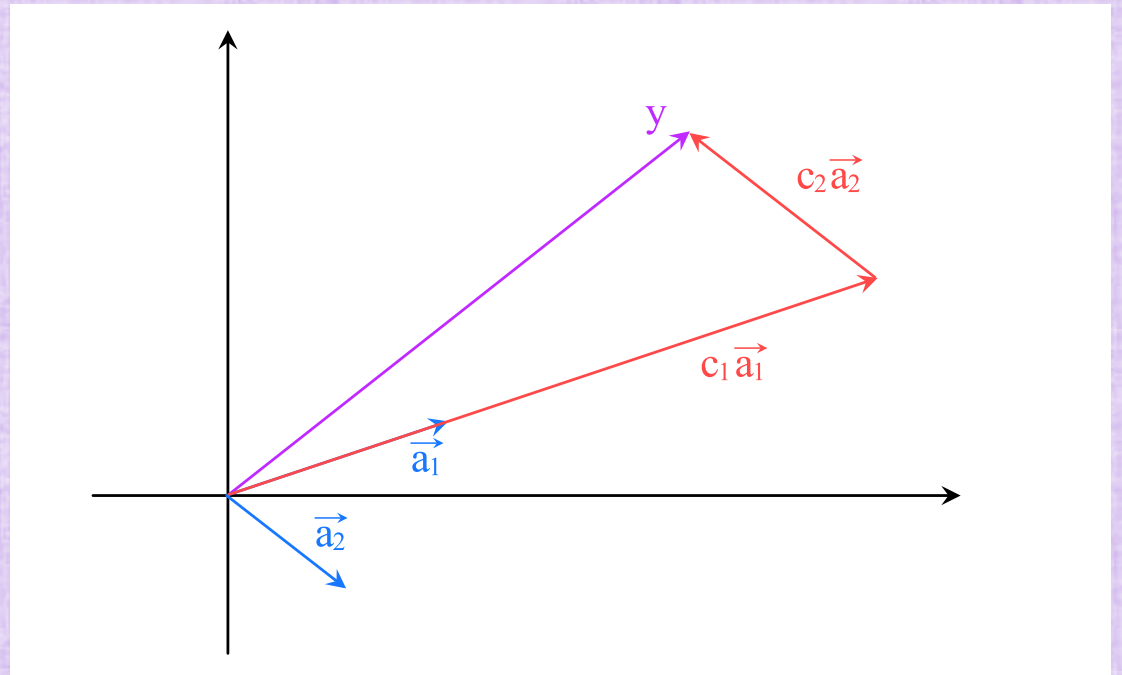
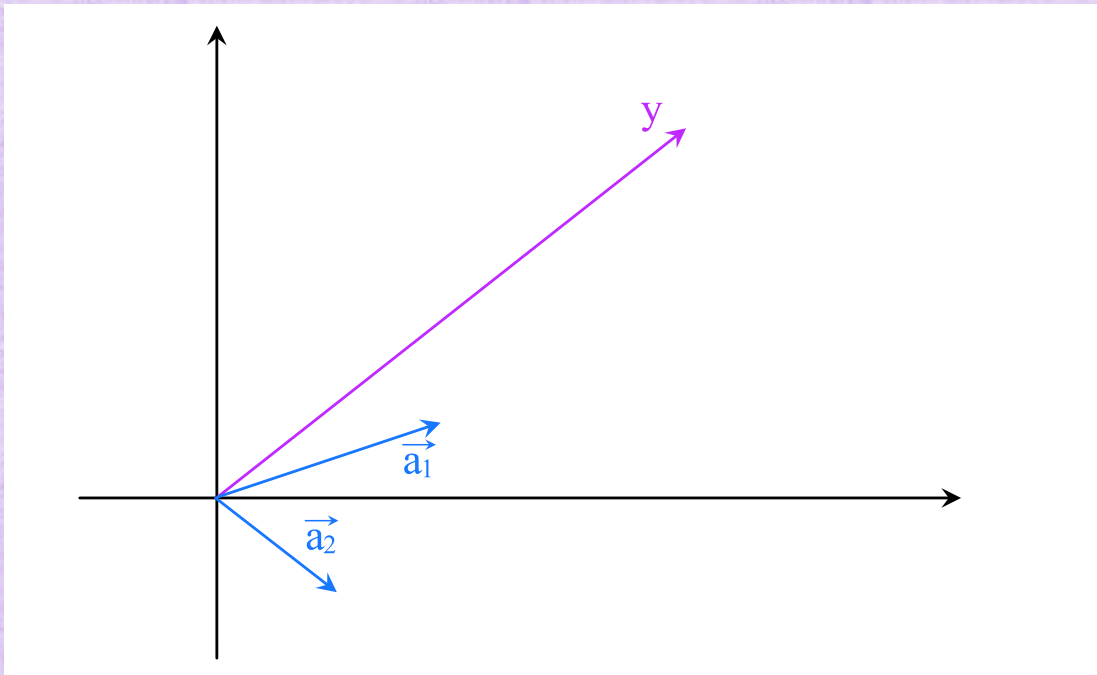


- Monomials look more similar to each other at higher powers
- This makes the rightmost columns of a Vandermonde matrix tend to become more parallel
- More parallel columns make the matrix less invertible, making it more difficult to solve for the c_k parameters
- More parallel columns make the matrix more **ill-conditioned** to invert, and it becomes difficult or impossible to invert it with a computer

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

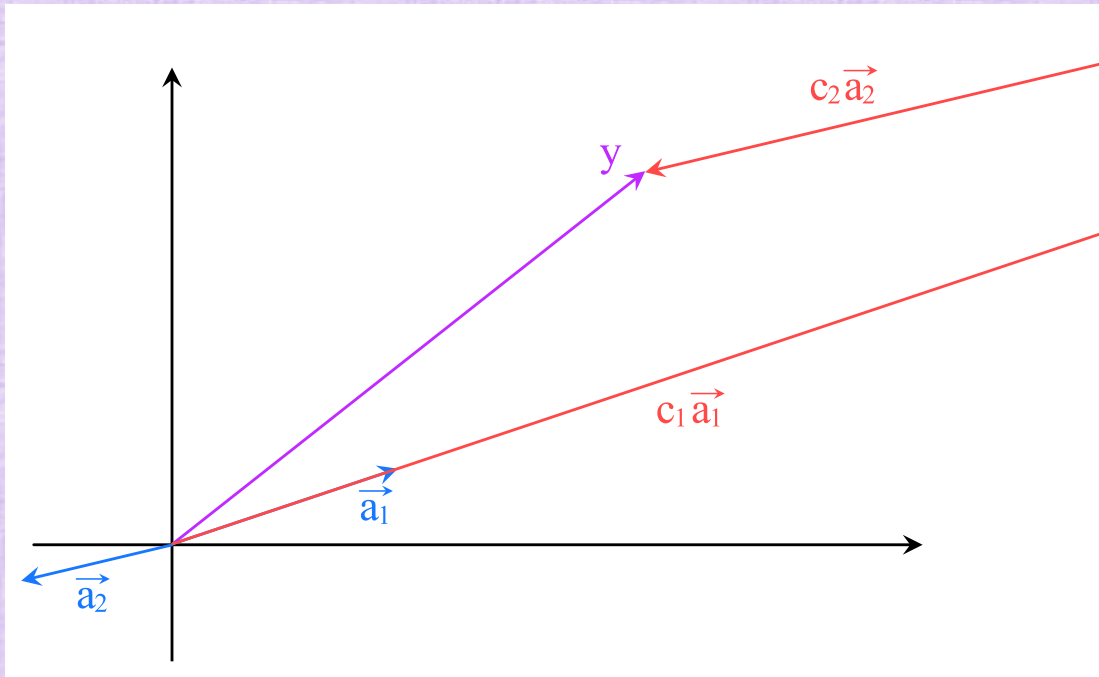
Matrix Columns as Vectors

- Define the k -th column of A as 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



- The **red** vectors go too far to the right 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
 - Thus, 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, 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
 - Thus, 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, the problem has no solution
 - Otherwise, there are infinite solutions

Near Singular Matrices

- With limited numerical precision, one struggles when columns (or linear combinations of columns) are too close to being parallel to each other
- Analytically invertible matrices may not be computationally invertible!
- A condition number can be used to describe how close a matrix is to being non-invertible on a computer
- The condition number is ∞ for a singular matrix and 1 for the identity matrix

Being Careful: Polynomial Interpolation

- Given basis functions ϕ and unknown parameters c :

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

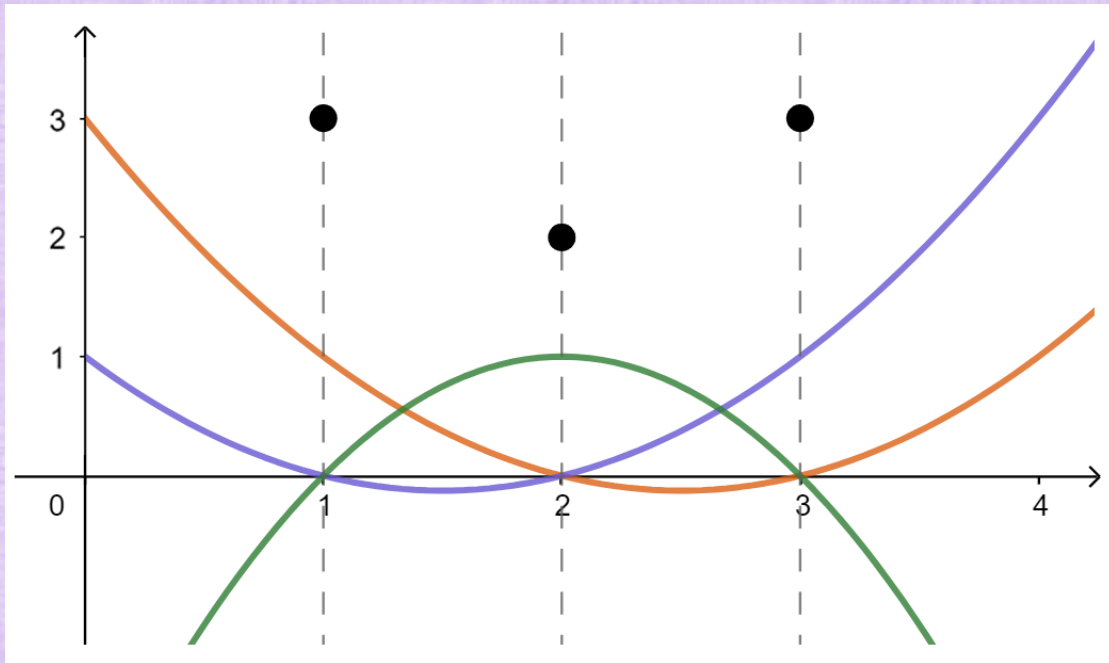
- Monomial basis: $\phi_k(x) = x^{k-1}$
- As we have seen, **the Vandermonde matrix may become near-singular and thus be computationally difficult to invert**

Lagrange Basis

- 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 now 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

- Consider $(1,3)$, $(2,2)$, $(3,3)$ 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

- Basis functions: $\phi_k(x) = \prod_{i=1}^{k-1} x - x_i$
- $Ac = y$ now has a lower triangular A (as opposed to being dense or diagonal)
- The 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

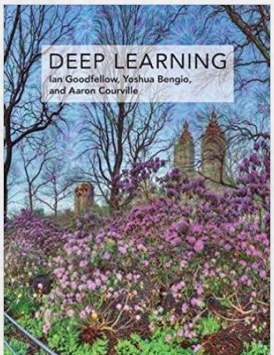
Summary

- The 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 find the polynomial as well as more/less computationally expensive to subsequently evaluate the polynomial
- The representation (or the basis in this case) matters!

Representation Matters

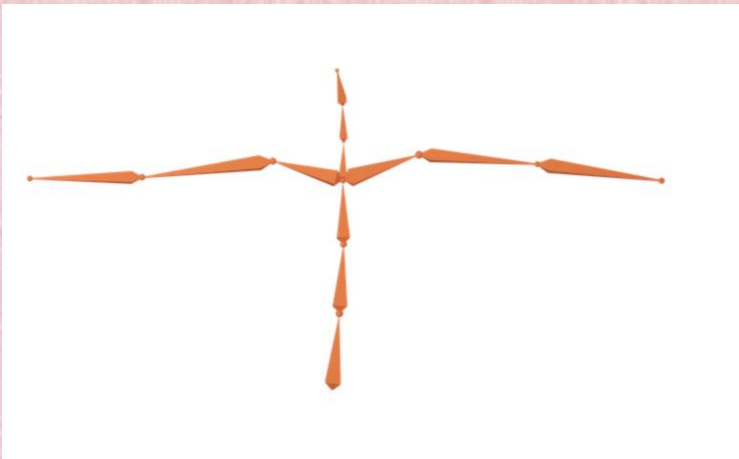
- Consider: Divide CCX by VI
- As compared to: Divide 210 by 6

- See Chapter 15 on Representation Learning in the Deep Learning book



Predict 3D Cloth Shape from Body Pose

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

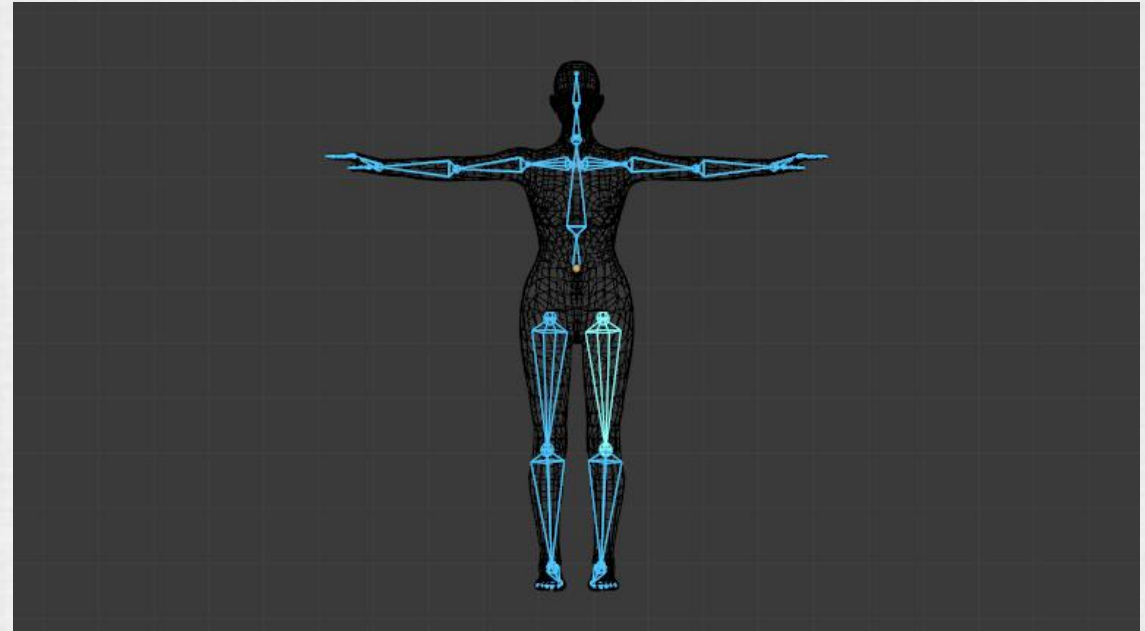


Approach

- Given: m training data points (θ_i, φ_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} using straightforward techniques
- How should the nonlinear rotations be handled?

Skinning

- Deforms a body surface mesh to match a skeletal pose
 - well studied and widely used in graphics
- Each vertex of the body surface mesh is associated with one or more nearby bones
- A weight (for each bone/vertex pair) dictates how much a change in a bone's position/orientation impacts a vertex's position
- As the pose changes, bone changes dictate new positions for body surface mesh vertices



Credit: Blender website

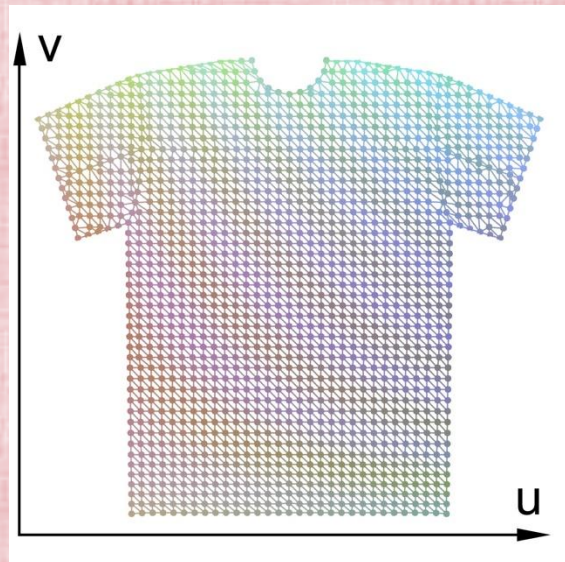
Being Careful: Leveraging Skinning

- Leverage the plethora of prior work on procedural skinning to estimate the body surface mesh S based on pose parameters θ
- 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 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

Shrink Wrap the Cloth Mesh

- Shrink-wrap the cloth vertices to the body triangle mesh
- Barycentrically embed the cloth vertices to follow body mesh triangles
- As the body deforms, cloth vertices move with their parent triangles
- This is $S(\theta)$



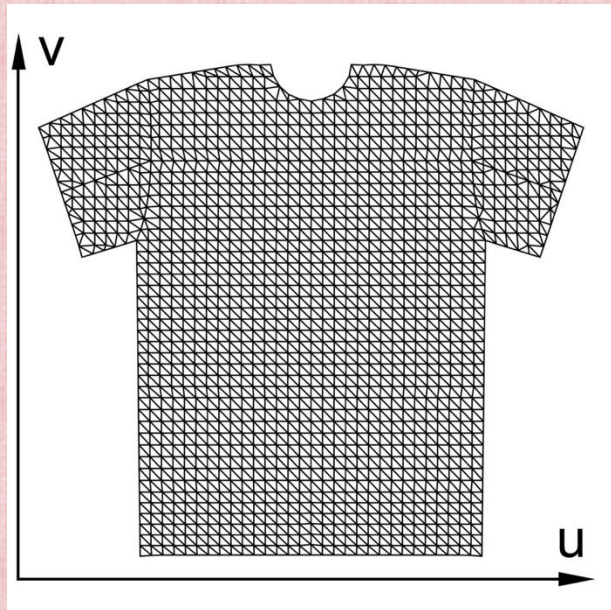
pattern space



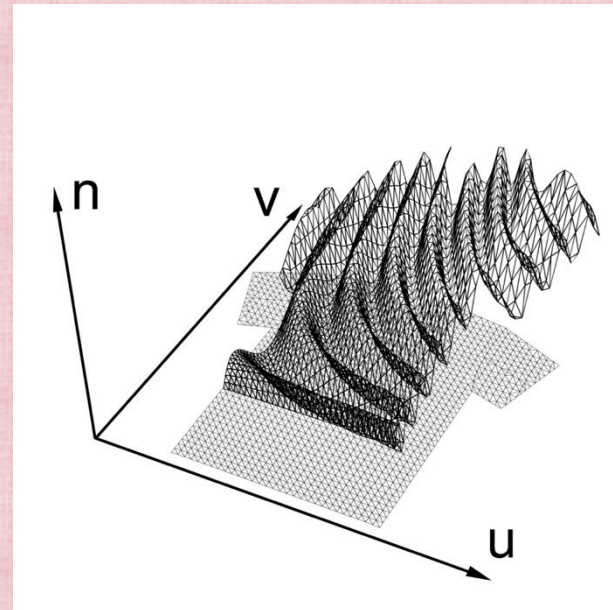
world space

Displacement Map

- Store per-vertex (u, v, n) offsets (n is the normal) in pattern (i.e. texture) space
- This is $D(\theta)$



pattern space



(u, v, n) offsets
in pattern space

Skinning + Displacement Map

$S(\theta)$

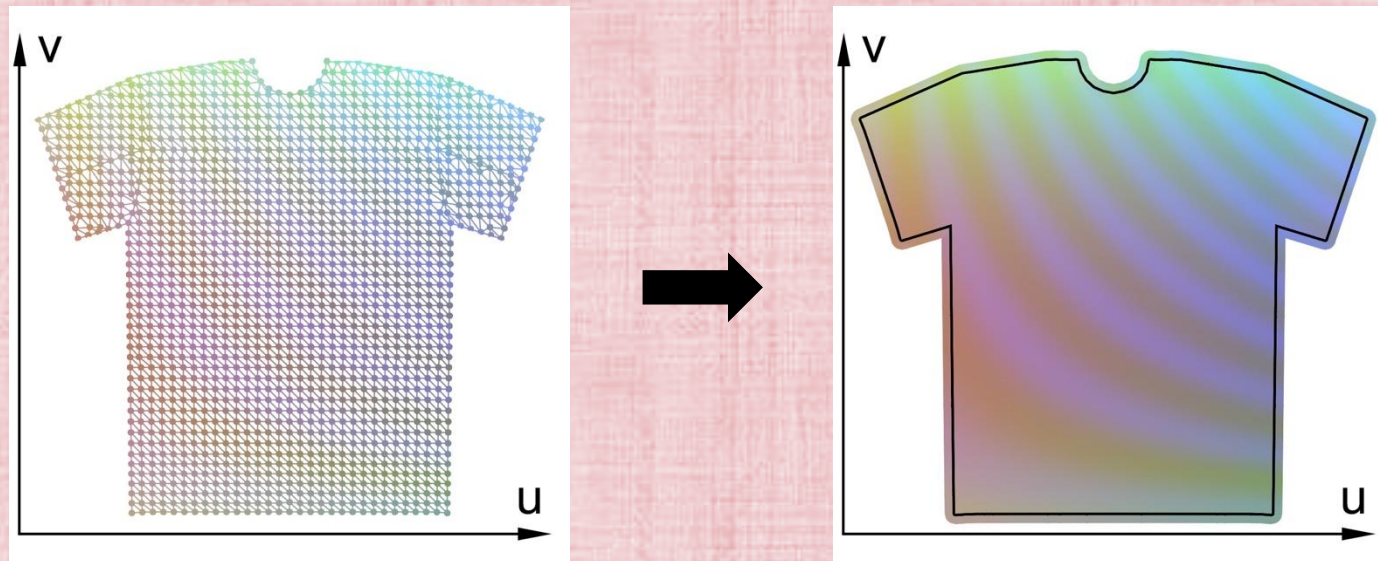


$S(\theta) + D(\theta)$



Image Based Cloth

- Convert the (u, v, n) offsets to RGB-triple color values
- Then, rasterize the triangle vertex colors to 2D image pixels (in the pattern space)
- The network learns to output a 2D RGB image (instead of 3D displacements)
- Images are more continuous than the cloth vertices, which have an irregular mesh/graph topology
- Learn to predict images as a function of pose θ , using Convolutional Neural Networks (CNNs) instead of Graph Neural Networks (GNNs)



Training Data

- For each pose in the training data, calculate per-vertex offsets and rasterize them into an image in pattern space
- Training data are $(\theta, I(\theta))$ pairs, where I is the image



Inference

- Learn to predict an image from pose parameters, i.e. learn $\hat{I}(\theta) \approx I(\theta)$
- Given an inferred image $\hat{I}(\theta)$, interpolate RGB colors to cloth vertices and convert colors to offsets; then, the offsets are added to the skinned vertex positions: $\hat{\phi}(\theta) = S(\theta) + \underbrace{\psi(\hat{I}(\theta))}_{D(\theta)}$



Unit 2

Linear Systems

Motivation

- “Matrices are bad, vector spaces are good”
 - Don’t think of matrices as a collection of numbers
 - Instead, think of the 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 contains a lot of machinery for dealing with, discussing, and gaining intuition about vectors in high dimensional spaces
- We will cover linear algebra 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 quite difficult to accomplish.
- Note: in this class, x is used for **data**, and c is used for **unknowns** (such as for the unknown parameters of a neural network)

“Zero”

- On the computer, defining “zero” is not straightforward
- 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 $O(\textit{“big”})$ or $O(\textit{“small”})$

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:

$$\begin{pmatrix} 3 & 2 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} c_3 \\ c_2 \end{pmatrix} = \begin{pmatrix} 5 \\ 6 \end{pmatrix}$$

- The final matrix is much easier to treat with finite precision arithmetic
- Solve for c_3 and c_2 ; then, $c_1 = (1e4)c_3$

Some Definitions...

- 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)
- Non-square matrices change size: $\begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}^T = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$

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

Square Matrices

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

Solvability

- A square matrix A is called singular when it is not invertible (does not have an inverse)
- Nonsingular square matrices are invertible: $AA^{-1} = A^{-1}A = I$
 - 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*
- Rank - maximum number of linearly independent columns
- A is singular if and only if its rank $< n$ (the # of columns), i.e. the columns are rank-deficient
 - A singular matrix has either no solution or infinite solutions (can't have a unique solution, see Unit 1)
- Various equivalent ways of determining whether A is singular:
 - At least one column is linearly dependent on others (not full rank)
 - It has a nonempty null space, i.e. $\exists c \neq 0$ with $Ac = 0$ (columns are linearly dependent)
 - The determinant is zero: $\det A = 0$ (zero n -dimensional "volume" in the geometry spanned by the vectors)
 - Etc.

Matrices as Vectors (an example)

- Recall $Ac = \sum_k c_k a_k$ where the a_k are the columns of A
- Consider $Ac = 0$, i.e. $\sum_k c_k a_k = 0$
- 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 so A is singular
- Alternatively: If the only solution to $Ac = 0$ is a 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
 - Either 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
$$\begin{pmatrix} 2 & 3 & 1 \\ 0 & 1 & -1 \\ 0 & 0 & 5 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 10 \\ 10 \end{pmatrix}$$

- Start at the bottom: $5c_3 = 10$; so, $c_3 = 2$
- Move up one row: $c_2 - c_3 = 10$; so, $c_2 - 2 = 10$ and $c_2 = 12$
- Move up one row: $2c_1 + 3c_2 + c_3 = 0$; so, $2c_1 + 36 + 2 = 0$ and $c_1 = -19$

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
$$\begin{pmatrix} 5 & 0 & 0 \\ -1 & 1 & 0 \\ 1 & 3 & 2 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 10 \\ 10 \\ 0 \end{pmatrix}$$

- Start at the top: $5c_1 = 10$; so, $c_1 = 2$
- Move down one row: $-c_1 + c_2 = 10$; so, $-2 + c_2 = 10$ and $c_2 = 12$
- Move down one row: $c_1 + 3c_2 + 2c_3 = 0$; so, $2 + 36 + 2c_3 = 0$ and $c_3 = -19$

LU Factorization

- In order to solve $Ac = b$ more generally, factor A into a lower triangular and an upper triangular matrix: $A = LU$
 - The LU factorization only needs to be computed once; then, it can be used on many right-hand side vectors (on many b vectors)

Process for solving $LUc = b$:

- 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

Elimination Matrix

- Given the k -th column vector $\begin{pmatrix} a_{1k} \\ \vdots \\ a_{ik} \\ a_{i+1,k} \\ \vdots \\ a_{nk} \end{pmatrix}$, first define $m_{ik} = \frac{1}{a_{ik}} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ a_{i+1,k} \\ \vdots \\ a_{nk} \end{pmatrix}$
- Then, the size $n \times n$ elimination matrix $M_{ik} = I_{n \times n} - m_{ik} \hat{e}_i^T$ subtracts multiples of row i from rows $> i$ in order to create zeroes in column k

- Standard basis vector $\hat{e}_i = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$ has a 1 in the i -th row

Elimination Matrix

- Let $a_k = \begin{pmatrix} 2 \\ 4 \\ -8 \end{pmatrix}$

- $M_{1k} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} - \frac{1}{2} \begin{pmatrix} 0 \\ 4 \\ -8 \end{pmatrix} (1 \ 0 \ 0) = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 4 & 0 & 1 \end{pmatrix}$ and $M_{1k}a_k = \begin{pmatrix} 2 \\ 0 \\ 0 \end{pmatrix}$

- $M_{2k} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} - \frac{1}{4} \begin{pmatrix} 0 \\ 0 \\ -8 \end{pmatrix} (0 \ 1 \ 0) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 2 & 1 \end{pmatrix}$ and $M_{2k}a_k = \begin{pmatrix} 2 \\ 4 \\ 0 \end{pmatrix}$

Elimination Matrix Inverse

- Inverse of an elimination matrix is $L_{ik} = M_{ik}^{-1} = I_{n \times n} + m_{ik} \hat{e}_i^T$
- L_{ik} is a size $n \times n$ elimination matrix that **adds** multiples of row i to rows $> i$ in order to reverse the effect of M_{ik}

- $L_{1k} = M_{1k}^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -4 & 0 & 1 \end{pmatrix}$

- $L_{2k} = M_{2k}^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -2 & 1 \end{pmatrix}$

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 $i_1 > i_2$)

$$M_{1k} M_{2k} = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 4 & 2 & 1 \end{pmatrix}, \text{ but } M_{2k} M_{1k} = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 0 & 2 & 1 \end{pmatrix}$$

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

$$L_{1k} L_{2k} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -4 & -2 & 1 \end{pmatrix}, \text{ but } L_{2k} L_{1k} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -8 & -2 & 1 \end{pmatrix}$$

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 $L_{i_1i_1}L_{i_2i_2} = I + m_{i_1i_1}\hat{e}_{i_1}^T + m_{i_2i_2}\hat{e}_{i_2}^T$ when $i_1 < i_2$, $L = L_{11}L_{22} \cdots L_{n-1,n-1}$ is lower triangular and $A = LU$

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

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

Pivoting

- $A = \begin{pmatrix} 0 & 4 \\ 4 & 9 \end{pmatrix}$ requires division by **zero** when creating 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. $P_{13} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$ swaps the first and third rows of A via $P_{13}A$
- Permutation matrices are their own inverses (swapping again restores the rows)
- Switching rows i_1 and i_2 moves a 1 from $a_{i_1i_1}$ to $a_{i_2i_1}$ as well as from $a_{i_2i_2}$ to $a_{i_1i_2}$, preserving symmetry (i.e. $P_{i_1i_2}^T = P_{i_1i_2}$)
- To swap the first and third unknowns: $Ac = AP_{13}P_{13}c = (AP_{13})(P_{13}c)$ where $P_{13}c$ row swaps the unknowns and AP_{13} swaps the columns
 - To understand AP_{13} , consider $(AP_{13})^{TT} = (P_{13}A^T)^T$ where $P_{13}A^T$ swaps the rows of A^T

LU Factorization with Pivoting

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

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

- 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)
- Thus, $Ac = b$ becomes $(P_r A P_c^T)(P_c c) = P_r b$ 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 (as usual); then, $c = P_c^T c_P$

Permuting before Elimination

- Assume $i > j$,

$$P_{r_i} M_{jj} P_{r_i} = I_{n \times n} - P_{r_i} m_{jj} \hat{e}_j^T P_{r_i} = I_{n \times n} - \hat{m}_{jj} \hat{e}_j^T = \hat{M}_{jj}$$
$$P_{r_i} M_{jj} = P_{r_i} M_{jj} P_{r_i} P_{r_i} = \hat{M}_{jj} P_{r_i}$$

- Thus, for some suitable definition of the hat notation (there are multiple permutation operators to consider for most M_{jj}):

$$M_{n-1,n-1} P_{r_{n-1}} \cdots M_{22} P_{r_2} M_{11} P_{r_1} A = M_{n-1,n-1} \cdots \hat{M}_{22} \hat{M}_{11} P_r A$$

- Thus, you can permute first and do elimination afterwards

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), with most entries identically 0
- Unfortunately, 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 a single 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 only has 7×10^6 nonzero entries
- But, the inverse could have as many as 10^{12} nonzero entries!

Computing the Inverse

- When A is relatively small (and dense), computing A^{-1} is fine
- Since $AA^{-1} = I$, the solution c_k to $Ac_k = \hat{e}_k$ is the k -th column of A^{-1}

Method:

- First, compute $A_P = L_P U_P$ as usual
- Then, solve $Ac_k = \hat{e}_k$ once for each column (i.e., n times)

Unit 3

Understanding Matrices

Eigensystems

- Eigenvectors - special directions v_k in which a matrix only applies scaling
- Eigenvalues - the amount λ_k of that scaling
- Right Eigenvectors (or just eigenvectors) satisfy $Av_k = \lambda_k v_k$
 - **Eigenvectors represent directions**, so $A(\alpha v_k) = \lambda_k(\alpha v_k)$ is also true for all α
- 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

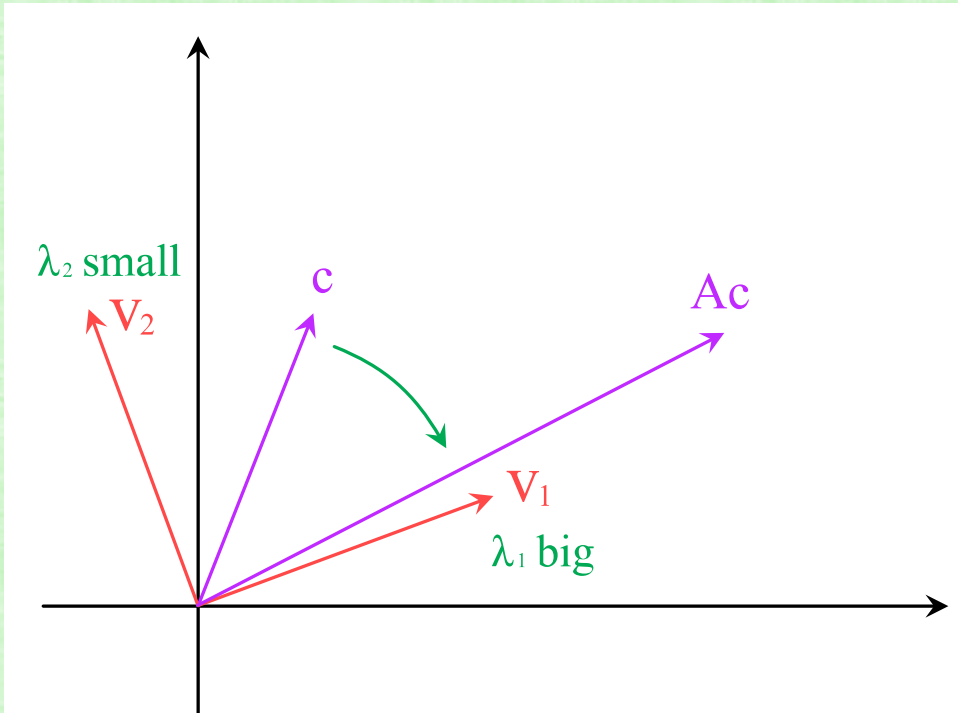
- Complex numbers may appear in both eigenvalues and eigenvectors

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

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

Vector Deformation

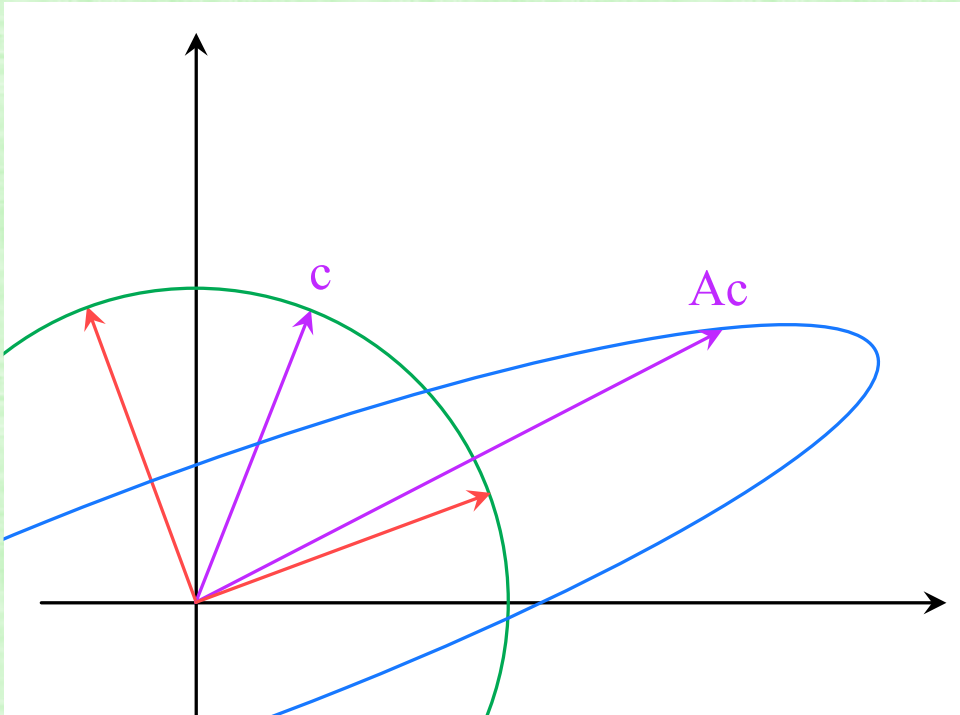
- Let $c = \sum_k \alpha_k v_k$, so that $Ac = \sum_k \alpha_k Av_k = \sum_k (\alpha_k \lambda_k) v_k$
- A tilts c away from directions with smaller eigenvalues and towards directions with larger eigenvalues



- Large λ_k stretch in their associated v_k directions
- Small λ_k squish in their associated v_k directions
- Negative λ_k flip the sign (i.e. direction) in their associated v_k directions

Spatial Deformation

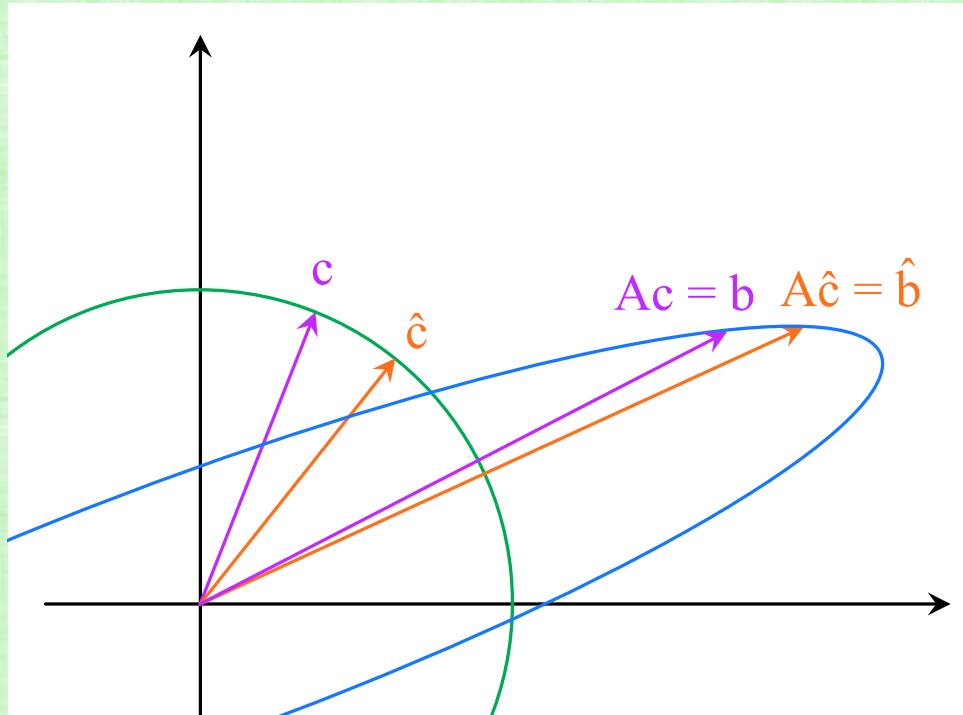
- Consider every point on the unit circle (green) as a vector $c = \sum_k \alpha_k v_k$, and remap each point via $Ac = \sum_k (\alpha_k \lambda_k) v_k$



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

Implications for 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: c and \hat{c} are more separated than b and \hat{b} , i.e. the solution is perturbed more than the right-hand side was perturbed



- Small changes in b lead to larger changes in the solution
- **Small algorithmic errors are also amplified:** they change $A^{-1}b$ to $\hat{A}^{-1}b$, 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

Aside: Preconditioning

- Suppose A has large eigenvalue ratios, making $Ac = b$ difficult to solve
- Let $\hat{A}^{-1} \approx A^{-1}$ be an approximate guess for the inverse
- 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 similar)
- \hat{I} is not the identity, so more computation is required to find c
- But, \hat{I} has similar magnitude eigenvalues (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 stretched ellipsoids along eigenvector axes, but scaling along 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 (the number of rows)
- Since the columns live in an m dimensional space, they can at best span that whole space
- Thus, 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

Rows vs. Columns

- One can find discussions on rows, row spaces, etc. that are used for various purposes
- Although these are fine mathematical discussions, they are unnecessary for an intuitive understanding of high dimensional vector spaces (so, we'll ignore them)
- The number of columns is equal to the 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 affect 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: $A = U \Sigma V^T$
- Σ 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_{ik} with $i \neq k$
- E.g. $\begin{pmatrix} 5 & 0 \\ 0 & 2 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} 10 \\ -1 \\ \alpha \end{pmatrix}$ has $5c_1 = 10$ and $2c_2 = -1$, so $c_1 = 2$ and $c_2 = -.5$
 - Note: $\alpha \neq 0$ imposes a “no solution” condition (even though c_1 and c_2 are well-specified)
- E.g. $\begin{pmatrix} 5 & 0 & 0 \\ 0 & 2 & 0 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 10 \\ -1 \end{pmatrix}$ has $5c_1 = 10$ and $2c_2 = -1$, so $c_1 = 2$ and $c_2 = -.5$
 - Note: 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, leading to “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 the square $A^T A$**

- $AA^T = U \Sigma V^T V \Sigma^T U^T = U (\Sigma \Sigma^T) U^T$, 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 the square 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**

- 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 has none

Example (Tall Matrix)

- Consider size 4×3 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
- Since $a_3 = 2a_2 - a_1$, 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 (only rank 2, not rank 3)
- 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 3×3 and has 3 eigenvectors (columns of V)
- The square roots of the 3 eigenvalues of $A^T A$ are in Σ (color coded to the eigenvectors)
- AA^T is size 4×4 and has 4 eigenvectors (columns of U)
- The square roots of 3 of the eigenvalues of AA^T are in Σ
 - The 4th 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

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

- 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

Extra Dimensions

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

$$\begin{pmatrix} .141 & .825 & -.420 & -.51 \\ .344 & .426 & .298 & .78 \\ .547 & .028 & .644 & -.59 \\ .750 & -.371 & -.542 & .09 \end{pmatrix} \begin{pmatrix} 25.5 & 0 & 0 \\ 0 & 1.29 & 0 \\ 0 & 0 & 0 \\ \blacksquare & \blacksquare & \blacksquare \end{pmatrix} \begin{pmatrix} .504 & .574 & .644 \\ -.761 & -.057 & .646 \\ .408 & -.816 & .408 \end{pmatrix}$$

- 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 Σ (and still obtain a valid factorization)

Zero Singular Values

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

$$\begin{pmatrix} .141 & .825 & -.410 & -.151 \\ .344 & .426 & .218 & .781 \\ .547 & .028 & .646 & -.519 \\ .750 & -.371 & -.512 & .019 \end{pmatrix} \begin{pmatrix} 25.5 & 0 \\ 0 & 1.29 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} .504 & .574 & .644 \\ -.761 & -.057 & .646 \\ .158 & .118 & .158 \end{pmatrix}$$

- The 3rd 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

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

$$\begin{pmatrix} .141 & .85 & -.40 & -.51 \\ .344 & .46 & .23 & .78 \\ .547 & .03 & .64 & -.59 \\ .750 & -.37 & -.54 & .09 \end{pmatrix} \begin{pmatrix} 25.5 & & & \\ & & & \\ & & & \\ & & & \end{pmatrix} \begin{pmatrix} .504 & .574 & .644 \\ & & & \\ & & & \\ & & & \end{pmatrix}$$

- 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 “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 3rd singular value is 0, and the associated (pink) terms make no contribution to the final result (so the null space is actually 2D)

Example (Wide Matrix)

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

$$\begin{pmatrix} .504 & -.761 & .418 & .141 \\ .574 & -.057 & -.816 & .344 \\ .644 & .646 & .418 & .547 \end{pmatrix} \begin{pmatrix} 25.5 & 0 & 0 & 0 \\ 0 & 1.29 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} .141 & .344 & .547 & .750 \\ .825 & .426 & .028 & -.371 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

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

More Theoretical than Practical

- 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 robust and computational algorithms

SVD Uniqueness

- Mandating that all the $\sigma_k \geq 0$ be listed in descending order makes Σ unique
- Non-repeated σ_k have unique singular vectors in both U and V
 - up to a ± 1 choice of sign (as discussed in the next two slides)
- Repeated σ_k lead to non-unique singular vectors, since there are an infinite number of ways to span the associated subspace
 - This non-uniqueness stems from the non-uniqueness in eigensystems with repeated eigenvalues
 - This non-uniqueness wreaks havoc on differentiability of the SVD

SVD Construction (an 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 , $Av_k = U\Sigma V^T v_k = U\Sigma \hat{e}_k = U\sigma_k \hat{e}_k = \sigma_k u_k$ 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

SVD Construction Strategy

- An orthogonal matrix has determinant equal to ± 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)
 - Regarding uniqueness: subsequently flipping any even number of columns preserves $\det V = 1$
- Then, for each v_k , compare Av_k to $\sigma_k u_k$ and flip the direction of u_k if necessary to make $Av_k = \sigma_k u_k$
- When $\det U = -1$, it contains a reflection
 - When $m > n$, this can be fixed by flipping a column of U that does not correspond to a singular value
 - When $m < n$, flip a column of U to make $\det U = 1$ along with the corresponding column V ; then, flip a column of V that does not correspond to a singular value
 - When $m = n$, column flips cannot remove the intrinsic reflection
- When $\det U = -1$ with $m = n$, one can flip the sign of the smallest singular value in Σ to be negative, while also flipping the direction of the corresponding column in U
 - This moves the reflection from U to Σ and is called the polar-SVD
 - See "Invertible Finite Elements For Robust Simulation of Large Deformation", 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
 - 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 σ_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)

Condition Number

- The condition number of A is $\frac{\sigma_{max}}{\sigma_{min}}$, and it measures closeness to being singular
- For a square A , it measures the difficulty in solving $Ac = b$
- For a rectangular (and square) A , it measures how close the columns are to being linearly dependent
 - For a wide (rectangular) A , it ignores the extra columns that must 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 ∞ for nearly singular matrices
- The condition number is ∞ for singular matrices, since $\sigma_{min} = 0$

Understanding 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 $\hat{c}_1 = \frac{\hat{b}_1}{\sigma_1}$, $\hat{c}_2 = \frac{\hat{b}_2}{\sigma_2}$
- When $\sigma_1 \neq 0$ 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 well 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 well determined)
- Consider: $\begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} \hat{c}_1 \\ \hat{c}_2 \end{pmatrix} = \begin{pmatrix} \hat{b}_1 \\ \hat{b}_2 \\ \hat{b}_3 \end{pmatrix}$ 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 (it's 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 well determined)
- Consider: $\begin{pmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \end{pmatrix} \begin{pmatrix} \hat{c}_1 \\ \hat{c}_2 \\ \hat{c}_3 \end{pmatrix} = \begin{pmatrix} \hat{b}_1 \\ \hat{b}_2 \end{pmatrix}$ 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 well determined)

Understanding Variables

Consider any column k of Σ :

- When $\sigma_k \neq 0$, a unique value can be determined for \hat{c}_k
- When $\sigma_k = 0$ or there is no σ_k (wide matrices), the data contains no information for \hat{c}_k
 - This does not mean that other variables cannot be adequately determined!

Consider a row i of Σ 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 the data contains conflicting information
 - 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 for those variables

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 **only** valid **theoretically**
- In practice, the "worst case scenario" (L^∞) and the "average" (L^1 , L^2 , etc.) are not interchangeable
- For example:
$$(100 \text{ people} * 98.6^\circ + 1 \text{ person} * 105^\circ) / (101 \text{ people}) = 98.66^\circ$$
- Their average temperature is 98.66° , but everything is not "ok"

Matrix Norms

- The norm of a matrix is defined via $\|A\| = \max_{c \neq 0} \frac{\|Ac\|}{\|c\|}$, 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 σ_{max} of A
 - Since $A^{-1} = V\Sigma^{-1}U^T$ and Σ^{-1} has diagonal entries $\frac{1}{\sigma_k}$, $\|A^{-1}\|_2 = \frac{1}{\sigma_{min}}$
- The condition number for solving $Ac = b$ for a square matrix A is:

$$\|A\|_2 \|A^{-1}\|_2 = \frac{\sigma_{max}}{\sigma_{min}}$$

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}

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 , $Av_k = U\Sigma V^T v_k = U\Sigma \hat{e}_k = U\sigma_k \hat{e}_k = \sigma_k u_k$ 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

- Since $A^T A = A A^T = A^2$, both the columns of U and the columns of V are eigenvectors of A^2
- They have 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 $\pm \sigma_k$

- Pulling negative signs out of the columns of U and into the σ_k (similar to the polar-SVD) leads to $U = V$ 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
 - Λ contains the positive and negative eigenvalues of A

Making/Breaking Symmetry

- Row/column scaling can make or 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 a symmetric $\begin{pmatrix} 5 & 6 \\ 6 & -16 \end{pmatrix}$
- Scaling corresponding rows/columns together preserves symmetry
- Important: a nonsymmetric matrix might be inherently symmetric when properly rescaled/rearranged

Symmetric Approximation

- A non-symmetric A can be approximated by a symmetric $\hat{A} = \frac{1}{2}(A + A^T)$, which averages off-diagonal components
- Solving the symmetric $\hat{A}c = b$ instead of the non-symmetric $Ac = b$ gives a faster/easier (but erroneous) approximation to c
 - This is useful when c might not require too much accuracy
- The inverse of a symmetric \hat{A} (or the notion of the inverse) may be used to devise a preconditioner for $Ac = b$

Inner Product

- The dot/inner product of two vectors (of the same length) 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 $m \times n$ matrix A
- $\langle u, v \rangle_A = u \cdot Av = u^T Av$
- Since $\langle u, v \rangle_A = u^T Av = v^T A^T u = \langle v, u \rangle_{A^T}$, weighted inner products commute when A is symmetric
- The standard dot product uses identity matrix weighting: $\langle u, v \rangle = \langle u, v \rangle_I$

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 ≥ 0
- Symmetric negative definite matrices have all eigenvalues < 0
- Symmetric negative semi-definite matrices have all eigenvalues ≤ 0
- Indefinite matrices have both positive and negative eigenvalues

SPD Matrices

- When A is SPD or SP(S)D, $\Lambda = \Sigma$ and the standard SVD is $A = V\Sigma V^T$ (i.e. $U = V$)

SPD matrices:

- The singular values are the (all positive) eigenvalues of A
- Construct V with $\det V = 1$ (as usual); then, 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 matrices:

- Has at least one $\sigma_k = 0$ and a null space
- Often, one can slightly modify SPD techniques for SP(S)D matrices
- Significantly easier to treat than indefinite matrices!

Cholesky Factorization

- SPD matrices have an LU factorization of LL^T , which can be computed without elimination

- 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 $l_{22} = \sqrt{a_{22} - l_{21}^2}$

for(j=1,n){

for(k=1,j-1) 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 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
 - That is, all zeros remain zeros

Rules Galore

- There are many rules/theorems regarding special matrices (e.g. 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.

Unit 5

Iterative Solvers

Iterative vs. Direct Solvers

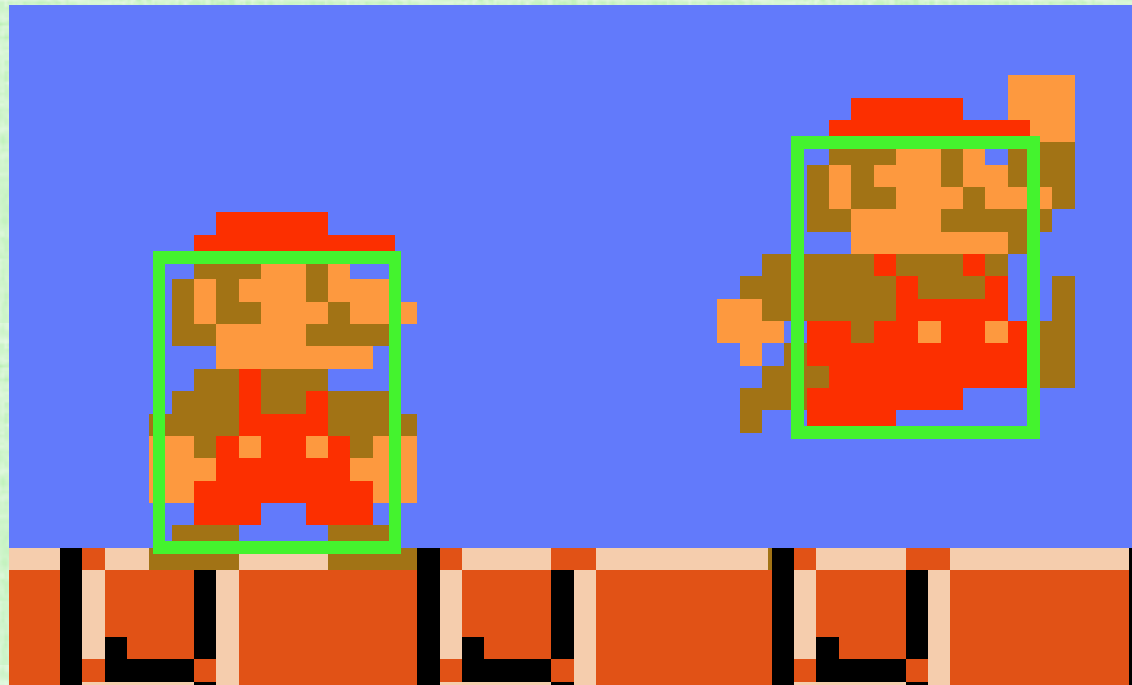
- Direct Solver/Method – closed form strategy, e.g. quadratic/Cardano formulas, Gaussian Elimination for LU , Cholesky factorization for LL^T , etc.
- Iterative Solver/Method
 - start with an initial guess: c^1
 - use a recursive approach to improve that guess: c^2, c^3, c^4, \dots
 - terminate based on stopping criteria, 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 too often have unacceptably high error
 - To highlight why one might need accurate cubic roots, let's consider collision detection...

Hit Box

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



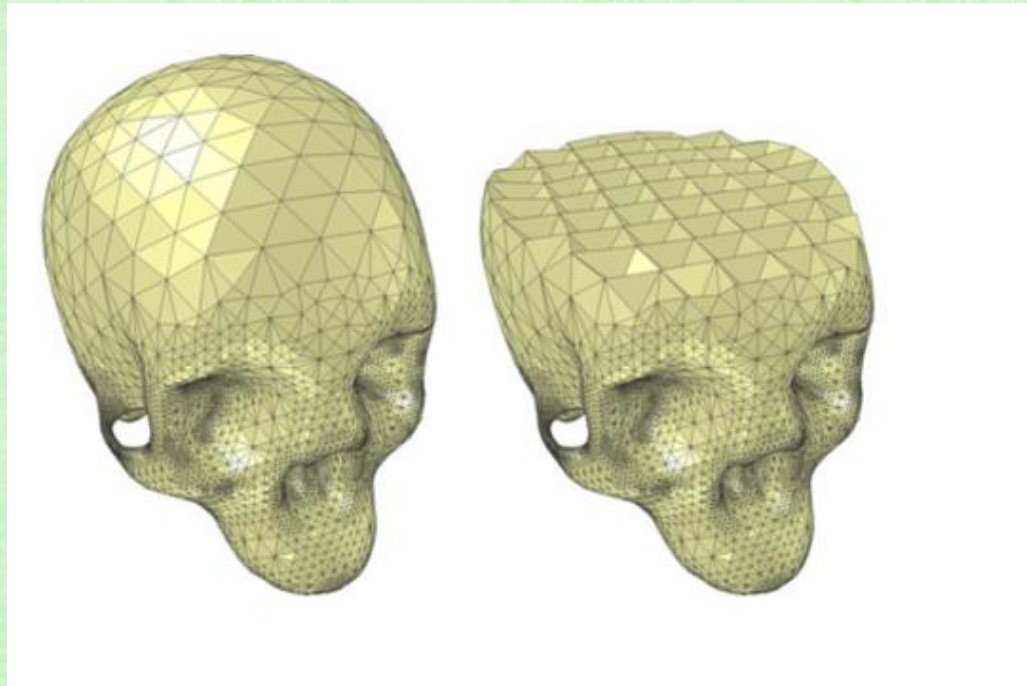
Better Hit Boxes

- 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



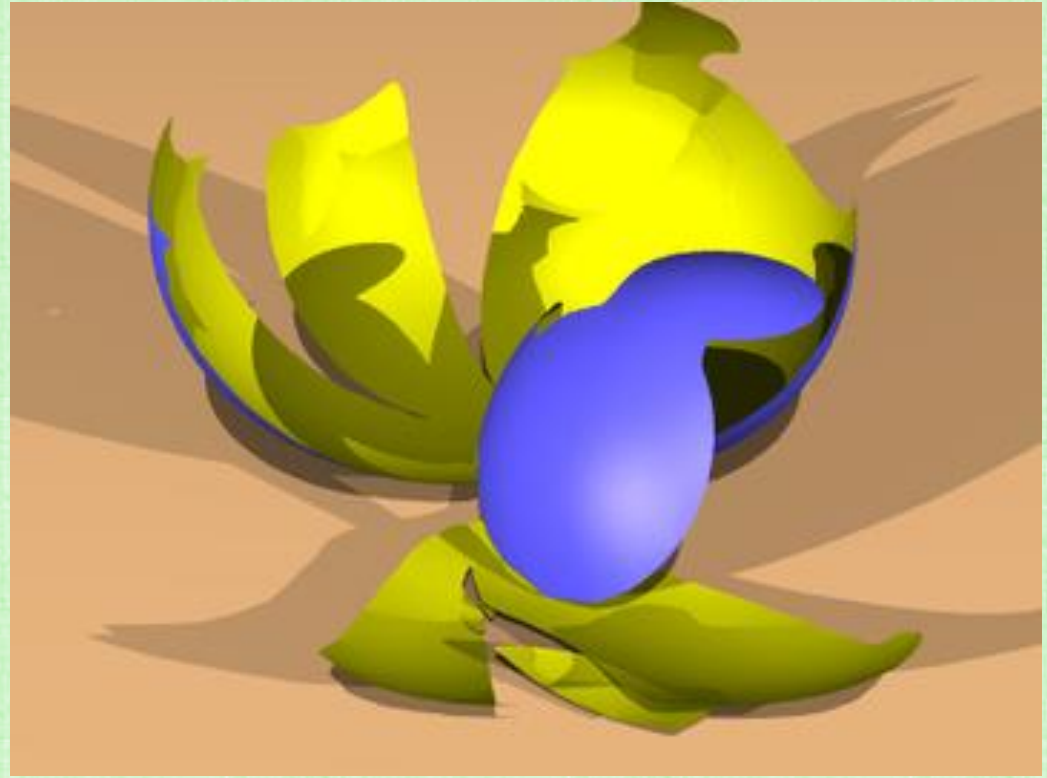
Accurate Collision Detection

- 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



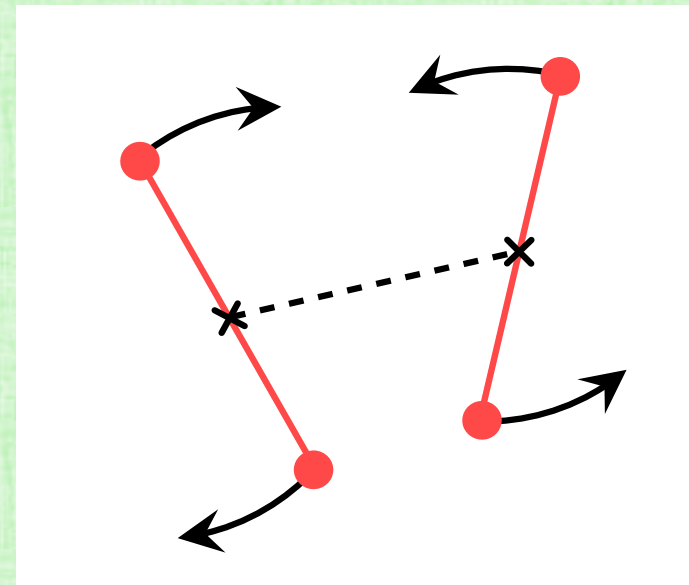
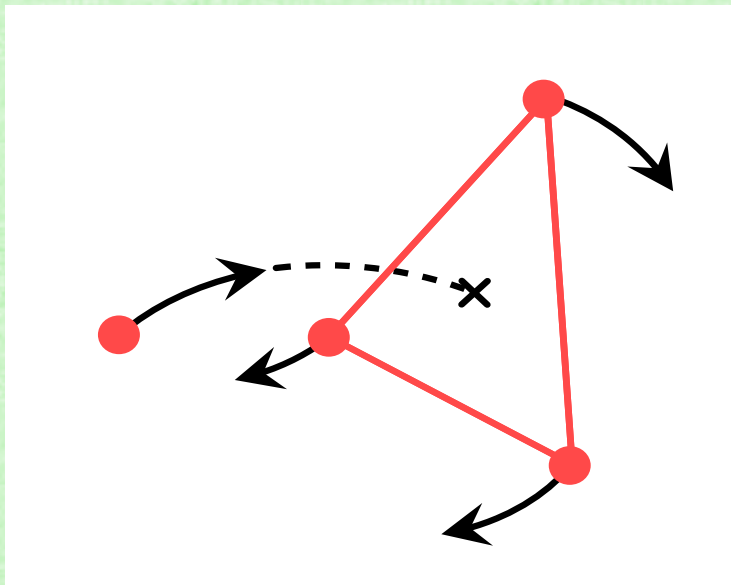
Objects Without Interiors

- 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



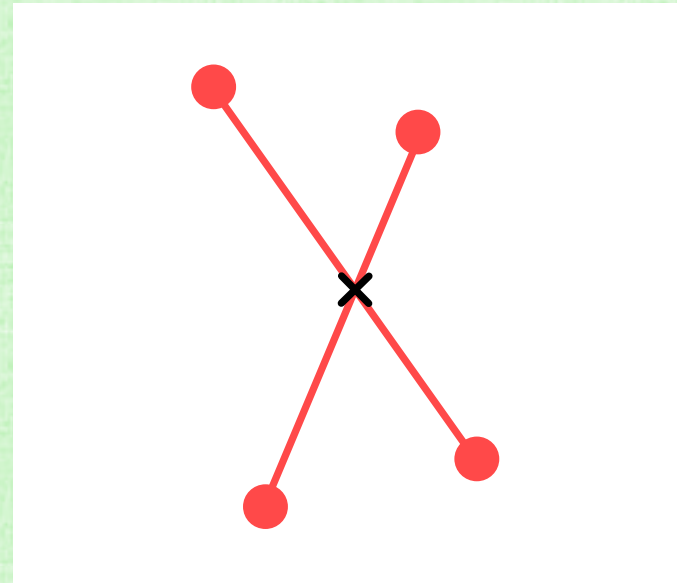
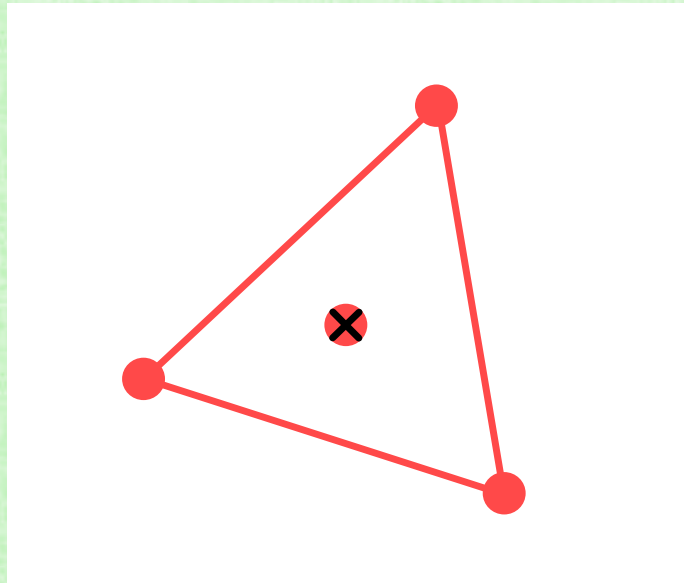
Continuous Collision Detection (CCD)

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



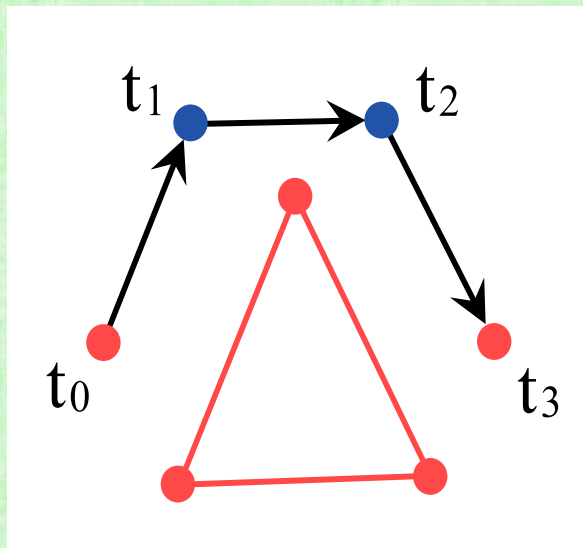
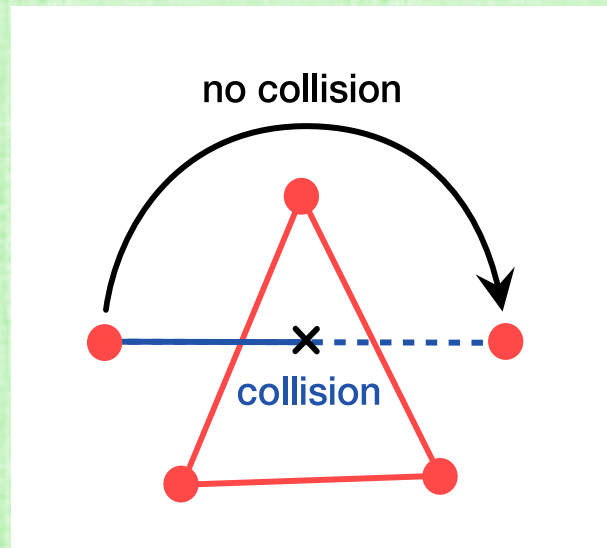
Continuous Collision Detection (CCD)

- In both cases, the 4 relevant points become coplanar when they collide
- Once coplanar, a second check determines whether: the 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 that the points have constant velocities during the 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-intersection



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 (leading to a cubic equation in t)
- 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 often in single precision)
 - See Bridson et al. “Robust Treatment of Collisions, Contact, and Friction for Cloth Animation” (2002)

Residual and Solution Error

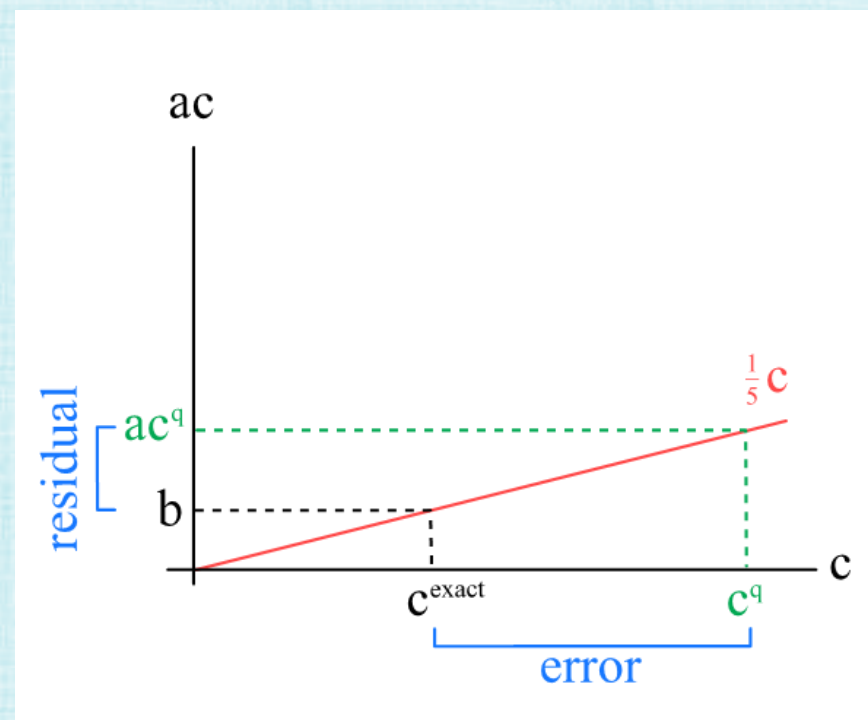
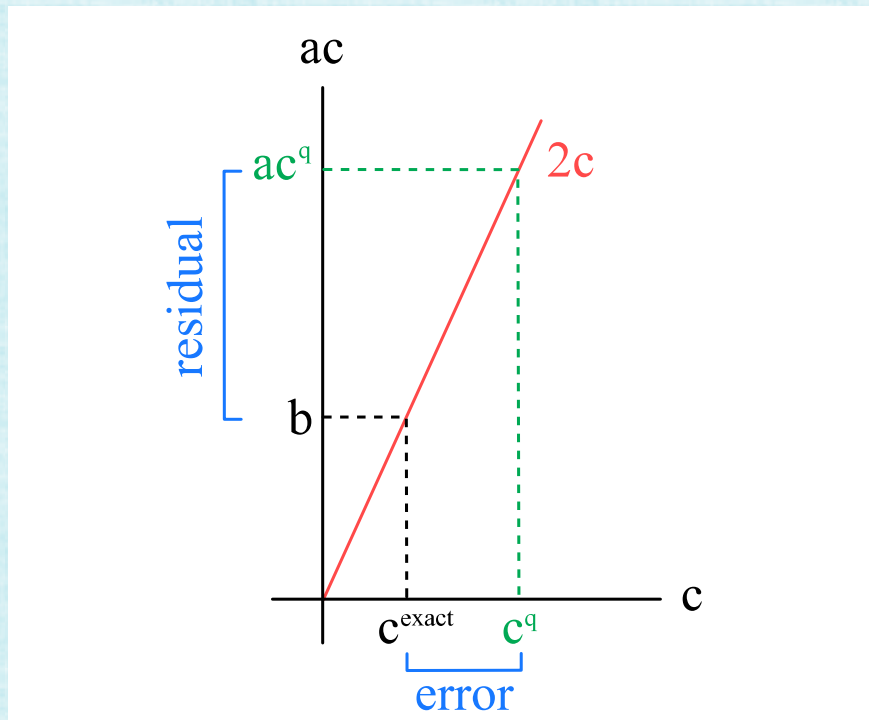
- When solving $Ac = b$, a current guess c^q has residual $r^q = b - Ac^q$
- The residual measures the errors in the equations, not the error in the solution
- The error in the solution $e^q = c^q - c^{exact}$ relates to the residual via:

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

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

1D example

- Consider a simple size 1×1 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



Diagonalizing the Residual/Error Equation

- "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 σ_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

- Choose a search direction s^q and move some distance α^q in that direction to update the current guess to the next guess: $c^{q+1} = c^q + \alpha^q s^q$
 - There are various strategies for choosing α^q , including the notion of safe sets that clamp its maximum magnitude
 - Subtract c^{exact} from both sides of this recursion to get $e^{q+1} = e^q + \alpha^q s^q$
 - Multiply through by $-A$ to get $r^{q+1} = r^q - \alpha^q A s^q$
- Optimally, one would follow s^q until all the error in that direction was eliminated
 - That is, until the remaining error is orthogonal to s^q , i.e. $e^{q+1} \cdot s^q = 0$
 - However, the error is unknown (otherwise, the solution would be known)
- Instead, follow s^q 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

- Steepest Descent chooses the steepest downhill direction as the search direction
 - That turns out to be the residual, i.e. choose $s^q = r^q$
- 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
- Can replace $r^q = b - Ac^q$ with $r^q = r^{q-1} - \alpha^{q-1} Ar^{q-1}$
 - Since Ar^{q-1} had already been computed to find α^{q-1} , this eliminates one of the two (possibly expensive) multiplications by A
- Drawback: Steepest Descent repeatedly searches in overlapping (non-orthogonal) directions, especially for higher condition number matrices (more on this later)

Conjugate Gradients (CG)

- A very efficient and robust method for SPD systems
- Converges (theoretically) in at most n -steps for an $n \times n$ matrix
 - Theoretically, only need one step for each distinct eigenvalue
 - Almost converged when taking one step for each eigenvalue cluster
 - Thus, preconditioning makes a big difference (assuming it clusters eigenvalues)
- Motivation: choosing **orthogonal** search directions would preclude repeatedly searching in overlapping directions (in contrast to Steepest Descent)
 - But, can't properly eliminate the error via $e^{q+1} \cdot s^q = 0$ in each search direction
- Trick: use **A-orthogonal** search directions
 - Instead of $\langle s^q, s^{\hat{q}} \rangle = 0$, use $\langle s^q, s^{\hat{q}} \rangle_A = 0$ for $q \neq \hat{q}$

Error Analysis for CG

- In the A-orthogonal basis of search directions, the initial error is $e^1 = \sum_{\hat{q}=1}^n \beta^{\hat{q}} s^{\hat{q}}$
- Error recursion (i.e., $e^{q+1} = e^q + \alpha^q s^q$) gives $e^q = e^1 + \sum_{\hat{q}=1}^{q-1} \alpha^{\hat{q}} s^{\hat{q}}$
- Thus, $e^q = \sum_{\hat{q}=q}^n \beta^{\hat{q}} s^{\hat{q}} + \sum_{\hat{q}=1}^{q-1} (\beta^{\hat{q}} + \alpha^{\hat{q}}) s^{\hat{q}}$
- Progressing until $r^{q+1} \cdot s^q = 0$ gives $\alpha^q = \frac{s^q \cdot r^q}{s^q \cdot A s^q} = \frac{-\langle s^q, e^q \rangle_A}{\langle s^q, s^q \rangle_A} = \frac{-\beta^q \langle s^q, s^q \rangle_A}{\langle s^q, s^q \rangle_A} = -\beta^q$
- Thus, $e^q = \sum_{\hat{q}=q}^n \beta^{\hat{q}} s^{\hat{q}}$
- This proves that the error is indeed cancelled out in n steps, i.e. $e^{n+1} = 0$

- Aside: If $q > \tilde{q}$, then $r^q \cdot s^{\tilde{q}} = -\langle s^{\tilde{q}}, e^q \rangle_A = 0$; so, **each residual is orthogonal to all previous search directions** (not just the previous one)

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 candidate 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
 - Note: $\langle s^{\hat{q}}, s^{\hat{q}} \rangle_A = \|s^{\hat{q}}\|_A^2$
 - Proof: $\langle s^q, s^{\tilde{q}} \rangle_A = \langle \bar{s}^q, s^{\tilde{q}} \rangle_A - \frac{\langle \bar{s}^q, s^{\tilde{q}} \rangle_A}{\langle s^{\tilde{q}}, s^{\tilde{q}} \rangle_A} \langle s^{\tilde{q}}, s^{\tilde{q}} \rangle_A = 0$ for $\tilde{q} \in \{1, 2, \dots, q-1\}$

Gram-Schmidt for CG

- Choose candidate search directions r^q to make A-orthogonal via Gram-Schmidt:

$$s^q = r^q - \sum_{\hat{q}=1}^{q-1} \frac{\langle r^q, s^{\hat{q}} \rangle_A}{\langle s^{\hat{q}}, s^{\hat{q}} \rangle_A} s^{\hat{q}}$$

- Dot product with $r^{\tilde{q}}$ to get: $s^q \cdot r^{\tilde{q}} = r^q \cdot r^{\tilde{q}} - \sum_{\hat{q}=1}^{q-1} \frac{\langle r^q, s^{\hat{q}} \rangle_A}{\langle s^{\hat{q}}, s^{\hat{q}} \rangle_A} s^{\hat{q}} \cdot r^{\tilde{q}}$

- If $\tilde{q} > q$, then $0 = r^q \cdot r^{\tilde{q}} + 0$ implies that all the residuals are orthogonal to each other

- If $\tilde{q} = q$, then $s^q \cdot r^q = r^q \cdot r^q + 0$ implies $\alpha^q = \frac{r^q \cdot r^q}{\langle s^q, s^q \rangle_A}$

- Dot product $r^q = r^{q-1} - \alpha^{q-1} A s^{q-1}$ with $r^{\tilde{q}}$ to get:

$$r^{\tilde{q}} \cdot r^q = r^{\tilde{q}} \cdot r^{q-1} - \alpha^{q-1} \langle r^{\tilde{q}}, s^{q-1} \rangle_A$$

- If $\tilde{q} = q$, then $r^q \cdot r^q = 0 - \alpha^{q-1} \langle r^q, s^{q-1} \rangle_A$ for the last term in the Gram-Schmidt sum

- If $\tilde{q} > q$, then $0 = 0 - \alpha^{q-1} \langle r^{\tilde{q}}, s^{q-1} \rangle_A$ implies that all the other terms are zero

- Thus, $s^q = r^q + \frac{r^q \cdot r^q}{\alpha^{q-1} \langle s^{q-1}, s^{q-1} \rangle_A} s^{q-1} = r^q + \frac{r^q \cdot r^q}{r^{q-1} \cdot r^{q-1}} s^{q-1}$

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

Non-Symmetric and/or Indefinite

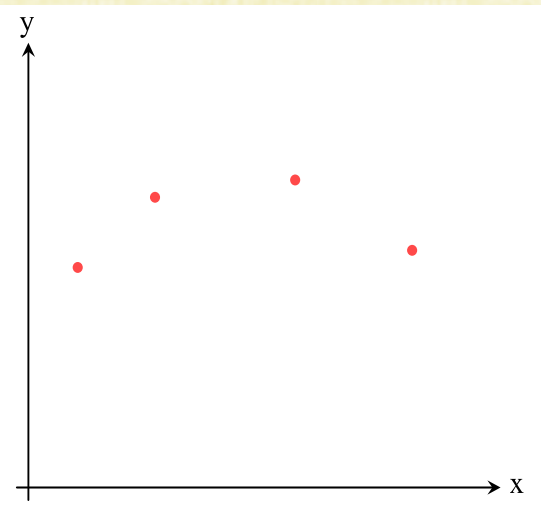
- GMRES, MINRES, BiCGSTAB, etc...
- Generally speaking, iterative methods for non-symmetric and/or indefinite matrices are less stable, more error prone, and slower than CG on an SPD matrix

Unit 6

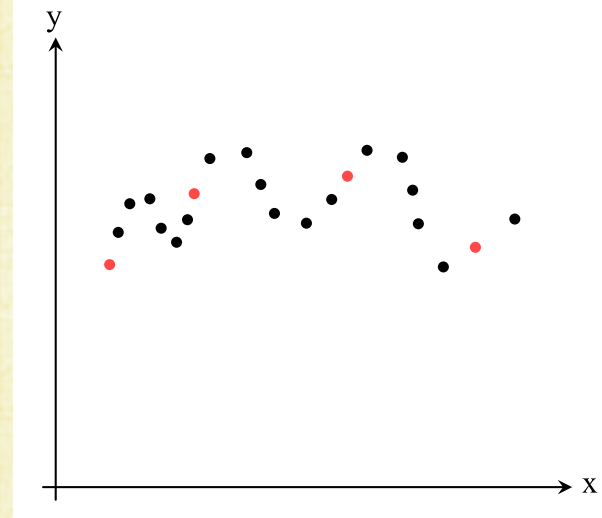
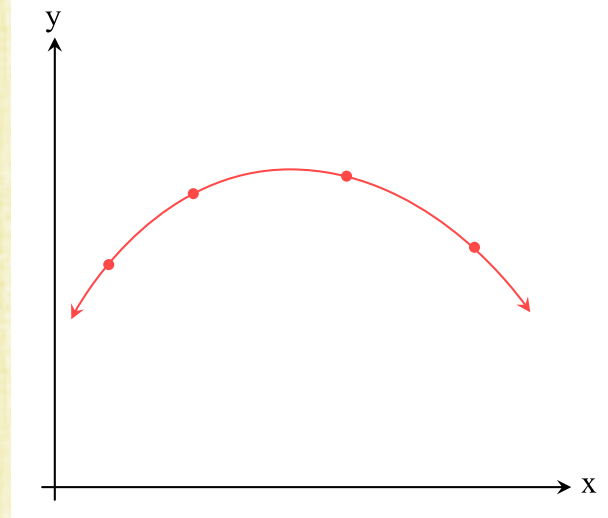
Local Approximations

Sampling

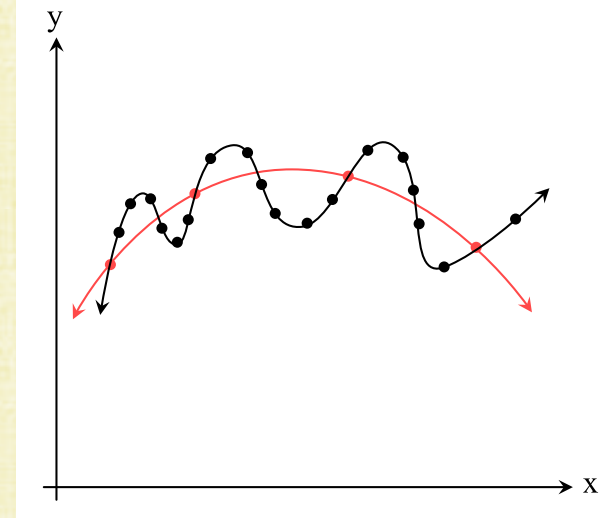
- Accurate approximation of a function is often limited by the amount of available data
- Given too few samples (left), one may “hallucinate” an incorrect function
- Adding more data allows for more accurate feature resolution (right)
- Given “enough” sample points, a function will eventually be well-represented by them



under-resolved



resolved better with more data



Taylor Expansion

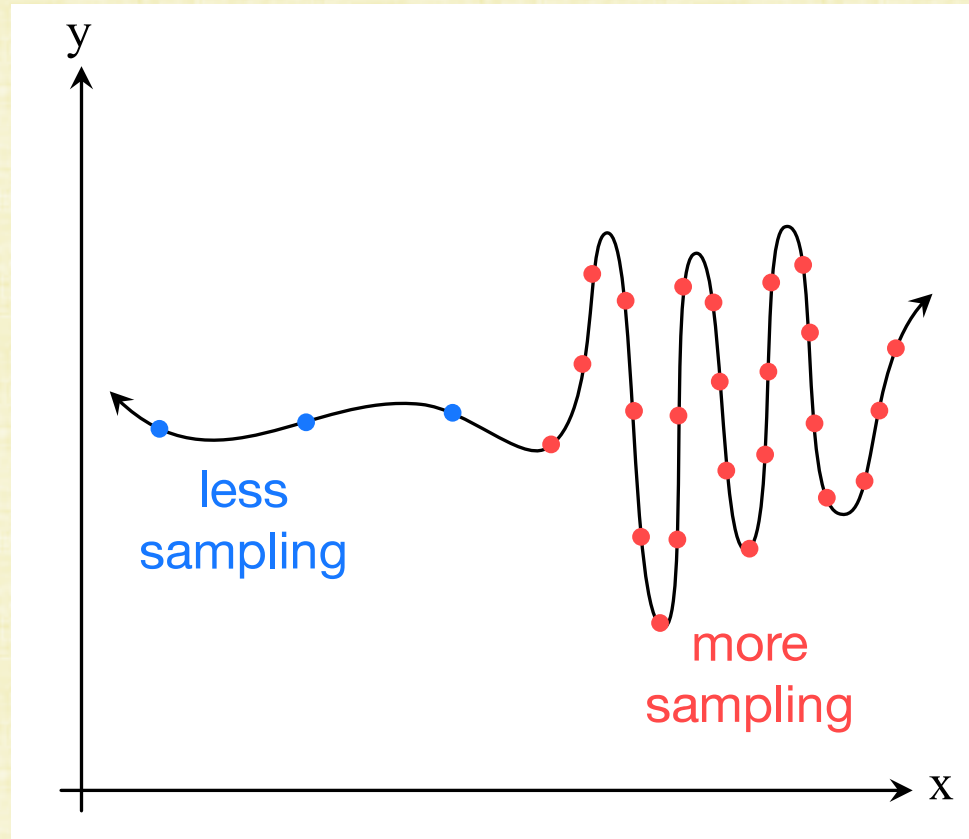
- $f(x + h) = \sum_{p=0}^{\infty} \frac{h^p}{p!} f^{(p)}(x) = \sum_{p=0}^{\hat{p}} \frac{h^p}{p!} f^{(p)}(x) + O(h^{\hat{p}+1})$
- Bounded derivatives would indicate that $O(h^{\hat{p}+1}) \rightarrow 0$ as $h \rightarrow 0$
- Examples:
 - $f(x + h) = f(x) + hf'(x) + \frac{h^2}{2} f''(x) + \frac{h^3}{6}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

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 derivatives 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

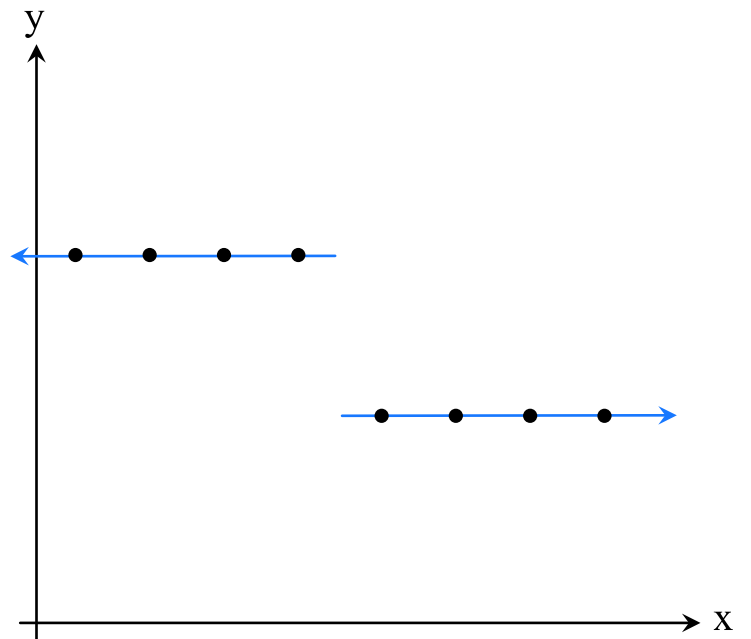
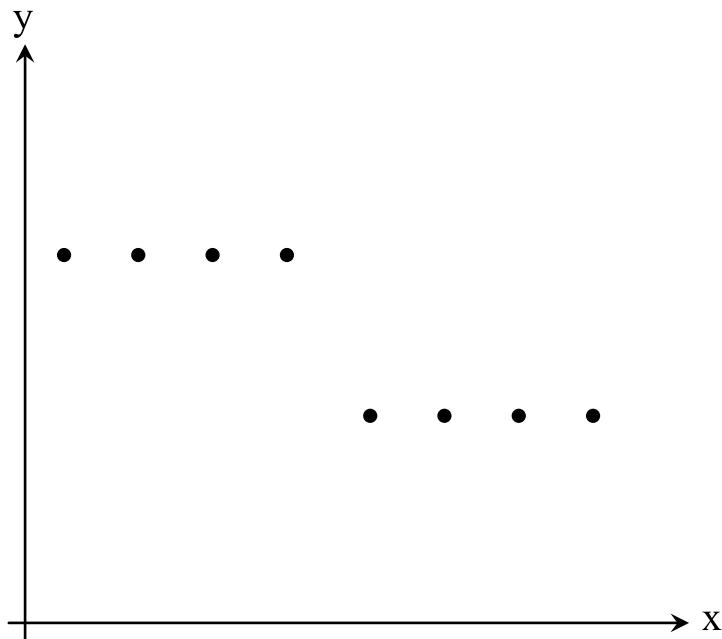
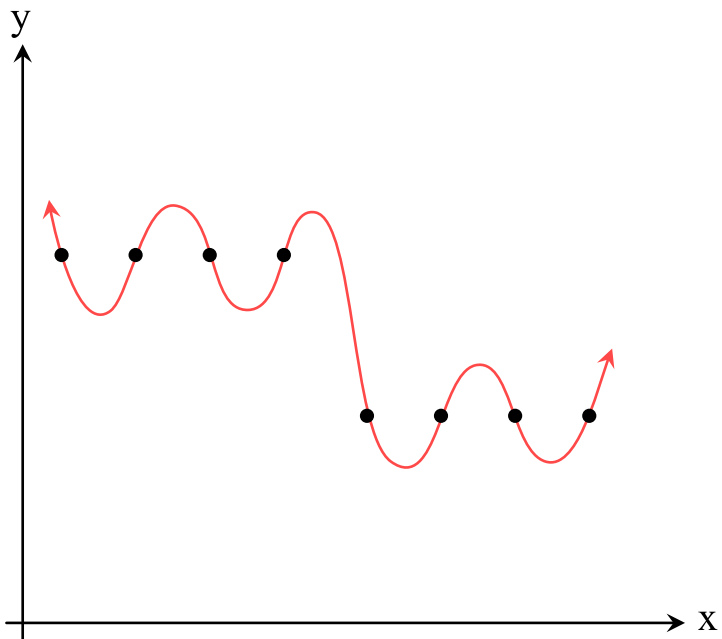
Well-Resolved Functions

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



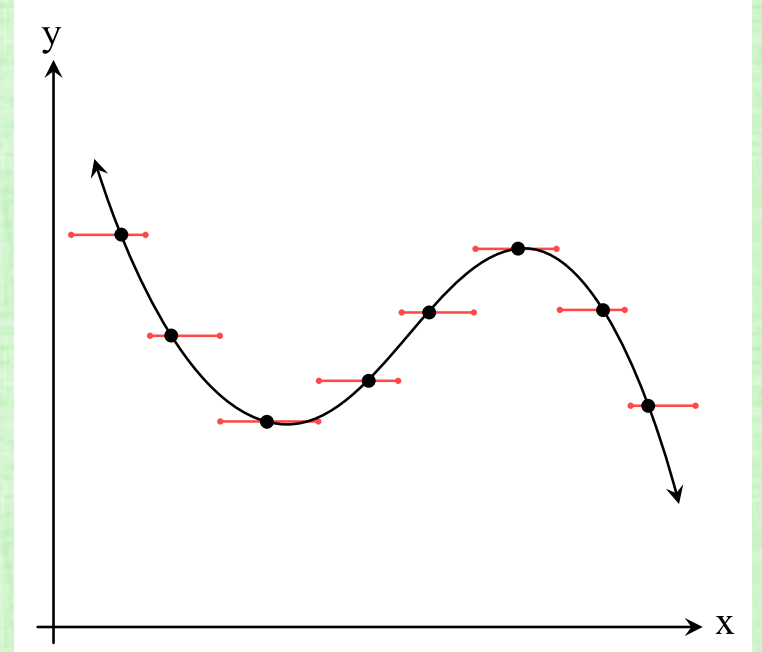
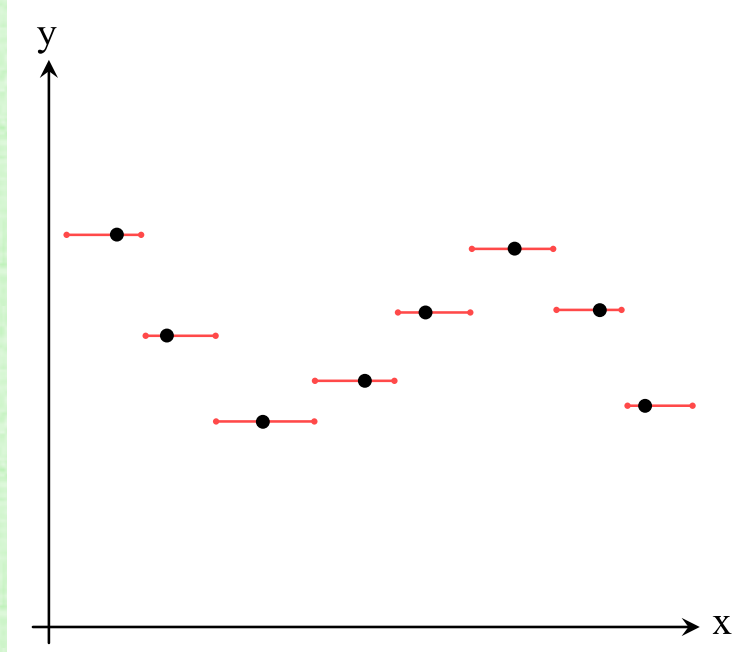
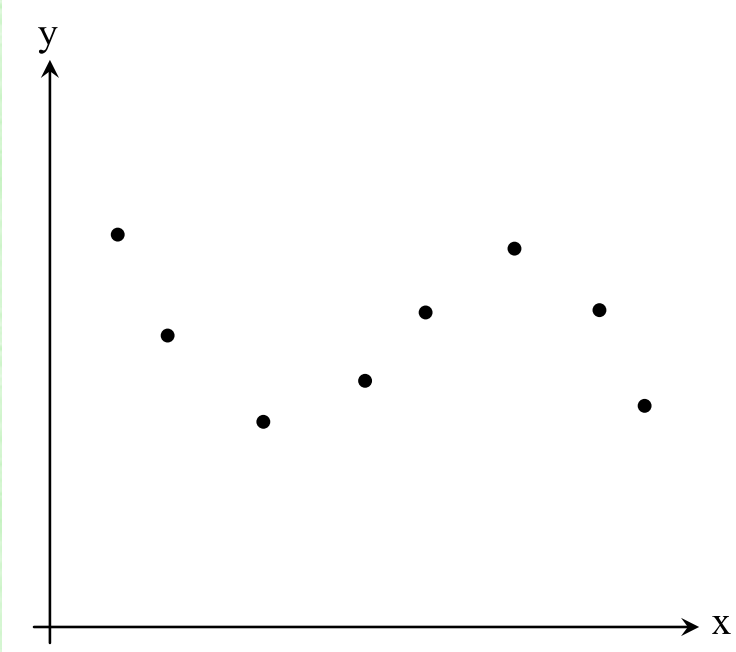
Piecewise Approximation

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



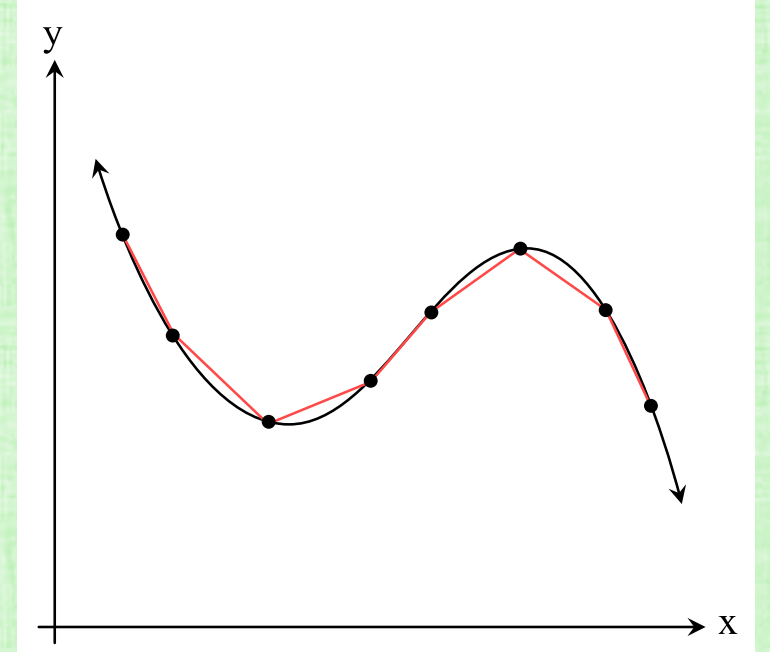
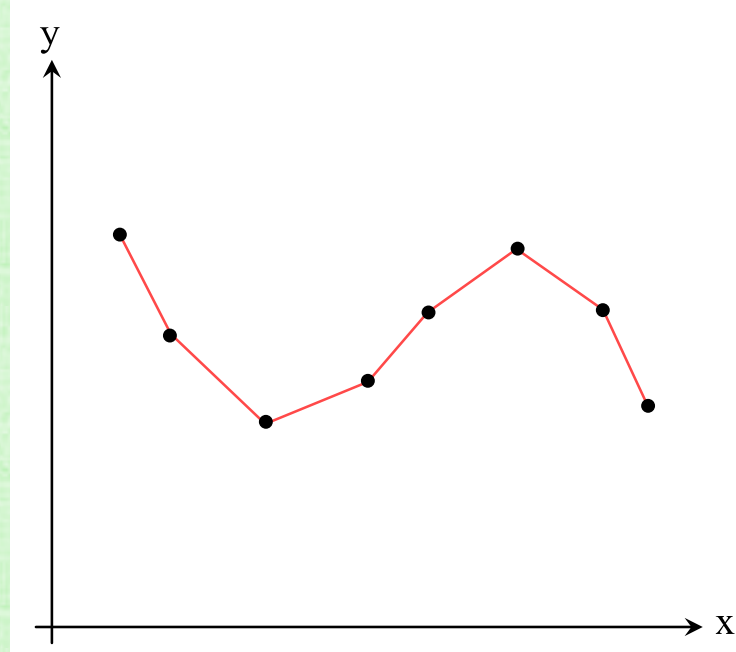
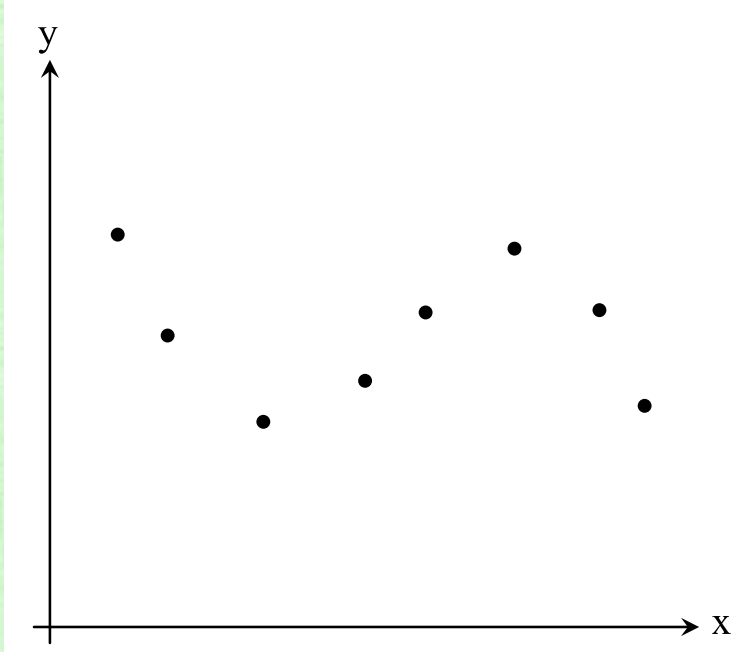
Piecewise Constant Interpolation

- 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 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

Piecewise Cubic Interpolation (B-Splines)

- Piecewise cubic splines are quite popular because of their ability to match derivatives across approximation boundaries
- B-splines – hierarchical family: ϕ_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
 - Recursively: $\phi_i^{p+1}(x) = w_i^p(x)\phi_i^p(x) + (1 - w_{i+1}^p(x))\phi_{i+1}^p(x)$
 - A linear $w_i^p(x) = \frac{x-x_i}{x_{i+p+1}-x_i}$ increases the polynomial degree of ϕ_i^p by one
 - Piecewise linear ϕ_i^1 , piecewise quadratic ϕ_i^2 , piecewise cubic ϕ_i^3 , etc.

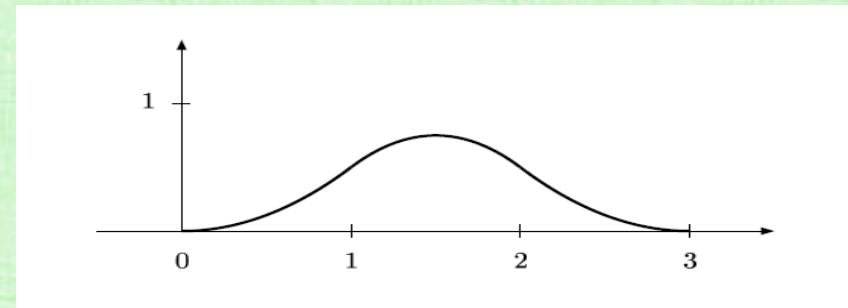
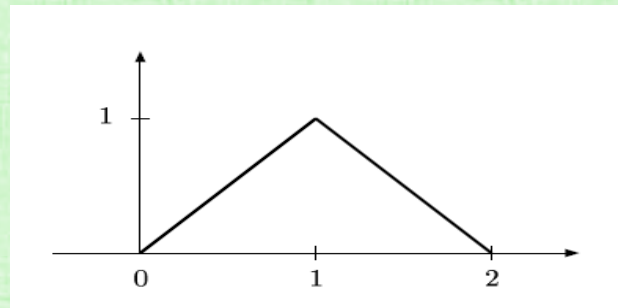
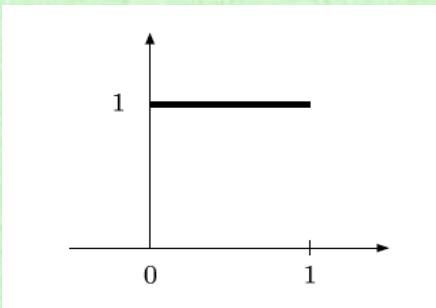


Image Segmentation

- The goal is to divide image pixels into separate regions, each representing separate objects or groups of objects
- Before neural networks: Various methods relied on clustering in color and/or space, graph-cuts, edge detection, etc.
- Since humans do well on this problem, can use neural networks to aim to mimic human perception/semantics
- Training examples:
 - Input: an image (all the pixel RGB values)
 - Output: labels on all the pixels, indicating what group each pixel is in

Bool Output Labels

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



Input



Output

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



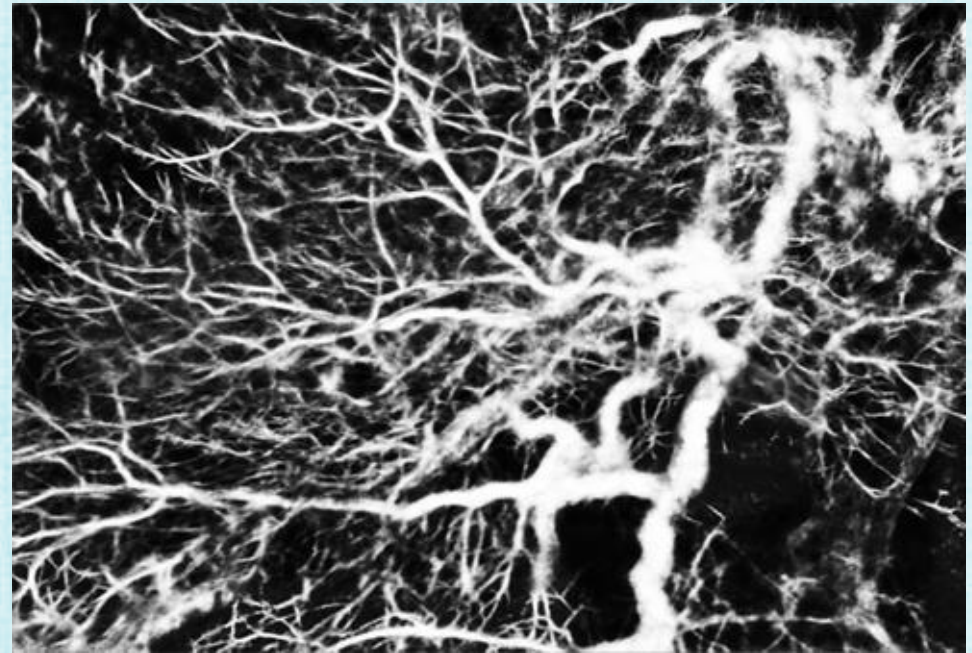
Output

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?

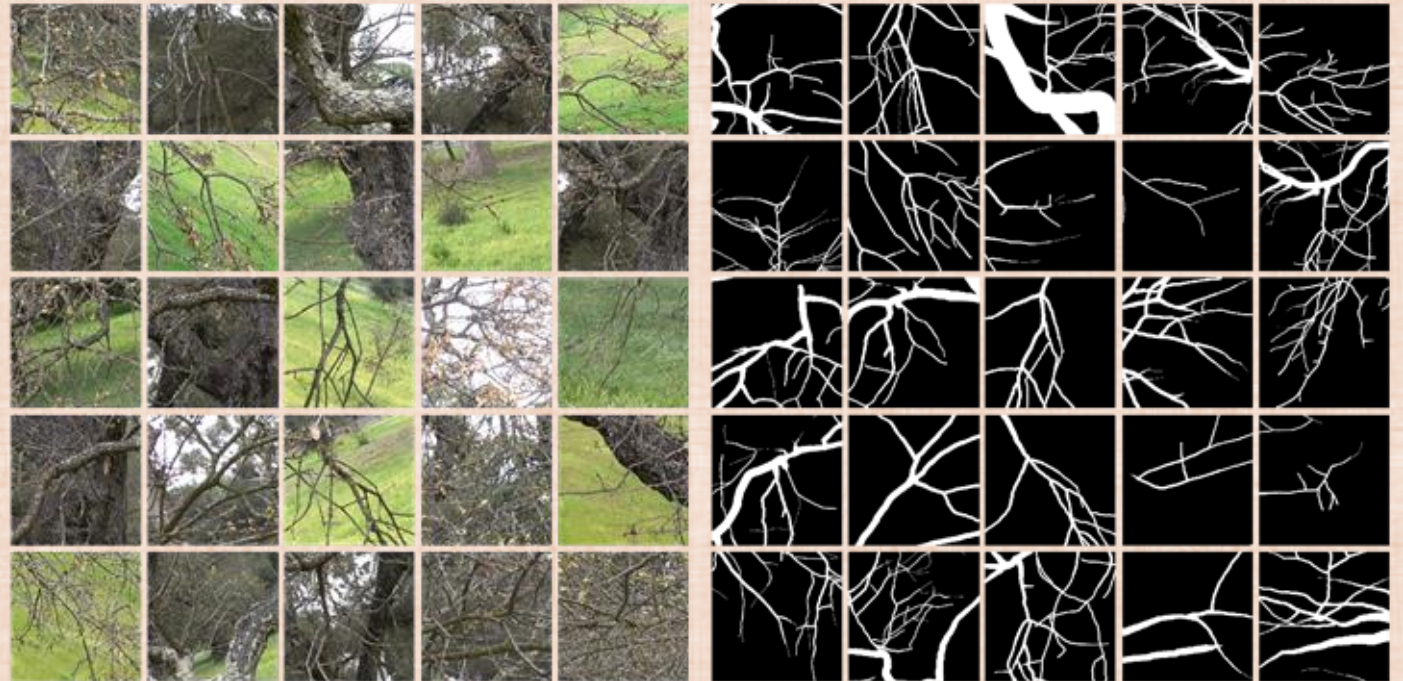
Constructing Training Data

- 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



Artificially Augmenting 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)

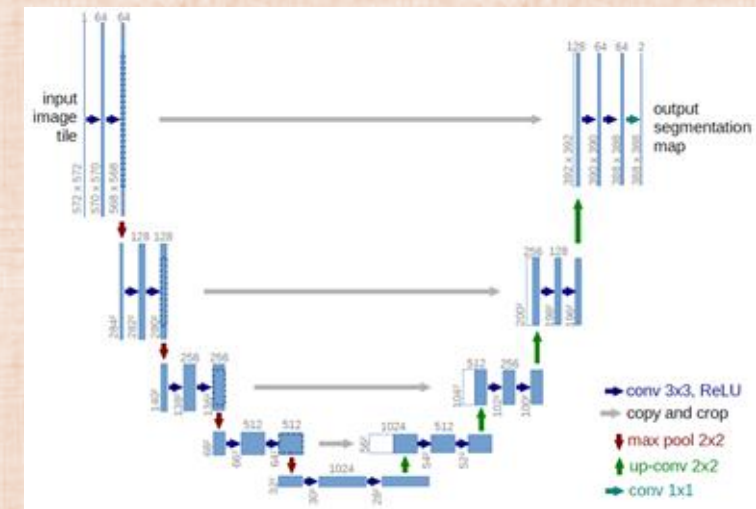


3840 pixels wide, 2160 pixels tall

each image: 512 pixels wide, 512 pixels tall

Training the Neural Network

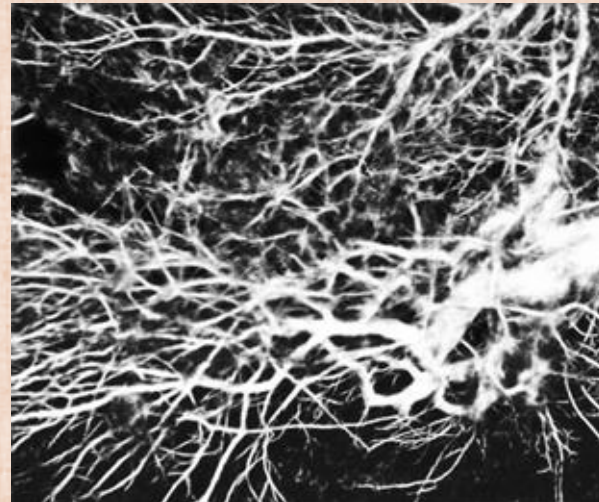
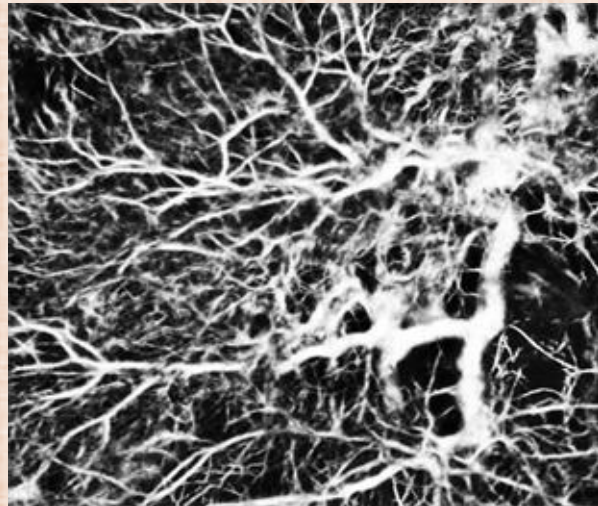
- Find function parameters c such that the network function $f_c(x)$ gives minimal error on the training data (i.e. minimize a “loss” function)
- The network should predict the known target labels (or close to it) from the input images



network architecture

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 green grass or (2) clusters of brown branches



Train 2 Neural Networks

- Divide the training data into these two disparate sets:
 - k-means clustering on hue/saturation can be used to divide the training images into two separate clusters
- Train a separate network on each of the two sets of training data:
 - separate architecture, separate trainable parameters, etc.
- Note: this approach works for more than 2 clusters/networks too!

Combining the Network Inferred Outputs

- Given an input image, inference it (separately) on each of the two networks
- Then, combine the two predictions, using the network that makes the most sense locally in each part of the image (blend predictions when appropriate)

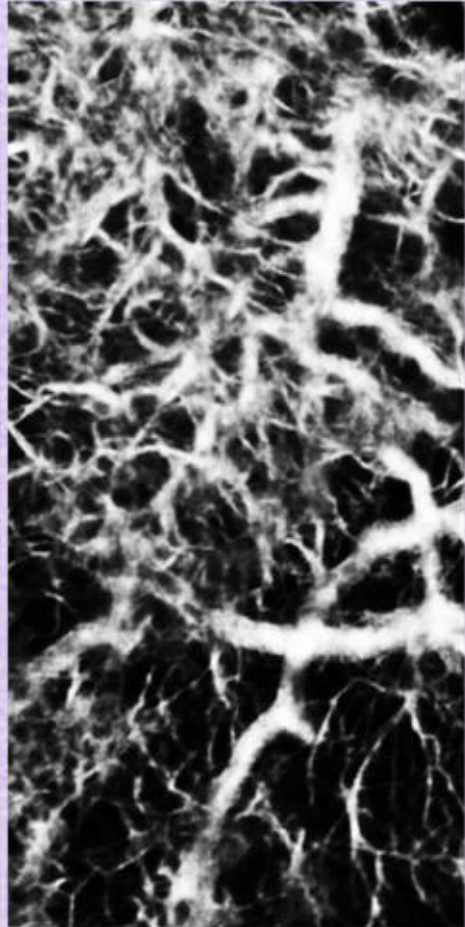
To inference each pixel:

- Compute hue/saturation values on a small patch around the pixel
- Find the distances from the patch hue/saturation values to the two cluster centers
- Interpolate the outputs from the two 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
- Note: this approach works for more than 2 clusters/networks too!

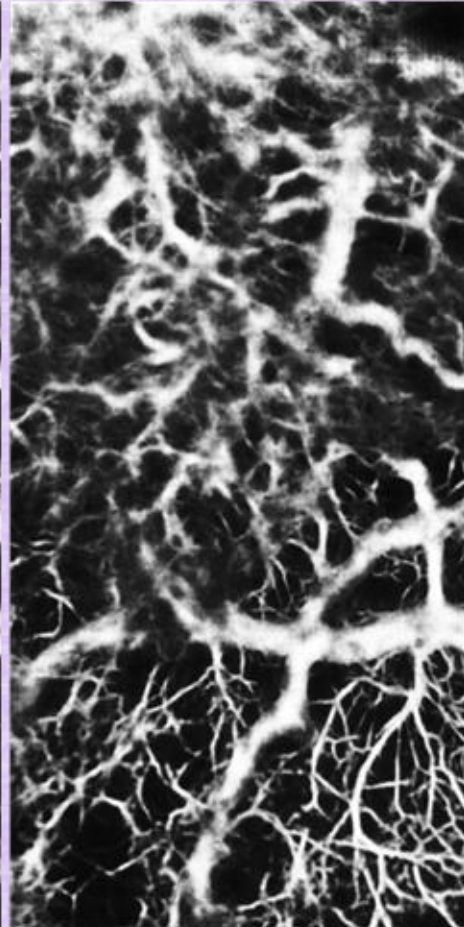
Example



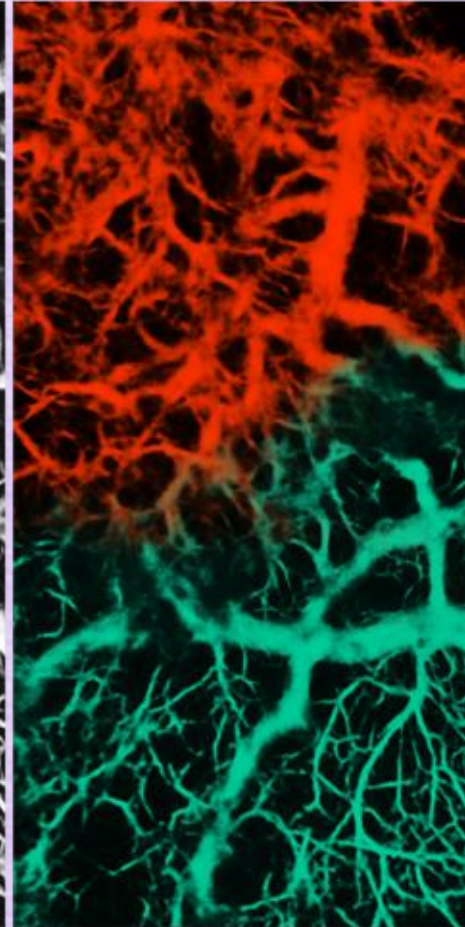
Input



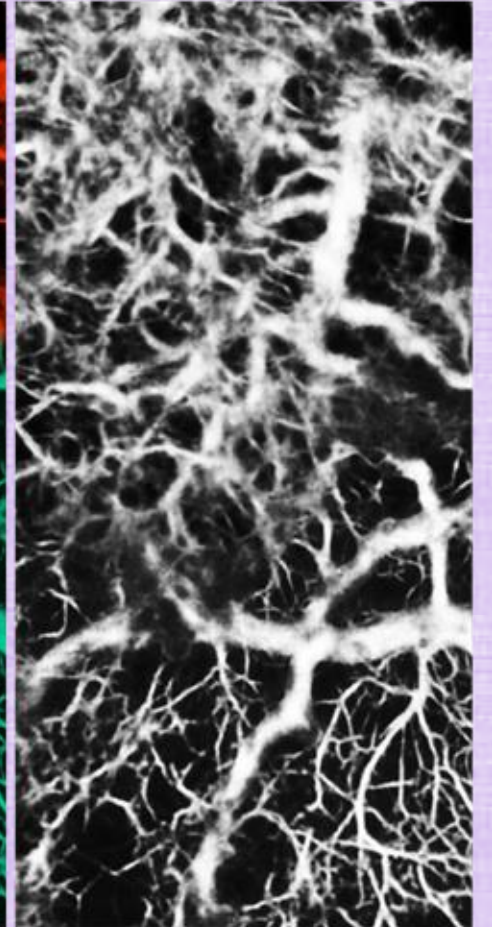
Network 1



Network 2

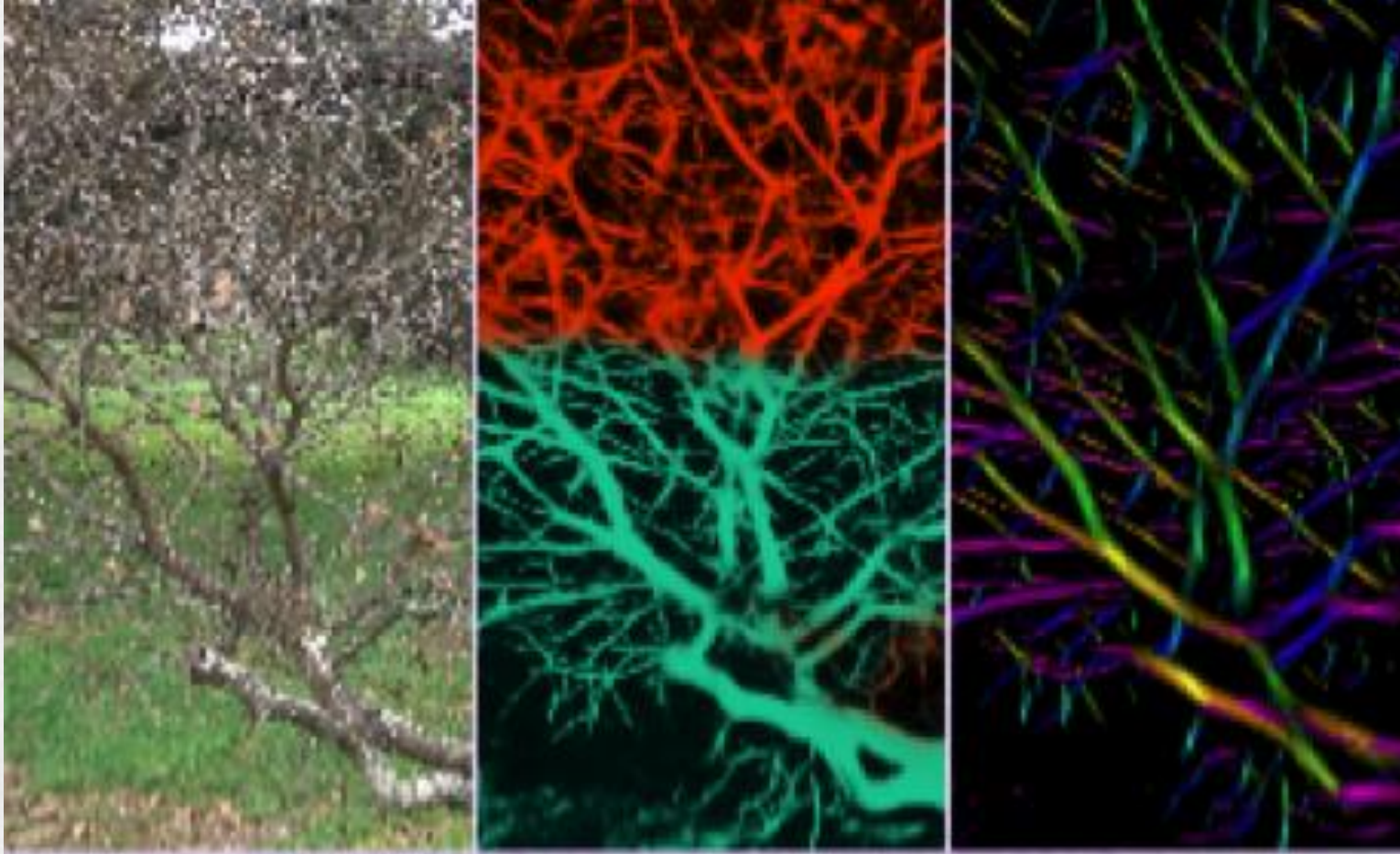


Combine



Final Result

Aside: Branch Estimation





Unit 7

Curse of Dimensionality

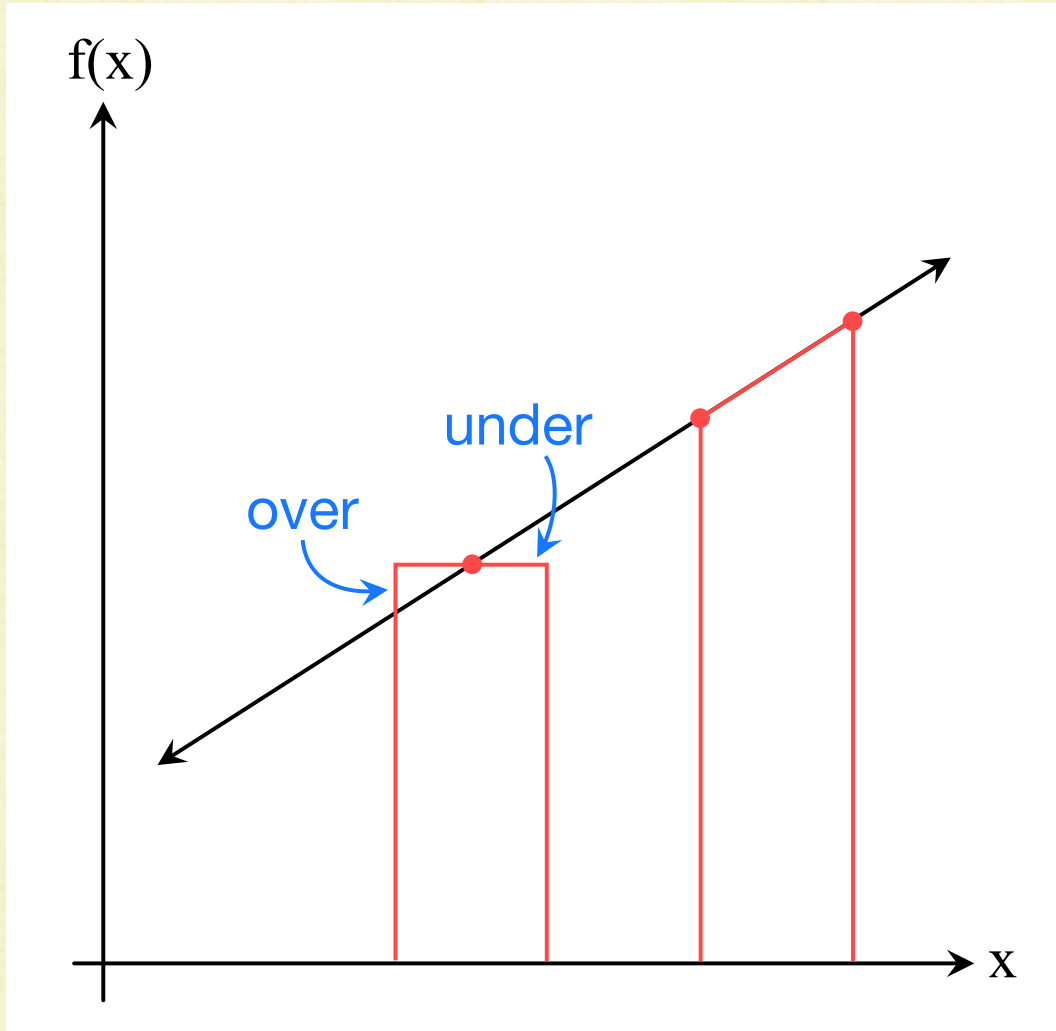
Numerical Integration (i.e., Quadrature)

- 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, it could be expensive to “evaluate” $f(x_i)$, especially when it requires performing experiments, gathering data, hand labeling, running code, etc.

Newton-Cotes Quadrature

- Uses a local approximation on each subinterval
- 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
- This gives 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 (one degree higher than expected)

Symmetric Cancellation



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

Newton-Cotes Quadrature

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

Recall: 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:
 - $f(x + h) = f(x) + hf'(x) + \frac{h^2}{2} f''(x) + \frac{h^3}{6} f'''(x) + O(h^4)$ looking forward
 - $f(x - h) = f(x) - hf'(x) + \frac{h^2}{2} f''(x) - \frac{h^3}{6} f'''(x) + O(h^4)$ looking backward
- Truncated Taylor expansions become more valid approximations as $h \rightarrow 0$
 - $f(x + h) \approx f(x) + hf'(x) + \frac{h^2}{2} f''(x) + \frac{h^3}{6} f'''(x)$ looking forward
 - $f(x - h) \approx f(x) - hf'(x) + \frac{h^2}{2} f''(x) - \frac{h^3}{6} f'''(x)$ looking backward

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 a per interval area error (local 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 error (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 $O\left(\left(\frac{h}{2}\right)^{p+1}\right) = \left(\frac{1}{2}\right)^{p+1} O(h^{p+1})$ total/global error, or $\left(\frac{1}{2}\right)^{p+1}$ as much total/global error
- This is denoted by an order of accuracy of $p + 1$

Newton-Cotes Quadrature (Methods)

- Midpoint Rule: $\sum_i h_i f(x_i^{mid})$
 - 1 point, piecewise constant approximation, exact for piecewise linear functions, $O(h^2)$ error, 2nd order accurate
- Trapezoidal Rule: $\sum_i h_i \frac{f(x_i^{left}) + f(x_i^{right})}{2}$
 - 2 points, piecewise linear approximation, exact for piecewise linear functions, $O(h^2)$ error, 2nd 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 approximation, exact for piecewise cubic functions, $O(h^4)$ error, 4th order accurate

Gaussian Quadrature

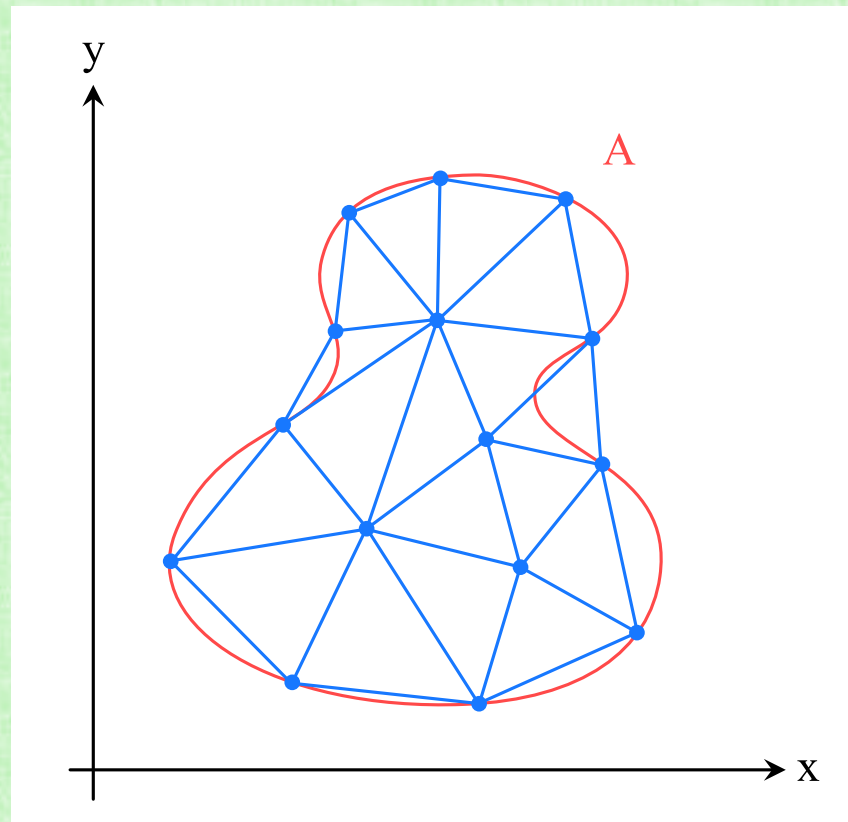
- Use p optimally chosen points to obtain a method that is exact on degree $2p - 1$ polynomials, and thus has an order of accuracy of $2p$
- For example:
$$\sum_i h_i \frac{f\left(x_i^{mid} - \frac{h_i}{2\sqrt{3}}\right) + f\left(x_i^{mid} + \frac{h_i}{2\sqrt{3}}\right)}{2}$$
- 2 points, piecewise cubic approximation, exact for piecewise cubic functions, $O(h^4)$ error, 4th order accurate
- Same accuracy as the 3 point Simpson's 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 each sub-region dA of area A is approximated locally
- 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 are just points instead of curves)

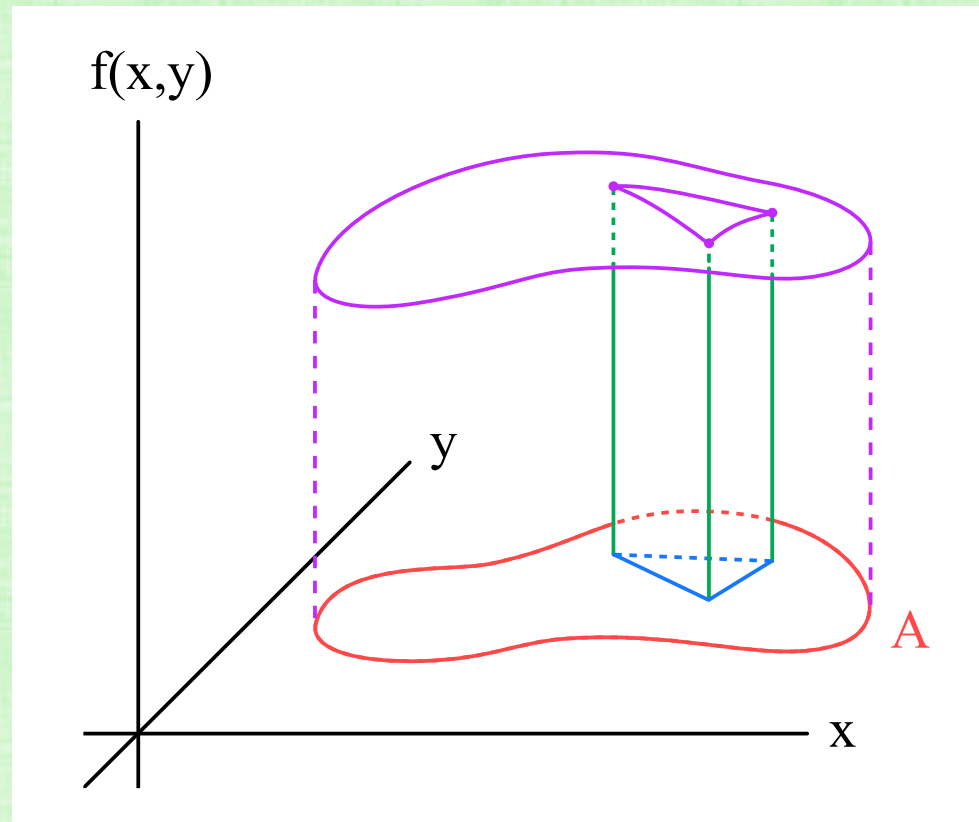
Domain Approximation Errors

- The difference between A and its approximation (via triangles here) leads to a new source of error in the integral (both missing and extra areas)



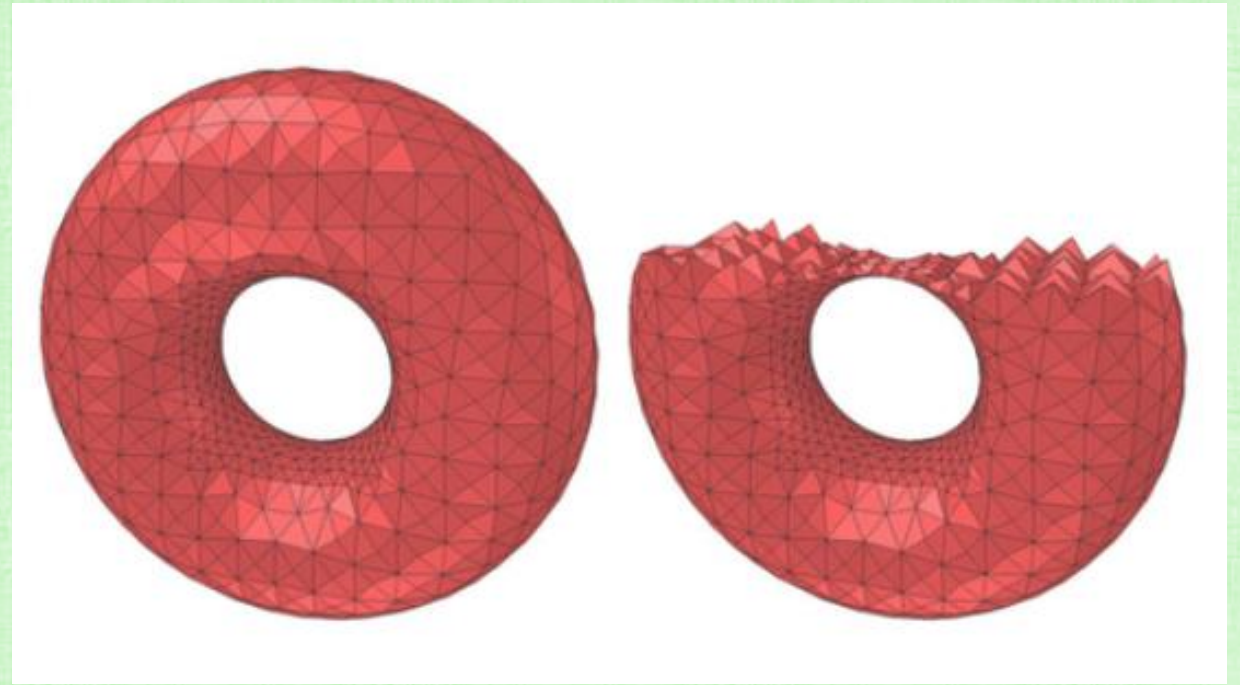
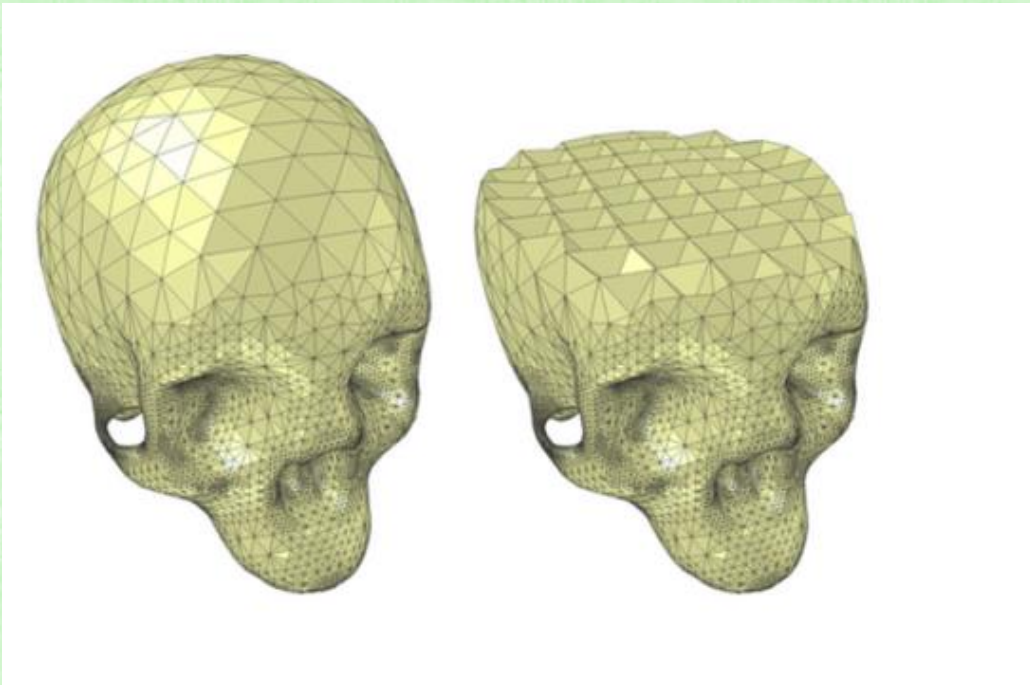
Integrating over Sub-regions

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



Three Dimensions

- $\iiint_V f(x, y, z) dV$ where each tetrahedral sub-region dV of volume V is approximated locally (with Gaussian quadrature points)



Curse of Dimensionality

- Consider a 1st order accurate method with $O(h)$ error
- 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 $32^2=1024$ x work
- Cutting error by a factor of 8 in 5D takes $32^3=32,768$ x 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

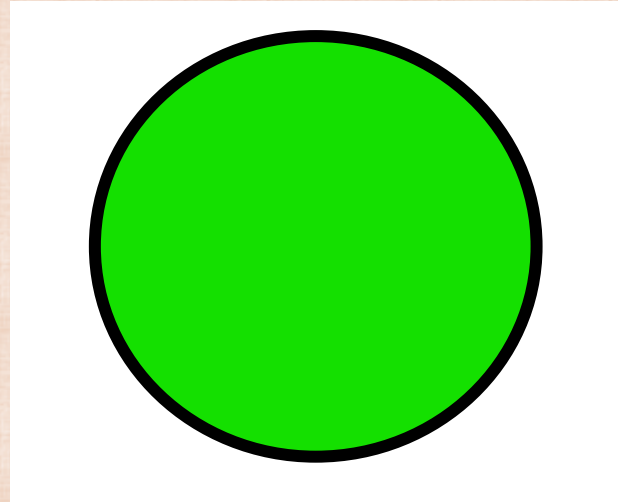
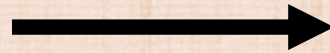
- Consider a 2nd order accurate method with $O(h^2)$ error
- In 1D/2D/3D/4D/5D/etc. halving the interval size gives 4 times less error
- Cutting error by a factor of 4 in 5D takes 32x work
- If the original code took 1 sec to run in 5D, cutting error by a factor of 16 takes only 17 min (much faster than the 12 days for the 1st order accurate method)
- Cutting error by a factor of 1024 (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 (or 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
- Sometimes they can work ok in 4D
- They are never a good idea in 5D+

A Simple Example

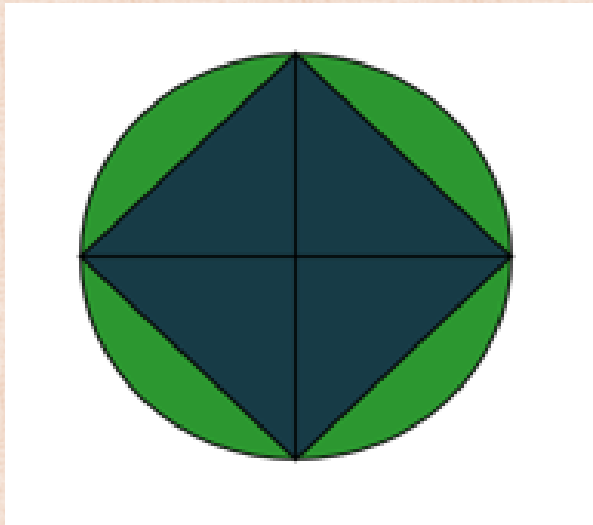
- Consider approximating $\pi = 3.1415926535 \dots$
- Use a compass to construct a circle with radius = 1
- Since $A = \pi r^2$, the area of this unit circle is π
- Integrate $f(x, y) = 1$ over the unit circle to obtain $\iint_A f(x, y) dA = \pi$



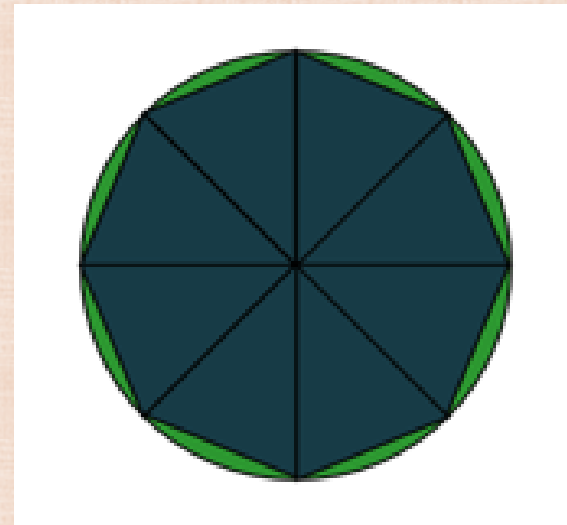
Area = π

Newton-Cotes Approach

- Inscribe triangles inside the circle
- Sum the area of all the triangles (no need to trivially multiply by the height = 1)
- The difference between the area A and its approximation with triangles leads to errors



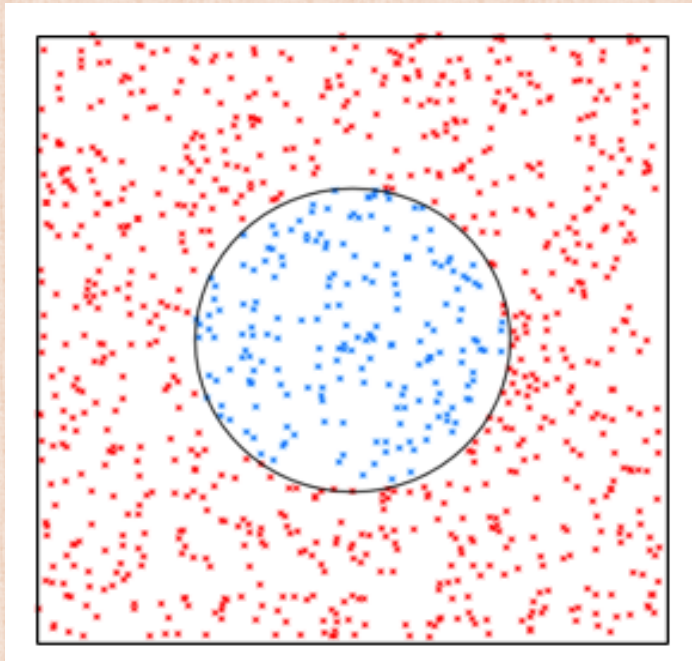
$$\pi \approx 2$$



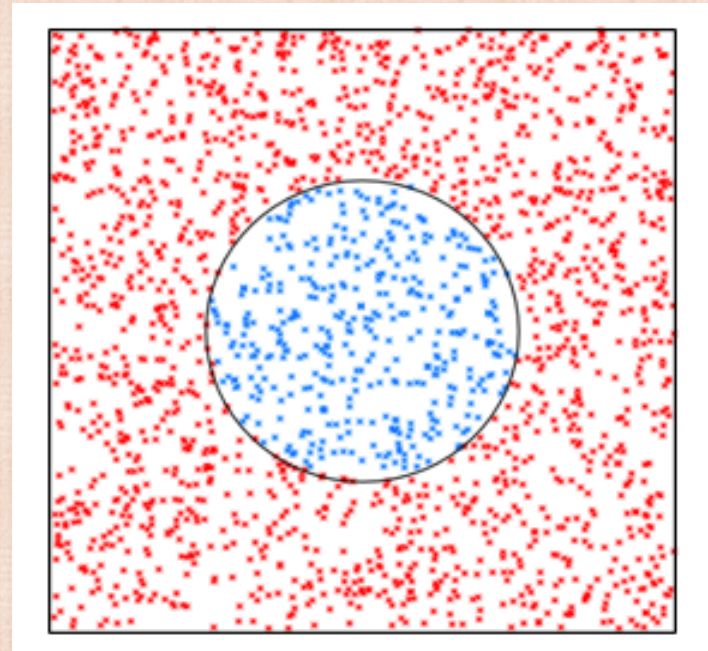
$$\pi \approx 2.8284$$

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 $\frac{A_{circle}}{A_{box}} = \frac{\pi}{16}$, can approximate $\pi \approx 16 \left(\frac{N_{blue}}{N_{blue} + N_{red}} \right)$



$$\pi \approx 3.136$$



$$\pi \approx 3.1440$$

Monte Carlo Integration

- 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 ½ order accurate)
 - E.g. 4 times more sample points are needed for ½ the error
 - 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 approach for high dimensional problems

Machine Learning Implications

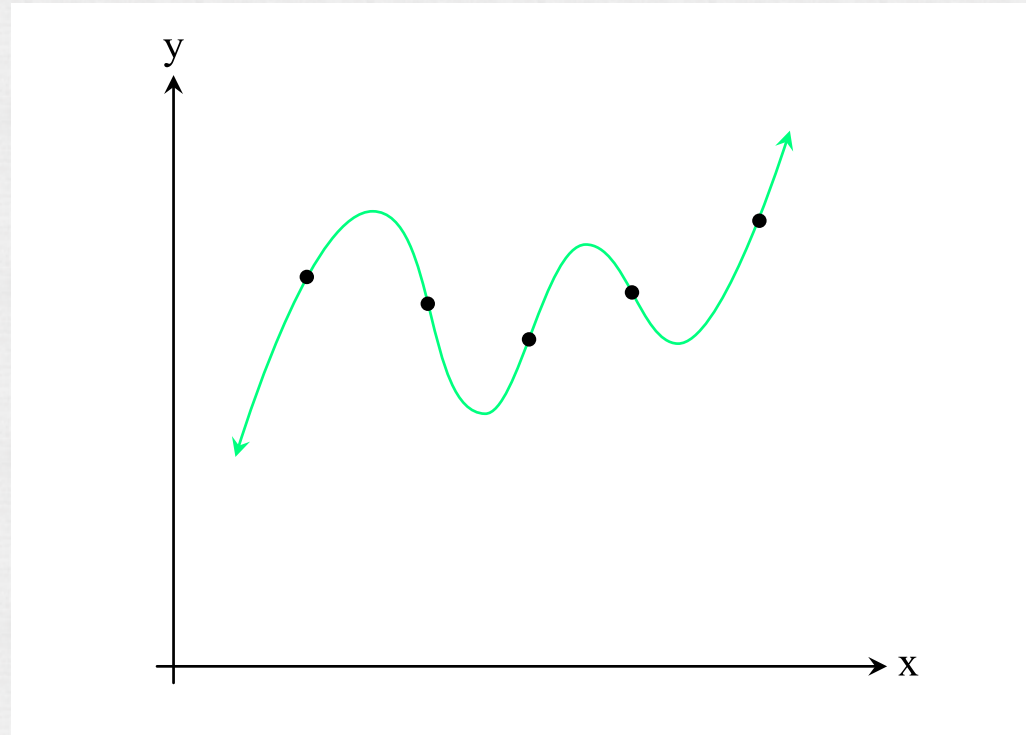
- Consider $y = f(x)$ where $x \in R^n$ with large n
- Newton-Cotes style approaches would first do polynomial interpolation, and then analytically integrate the result
- An intractable number of points and control volumes is required to construct polynomial functions in higher dimensions (the curse of dimensionality)
- The same is true when constructing an approximation to $y = f(x)$ for function interpolation for the sake of neural network inference, i.e. it is intractable in higher dimensions (too many control volumes, too much data, etc.)
 - However, it's still a good way to visualize or think about things to build intuition, especially since intuition is best built in R^1 , R^2 , and R^3
- Monte Carlo approaches are far more efficient in higher dimensions
 - This is a major reason for the close collaborations between CS ML/DL folks and Statistics departments
 - Classical engineering disciplines tend to operate in a lower dimensional 3D model of the physical world, and thus they typically have closer ties to Applied Mathematics (than to Statistics)

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're not degenerate (e.g. 3 points on a line)



Recall: Basis Functions (Unit 1)

- Given basis functions ϕ and unknown parameters c :

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

- Monomial basis: $\phi_k(x) = x^{k-1}$

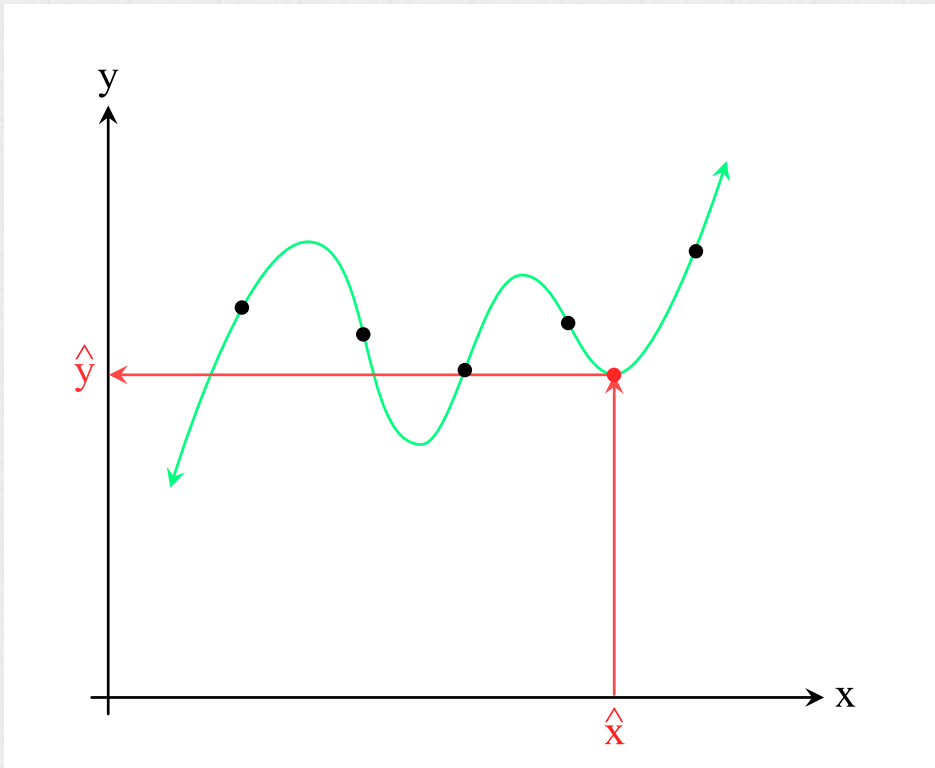
- Lagrange basis: $\phi_k(x) = \frac{\prod_{i \neq k} x - x_i}{\prod_{i \neq k} x_k - x_i}$

- Newton basis: $\phi_k(x) = \prod_{i=1}^{k-1} x - x_i$

- Write a (linear) equation for each data point, and put into matrix form: $Ac = y$
- Monomial/Lagrange/Newton basis all give the same polynomial, but different matrices need to be inverted in each case

Recall: Overfitting (Unit 1)

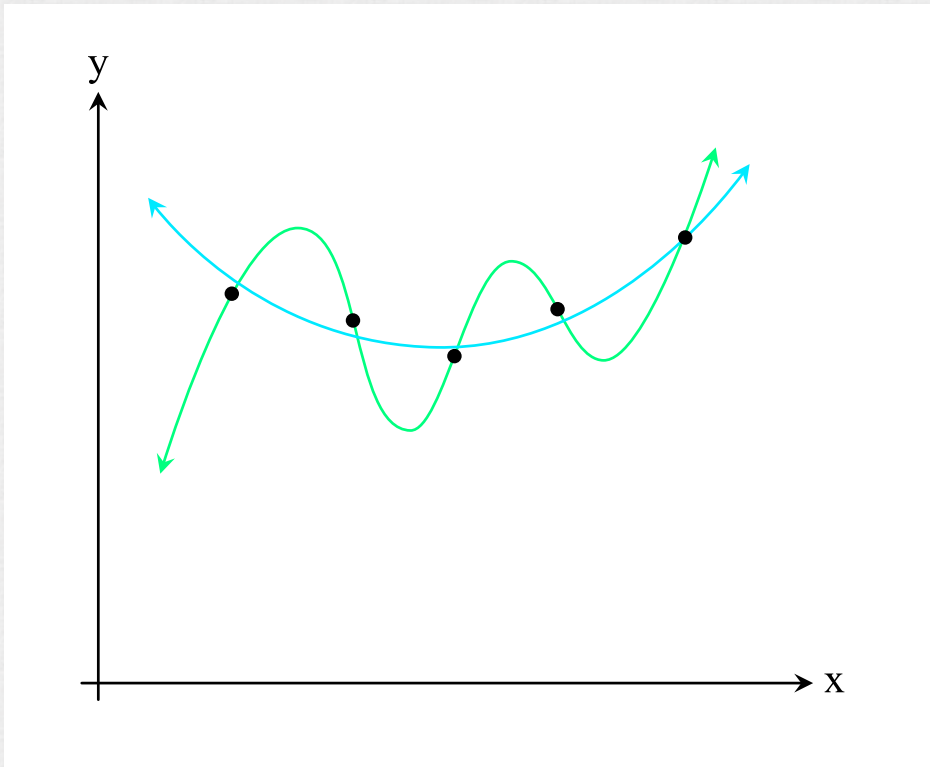
- Given a new input \hat{x} , the interpolating polynomial infers/predicts an output \hat{y} that may be far from what one may expect



- 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)
- Overfitting leads to inference/predictions that can vary wildly from the training data

Recall: Regularization (Unit 1)

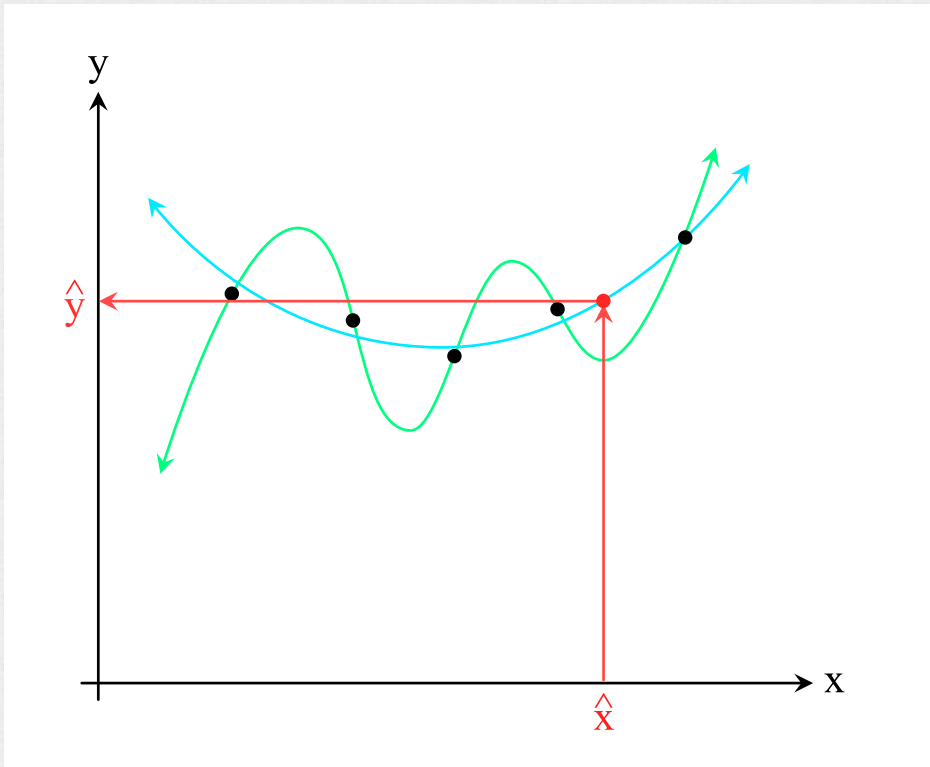
- Using **a lower order polynomial** that doesn't (can't) exactly fit the data points **provides regularization**



- A regularized interpolant contains intentional error, 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)

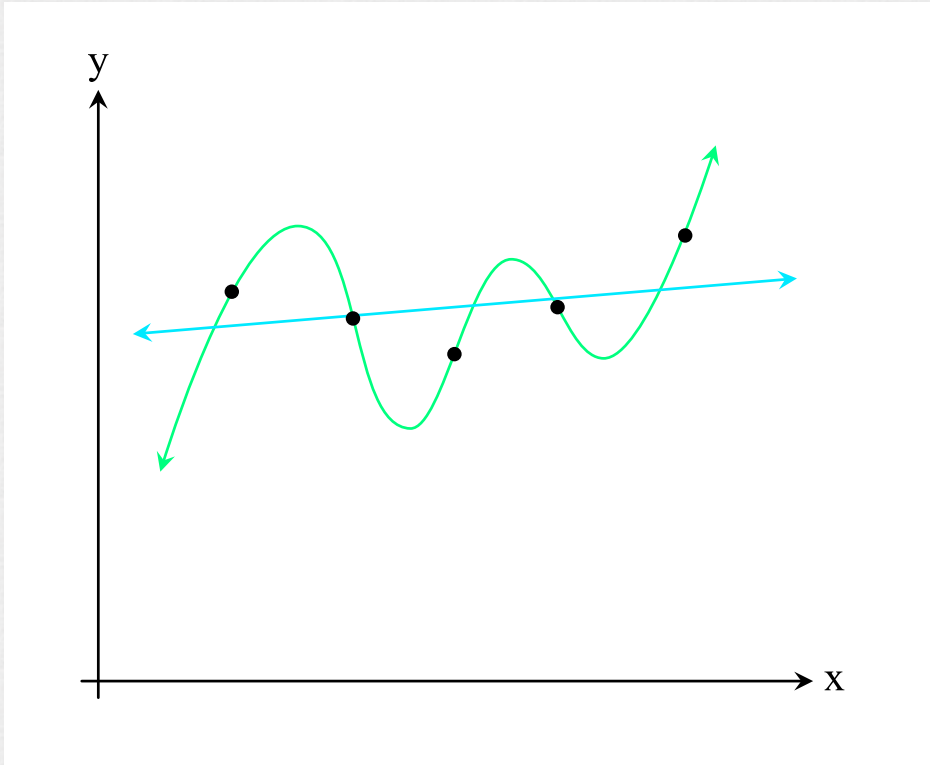
- Given \hat{x} , the regularized interpolant infers/predicts a more reasonable \hat{y}



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

Recall: Underfitting (Unit 1)

- Using too low of an order polynomial causes it 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

Eliminating Basis Functions

- Consider $Ac = y$:
 - Each row of A evaluates all n basis functions ϕ_k on a single data point x_i
 - Each column of A evaluates all m data points x_i on a single basis function ϕ_k
- Regularize by reducing the number of basis functions (**remove columns**)
 - This reduces the degree of the polynomial
 - Still write an equation for each point, and put into matrix form $Ac = y$ (as usual)
- When there are more points than basis functions, there will be more rows than columns (the matrix will be tall/rectangular)
 - This tall matrix has full (column) rank when the columns are linearly independent
 - This requires both well-chosen basis functions and non-degenerate data

Understanding Tall **Full Rank** Matrices

- Let A be a size $m \times n$ tall (i.e. $m \geq n$) 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 other equations
 - 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 with: **the last $m - n$ rows of Σ are 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 (only rank 2, not rank 3)
- The rank of a matrix is equal to its number of nonzero singular values

Recall: Extra Dimensions (Unit 3)

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

$$\begin{pmatrix} .141 & .825 & -.420 & -.51 \\ .344 & .426 & .298 & .78 \\ .547 & .028 & .644 & -.59 \\ .750 & -.371 & -.542 & .09 \end{pmatrix} \begin{pmatrix} 25.5 & 0 & 0 \\ 0 & 1.29 & 0 \\ 0 & 0 & 0 \\ \blacksquare & \blacksquare & \blacksquare \end{pmatrix} \begin{pmatrix} .504 & .574 & .644 \\ -.761 & -.057 & .646 \\ .408 & -.816 & .408 \end{pmatrix}$$

- 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 Σ (and still obtain a valid factorization)

Understanding Tall **Full Rank** Linear Systems

- $Ac = b$ becomes $U\Sigma V^T c = b$ or $\Sigma(V^T c) = (U^T b)$ or $\Sigma\hat{c} = \hat{b}$
- Solve $\Sigma\hat{c} = \hat{b}$ by dividing the entries of \hat{b} by the all nonzero (because full rank) singular values σ_k ; then, $c = V\hat{c}$
- The last $m - n$ equations are identically zero on the left, and need to be identically zero on the right as well in order for a solution to exist
 - 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 for a solution to exist
- 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

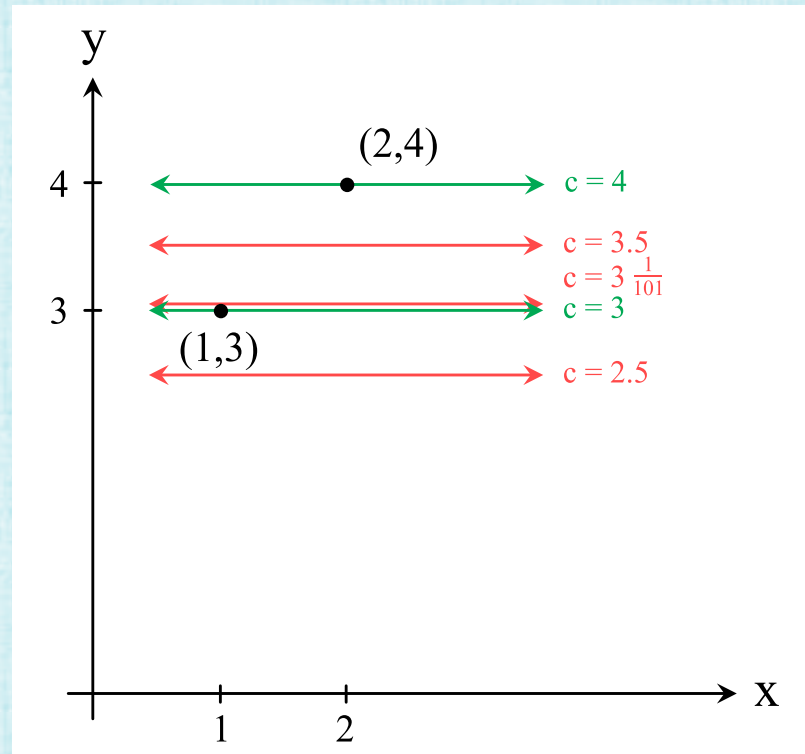
- Consider a very simple $Ac = b$ given by: $\begin{pmatrix} 1 \\ 1 \end{pmatrix} (c) = \begin{pmatrix} 3 \\ 4 \end{pmatrix}$
- This contains the equations $c = 3$ and $c = 4$, and as such is a false statement
- Solve via $\begin{pmatrix} 1 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} (c) = \begin{pmatrix} 1 & -1 \end{pmatrix} \begin{pmatrix} 3 \\ 4 \end{pmatrix}$, 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 $\begin{pmatrix} 10 & -1 \end{pmatrix} \begin{pmatrix} 10 \\ 1 \end{pmatrix} (c) = \begin{pmatrix} 10 & -1 \end{pmatrix} \begin{pmatrix} 30 \\ 4 \end{pmatrix}$, so $101c = 304$ or $c = 3 \frac{1}{101} \approx 3.01$
- 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 $\text{row } 2$ 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!
- Issue: $\begin{pmatrix} 3 \\ 4 \end{pmatrix}$ 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 $\forall c \in \mathcal{R}$

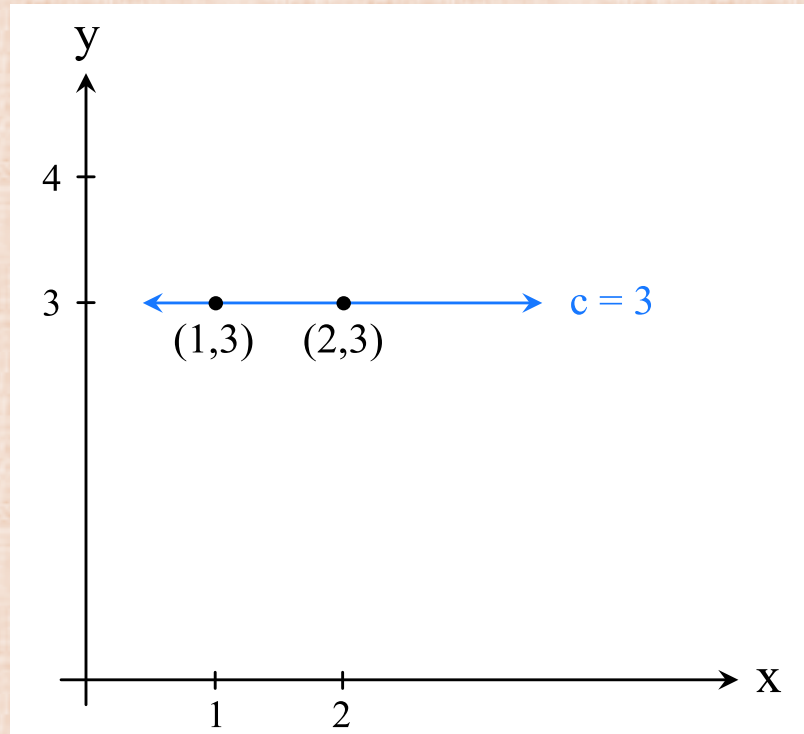
False Statements

- 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 $\begin{pmatrix} 1 \\ 1 \end{pmatrix} (c_1) = \begin{pmatrix} 3 \\ 3 \end{pmatrix}$ 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 that satisfies 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 $c_1 = 3\frac{1}{3}$ 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 typically change the answer
- Thus, **one should carefully choose the residual they want to minimize**
- 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 by minimizing $\|Dr\|$ for a diagonal matrix D with non-zero diagonal entries
- This is equivalent to row scaling: $Dr = Db - DAc$
- Column scaling doesn't affect the residual, e.g. $Dr = Db - DA\hat{D}^{-1}(\hat{D}c)$
- 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 (it's really **a** least squares solution)
 - A 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 $J(c) = \left(\frac{\partial f}{\partial c_1}(c) \quad \frac{\partial f}{\partial c_2}(c) \quad \cdots \quad \frac{\partial f}{\partial c_n}(c) \right)$
- 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 $J(\nabla f(c)) = \begin{pmatrix} \frac{\partial^2 f}{\partial c_1 \partial c_1}(c) & \frac{\partial^2 f}{\partial c_2 \partial c_1}(c) & \dots & \frac{\partial^2 f}{\partial c_n \partial c_1}(c) \\ \frac{\partial^2 f}{\partial c_1 \partial c_2}(c) & \frac{\partial^2 f}{\partial c_2 \partial c_2}(c) & \dots & \frac{\partial^2 f}{\partial c_n \partial c_2}(c) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial c_1 \partial c_n}(c) & \frac{\partial^2 f}{\partial c_2 \partial c_n}(c) & \dots & \frac{\partial^2 f}{\partial c_n \partial c_n}(c) \end{pmatrix}$

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

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

- When the order of differentiation doesn't matter, $H(c)$ is symmetric
- In 1D, this is the usual $f''(c)$

Differential Forms

- Vector valued function: $dF(c) = J(F(c))dc$
- Substitute ∇f for F to get: $d\nabla f(c) = J(\nabla f(c))dc = H^T(c)dc$

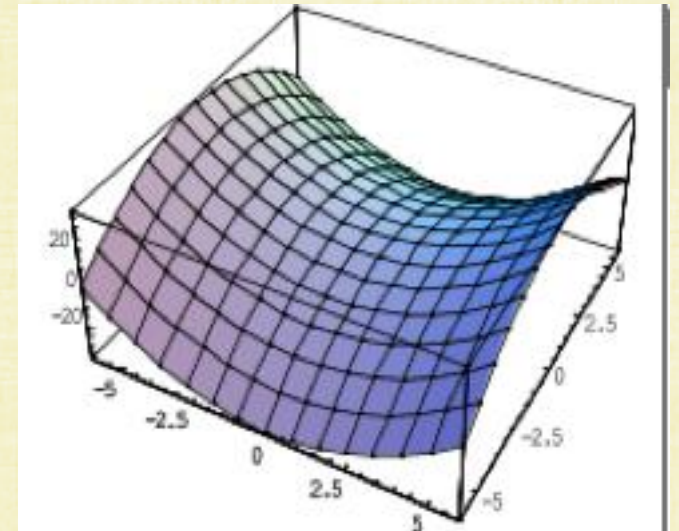
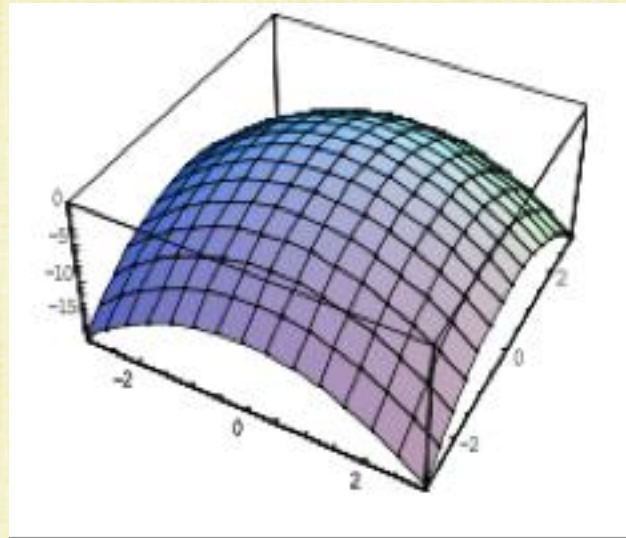
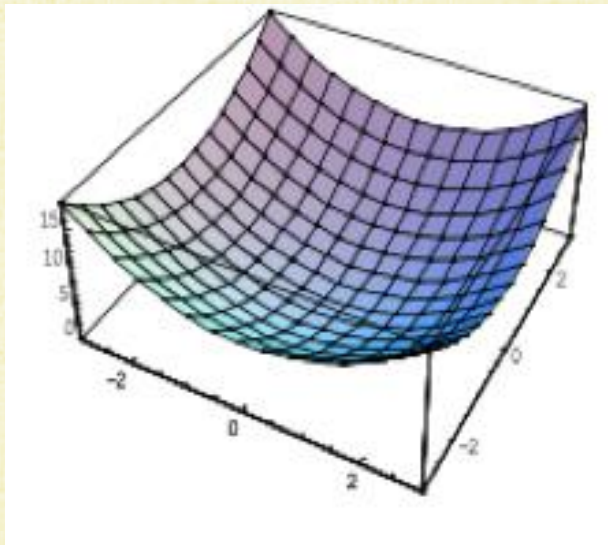
- Scalar valued function: $df(c) = J(f(c))dc$
- Take the transpose: $df(c) = dc^T \nabla f(c)$

- Take (another) differential: $d^2 f(c) = J(dc^T \nabla f(c))dc$
- Some hand waving: $d^2 f(c) = dc^T H^T(c)dc = dc \cdot H^T(c)dc$

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 1×1 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 $f(c) = \frac{1}{2} c^T \tilde{A} c - \tilde{b}^T c + \tilde{c}$
 - 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 $\nabla f(c) = \frac{1}{2} \tilde{A} c + \frac{1}{2} \tilde{A}^T c - \tilde{b}$, since $J(c^T v) = J(v^T c) = v^T$ (the **gradient is v**)
 - Solve the symmetric system $\frac{1}{2} (\tilde{A} + \tilde{A}^T) c = \tilde{b}$ to find critical points
- When \tilde{A} is symmetric, $\nabla f(c) = \tilde{A} c - \tilde{b} = 0$ is satisfied when $\tilde{A} c = \tilde{b}$
 - In 1D, the critical point is on the line of symmetry $\tilde{c} = \frac{\tilde{b}}{\tilde{a}}$
- That is, solve $\tilde{A} c = \tilde{b}$ 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 1×1 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 (it's really a least squares solution)
 - A 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 $c^T A^T D^2 Ac - 2b^T D^2 Ac$

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 $A^T D^2 A c = A^T D^2 b$
- For (unweighted) least squares where $D = I$, 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 $H = \tilde{A} = A^T A = V \Sigma^T U^T U \Sigma V^T = V \Sigma^T \Sigma V^T = V \Lambda V^T$
 - $\Lambda = \Sigma^T \Sigma$ is a size $n \times n$ matrix of nonzero singular values squared
- $HV = V\Lambda$ illustrates that H has all positive eigenvalues; so, H 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
- So, 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 it does change the answer when both $D \neq I$ and $m \neq n$ (when A is square with $m = n$, the unique solution doesn't change via row scaling)
- Minimizing $\|r\|_2 = \|b - Ac\|_2$ leads to the normal equations $A^T Ac = A^T b$ for the critical point
- Since $A^T A$ is SPD, $A^T Ac = A^T b$ has a unique solution obtainable via fast/efficient SPD solvers
- When b is in the range of A , the unique solution to $A^T Ac = A^T b$ makes $r = 0$, and thus it is the unique solution to $Ac = b$
 - Note: when A is square ($m = n$) and full rank, b is always in the range of A

Condition Number for the Normal Equations

- 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 $\frac{\sigma_{max}^2}{\sigma_{min}^2}$ of $A^T A$ is also squared as compared to $\frac{\sigma_{max}}{\sigma_{min}}$ for 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 for theoretical purposes
 - Can transform any full rank matrix into an SPD system

Understanding the Normal Equations

- When $A = U\Sigma V^T$ has full column rank, $\Sigma = \begin{pmatrix} \hat{\Sigma} \\ 0 \end{pmatrix}$ 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: $A^T A = V \begin{pmatrix} \hat{\Sigma} & 0^T \\ 0 \end{pmatrix} \begin{pmatrix} \hat{\Sigma} \\ 0 \end{pmatrix} V^T = V \hat{\Sigma}^2 V^T$ and $(A^T A)^{-1} = V \hat{\Sigma}^{-2} V^T$
- $c = (A^T A)^{-1} A^T b = V \hat{\Sigma}^{-2} V^T V \begin{pmatrix} \hat{\Sigma} & 0^T \\ 0 \end{pmatrix} U^T b = V \begin{pmatrix} \hat{\Sigma}^{-1} & 0^T \\ 0 \end{pmatrix} U^T b$
- $Ac = U \begin{pmatrix} \hat{\Sigma} \\ 0 \end{pmatrix} V^T V \begin{pmatrix} \hat{\Sigma}^{-1} & 0^T \\ 0 \end{pmatrix} U^T b = U \begin{pmatrix} I_{n \times n} & 0^T \\ 0 & 00^T \end{pmatrix} U^T b$
- $r = b - Ac = U I_{m \times m} U^T b - U \begin{pmatrix} I_{n \times n} & 0 \\ 0 & 00^T \end{pmatrix} U^T b = U \begin{pmatrix} 0^T 0 & 0^T \\ 0 & I_{(m-n) \times (m-n)} \end{pmatrix} U^T b$

Recall: Summary (Unit 3)

- 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 “smaller” singular values and still obtain a reasonable approximation of A

Understanding the Normal Equations

- A has n singular values
- So, only the first n columns of U (which has m columns) span the range of A
- Write $\begin{pmatrix} \hat{b}_r \\ \hat{b}_z \end{pmatrix} = U^T b$
- \hat{b}_r is size $n \times 1$, and it corresponds to the part of b in the range of A
- \hat{b}_z is size $(m - n) \times 1$, and it corresponds to the part of b orthogonal to the range of A
- Then: $c = V\hat{\Sigma}^{-1}\hat{b}_r$, $Ac = U \begin{pmatrix} \hat{b}_r \\ 0 \end{pmatrix}$, and $r = U \begin{pmatrix} 0 \\ \hat{b}_z \end{pmatrix}$

Recall: Singular Value Decomposition (Unit 3)

- Factorization of any (size $m \times n$) matrix: $A = U\Sigma V^T$
- Σ 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.

Aside: Orthogonal Matrices and the L2 norm

- An orthogonal \hat{Q} has $\hat{Q}\hat{Q}^T = \hat{Q}^T\hat{Q} = I$

- So,

$$\|\hat{Q}r\|_2 = \sqrt{\hat{Q}r \cdot \hat{Q}r} = \sqrt{r^T \hat{Q}^T \hat{Q}r} = \sqrt{r^T r} = \|r\|_2$$

$$\|\hat{Q}^T r\|_2 = \sqrt{\hat{Q}^T r \cdot \hat{Q}^T r} = \sqrt{r^T \hat{Q} \hat{Q}^T r} = \sqrt{r^T r} = \|r\|_2$$

- That is, orthogonal transformations preserve Euclidean distance

Understanding the Normal Equations

- Since U is orthogonal, $\|r\|_2 = \left\| U \begin{pmatrix} 0 \\ \hat{b}_z \end{pmatrix} \right\|_2 = \left\| \begin{pmatrix} 0 \\ \hat{b}_z \end{pmatrix} \right\|_2 = \|\hat{b}_z\|_2$

Recall the SVD view of $Ac = b$ in the case where A is full rank:

- Then, $U\Sigma V^T c = b$ gives $\begin{pmatrix} \hat{\Sigma} \\ 0 \end{pmatrix} \hat{c} = \begin{pmatrix} \hat{b}_r \\ \hat{b}_z \end{pmatrix}$
- The first block row leads to $c = V\hat{\Sigma}^{-1}\hat{b}_r$, just like for the normal equations
- The norm of the residual is $\left\| \begin{pmatrix} \hat{b}_r \\ \hat{b}_z \end{pmatrix} - \begin{pmatrix} \hat{\Sigma} \\ 0 \end{pmatrix} \hat{c} \right\|_2 = \left\| \begin{pmatrix} 0 \\ \hat{b}_z \end{pmatrix} \right\|_2 = \|\hat{b}_z\|_2$, just like for the normal equations
- **The normal equations approach (derived from minimizing $\|r\|_2$) agrees with the solution obtained via the SVD**

Summary

- Minimizing $\|r\|_2$ leads to the normal equations
- The SVD leads to the same solution (and residual) as the normal equations

- Neither approach is a good idea
- Instead, we will use a new matrix factorization
- The QR factorization:
$$A = QR \text{ where } R \text{ is upper triangular and } Q^T Q = I$$

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 candidate 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
 - Note: $\langle s^{\hat{q}}, s^{\hat{q}} \rangle_A = \|s^{\hat{q}}\|_A^2$
 - Proof: $\langle s^q, s^{\tilde{q}} \rangle_A = \langle \bar{s}^q, s^{\tilde{q}} \rangle_A - \frac{\langle \bar{s}^q, s^{\tilde{q}} \rangle_A}{\langle s^{\tilde{q}}, s^{\tilde{q}} \rangle_A} \langle s^{\tilde{q}}, s^{\tilde{q}} \rangle_A = 0$ for $\tilde{q} \in \{1, 2, \dots, q-1\}$

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 for QR (an 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, $Q Q^T \neq I_{5 \times 5}$ since Q is only a subset of an orthogonal matrix

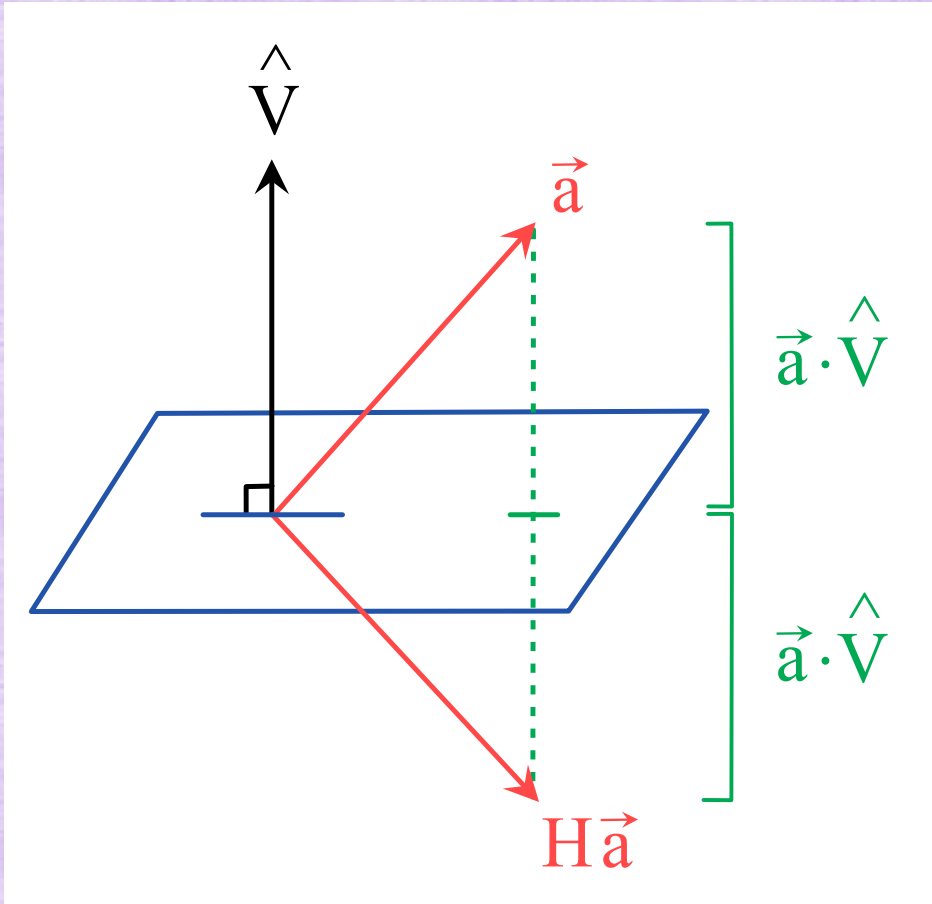
Not Good for Large Matrices

- Gram-Schmidt has too much numerical drift, when used on a large number of vectors
- So, we won't use Gram-Schmidt to find $A = QR$
- However, Gram-Schmidt is a good way to conceptualize the QR factorization

Understanding QR Factorization

- Consider $A = QR$ with an 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 that contains the $m - n$ orthonormal columns that span the space perpendicular to the range of Q
- Then, the size $m \times m$ matrix $\hat{Q} = \begin{pmatrix} Q & \tilde{Q} \end{pmatrix}$ is an orthogonal matrix
- $\|r\|_2 = \|\hat{Q}^T r\|_2 = \left\| \begin{pmatrix} Q^T \\ \tilde{Q}^T \end{pmatrix} (b - QRc) \right\|_2 = \left\| \begin{pmatrix} Q^T b - Rc \\ \tilde{Q}^T b \end{pmatrix} \right\|_2 = \left\| \begin{pmatrix} \hat{b}_Q - Rc \\ \hat{b}_{\tilde{Q}} \end{pmatrix} \right\|_2$
- The first block row shows that $\|r\|_2$ is minimized by solving $Rc = Q^T b$
 - Since R is upper triangular, this can be solved via back-substitution
 - This results in $\|r\|_2 = \|\tilde{Q}^T b\|_2 = \|\hat{b}_{\tilde{Q}}\|_2$ as the minimum value for $\|r\|_2$

Householder Transform



- A unit normal \hat{v} implicitly defines a plane orthogonal to it
- $H = I - 2\hat{v}\hat{v}^T$ 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 size $m \times m$ matrix H
- Instead, apply it via \hat{v}

Householder Transform for QR

- Choose the directions $v_k = a - Ha$ in order to zero out elements

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

- As a reflection, Ha has the same length as a , so $\gamma = \pm \|\hat{a}_k\|_2$
- 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
- Finally, v_k is normalized to $\hat{v}_k = \frac{v_k}{\|v_k\|_2}$

Householder Transform (an example)

- Let $a_1 = \begin{pmatrix} 2 \\ 1 \\ 2 \end{pmatrix}$ and consider $v_1 = \hat{a}_1 + S(a_1)\|\hat{a}_1\|_2\hat{e}_1$

- Then, $\hat{a}_1 = \begin{pmatrix} 2 \\ 1 \\ 2 \end{pmatrix}$, $v_1 = \begin{pmatrix} 2 \\ 1 \\ 2 \end{pmatrix} + S(2)\sqrt{9} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 5 \\ 1 \\ 2 \end{pmatrix}$, $\hat{v}_1 = \frac{1}{\sqrt{30}} \begin{pmatrix} 5 \\ 1 \\ 2 \end{pmatrix}$

- So, $H_1 a_1 = a_1 - 2(\hat{v}_1^T a_1) \hat{v}_1 = \begin{pmatrix} 2 \\ 1 \\ 2 \end{pmatrix} - 2 \frac{15}{\sqrt{30}} \frac{1}{\sqrt{30}} \begin{pmatrix} 5 \\ 1 \\ 2 \end{pmatrix} = \begin{pmatrix} -3 \\ 0 \\ 0 \end{pmatrix}$

Householder Transform (another example)

• Let $a_2 = \begin{pmatrix} 6 \\ 3 \\ 4 \end{pmatrix}$ and consider $v_2 = \hat{a}_2 + S(a_2)\|\hat{a}_2\|_2\hat{e}_2$

• Then, $\hat{a}_2 = \begin{pmatrix} 0 \\ 3 \\ 4 \end{pmatrix}$, $v_2 = \begin{pmatrix} 0 \\ 3 \\ 4 \end{pmatrix} + S(3)\sqrt{25}\begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 8 \\ 4 \end{pmatrix}$, $\hat{v}_2 = \frac{1}{\sqrt{80}}\begin{pmatrix} 0 \\ 8 \\ 4 \end{pmatrix}$

• So, $H_2 a_2 = a_2 - 2(\hat{v}_2^T a_2)\hat{v}_2 = \begin{pmatrix} 6 \\ 3 \\ 4 \end{pmatrix} - 2\frac{40}{\sqrt{80}}\frac{1}{\sqrt{80}}\begin{pmatrix} 0 \\ 8 \\ 4 \end{pmatrix} = \begin{pmatrix} 6 \\ -5 \\ 0 \end{pmatrix}$

Householder Transform for QR

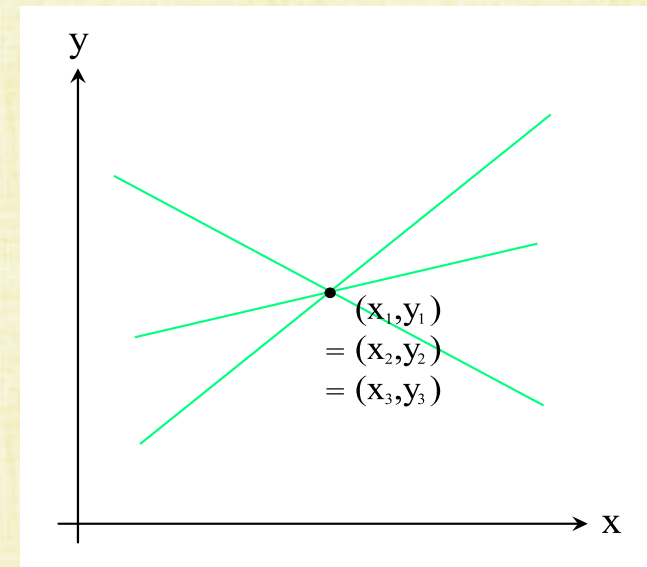
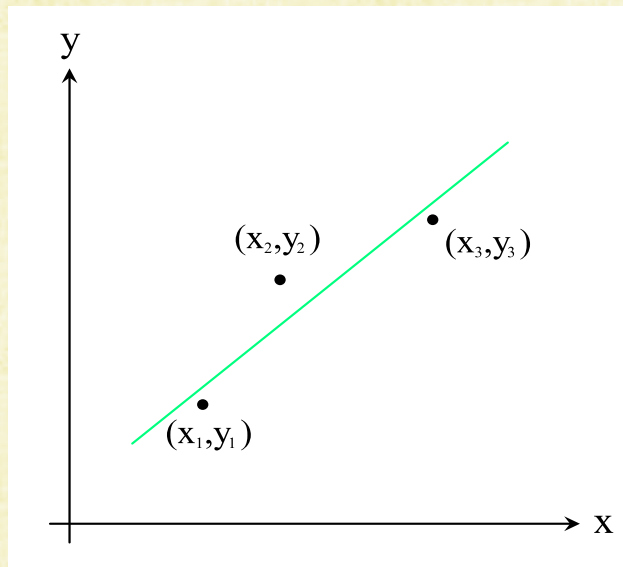
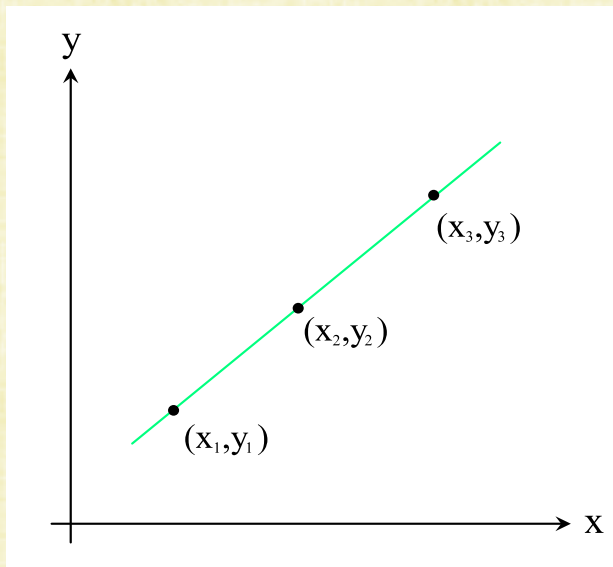
- For each column of A , construct the Householder transform that zeroes out entries below the diagonal (similar to Gaussian elimination)
- Then $H_n H_{n-1} \cdots H_2 H_1 A = \begin{pmatrix} R \\ 0 \end{pmatrix}$ and $H_n H_{n-1} \cdots H_2 H_1 b = \begin{pmatrix} \hat{b}_Q \\ \hat{b}_{\tilde{Q}} \end{pmatrix}$
 - H_n is required to get zeroes at the bottom of the last column
- Apply each H_k efficiently via \hat{v}_k to all columns $\geq k$
 - For columns \hat{k} with $\hat{k} < k$, $\hat{v}_k^T a_{\hat{k}} = 0$ implies that H_k has no effect
- 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 $A = \hat{Q} \begin{pmatrix} R \\ 0 \end{pmatrix}$
- So, $\|r\|_2 = \|\hat{Q}^T r\|_2 = \left\| \hat{Q}^T \left(b - \hat{Q} \begin{pmatrix} R \\ 0 \end{pmatrix} c \right) \right\|_2 = \left\| \begin{pmatrix} \hat{b}_Q - Rc \\ \hat{b}_{\tilde{Q}} \end{pmatrix} \right\|_2$
- Solve $Rc = \hat{b}_Q$ via back-substitution to minimize $\|r\|_2$ to a value of $\|\hat{b}_{\tilde{Q}}\|_2$

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
$$\begin{pmatrix} 1 & x_1 \\ 1 & x_2 \\ 1 & x_3 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$$
- Suppose that $x_1 = x_2 = x_3$, so that the columns are multiples of each other and the matrix is only rank 1
- If $y_1 = y_2 = y_3$, the right-hand side is in the range of the (rank 1) columns implying infinite solutions
- Otherwise, the right-hand side is not in the range of the columns implying no solutions
 - Toss away the linearly dependent second column along with c_2 , and do least squares on c_1

(Careful) Variable Classification

- 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 third column, $\vec{0}c_3$, leaves c_3 underdetermined with infinite possibilities
- It's often 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 each variable

Understanding Underdetermined Systems

- 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 σ_k is dubious)
- Wide matrices have extra columns of zeros, leaving some \hat{c}_k undetermined (just like $\sigma_k = 0$ columns)
- Tall matrices have extra rows with $0 = \hat{b}_k$ ($\sigma_k = 0$ rows contribute to this too)
 - Nonzero \hat{b}_k in such rows implies a nonzero residual

Understanding Underdetermined Systems

- Could write $U(\hat{\Sigma} \quad 0)V^T$ for wide matrices, similar to $A = U \begin{pmatrix} \hat{\Sigma} \\ 0 \end{pmatrix} V^T$ for tall matrices
 - But, $\hat{\Sigma}$ may contain zeros on the diagonal (this is true for tall rank-deficient matrices too)
- For any matrix, 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 $\begin{pmatrix} \hat{\Sigma} & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} \hat{c}_r \\ \hat{c}_z \end{pmatrix} = \begin{pmatrix} \hat{b}_r \\ \hat{b}_z \end{pmatrix}$
- $\|r\|_2 = \|U^T(b - Ac)\|_2 = \left\| \begin{pmatrix} \hat{b}_r \\ \hat{b}_z \end{pmatrix} - \begin{pmatrix} \hat{\Sigma} & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} \hat{c}_r \\ \hat{c}_z \end{pmatrix} \right\|_2 = \left\| \begin{pmatrix} \hat{b}_r \\ \hat{b}_z \end{pmatrix} - \begin{pmatrix} \hat{\Sigma} \hat{c}_r \\ 0 \end{pmatrix} \right\|_2$
- Solving $\hat{\Sigma} \hat{c}_r = \hat{b}_r$ to obtain $\hat{c}_r = \hat{\Sigma}^{-1} \hat{b}_r$ minimizes the residual to $\|r\|_2 = \|\hat{b}_z\|_2$
- Any values are acceptable for the non-determined \hat{c}_z

Minimum Norm Solution

- Set $\hat{c}_z = 0$ to stress that these parameters have no bearing on the solution
- This is more sensible than setting the \hat{c}_z to some arbitrary nonzero values 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 longer hair away from the eyes
 - Someone with short hair driving at night would likely have no driving dependence on a hat; so, reporting information about hats is just misleading in those cases
- Finally, $\mathbf{c} = V\hat{\mathbf{c}} = V \begin{pmatrix} \hat{\mathbf{c}}_r \\ \hat{\mathbf{c}}_z \end{pmatrix} = V \begin{pmatrix} \hat{\Sigma}^{-1} \hat{\mathbf{b}}_r \\ 0 \end{pmatrix} = \sum_{\sigma_k \neq 0} \mathbf{v}_k \frac{\hat{b}_k}{\sigma_k} = \sum_{\sigma_k \neq 0} \mathbf{v}_k \frac{\mathbf{u}_k^T \mathbf{b}}{\sigma_k}$

Pseudo-Inverse

- The minimum norm solution is $c = \left(\sum_{\sigma_k \neq 0} \frac{v_k u_k^T}{\sigma_k} \right) b = A^+ b$ where the pseudo-inverse is $A^+ = \sum_{\sigma_k \neq 0} \frac{1}{\sigma_k} v_k u_k^T$
- 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 size $n \times m$ matrix, so this a sum of matrices

Sum of Rank One Matrices

- $$Ac = U \begin{pmatrix} \hat{\Sigma} & 0 \\ 0 & 0 \end{pmatrix} V^T c = U \begin{pmatrix} \hat{\Sigma} & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} \hat{c}_r \\ \hat{c}_z \end{pmatrix} = U \begin{pmatrix} \hat{\Sigma} \hat{c}_r \\ 0 \end{pmatrix} = \sum_{\sigma_k \neq 0} u_k \sigma_k \hat{c}_k =$$
$$\sum_{\sigma_k \neq 0} u_k \sigma_k v_k^T c = \left(\sum_{\sigma_k \neq 0} \sigma_k u_k v_k^T \right) c$$

- That is, $A = \sum_{\sigma_k \neq 0} \sigma_k u_k v_k^T$

- Each term is an outer product between corresponding columns of U and V , weighted by their corresponding singular value
- Each term is a size $m \times n$ matrix (the same size as A)
- Each term is rank 1, since every column in the term 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: $A^+ \approx \sum_{k=1}^p \frac{1}{\sigma_k} v_k u_k^T$
- This **is** the best rank p approximation to A , and the main idea behind principal component analysis (PCA)
 - Often, thousands/millions of terms can be thrown away keeping only 10 to 100 terms
- Can instead drop small singular values: $A \approx \sum_{\sigma_k > \epsilon} \sigma_k u_k v_k^T$
- This makes the pseudo-inverse better conditioned: $A^+ \approx \sum_{\sigma_k > \epsilon} \frac{1}{\sigma_k} v_k u_k^T$
 - This obviously relies on a good choice of $\epsilon > 0$

Recall: Approximating A (unit 3)

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

$$\begin{pmatrix} .141 & .85 & -.40 & -.51 \\ .344 & .46 & .23 & .78 \\ .547 & .03 & .64 & -.59 \\ .750 & -.37 & -.54 & .09 \end{pmatrix} \begin{pmatrix} 25.5 & & & \\ & & & \\ & & & \\ & & & \end{pmatrix} \begin{pmatrix} .504 & .574 & .644 \\ & & & \\ & & & \\ & & & \end{pmatrix}$$

- 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 (real time decision making, etc.), 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 ordering of the operations:
 - $u_k^T b$ is m multiplies, and (after 1 division) the result times v_k is n multiplies (for a total of $m + n$ multiplies)
 - Don't form the size $n \times m$ matrix!
 - Multiplying the size $n \times m$ 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 σ_k^2 is an eigenvalue of both $A^T A$ and AA^T (with eigenvectors v_k and u_k respectively)
- Work with the smaller of $A^T A$ and AA^T , which are both SP(S)D, to find the eigenvalues σ_k^2
- Then, σ_k^2 can be used in both $A^T A$ and AA^T to find the corresponding eigenvectors (v_k and u_k respectively)

Finding Eigenvectors from Eigenvalues

- Given an eigenvalue λ , form the matrix $\hat{A} - \lambda I$
- If \hat{A} is symmetric, then $\hat{A} - \lambda I$ is symmetric
- $\hat{A} - \lambda I$ has (at least) a rank 1 null space (from the definition of eigenvalues)
 - Proof: $(\hat{A} - \lambda I)v = \hat{A}v - \lambda v = \lambda v - \lambda v = 0$ for eigenvector v
- Solve the linear system $(\hat{A} - \lambda I)v = 0$ to find the eigenvector v

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 and right eigenvectors
- Symmetric (Hermitian) matrices have identical left and right eigenvectors; so, $v_L^T v_R = 1$ and the condition number is 1

Characteristic Polynomial

- The eigenvalue problem is typically written as $\hat{A}v = \lambda v$
- So, $(\hat{A} - \lambda I)v = 0$ implying that $\hat{A} - \lambda I$ is singular
- Setting $\det(\hat{A} - \lambda I) = 0$ leads to a degree n characteristic polynomial equation in λ (for a size $n \times n$ matrix \hat{A})
- Finding the roots of this polynomial equation can be quite difficult
 - Recall how difficult it was to find roots for a mere cubic equation
- Finding roots for degree $n > 3$ polynomials is undesirable!

Similarity Transforms

- Similarity transforms, $T^{-1}\hat{A}T$, preserve the eigenstructure
 - Proof: $T^{-1}\hat{A}Tv = \lambda v$ or $\hat{A}(Tv) = \lambda(Tv)$ still has eigenvalue λ with a modified eigenvector $T^{-1}v$
- When \hat{A} is real and symmetric (complex and Hermitian), there exists an orthogonal (unitary) T that makes $T^{-1}\hat{A}T$ diagonal with real eigenvalues
 - In fact, $T = V$ for $A^T A = V\Sigma^T \Sigma V^T$ and $T = U$ for $AA^T = U\Sigma\Sigma^T U^T$
- Other interesting facts:
 - When \hat{A} has distinct eigenvalues, a T exists that makes $T^{-1}\hat{A}T$ diagonal
 - Schur form: For any (square) matrix, a unitary T exists to make $T^{-1}\hat{A}T$ upper triangular with eigenvalues on the diagonal
 - Jordan form: Any (square) matrix can be put into a form with eigenvalues on the diagonal and nonzero off-diagonal elements only occurring on the band above the diagonal and only for defective eigenvalues (repeated eigenvalues that don't possess a full set of eigenvectors)

Similarity Transforms via QR Iteration

- 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
- When the eigenvalues are distinct, \hat{A}^q converges to a triangular matrix
- When \hat{A} is symmetric, \hat{A}^q converges to a diagonal matrix

Power Method

- Computes the largest eigenvalue (great for rank 1 updates)
- Start with a $c^0 \neq 0$, and iterate $c^{q+1} = \hat{A}c^q$

Proof:

- Write c^0 as 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 $\lambda_k < \lambda_{max}$; so, $c^q \rightarrow \lambda_{max}^q \alpha_{max} v_{max}$
- As $q \rightarrow \infty$, $\frac{(c^{q+1})_i}{(c^q)_i} \rightarrow \frac{\lambda_{max}^{q+1} \alpha_{max} (v_{max})_i}{\lambda_{max}^q \alpha_{max} (v_{max})_i} = \lambda_{max}$ for every component i of c
- Deflation removes an eigenvalue from \hat{A} by subtracting off its rank 1 update
 - 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 then be used to compute the next largest eigenvalue (repeatedly)

Power Method (a few notes)

- If $c^0 = \sum_k \alpha_k v_k$ happens to have $\alpha_{max} = 0$, the method might fail (but roundoff errors can help)
- c^q needs to be periodically renormalized to stop it from growing too large
- When c^0 and \hat{A} are real-valued, cannot obtain complex numbers
- When the largest eigenvalue is repeated, one needs to determine a basis for the multiple associated eigenvectors
- 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.
- Normal equations: $(A^T \quad I) \begin{pmatrix} A \\ I \end{pmatrix} c = (A^T \quad I) \begin{pmatrix} b \\ 0 \end{pmatrix}$ or $(A^T A + I)c = A^T b$
- Use $A = U\Sigma V^T$ to get $(V\Sigma^T \Sigma V^T + I)c = V\Sigma^T \hat{b}$ or $(\Sigma^T \Sigma + I)\hat{c} = \Sigma^T \hat{b}$
- Use $\Sigma = \begin{pmatrix} \hat{\Sigma} & 0 \\ 0 & 0 \end{pmatrix}$ to get $\left(\begin{pmatrix} \hat{\Sigma} & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} \hat{\Sigma} & 0 \\ 0 & 0 \end{pmatrix} + I \right) \begin{pmatrix} \hat{c}_r \\ \hat{c}_z \end{pmatrix} = \begin{pmatrix} \hat{\Sigma} & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} \hat{b}_r \\ \hat{b}_z \end{pmatrix}$
- Then $\left(\begin{pmatrix} \hat{\Sigma}^2 & 0 \\ 0 & 0 \end{pmatrix} + I \right) \begin{pmatrix} \hat{c}_r \\ \hat{c}_z \end{pmatrix} = \begin{pmatrix} \hat{\Sigma} \hat{b}_r \\ 0 \end{pmatrix}$, which gives $\hat{c}_z = 0$ as desired

Perturbation

- However, $\left(\begin{pmatrix} \hat{\Sigma}^2 & 0 \\ 0 & 0 \end{pmatrix} + I \right) \begin{pmatrix} \hat{c}_r \\ \hat{c}_z \end{pmatrix} = \begin{pmatrix} \hat{\Sigma} \hat{b}_r \\ 0 \end{pmatrix}$ perturbs the equations for the \hat{c}_r terms
- Instead of the usual $\hat{c}_k = \frac{1}{\sigma_k} \hat{b}_k$, the new solution is $\hat{c}_k = \frac{\sigma_k}{\sigma_k^2 + 1} \hat{b}_k$
 - This perturbs these \hat{c}_k away from their correct (unique or least squares) solution
 - Adding $Ic = 0$ interferes with $Ac = b$ for the \hat{c}_k with $\sigma_k \neq 0$
- 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**
- For $\sigma_k \approx 1$, the **perturbation is quite large**
- For $\sigma_k \ll 1$, $\frac{\sigma_k}{\sigma_k^2 + 1} \approx 0$ **drives the associated \hat{c}_k towards zero**

Regularization

- Add $\epsilon I c = 0$ (with $\epsilon > 0$) instead of $I c = 0$, that is $\begin{pmatrix} A \\ \epsilon I \end{pmatrix} c = \begin{pmatrix} b \\ 0 \end{pmatrix}$
- Normal equations: $\begin{pmatrix} A^T & \epsilon I \end{pmatrix} \begin{pmatrix} A \\ \epsilon I \end{pmatrix} c = \begin{pmatrix} A^T & \epsilon I \end{pmatrix} \begin{pmatrix} b \\ 0 \end{pmatrix}$ or $(A^T A + \epsilon^2 I) c = A^T b$
- This results in a modified $\begin{pmatrix} \left(\begin{pmatrix} \hat{\Sigma}^2 & 0 \\ 0 & 0 \end{pmatrix} + \epsilon^2 I \right) \begin{pmatrix} \hat{c}_r \\ \hat{c}_z \end{pmatrix} = \begin{pmatrix} \hat{\Sigma} \hat{b}_r \\ 0 \end{pmatrix}$
- Instead of the usual $\hat{c}_k = \frac{1}{\sigma_k} \hat{b}_k$, the new solution is $\hat{c}_k = \frac{\sigma_k}{\sigma_k^2 + \epsilon^2} \hat{b}_k$
- This has limited effect on $\sigma_k \gg \epsilon$
- This helps to stabilize/regularize the solution for $\sigma_k \approx \epsilon$ and $\sigma_k < \epsilon$, driving the associated \hat{c}_k towards zero

A Nonzero Initial Guess

- Can view setting $Ic = 0$ as guessing a solution of $c = 0$
- Instead, suppose there were a better initial guess of $c = c^*$
- Add $\epsilon Ic = \epsilon c^*$ to the equations to get: $\begin{pmatrix} A \\ \epsilon I \end{pmatrix} c = \begin{pmatrix} b \\ \epsilon c^* \end{pmatrix}$
- Normal equations: $(A^T A + \epsilon^2 I)c = A^T b + \epsilon^2 c^*$
- This leads to $(\Sigma^T \Sigma + \epsilon^2 I)\hat{c} = \Sigma^T \hat{b} + \epsilon^2 V^T c^* = \Sigma^T \hat{b} + \epsilon^2 \hat{c}^*$
- Then, $\hat{c}_k = \frac{\sigma_k}{\sigma_k^2 + \epsilon^2} \hat{b}_k + \frac{\epsilon^2}{\sigma_k^2 + \epsilon^2} \hat{c}_k^*$ tends towards $\frac{1}{\sigma_k} \hat{b}_k$ for larger σ_k (as desired), and tends towards \hat{c}_k^* for smaller σ_k
 - In addition, $\hat{c}_k = \hat{c}_k^*$ for any $\sigma_k = 0$, as desired

A Nonzero 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 **(unique or least squares) 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** σ_k ($\sigma_k \gg \epsilon$) tend toward the usual solution: $\hat{c}_k = \frac{\hat{b}_k}{\sigma_k}$
- **Small** σ_k ($\sigma_k \ll \epsilon$) tend toward the initial guess: $\hat{c}_k = \hat{c}_k^*$

An Iterative Approach

- 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 that 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 that solution as an initial guess and solve again to get:

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

Convergence

- 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 $\hat{c}_k = \frac{1}{\sigma_k} \hat{b}_k$ 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 σ_k

Convergence Rate

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

Comparison with PCA

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

Aside

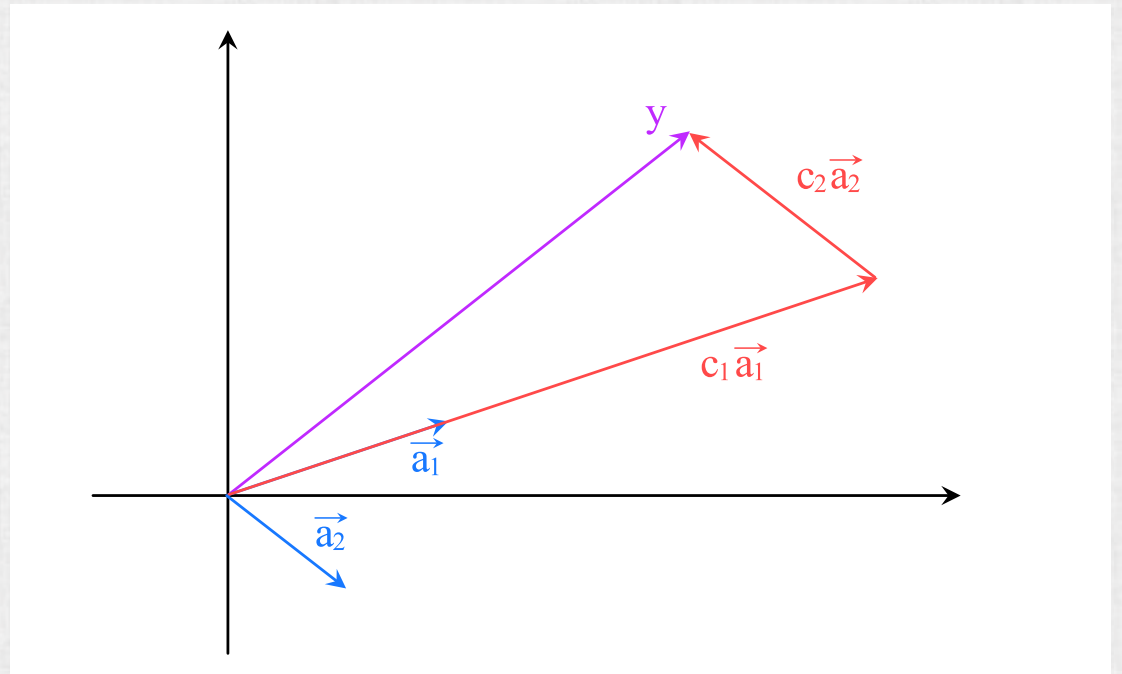
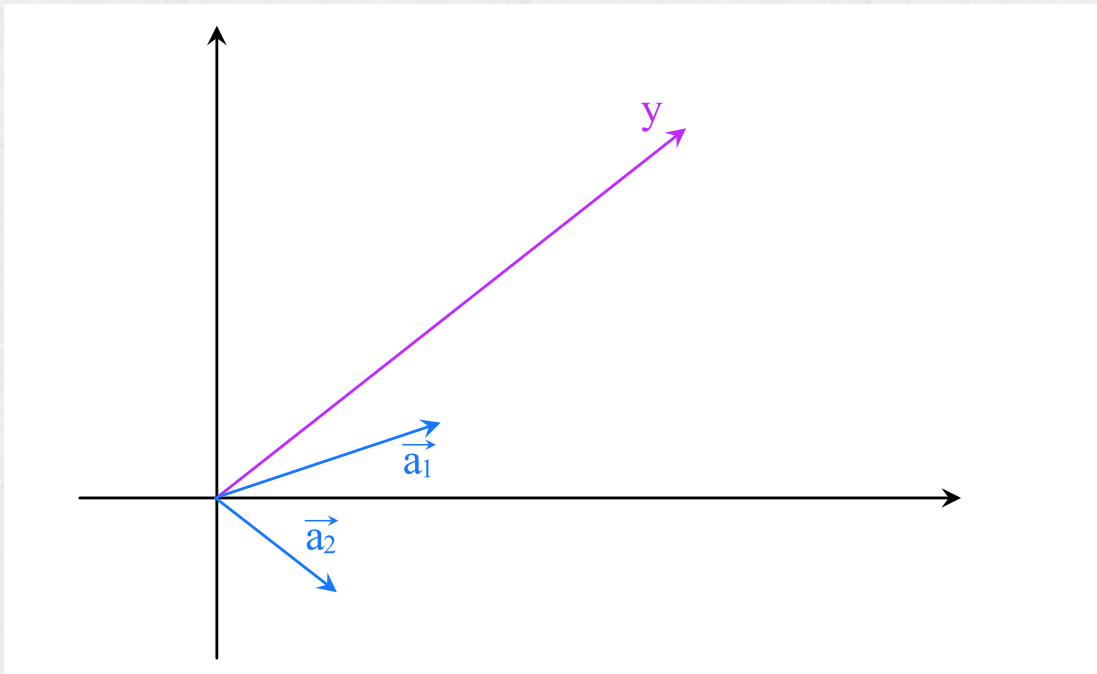
- This iterative method and the analysis via a geometric series (slides 7-10) were derived in preparation for the Winter 2019 offering of this course
 - Hyde, D., Bao, M., and Fedkiw, R., "On Obtaining Sparse Semantic Solutions for Inverse Problems, Control, and Neural Network Training", J. Comp. Phys.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

Adding a Diagonal Matrix

- Adding $Dc = 0$ to obtain: $\begin{pmatrix} A \\ D \end{pmatrix} c = \begin{pmatrix} b \\ 0 \end{pmatrix}$ drives 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 same 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
- Alternatively, column scale $\begin{pmatrix} AD^{-1} \\ I \end{pmatrix} Dc = \begin{pmatrix} b \\ 0 \end{pmatrix}$ to obtain an $\begin{pmatrix} \tilde{A} \\ I \end{pmatrix} \tilde{c} = \begin{pmatrix} b \\ 0 \end{pmatrix}$ that can be treated in the original way (as if $I\tilde{c} = 0$ were added)

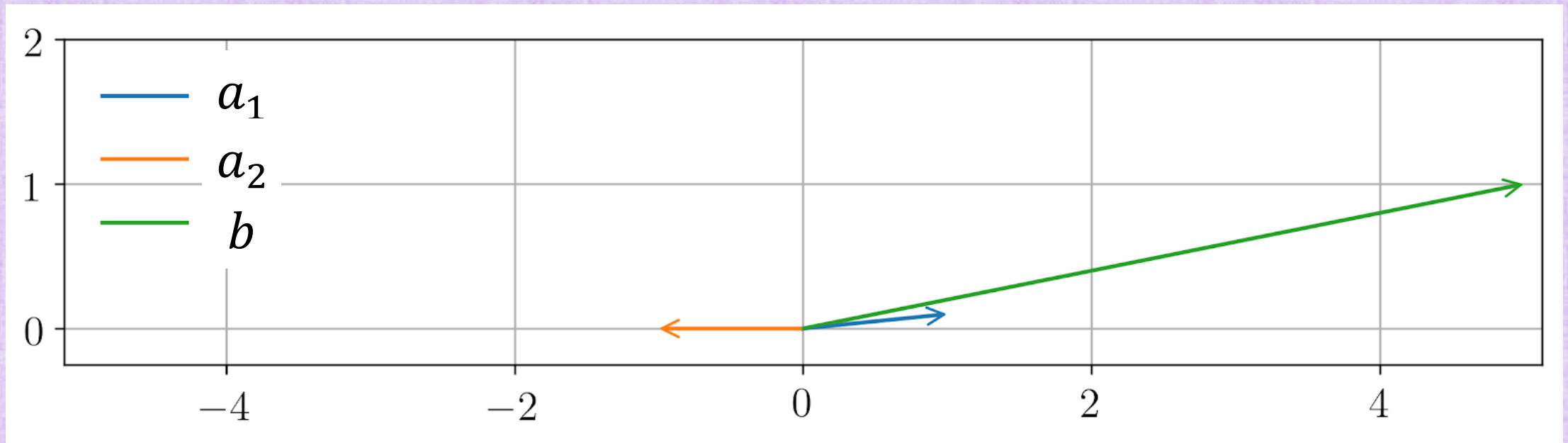
Recall: Matrix Columns as Vectors (unit 1)

- Define the k -th column of A as 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



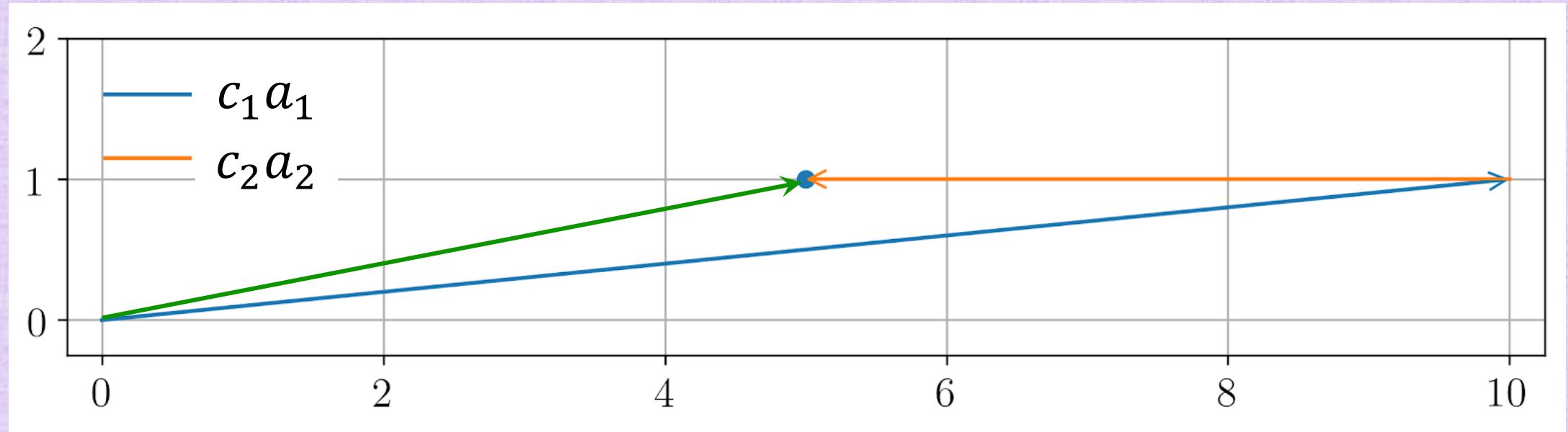
An Example

- Determine c_1 and c_2 such that $c_1 a_1 + c_2 a_2 = b$ or $Ac = b$



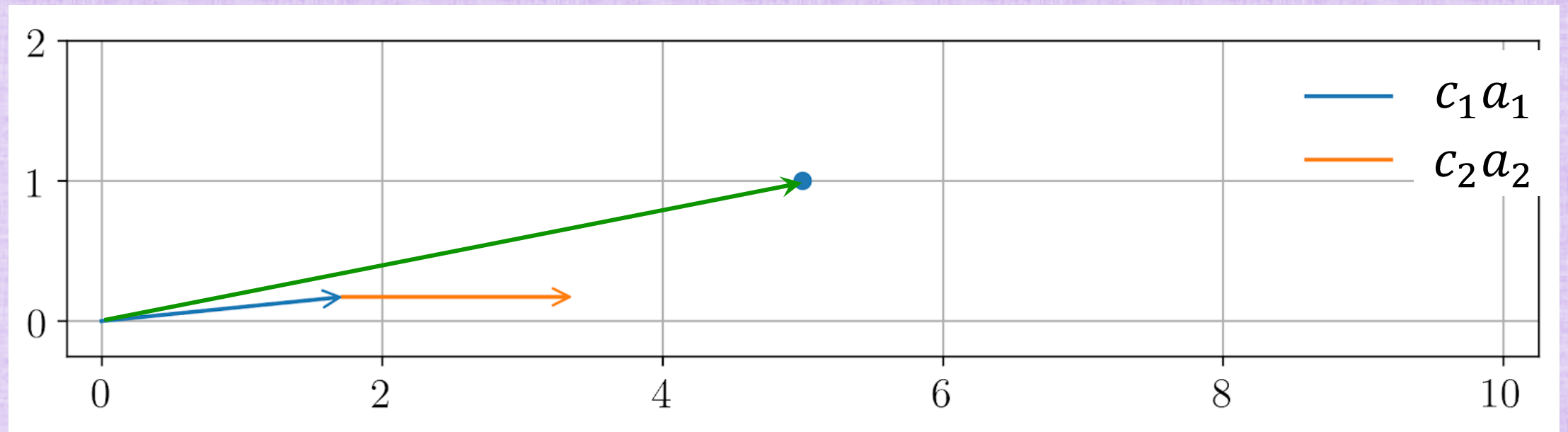
Overshooting

- 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



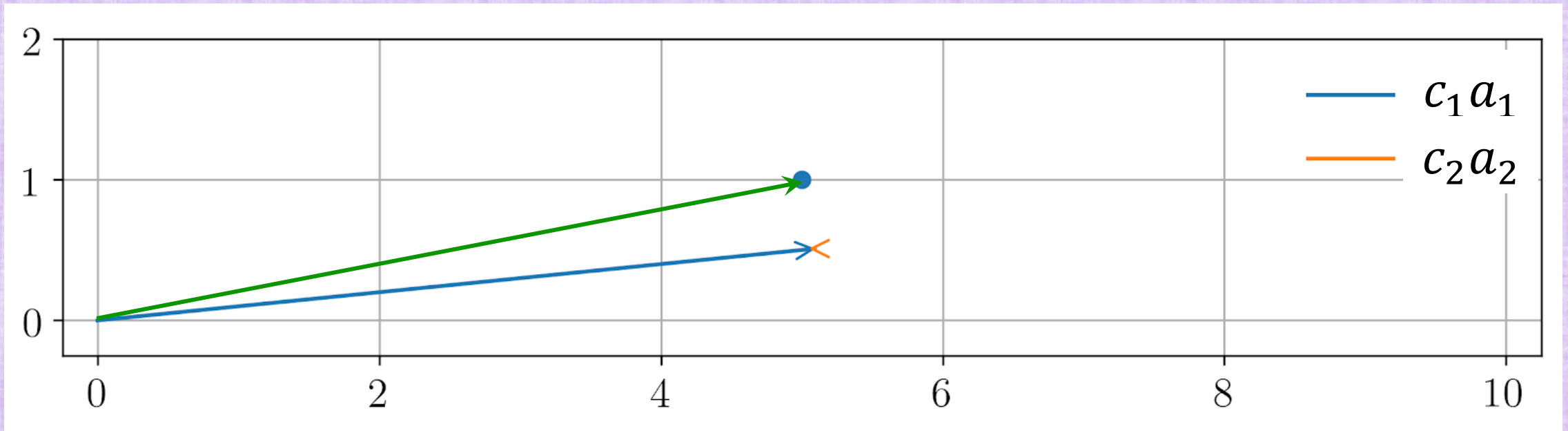
Regularization/Damping

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



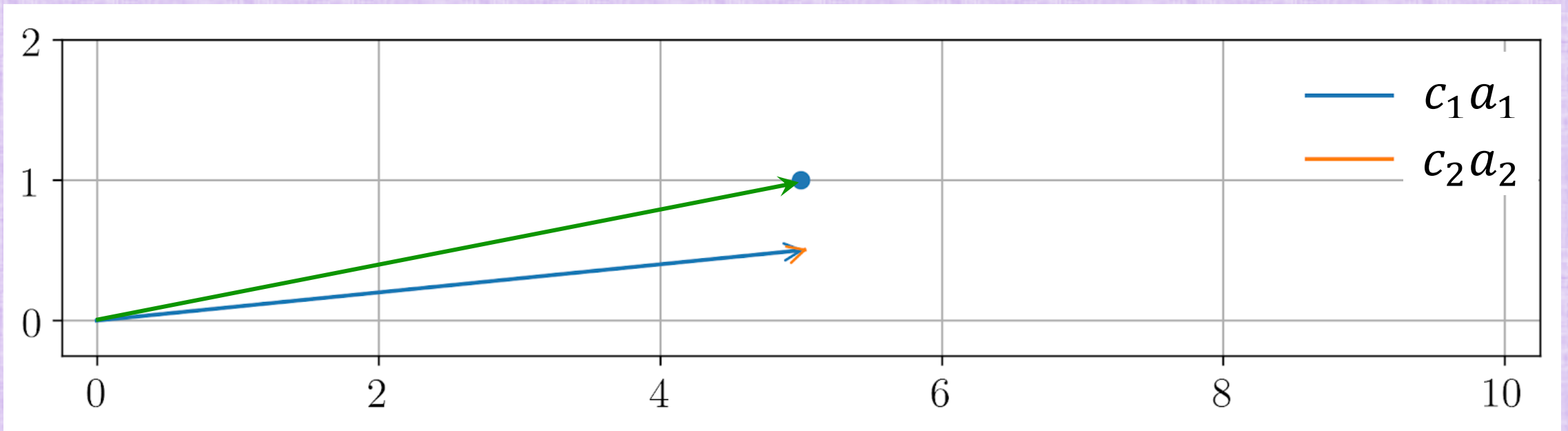
Smarter Regularization

- 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



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 Approaches

- 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 that are 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)

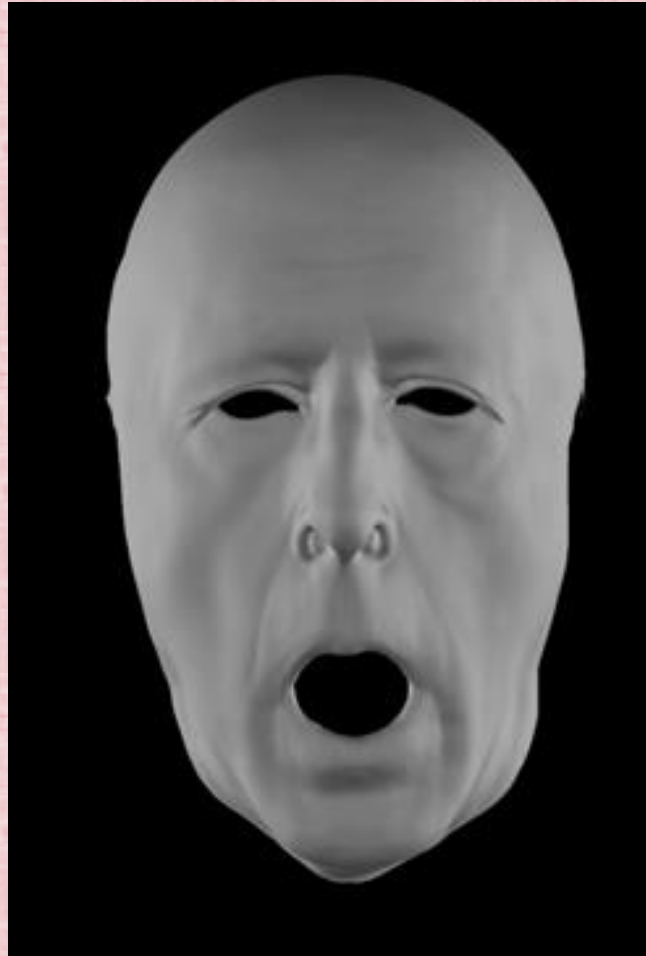
Correlation vs. Gains

- Consider $a_k \cdot b = \|a_k\|_2 \|b\|_2 \cos \Theta$ where Θ measures how parallel a_k and b are
- Correlation preference uses the columns a_k with a larger $\cos \Theta$, 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** are 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 θ lead to a 3D position (output) for every vertex of the face mesh $\varphi(\theta)$
- E.g. in blend shape systems, each component of θ corresponds to a different expression (or sub-expression), and setting multiple components to be nonzero mixes expressions

Facial Tracking



2D RGB Image



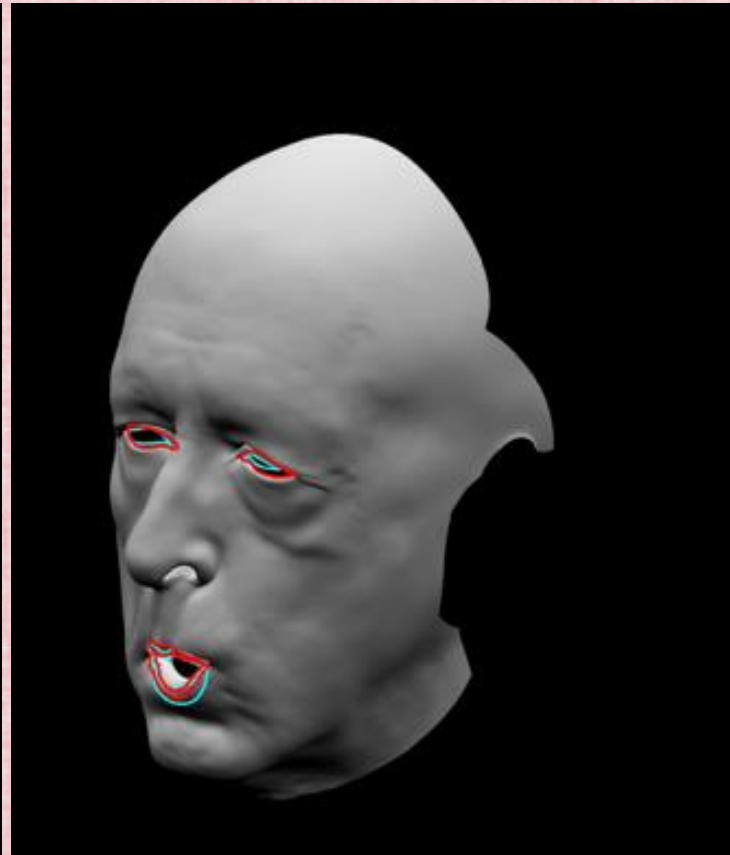
3D model

- 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 θ for the 2D RGB image)

Facial Tracking



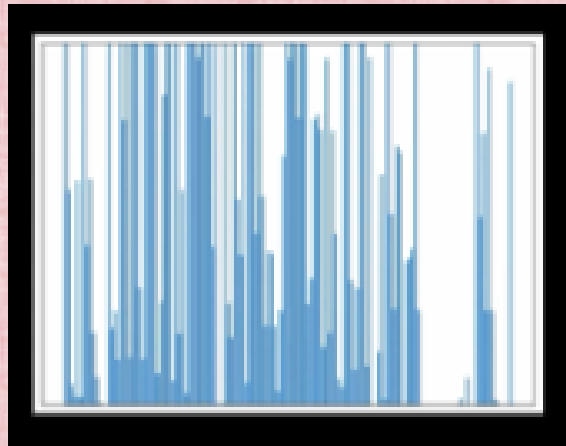
2D RGB Image



3D model

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

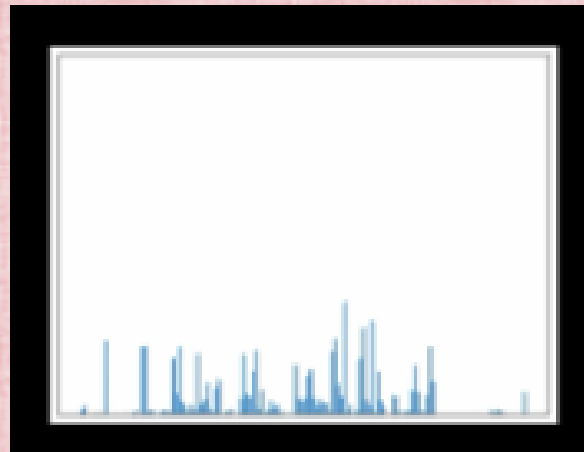
Solving for the Animation Parameters



- This nonlinear problem can be solved via optimization
- At every step of optimization, the problem is linearized
- Solving the resulting linear system $Ac = b$ gives a search direction, which is used to make progress towards the solution
- The optimization performs poorly without regularization
- The resulting θ 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 $Ic = 0$ 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, θ is overly damped



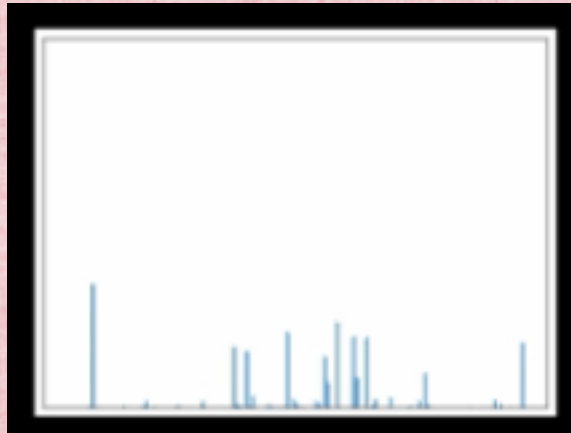
- Also, a large number of animation parameters θ are nonzero, even for this relatively simple expression
- This hinders the interpretability (semantics) of θ

“Soft L1” Regularization

- There are many options for regularization
- In particular, “soft L1” typically produces a sparser set of solution parameters than L2 regularization
- A sparser solution allows one to better ascertain semantic meaning from the animation parameters θ
 - But, θ is still overly damped



Soft L1 regularization

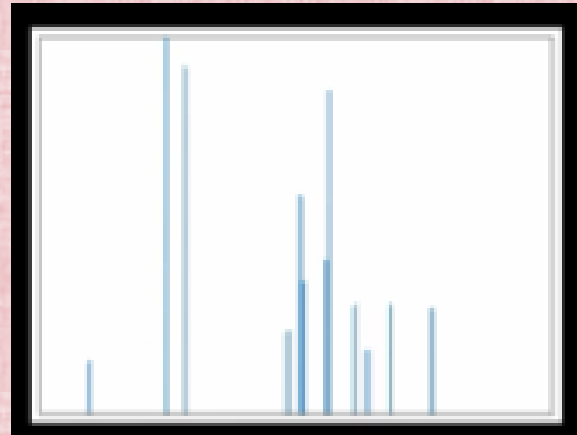


Geometric Approach (Column Space Search)

- 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 θ



Column Space Search



Unit 13

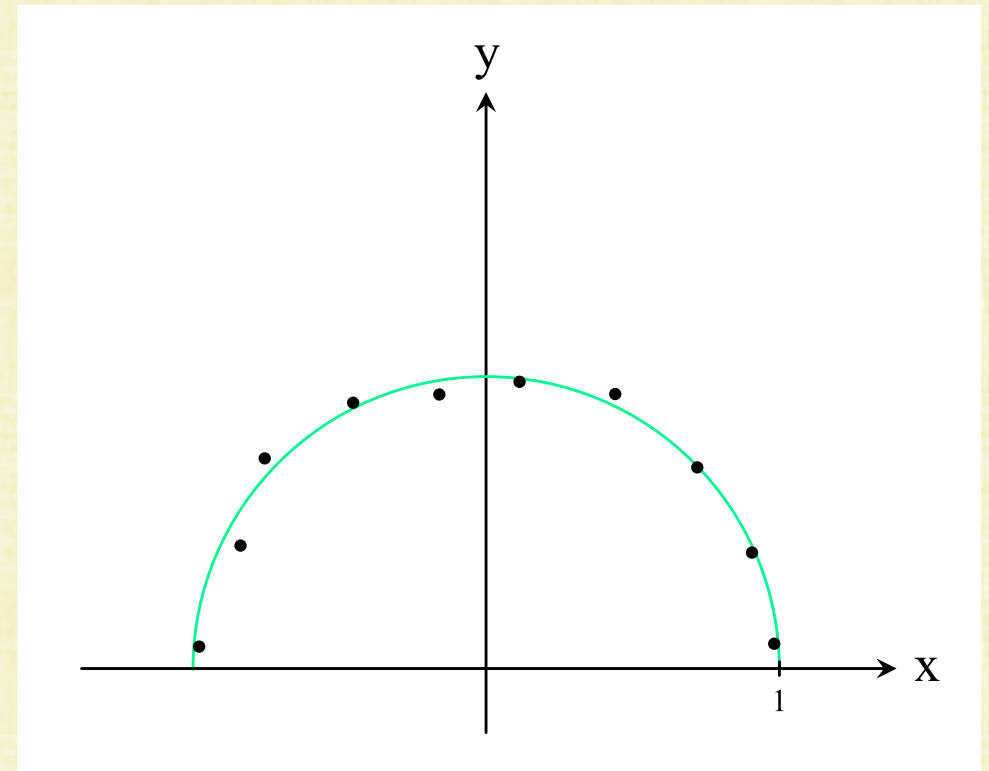
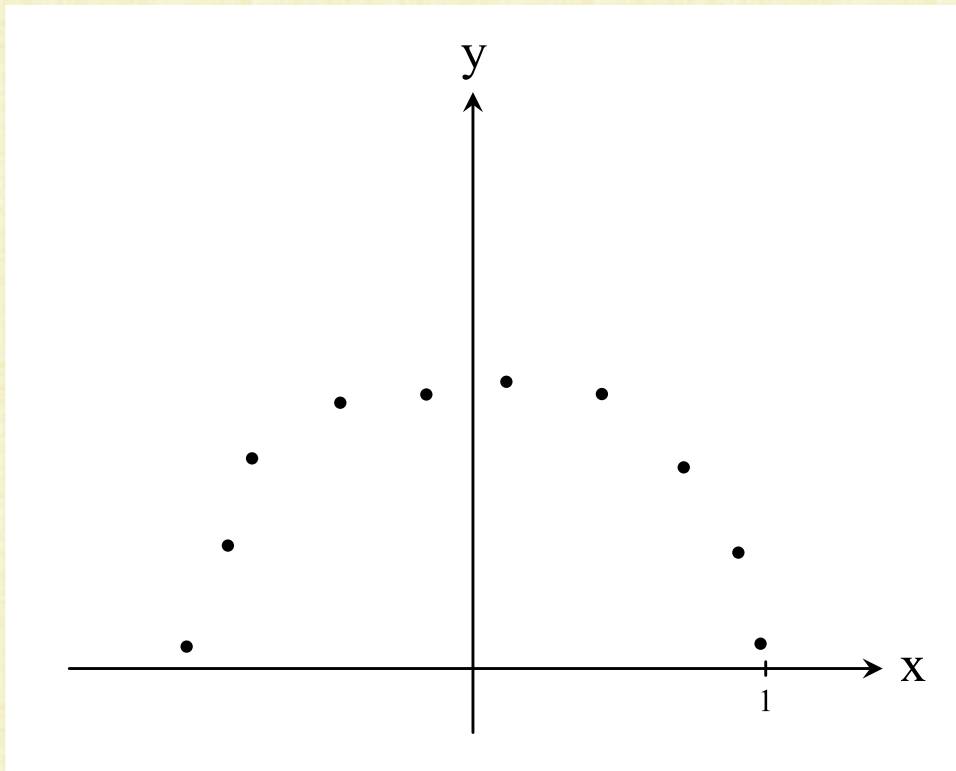
Optimization

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; P1[Part I - Linear Algebra (units 1-12) Ac = b]; P2[Part II - Optimization (units 13-20)]; P2 -- linearize --> P1; P1 -- line search --> P2; P2 -- Theory --> P2_1["(units 13-16) Optimization -> Nonlinear Equations -> 1D roots/minima"]; P2_1 --- P2_2["(units 17-18) Computing/Avoiding Derivatives"]; P2_2 --- P2_3["(unit 19) Hack 1.0: 'I give up' H = I and J is mostly 0 (descent methods)"]; P2_3 --- P2_4["(unit 20) Hack 2.0: 'It's an ODE!?' (adaptive learning rate and momentum)"]; P2_2 --- P2_4; P2_2 --- P2_4 --- Methods[Methods];
```

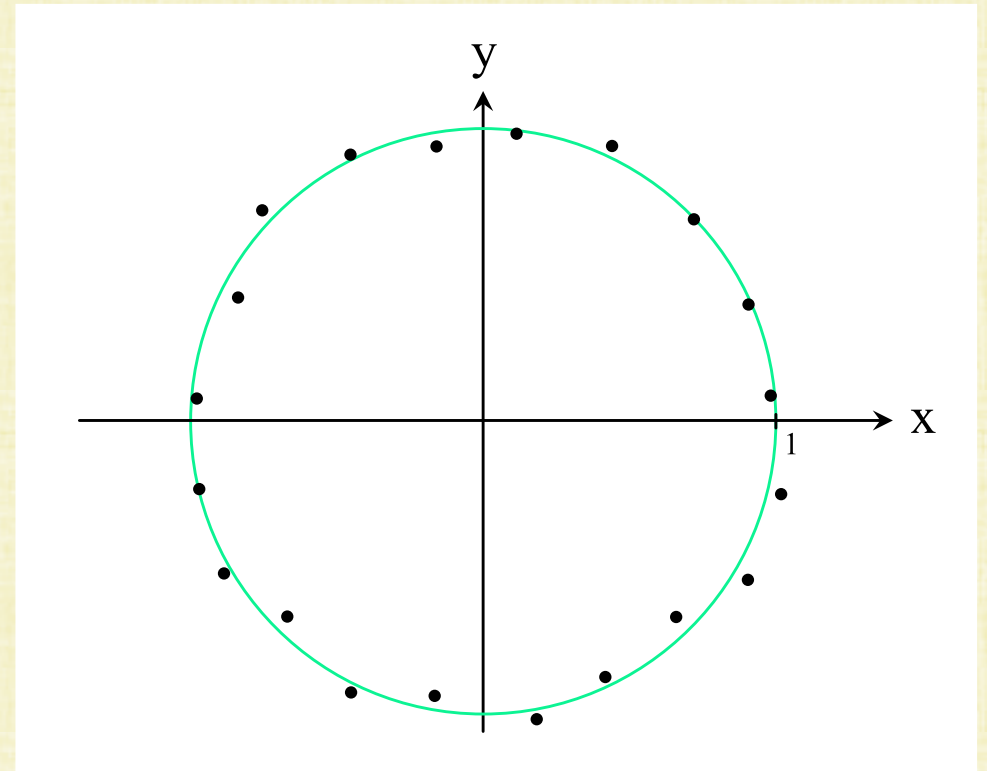
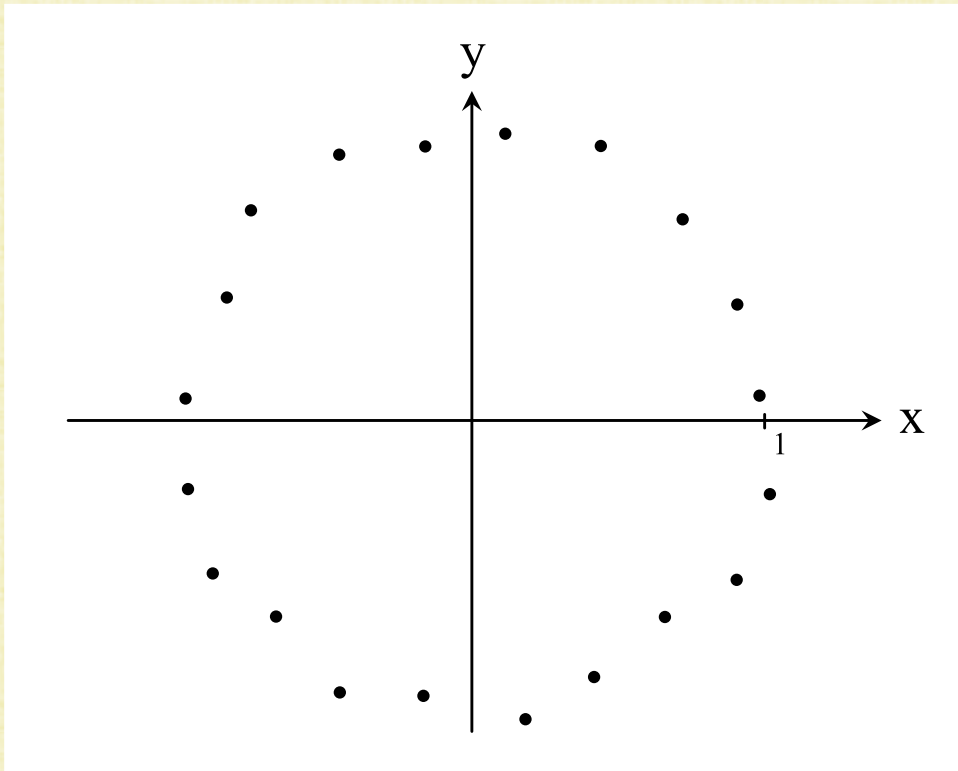
# Approximating Functions

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



# Approximating Functions

- Consider the  $(x_i, y_i)$  data shown below
- Here,  $x^2 + y^2 = 1$  looks like a good approximation (**fails the vertical line test**)



# Approximating Functions

- A function does not need to be explicit in  $y$
- Any relationship between  $x$  and  $y$  is fine, i.e.  $f(x, y) = 0$
- It is difficult to consider all possible functions at the same time; so, one typically chooses a parametric family of possible functions (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 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; for vectors, 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.,  $\sqrt{\sum_i \|f(x_i, y_i; c)\|_2^2} = \sqrt{\sum_i f(x_i, y_i; c)^T f(x_i, y_i; c)}$
- Minimize  $\sqrt{\sum_i f(x_i, y_i; c)^T f(x_i, y_i; c)}$  or equivalently  $\sum_i f(x_i, y_i; c)^T f(x_i, y_i; c)$
- 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 the cost function  $\hat{f}(c)$
- Since maximizing  $\hat{f}(c)$  is equivalent to minimizing  $-\hat{f}(c)$ , optimization is typically 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)
  - Finding lots of local minima and choosing the smallest of those is often a good strategy
- When constraints are present, it's denoted as constrained (as opposed to unconstrained) optimization
  - 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)

# Conditioning for Optimization

- Recall: Minimizing the residual  $r = b - Ac$  with an  $L^2$  norm led to the 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  $k$
  - Partial derivatives approaching 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  $\hat{f}(c) = 0$  (i.e. for finding the roots of  $\hat{f}(c) = 0$ )
  - 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)  **$\sqrt{\epsilon}$**  error tolerance is more appropriate for minimizing  $\hat{f}(c)$

# Nonlinear Systems of Equations (for Optimization)

- Critical points have  $\frac{\partial \hat{f}}{\partial c_k}(c) = 0$  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, can add a penalty term  $\hat{g}^T(c)D\hat{g}(c) \geq 0$  to the cost function and proceed via unconstrained optimization
  - I.e., minimize  $\hat{f}(c) + \hat{g}^T(c)D\hat{g}(c)$  via unconstrained optimization
- Various other options also exist:
  - E.g. Add Lagrange multipliers  $\eta$  as new variables, and minimize  $\hat{f}(c) + \eta^T \hat{g}(c)$

# Lagrange Multipliers

- Minimize  $\hat{f}(c) + \eta^T \hat{g}(c)$
- Critical Points:  $\nabla \begin{pmatrix} \hat{f}(c) + \eta^T \hat{g}(c) \\ \hat{g}(c) \end{pmatrix} = \begin{pmatrix} J_{\hat{f}}^T(c) + J_{\hat{g}}^T(c)\eta \\ \hat{g}(c) \end{pmatrix} = 0$ 
  - Note how the  $\hat{g}(c) = 0$  constraints 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 (when  $H_{\hat{f}}(c)$  is) 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  $J_{\hat{g}}$ ); so, it's ok

# Lagrange Multipliers (an example)

- Minimize  $\hat{f}(c) = \frac{1}{2}c_1^2 + \frac{5}{2}c_2^2$  subject to  $\hat{g}(c) = c_1 - c_2 - 1 = 0$

- So, minimize  $\frac{1}{2}c_1^2 + \frac{5}{2}c_2^2 + \eta_1(c_1 - c_2 - 1)$

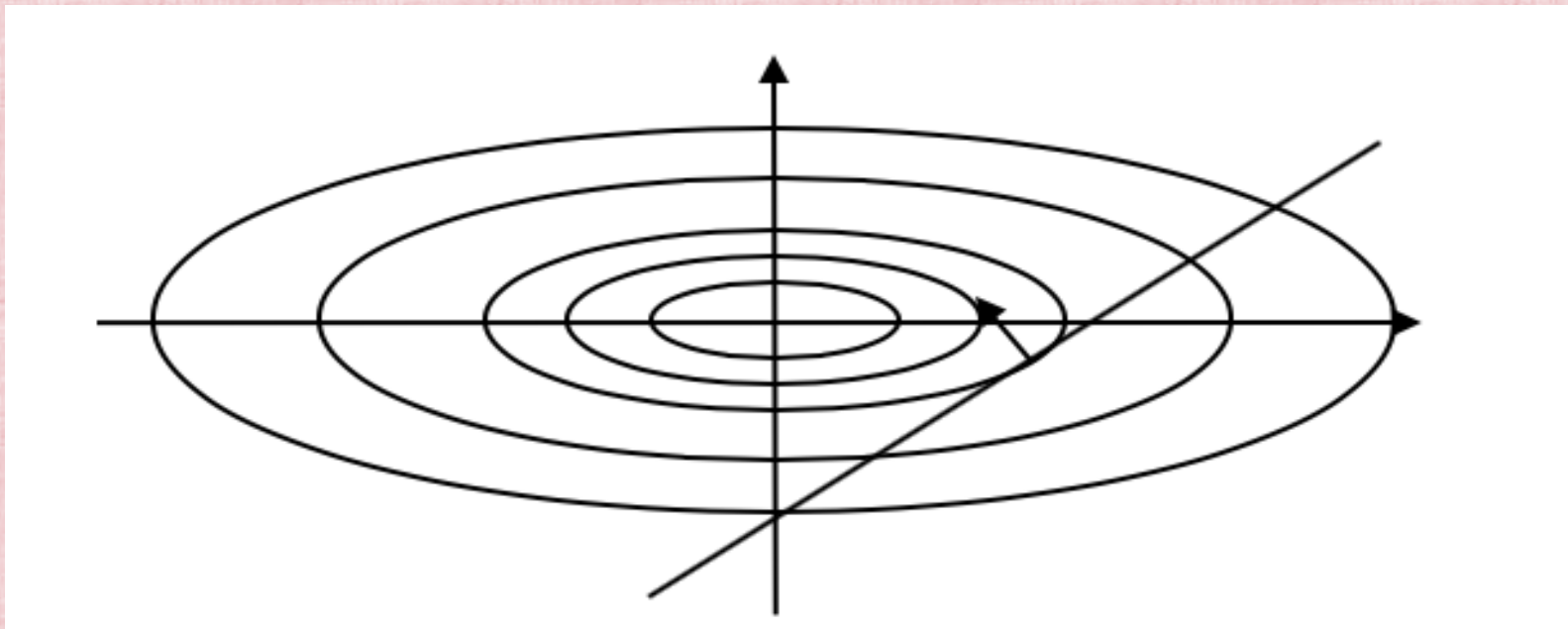
- Critical Points: 
$$\begin{pmatrix} \begin{pmatrix} c_1 \\ 5c_2 \end{pmatrix} + \begin{pmatrix} 1 \\ -1 \end{pmatrix} \eta_1 \\ c_1 - c_2 - 1 \end{pmatrix} = \begin{pmatrix} c_1 + \eta_1 \\ 5c_2 - \eta_1 \\ c_1 - c_2 - 1 \end{pmatrix} = 0$$

- Equivalently, 
$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 5 & -1 \\ 1 & -1 & 0 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \eta_1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \text{ or } \begin{pmatrix} c_1 \\ c_2 \\ \eta_1 \end{pmatrix} = \begin{pmatrix} 5/6 \\ -1/6 \\ -5/6 \end{pmatrix}$$

- The Hessian is 
$$\begin{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 5 \end{pmatrix} & \begin{pmatrix} 1 \\ -1 \end{pmatrix} \\ \begin{pmatrix} 1 & -1 \end{pmatrix} & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 5 & -1 \\ 1 & -1 & 0 \end{pmatrix}$$

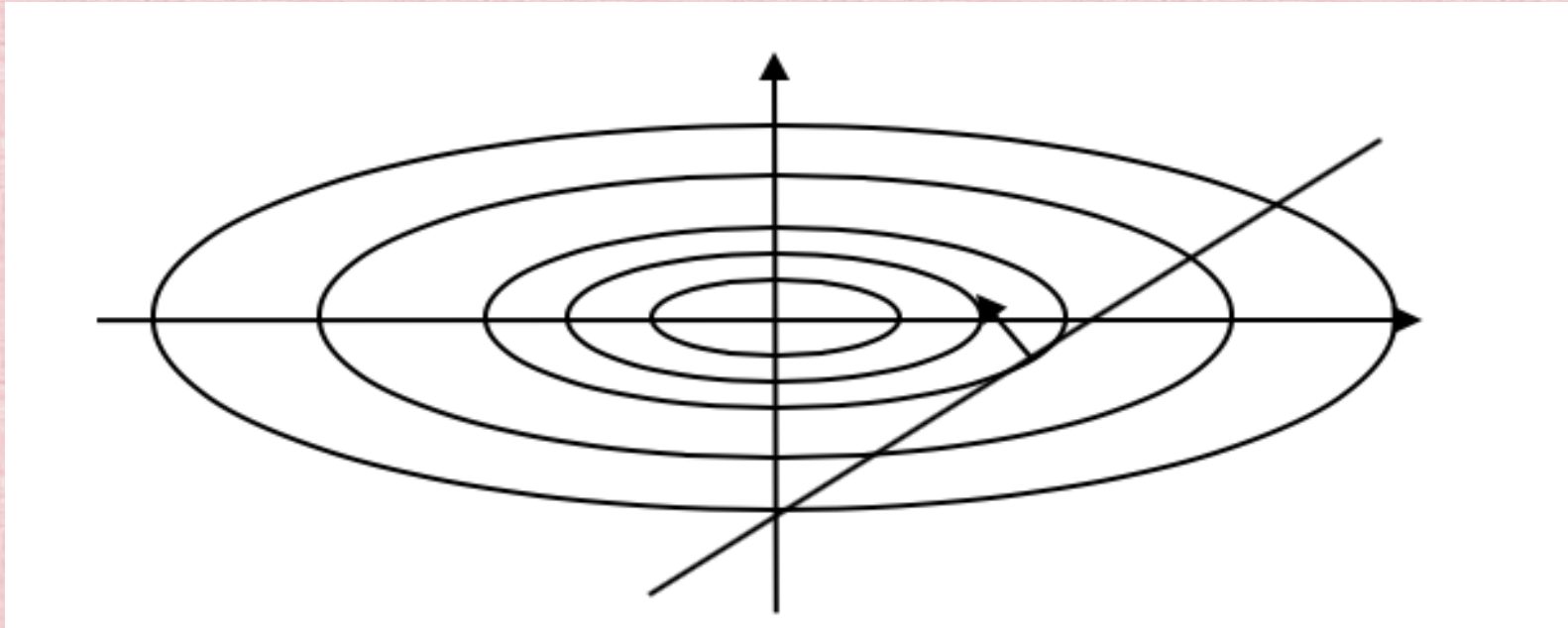
# Lagrange Multipliers (an example)

- 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**  $-\nabla\hat{f} = \begin{pmatrix} -5/6 \\ 5/6 \end{pmatrix}$  is perpendicular to the constraint surface (which has  $(1,1)$  as the line direction)



# Lagrange Multipliers (an example)

- Plug  $c_2 = c_1 - 1$  into  $\hat{f}(c)$  to get  $\frac{1}{2}c_1^2 + \frac{5}{2}(c_1 - 1)^2 = 3c_1^2 - 5c_1 + \frac{5}{2}$ , 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; P1[Part I - Linear Algebra (units 1-12) Ac = b]; P2[Part II - Optimization (units 13-20)]; P2 -- linearize --> P1; P1 -- line search --> P2; P2 -- Theory --> NLE[Nonlinear Equations -> 1D roots/minima]; subgraph Methods; P2 -- Methods --> P2; end
```

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 of equations $F(c) = 0$ is difficult
- Linearize via the multidimensional version of the Taylor expansion:

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

- More valid when $\Delta c = c - c^*$ 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 $\frac{dF(c)}{dt} = F'(c) \frac{dc}{dt}$ is valid for any variable t , and thus can be written in differential form as $dF(c) = F'(c)dc$
 - 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 equivalent to $\frac{df}{dc} = f'(c)$ and $\frac{\Delta f}{\Delta c} \approx f'(c)$ respectively

Newton's Method

- An iterative method: start with c^0 , recursively find: c^1, c^2, c^3, \dots
- Based on $\Delta F(c) \approx F'(c)\Delta c$, write $F(c^{q+1}) - F(c^q) = F'(c^q)\Delta c^q$
 - 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
- Solve the linear system $F'(c^q)\Delta c^q = (\beta - 1)F(c^q)$ for Δc^q
- Use $\Delta c^q = c^{q+1} - c^q$ 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 already contains linearization errors, so approximating $F'(c^q)$ is often justified and worthwhile (e.g. symmetric approximation, etc.)
 - More on this in units 17/18 on 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, 10, 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

- The linearization errors in $F'(c^q)\Delta c^q = (\beta - 1)F(c^q)$ can cause the resulting Δc^q to lead to a poor estimate for c^{q+1} via $c^{q+1} = c^q + \Delta c^q$
- Instead, Δc^q is often just used as a search direction, i.e. $c^{q+1} = c^q + \alpha^q \Delta c^q$
- Then, the new domain is the 1D parameterized line $c^{q+1}(\alpha) = c^q + \alpha \Delta c^q$
- Find an α with $F(c^{q+1}(\alpha)) = 0$ simultaneously for all equations
- Trust Region methods restrict α in various ways, e.g. $0 \leq \alpha \leq 1$
 - The intention is to restrict the domain to a region where the linear approximation is trusted

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 an α with $F(c^{q+1}(\alpha)) = 0$

- Option 1: find simultaneous roots of the vector valued $F(c^{q+1}(\alpha)) = 0$
- Option 2: find roots of or minimize $g(\alpha) = \frac{1}{2} F^T(c^{q+1}(\alpha)) F(c^{q+1}(\alpha))$, to find or make progress towards an α with $F(c^{q+1}(\alpha)) = 0$

Optimization

- Minimize the scalar cost function $\hat{f}(c)$ by finding the critical points where $\nabla \hat{f}(c) = J_{\hat{f}}^T(c) = F(c) = 0$ (as usual)
 - This is a nonlinear system of equations
- $F'(c^q)\Delta c^q = (\beta - 1)F(c^q)$ gives the search direction (as usual)
 - Here, $F'(c) = J_F(c) = H_{\hat{f}}^T(c)$
 - So, actually solving $H_{\hat{f}}^T(c^q)\Delta c^q = (\beta - 1)J_{\hat{f}}^T(c^q)$ to find the search direction Δc^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 towards 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) $Ac = b$
 - 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; P1[Part I - Linear Algebra (units 1-12) Ac = b]; P2[Part II - Optimization (units 13-20)]; P2 -- linearize --> P1; P1 -- line search --> P2; P2 -- Theory --> T[Theory]; P2 -- Methods --> M[Methods];
```

# 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
  - A converged  $t^*$  satisfies  $t^* = \hat{g}(t^*) = g(t^*) + t^*$  implying that  $g(t^*) = 0$

Error Analysis:

- $e^{q+1} = t^{q+1} - t^* = \hat{g}(t^q) - \hat{g}(t^*) = \hat{g}'(\hat{t})(t^q - t^*) = \hat{g}'(\hat{t})e^q$  for some  $\hat{t}$  between  $t^q$  and  $t^*$  by the Mean Value Theorem
  - If all such  $\hat{t}$  had  $|\hat{g}'(\hat{t})| \leq C < 1$ , then  $|e^q| \leq C^q |e^0|$  would prove convergence
  - So, convergence requires:  $|\hat{g}'(t^*)| < 1$ ,  $g$  sufficiently smooth, and an initial guess close enough to  $t^*$

# Convergence Rate

- Consider the bound  $\|e^{q+1}\| \leq C\|e^q\|^p$  asymptotically 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

## Notes:

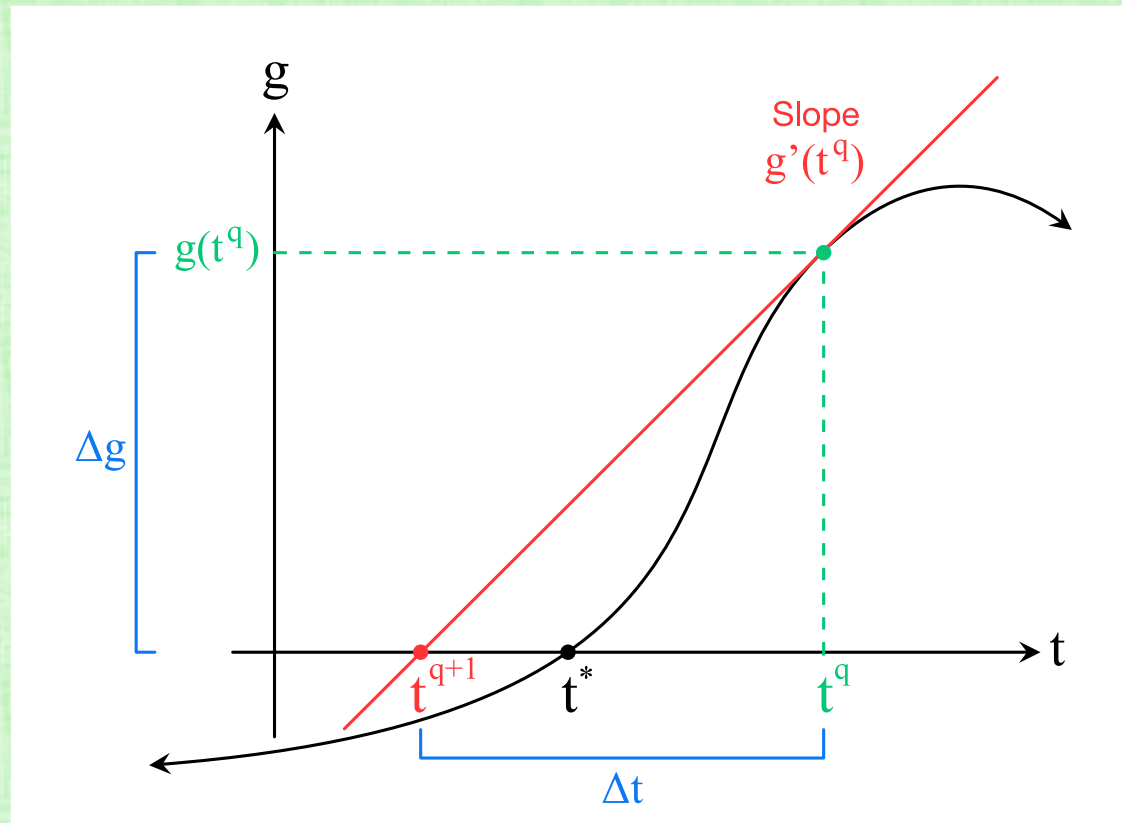
- Statements only apply asymptotically (i.e. once convergence is happening)
- Might converge to a different non-desired root (when other roots are present)
- The root finder requires less accuracy when solving  $g(t) = 0$  only approximates the problem being solved

# 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  $|t^{q+1} - t^q| < \frac{\epsilon}{|g'(t^q)|}$ 
  - Poorly conditioned when  $g'(t^*)$  is small
  - Thus, especially problematic for repeated roots where  $g'(t^*) = 0$
- Quadratic convergence rate ( $p = 2$ ), unless  $g'(t^*) = 0$
- Requires computing  $g$  and  $g'$  every iteration
  - Computing derivatives can be both difficult and expensive (see units 17/18 on Computing/Avoiding Derivatives)

# 1D Newton's Method

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

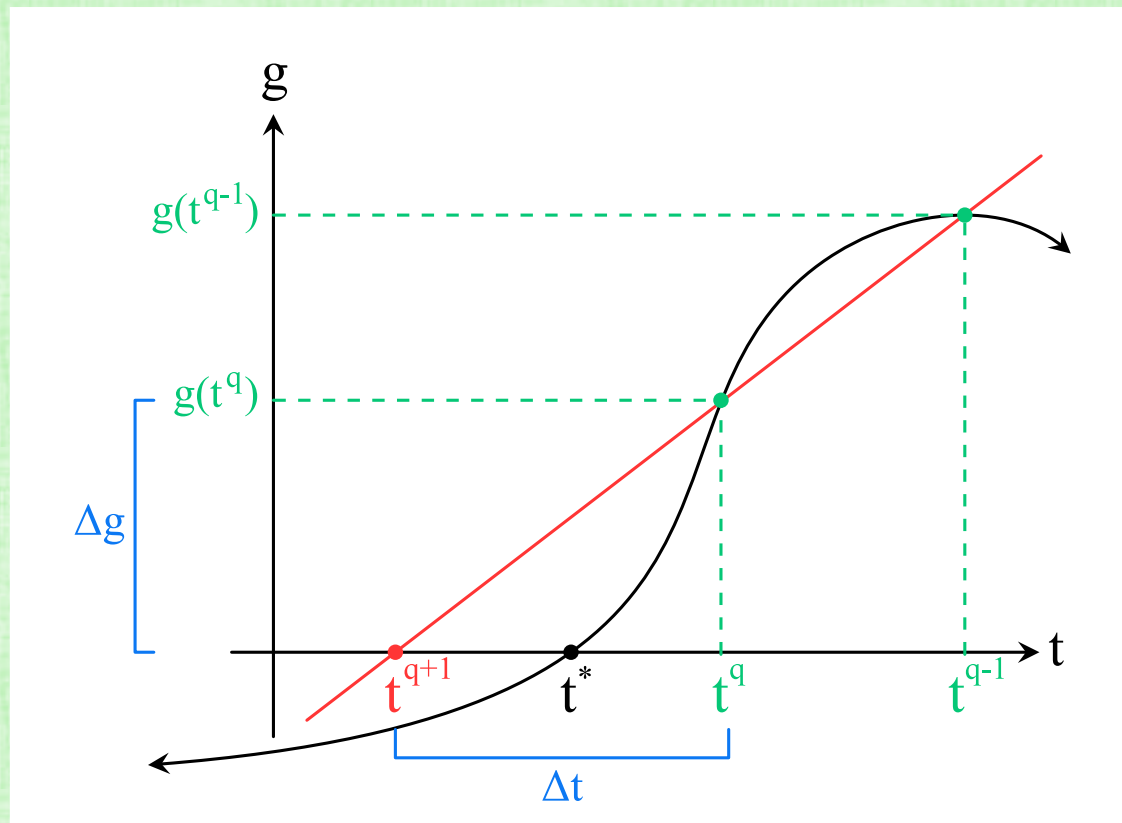


# Secant Method

- Replace  $g'(t^q)$  in Newton's method with an estimate (a few choices for this)
- The standard estimate 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  $p \approx 1.618$ , unless  $g'(t^*) = 0$
- Often (typically) faster than Newton
  - $g'$  is not needed, making each iteration (potentially much) faster
  - only a few extra iterations are required to obtain the same accuracy, unless very high accuracy is desired

# 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  $g'(t^q) \approx \frac{g(t^q) - g(t^{q-1})}{t^q - t^{q-1}}$

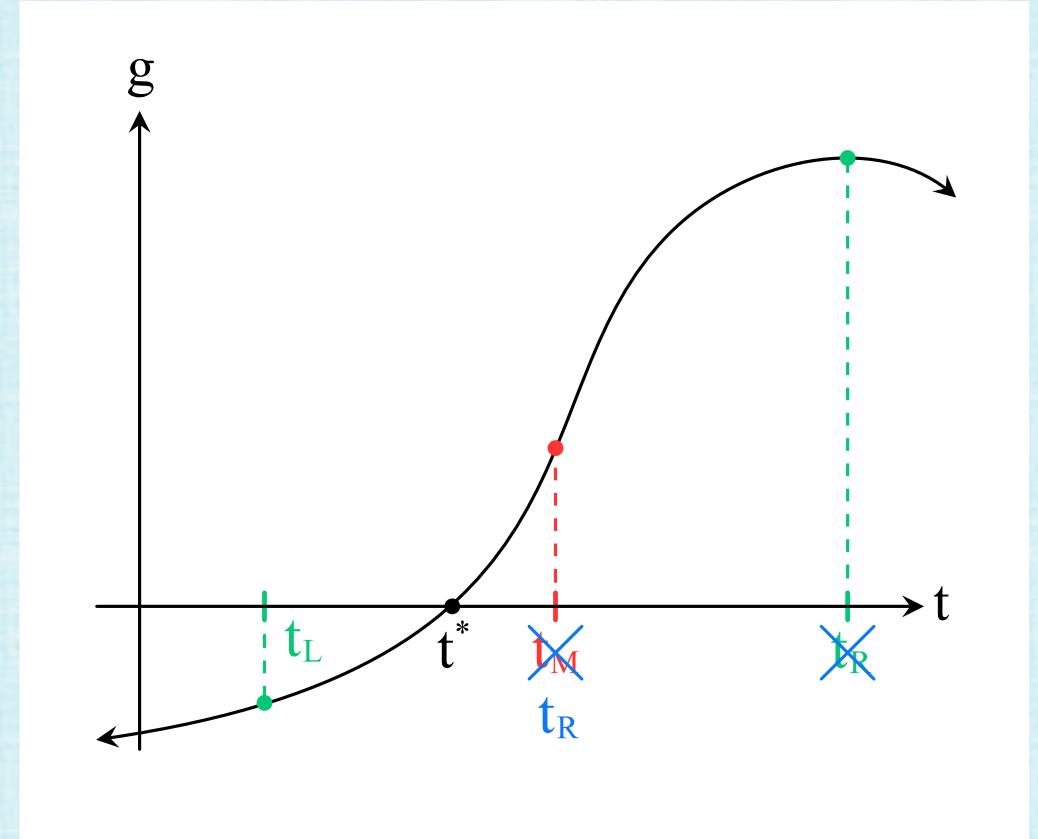
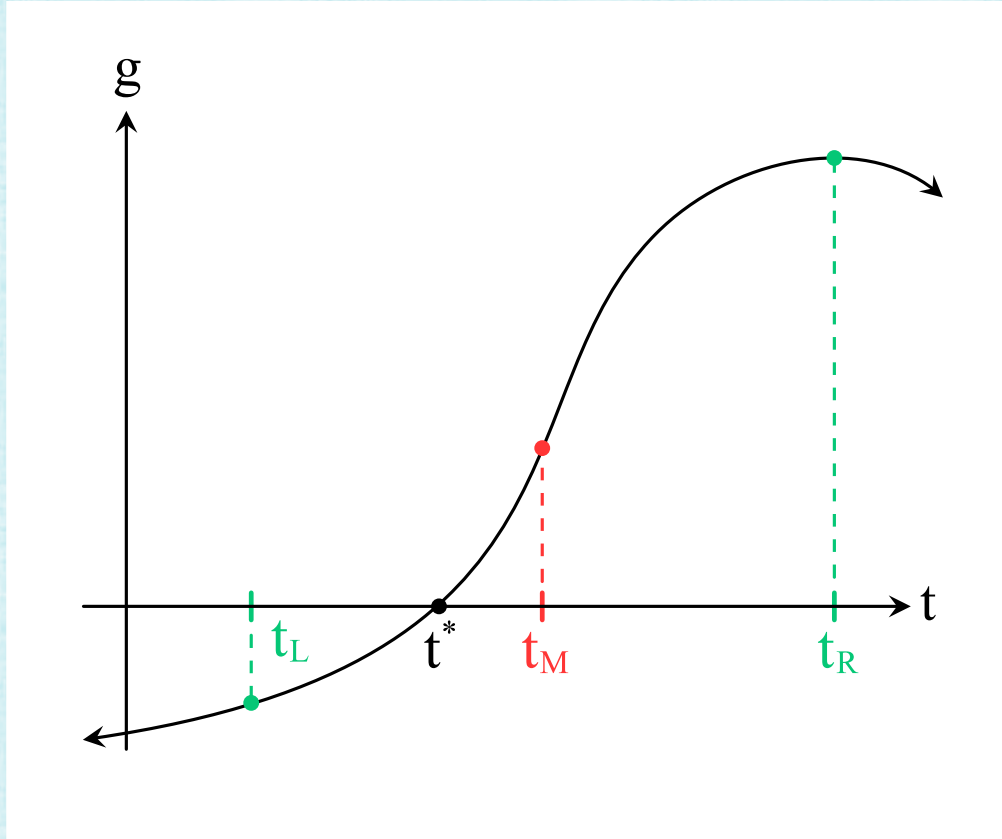


# Bisection Method

- 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  $t_M = \frac{t_L + t_R}{2}$ ,
  - If  $g(t_L)g(t_M) < 0$ , set  $t_R = t_M$
  - Otherwise, set  $t_L = t_M$  knowing that  $g(t_R)g(t_M) < 0$  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, would have 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)
- If Newton/Secant attempt to leave the current interval, instead use Bisection to continue shrinking the 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)

# Some Useful Derivatives

For line search:

- $\frac{\partial}{\partial t} c^{q+1}(t) = \Delta c^q$ , since  $c^{q+1}(t) = c^q + t\Delta c^q$

For solving nonlinear systems  $F(c) = 0$ :

- $\frac{\partial}{\partial t} F(c^{q+1}(t)) = J_F(c^{q+1}(t))\Delta c^q$  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  $F_i(c^{q+1}(t))$  is the scalar entry in the  $i$ -th row of  $F^T(c^{q+1}(t))$

For minimizing a scalar cost function  $\hat{f}(c)$ :

- Find critical points of  $\hat{f}(c^{q+1}(t))$  by solving the nonlinear system  $J_{\hat{f}}^T(c^{q+1}(t)) = 0$
- $\frac{\partial}{\partial t} J_{\hat{f}}^T(c^{q+1}(t)) = H_{\hat{f}}^T(c^{q+1}(t))\Delta c^q$  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)

- The linearization errors in  $F'(c^q)\Delta c^q = (\beta - 1)F(c^q)$  can cause the resulting  $\Delta c^q$  to lead to a poor estimate for  $c^{q+1}$  via  $c^{q+1} = c^q + \Delta c^q$
- Instead,  $\Delta c^q$  is often just used as a search direction, i.e.  $c^{q+1} = c^q + \alpha^q \Delta c^q$
- Then, the new domain is the 1D parameterized line  $c^{q+1}(\alpha) = c^q + \alpha \Delta c^q$
- Find an  $\alpha$  with  $F(c^{q+1}(\alpha)) = 0$  simultaneously for all equations
- Trust Region methods restrict  $\alpha$  in various ways, e.g.  $0 \leq \alpha \leq 1$ 
  - The intention is to restrict the domain to a region where the linear approximation is trusted

## 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 an  $\alpha$  with  $F(c^{q+1}(\alpha)) = 0$
  
- Option 1: find simultaneous **roots** of the vector valued  $F(c^{q+1}(\alpha)) = 0$
- Option 2: find **roots** of or minimize  $g(\alpha) = \frac{1}{2} F^T(c^{q+1}(\alpha)) F(c^{q+1}(\alpha))$ , to find or make progress towards an  $\alpha$  with  $F(c^{q+1}(\alpha)) = 0$

# Nonlinear Systems

- Solve  $J_F(c^q)\Delta c^q = (\beta - 1)F(c^q)$  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: 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,  $g'(t) = \frac{1}{2}F^T(c^{q+1}(t))J_F(c^{q+1}(t))\Delta c^q + \frac{1}{2}(\Delta c^q)^T J_F^T(c^{q+1}(t))F(c^{q+1}(t))$
  - Since both terms are scalars,  $g'(t) = F^T(c^{q+1}(t))J_F(c^{q+1}(t))\Delta c^q$

# Recall: Optimization (Unit 14)

- Minimize the scalar cost function  $\hat{f}(c)$  by finding the critical points where  $\nabla \hat{f}(c) = J_{\hat{f}}^T(c) = F(c) = 0$  (as usual)
  - This is a nonlinear system of equations
- $F'(c^q)\Delta c^q = (\beta - 1)F(c^q)$  gives the search direction (as usual)
  - Here,  $F'(c) = J_F(c) = H_{\hat{f}}^T(c)$
  - So, actually solving  $H_{\hat{f}}^T(c^q)\Delta c^q = (\beta - 1)J_{\hat{f}}^T(c^q)$  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 towards critical points of  $\hat{f}(c)$
- Option 3: minimize  $\hat{f}(c^{q+1}(\alpha))$  directly

# Optimization

- Solve  $H_{\hat{f}}^T(c^q)\Delta c^q = (\beta - 1)J_{\hat{f}}^T(c^q)$  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: 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  $g(t) = \frac{1}{2}J_{\hat{f}}(c^{q+1}(t))J_{\hat{f}}^T(c^{q+1}(t)) = 0$  to find or make progress towards critical points of  $\hat{f}(c)$ 
  - Here,  $g'(t) = \frac{1}{2}J_{\hat{f}}(c^{q+1}(t))H_{\hat{f}}^T(c^{q+1}(t))\Delta c^q + \frac{1}{2}(\Delta c^q)^T H_{\hat{f}}(c^{q+1}(t))J_{\hat{f}}^T(c^{q+1}(t))$
  - Since both terms are scalars,  $g'(t) = J_{\hat{f}}(c^{q+1}(t))H_{\hat{f}}^T(c^{q+1}(t))\Delta c^q$
- Option 3: **minimize**  $\hat{f}(c^{q+1}(t))$  directly (see **unit 16**)

# Unit 16

# 1D Optimization

# Part II Roadmap

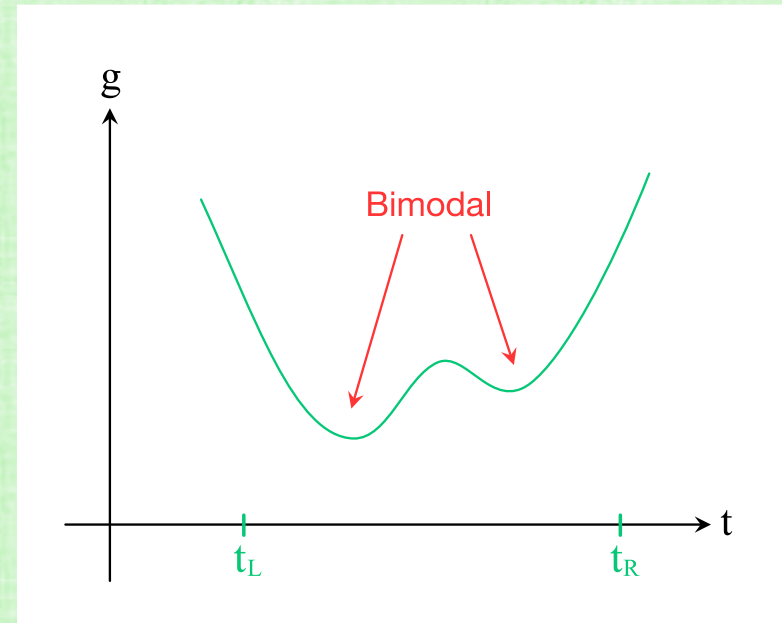
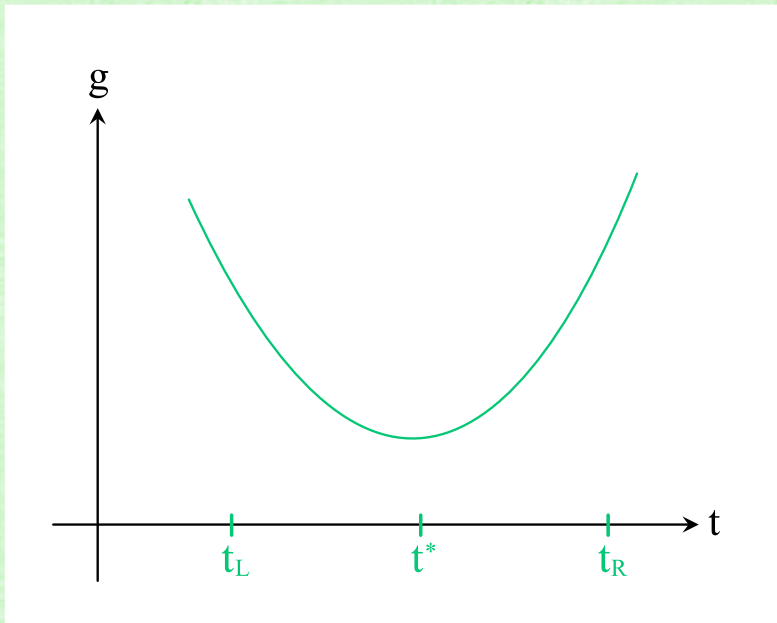
- Part I – Linear Algebra (units 1-12)  $Ac = b$
  - Part II – Optimization (units 13-20)
    - (units 13-16) Optimization -> Nonlinear Equations -> 1D roots/minima
    - (units 17-18) Computing/Avoiding Derivatives
    - (unit 19) Hack 1.0: "I give up"  $H = I$  and  $J$  is mostly 0 (descent methods)
    - (unit 20) Hack 2.0: "It's an ODE!?" (adaptive learning rate and momentum)
- 
- ```
graph TD; P1[Part I - Linear Algebra (units 1-12) Ac = b]; P2[Part II - Optimization (units 13-20)]; P2 -- linearize --> P1; P1 -- line search --> P2; P2 -- Theory --> T[1D roots/minima]; P2 -- Methods --> M["(unit 19) Hack 1.0: 'I give up' H = I and J is mostly 0 (descent methods)"]; P2 -- Methods --> M2["(unit 20) Hack 2.0: 'It's an ODE!?' (adaptive learning rate and momentum)"];
```

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: $t^{q+1} = t^q - g'(t^q) \frac{t^q - t^{q-1}}{g'(t^q) - g'(t^{q-1})}$ (can replace g' with approximations too)
- Bisection: $g'(t_L)g'(t_R) < 0$ is the new condition
- Mixed Methods: mixing the above (as in unit 15)

Unimodal

- Unimodal means one mode (bimodal means two modes)
- In the context of 1D optimization, this means that a function has a unique 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 $[t^*, t_R]$

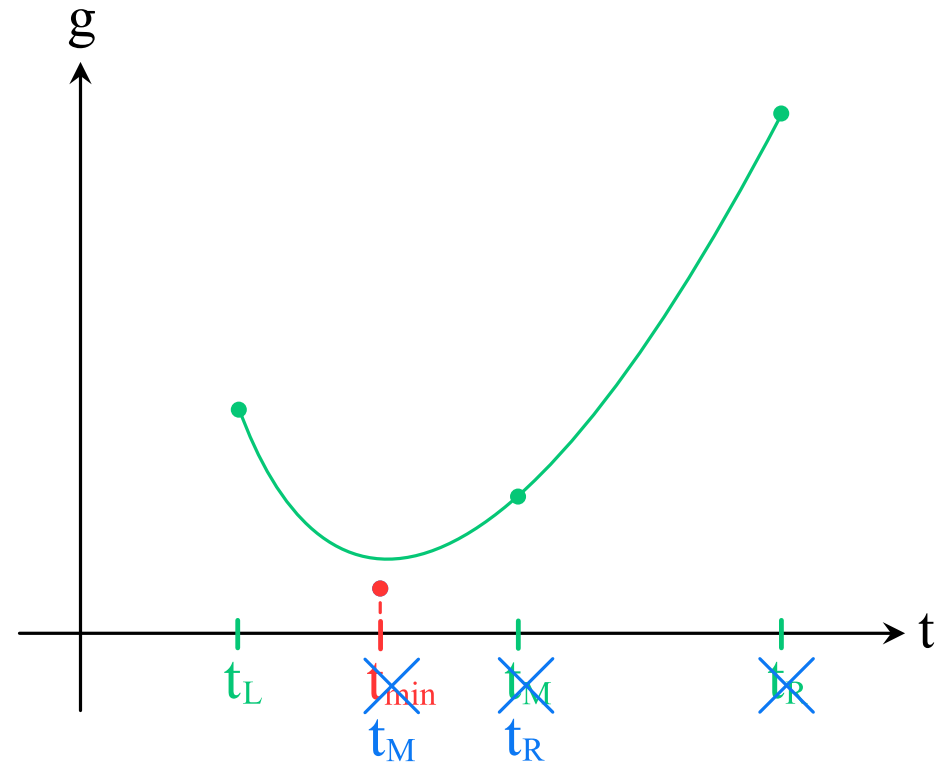
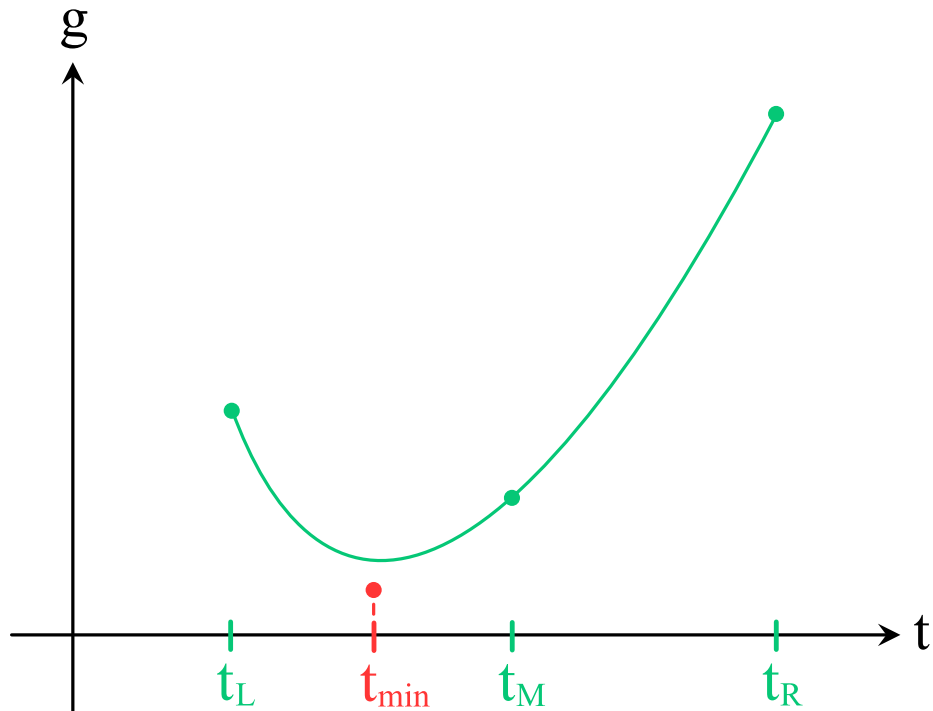


Successive Parabolic Interpolation

- Use parabolas to find candidates for minima, motivated by using lines to find candidates for roots in Newton/Secant methods
- 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 swap their names)
- 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 $t_M = t_{min}$



Discarding Intervals

- Bisection uses 3 points to discard an interval during root finding
- Successive Parabolic Interpolation uses 4 points to discard an interval during minimization

Generalization of 4-point discarding:

- Let $[t_L, t_R]$ have two 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

- Let the interval size be $\delta = t_R - t_L$
- Place the interior points symmetrically, i.e. $t_{M1} = t_L + \lambda\delta$ and $t_{M2} = t_R - \lambda\delta$ with inward displacement $\lambda \in (0, .5)$
- After discarding an interval, an interior point becomes an endpoint
- Key idea: force the other interior point to stay and interior point (to efficiently reduce evaluations of g)

Derivation:

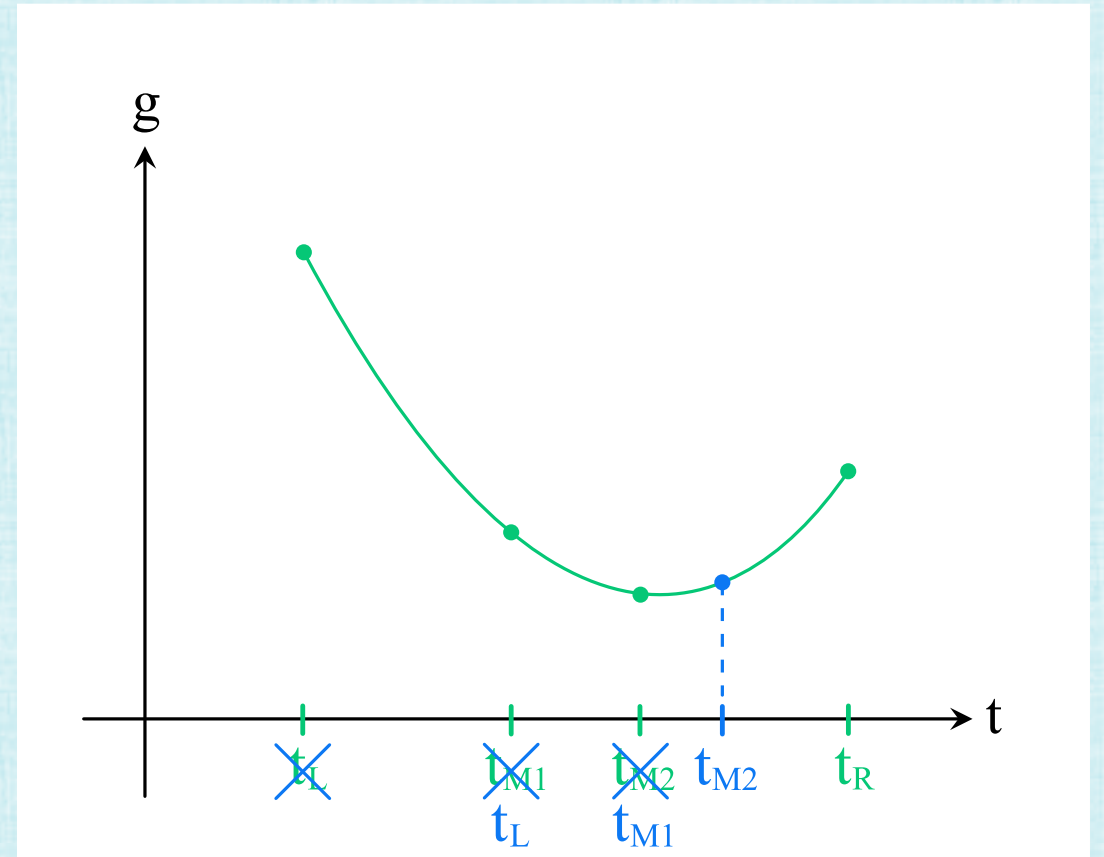
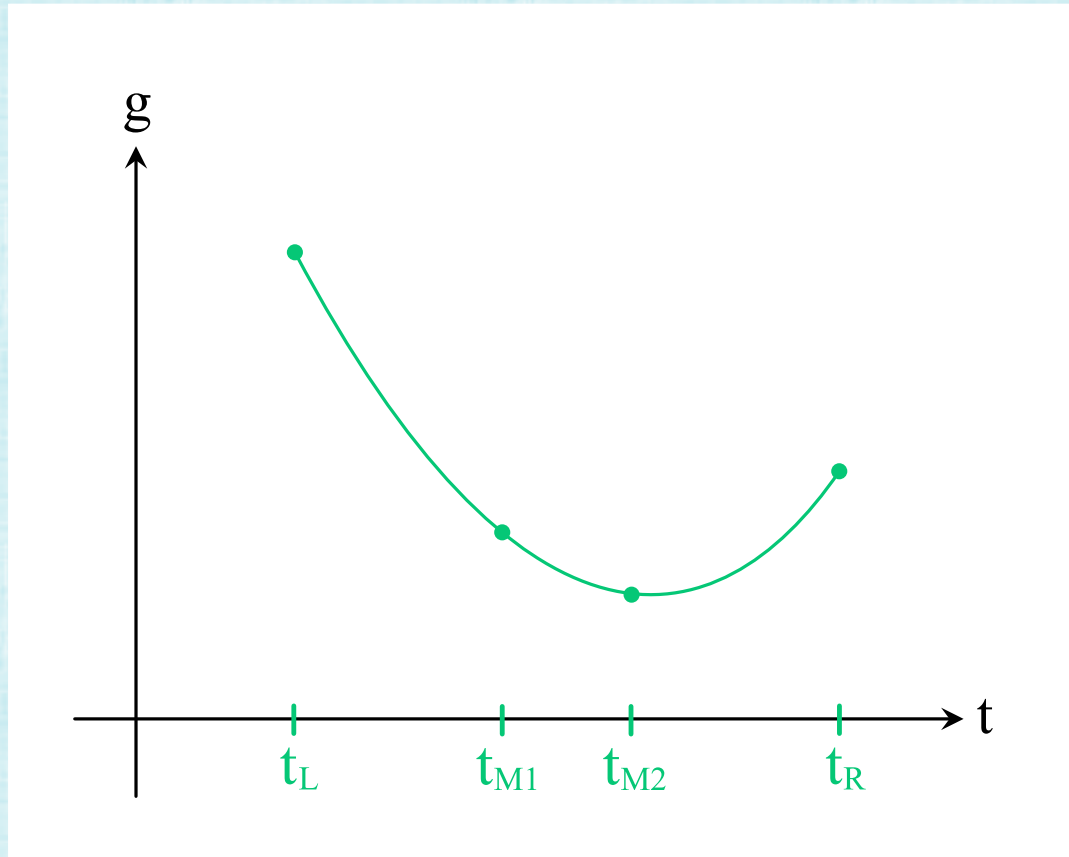
- Discard the right interval to obtain $t_R^{new} = t_{M2}$ and $\delta^{new} = (1 - \lambda)\delta$
 - Discarding the left interval gives similar math
- t_{M1} cannot become t_{M1}^{new} , since it is not $\lambda\delta^{new}$ inward from t_L ; so, t_{M1} becomes t_{M2}^{new}
 - That is, $t_L + \lambda\delta = t_R^{new} - \lambda\delta^{new}$
 - Or $t_L + \lambda\delta = t_L + \delta - \lambda\delta - \lambda(1 - \lambda)\delta = t_L + (1 - 2\lambda + \lambda^2)\delta$
 - In other words, $\lambda^2 - 3\lambda + 1 = 0$ implying $\lambda = \frac{3 - \sqrt{5}}{2}$ since $\lambda = \frac{3 + \sqrt{5}}{2} \notin (0, .5)$

Golden Section Search

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

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 $[t_L, t_R]$
- 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)
- If Successive Parabolic Interpolation attempts to leave the current interval, instead use Golden Section Search to continue shrinking the 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 evaluation of the function g
- The root-finding approach differentiates g and solves $g'(t) = 0$ to identify critical points
 - The root finding methods require evaluation of the function, which is g' here
 - **Newton** (and mixed methods using Newton) requires the derivative of the function, which is g'' here

Recall: Some Useful Derivatives (unit 15)

For line search:

- $\frac{\partial}{\partial t} c^{q+1}(t) = \Delta c^q$, since $c^{q+1}(t) = c^q + t\Delta c^q$

For solving nonlinear systems $F(c) = 0$:

- $\frac{\partial}{\partial t} F(c^{q+1}(t)) = J_F(c^{q+1}(t))\Delta c^q$ 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 $F_i(c^{q+1}(t))$ is the scalar entry in the i -th row of $F(c^{q+1}(t))$

For minimizing a scalar cost function $\hat{f}(c)$:

- Find critical points of $\hat{f}(c^{q+1}(t))$ by solving the nonlinear system $J_{\hat{f}}^T(c^{q+1}(t)) = 0$
- $\frac{\partial}{\partial t} J_{\hat{f}}^T(c^{q+1}(t)) = H_{\hat{f}}^T(c^{q+1}(t))\Delta c^q$ and $\frac{\partial}{\partial t} J_{\hat{f}}(c^{q+1}(t)) = (\Delta c^q)^T H_{\hat{f}}(c^{q+1}(t))$
 - $\frac{\partial}{\partial t} (J_{\hat{f}}^T)_i(c^{q+1}(t)) = (H_{\hat{f}}^T)_i(c^{q+1}(t))\Delta c^q$

Additional Useful Derivatives

- $\frac{\partial}{\partial t} J_F(c^{q+1}(t)) = (\Delta c^q)^T H_F(c^{q+1}(t))$
 - H_F is a rank 3 tensor of all 2nd 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 3rd 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 (unit 15)

- Solve $J_F(c^q)\Delta c^q = (\beta - 1)F(c^q)$ for Δc^q and use $c^{q+1}(t) = c^q + t\Delta c^q$ in $F(c^{q+1}(t)) = 0$
- Option 1: 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, $g'(t) = \frac{1}{2}F^T(c^{q+1}(t))J_F(c^{q+1}(t))\Delta c^q + \frac{1}{2}(\Delta c^q)^T J_F^T(c^{q+1}(t))F(c^{q+1}(t))$
 - Since both terms are scalars, $g'(t) = F^T(c^{q+1}(t))J_F(c^{q+1}(t))\Delta c^q$

Nonlinear Systems

- Solve $J_F(c^q)\Delta c^q = (\beta - 1)F(c^q)$ for Δc^q and use $c^{q+1}(t) = c^q + t\Delta c^q$ in $F(c^{q+1}(t)) = 0$
- Option 1: 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 (unit 15)

- Solve $H_{\hat{f}}^T(c^q)\Delta c^q = (\beta - 1)J_{\hat{f}}^T(c^q)$ for Δc^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: 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 $g(t) = \frac{1}{2}J_{\hat{f}}(c^{q+1}(t))J_{\hat{f}}^T(c^{q+1}(t)) = 0$ to find or make progress towards critical points of $\hat{f}(c)$
 - Here, $g'(t) = \frac{1}{2}J_{\hat{f}}(c^{q+1}(t))H_{\hat{f}}^T(c^{q+1}(t))\Delta c^q + \frac{1}{2}(\Delta c^q)^T H_{\hat{f}}(c^{q+1}(t))J_{\hat{f}}^T(c^{q+1}(t))$
 - Since both terms are scalars, $g'(t) = J_{\hat{f}}(c^{q+1}(t))H_{\hat{f}}^T(c^{q+1}(t))\Delta c^q$
- Option 3: **minimize** $\hat{f}(c^{q+1}(t))$ directly (see **unit 16**)

Optimization

- Solve $H_{\hat{f}}^T(c^q)\Delta c^q = (\beta - 1)J_{\hat{f}}^T(c^q)$ for Δc^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: 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 $g''_i(t) = (\Delta c^q)^T (OMG_{\hat{f}}^T)_i(c^{q+1}(t)) \Delta c^q$
- Option 2: **minimize** $g(t) = \frac{1}{2}J_{\hat{f}}(c^{q+1}(t))J_{\hat{f}}^T(c^{q+1}(t))$ 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** $g(t) = \hat{f}(c^{q+1}(t))$ 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) $Ac = b$
 - 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)
-
- The diagram illustrates the flow of concepts from Part I to Part II. A red arrow labeled "linearize" points from the optimization section back to the linear algebra equation $Ac = b$. Another red arrow labeled "line search" points from the linear algebra equation to the optimization section. On the right side, a blue arrow labeled "Theory" points to the first item in the optimization section. A blue arrow labeled "Methods" points to a bracketed group of the last three items in the optimization section.

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
 - Specialized approaches exist for certain 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 towards mimicking biological (typically human) neural networks and learning
- Biological neurons are “all or none”, motivating similar strategies in artificial neural networks
 - This leads to discontinuous functions with identically zero derivatives away from the discontinuity, which is disastrous for optimization
- Biological neurons fire with increased frequency for stronger signals
 - This leads to a piecewise constant discontinuous derivative, which is problematic for optimization
- Smoothing allows optimization to “work”
 - That is, smoothing allows one to minimize the loss (objective/cost function) to find the parameters/coefficients for the network architecture

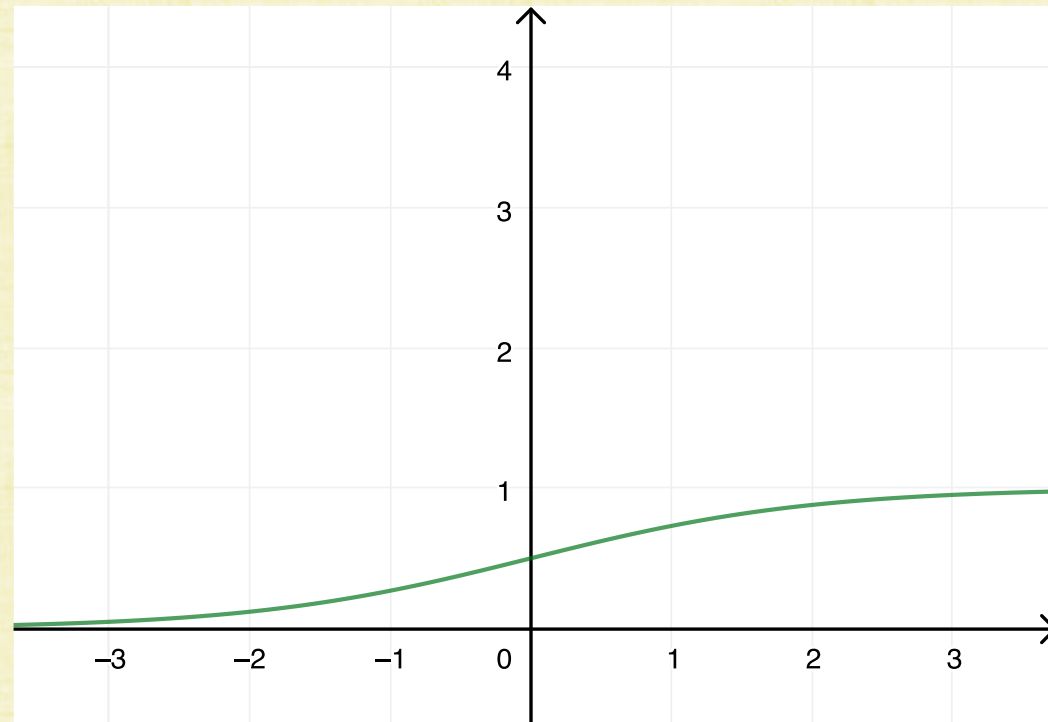
Heaviside Function

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



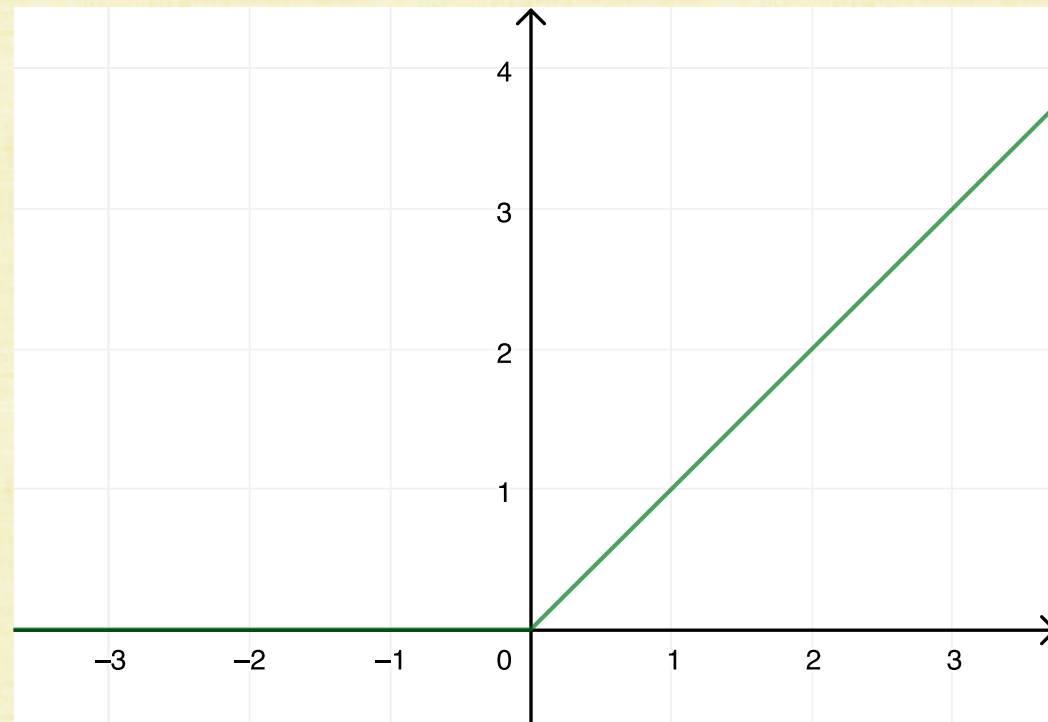
Sigmoid Function

- 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 further away from $x = 0$



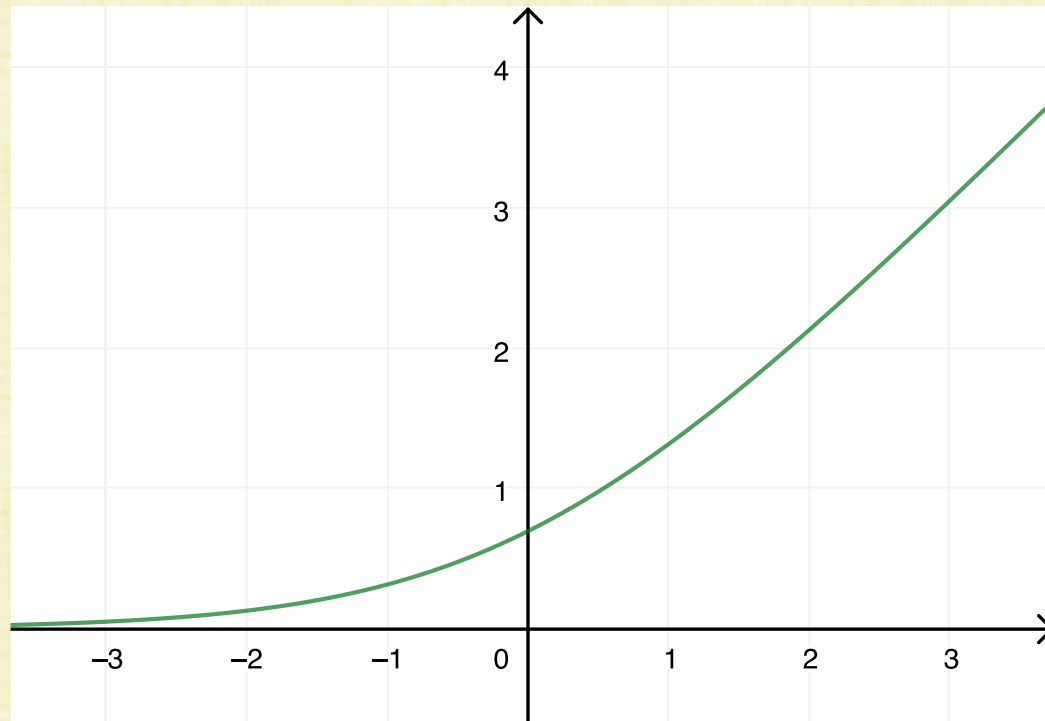
Rectifier Functions

- $R(x) = \max(x, 0)$ or similar functions that 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. $(.99, 1) \rightarrow (0, 1)$, $(1, .99) \rightarrow (1, 0)$, etc.
 - Highly discontinuous!
- Soft Max is a **smoothed** 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

- The training data (x_i, y_i) has binary class labels $y_i = \pm 1$
- Find a plane $\hat{n}^T (x - x_o) = 0$ that separates the two classes
 - \hat{n} is the unit normal and x_o is a point on the plane
- The closest data point x_i on each side of the plane are called support vectors
- When the separating plane is equidistant between the two support vectors, then they lie on parallel planes: $\hat{n}^T (x - x_o) = \pm \epsilon$ (where ϵ is the margin)
- Divide by ϵ to normalize, obtaining $c^T (x - x_o) = \pm 1$ where c points in the normal direction (but is not unit length); then, maximizing the margin ϵ 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 these 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 a scalar unknown $b = c^T x_o$
- Can fold the constraints into the cost function using Heaviside functions
 - (see next slide)
- Subsequently **smoothing** those Heaviside functions is called soft-margin
- Note: new data is classified (via inference) based on the sign of $c^T x_{new} - b$

(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 zero 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, 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 discontinuous 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
 - Subsequently allows access to higher derivatives
 - Some of the aforementioned benefits enable the use of better solvers
 - Helps to write/maintain more efficient code with less bugs
 - Etc.

An 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 the denominators 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
 - It would be better to implement $h(t) = t + 2$ and $h'(t) = 1$ directly, so that the simplifications can be done by hand (for accuracy and robustness)

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 have the derivative treatment simplified for accuracy/robustness/efficiency
 - For example (from the previous slide): Implementing $h(t) = t + 2$ and $h'(t) = 1$ directly is more accurate and robust

Differentiate the Right Thing (an example)

- Consider an iterative solver (e.g. CG, Minres, etc.) that solves $Ac = b$ to find c given b
- Source code can be enormous, complicated, confusing, impenetrable, a black box, etc.
- It is tempting to consider software that claims to differentiate “chunks of code”
 - Sometimes these approaches work, and the answers are reasonable
 - But, it can be difficult to ascertain when computational inaccuracies in a black box approach are having an adverse effect
- Alternatively, when invertible: $c = A^{-1}b$ and $\frac{\partial c_i}{\partial b_j} = \tilde{a}_{ij}$ where \tilde{a}_{ij} 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.
- Note that the derivative is actually independent of the iterative solver (CG, Minres, etc.) and the code written to implement it
- Computing the derivative based on the result of an algorithm, instead of differentiating the algorithm itself, allows one to avoid errors in the algorithm
 - In this case, one avoids any errors and error accumulation in the iterative solver that might result from poor conditioning
 - This sort of approach is often referred to as an **implicit layer**

The Used Car Salesman

- 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 of this knowledge/expertise
- The question is: **what kind of computer scientist do you want to be?**

Oversimplified Thinking

- Beware of claims that drastically oversimplify
- For example, 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 in the first place (leaving one to question these sorts of claims)

Finite Differences

- Derivatives can be approximated by various formulas
 - E.g., this is 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: $g'(t) = \frac{g(t+h)-g(t)}{h} + O(h)$, 1st order accurate
 - Backward Difference: $g'(t) = \frac{g(t)-g(t-h)}{h} + O(h)$, 1st order accurate
 - Central Difference: $g'(t) = \frac{g(t+h)-g(t-h)}{2h} + O(h^2)$, 2nd order accurate
 - Second Derivative: $g''(t) = \frac{g(t+h)-2g(t)+g(t-h)}{h^2} + O(h^2)$, 2nd order accurate
- These approximations can be evaluated even when $g(t)$ is not known precisely, but when it merely represents the output of some code with input t

Drawbacks of Finite Differences

- 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 errors (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

- Often referred to as Back Propagation
- For each (potentially vector valued) function $F(c_{input})$ in the code, include an analytically correct companion function for the Jacobian matrix $\frac{\partial F}{\partial c}(c_{input})$
- 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 on conditioning, etc.); but, it does not contain the (typically) much larger truncation errors present in finite differencing
- **Code can be considered in chunks** that 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 and other errors will accumulate, of course, and the total error has the potential to be catastrophic (this is typically even worse for the much larger truncation errors)

Second Derivatives

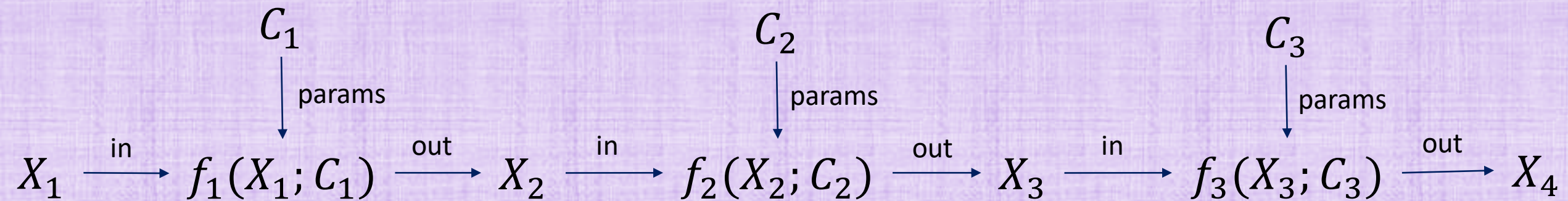
- If c_{input} is size n and $F(c_{input})$ is size m , the Jacobian matrix $\frac{\partial F}{\partial c}(c_{input})$ is size $m \times n$
- The Hessian of second derivatives is size $m \times n \times n$
 - Recall: $m = 1$ for optimization, i.e. for $\hat{f}(c_{input})$
- Writing automatic differentiation 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 idea for combating overfitting is to train several different network architectures on the same data, inference them all, and average the result (model averaging)
 - This can be costly, especially if there are many networks
- Dropout is a “hacky” approach to model averaging
 - 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 derivatives

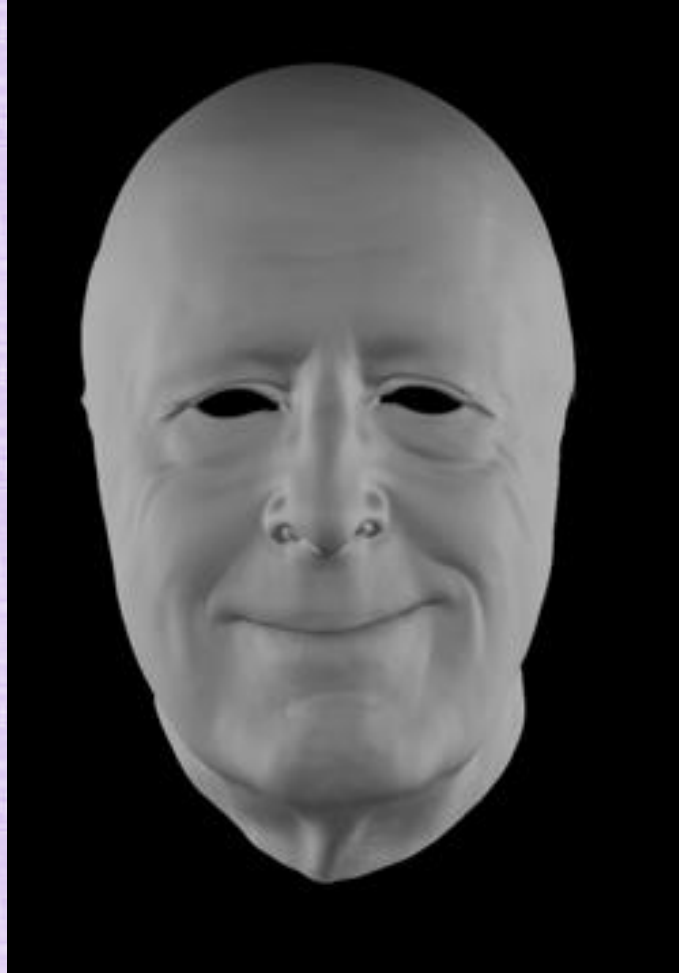
Function Layers

- 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 (an example)



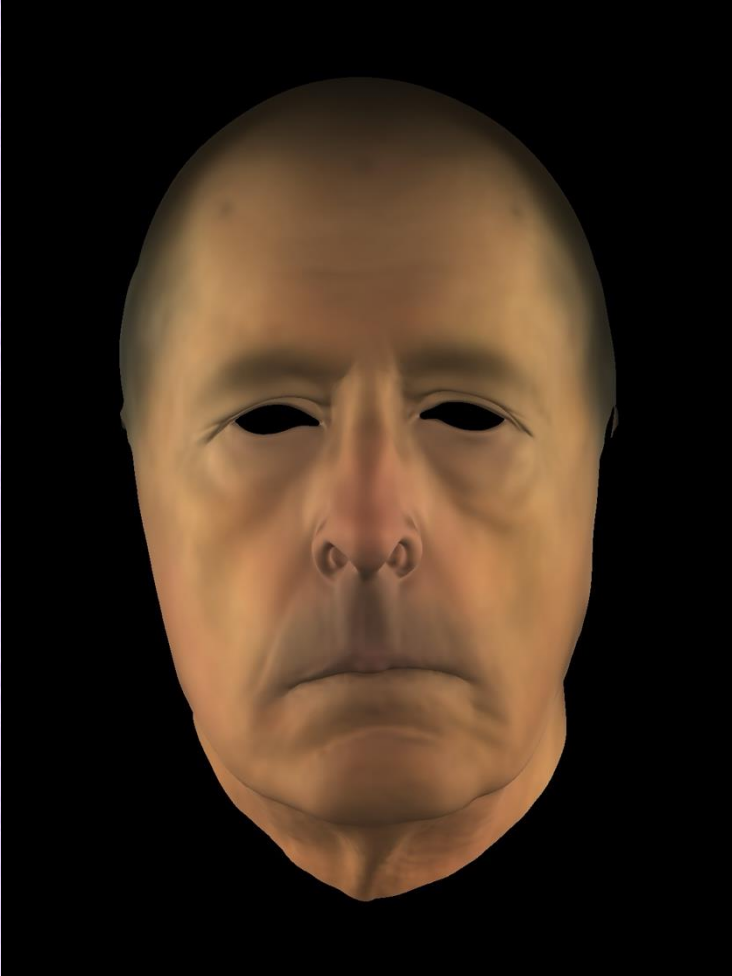
LAYER 1

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

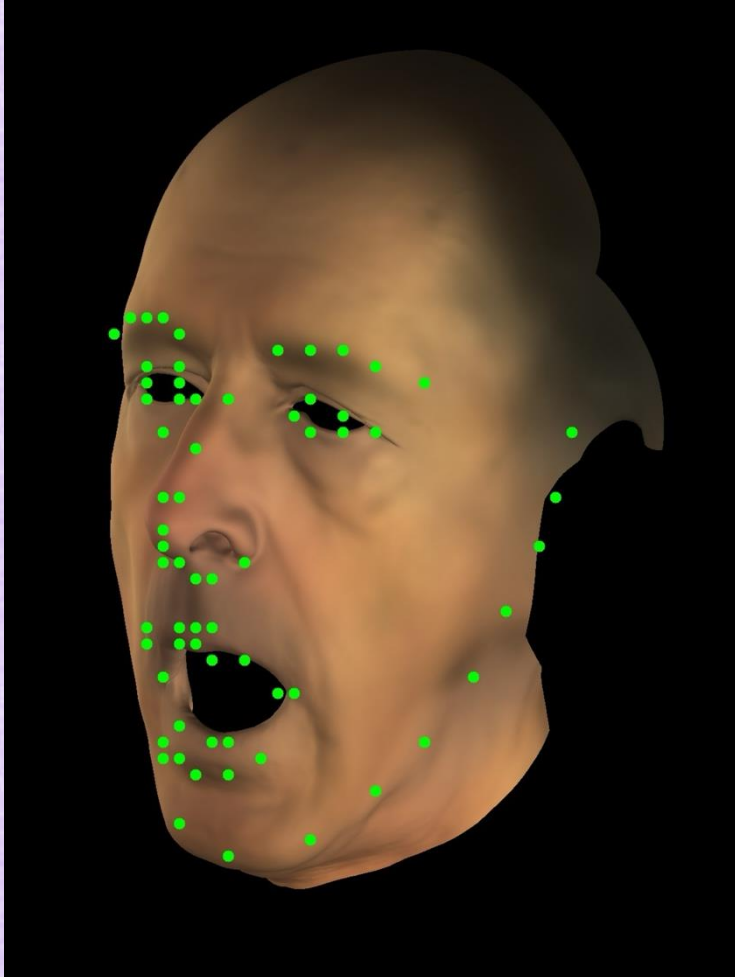
Function Layers (an example)

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 (an 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 (an example)

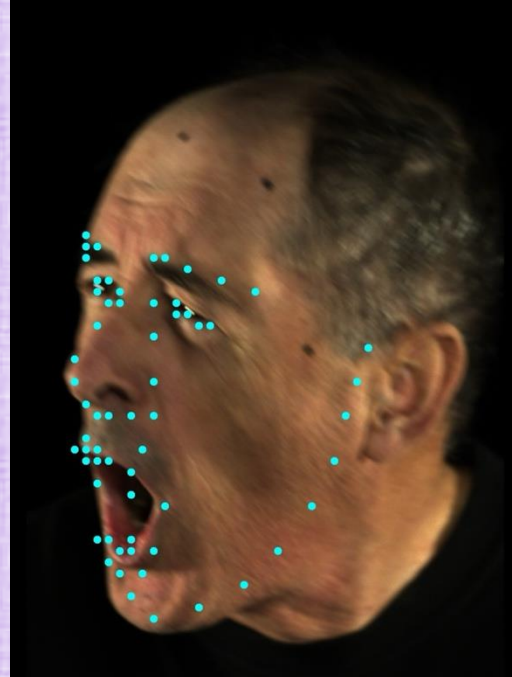


TARGET

- Run a landmark detector on a photograph of the individual to obtain 2D landmark locations
 - Alternatively, can label by hand
- The goal is to have the 2D landmarks output from the complex multi-layered function (on the prior three slides) match the 2D landmarks on the photograph

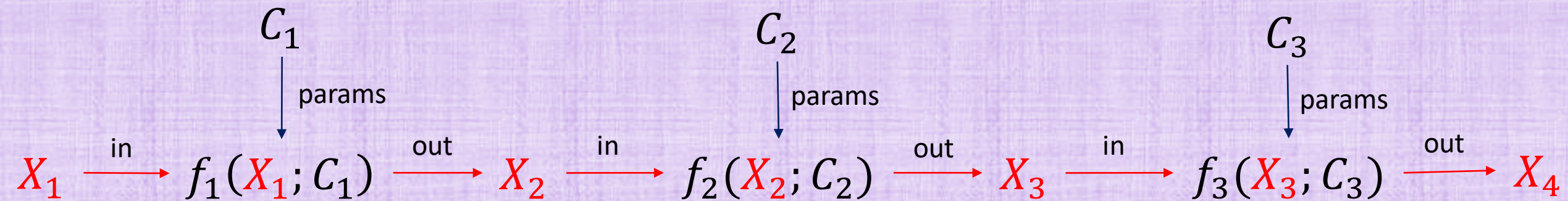
Function Layers (Example)

- Modifying animation controls changes the triangulated surface which changes the rendered pixels in the 2D image which changes the network's determination of the landmark locations
- When the two sets of landmarks agree, the animation controls give some indication of what the person in the photograph was doing



Classical Optimization

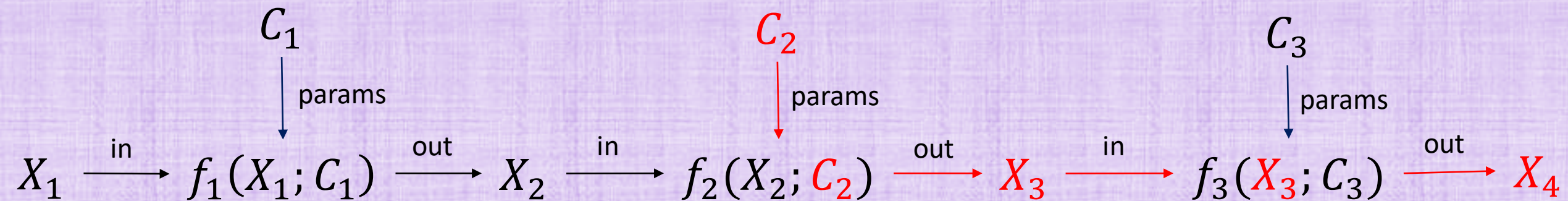
- Find the input X_1 that minimizes $\hat{f}(X_4)$
- Chain rule: $\frac{\partial \hat{f}(X_4)}{\partial X_1} = \frac{\partial \hat{f}(X_4)}{\partial X_4} \frac{\partial X_4}{\partial X_3} \frac{\partial X_3}{\partial X_2} \frac{\partial X_2}{\partial X_1} = \frac{\partial \hat{f}(X_4)}{\partial X_4}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



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

Network Training

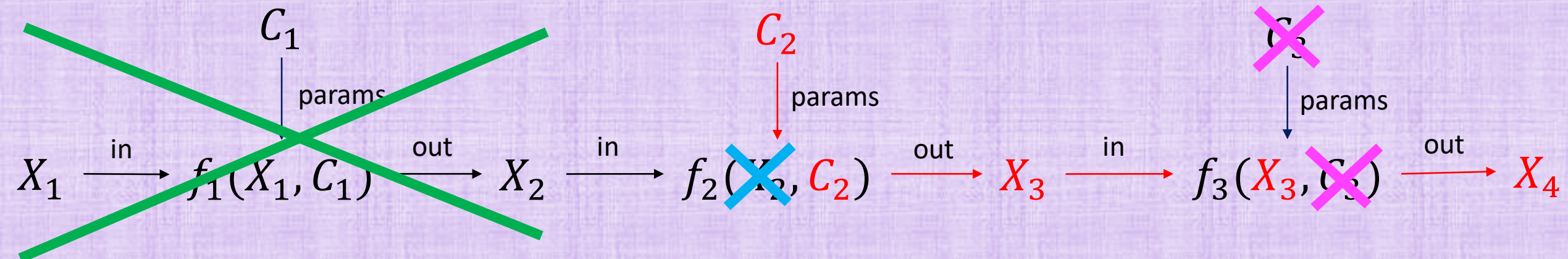
- Train network f_2 by finding parameters C_2 that minimize $\hat{f}(X_4)$
- Chain rule: $\frac{\partial \hat{f}(X_4)}{\partial C_2} = \frac{\partial \hat{f}(X_4)}{\partial X_4} \frac{\partial X_4}{\partial X_3} \frac{\partial X_3}{\partial C_2} = \frac{\partial \hat{f}(X_4)}{\partial X_4} \frac{\partial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



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

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



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

Smoothness and Inference

- Smoothing is required for optimization
- So, smoothing is required for training

- Smoothing is not required for inference

- After training, it's worth considering the replacement of smoothed out functions with their non-smoothed (pre-smoothed) counterparts

Unit 18

Avoiding Derivatives

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)
-
- The diagram illustrates the flow of concepts from Part II to Part I and back. A red arrow labeled "linearize" points from the optimization sub-items back to the equation $Ac = b$. A red arrow labeled "line search" points from the equation $Ac = b$ to the optimization sub-items. Blue arrows labeled "Theory" and "Methods" point to the optimization sub-items, with "Methods" encompassing all four sub-items.

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 through two prior iterates
- Finite differencing (unit 17) may be used to approximate g' 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 trust region methods (such as mixed methods with bisection)
- Trust region approaches (such as mixed methods with bisection) 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, which is g' here
 - Newton (and mixed methods using Newton) require the derivative of the function, which is 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''
- Trust region approaches (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)

- $J_F(c^q)\Delta c^q = (\beta - 1)F(c^q)$ is solved to find the search direction Δc^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)
- Making various approximations to the Jacobian $J_F(c^q)$ perturbs the search direction; so, **robust approaches to 1D line search are important for making “progress” towards solutions**

Quasi-Newton Methods

- $J_F(c^q)\Delta c^q = (\beta - 1)F(c^q)$ is solved to find a search direction Δc^q
- This requires computing the Jacobian matrix of first derivatives $J_F(c^q)$ for a given iterate c^q (i.e., evaluated at c^q)
- Quasi-Newton approaches make various **aggressive** approximations to $J_F(c^q)$
- Quasi-Newton can wildly perturb the search direction
 - So, robust approaches to the 1D line search become quite important for 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 Δc^q
 - Use 1D line search to find c^{q+1} and thus $F(c^{q+1})$; then, update $\Delta c^q = c^{q+1} - c^q$ overwrite Δc^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)
- $H_{\hat{f}}^T(c^q)\Delta c^q = (\beta - 1)J_{\hat{f}}^T(c^q)$ is solved to find a search direction Δc^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

Quasi-Newton Methods (for optimization)

- $H_{\hat{f}}^T(c^q)\Delta c^q = (\beta - 1)J_{\hat{f}}^T(c^q)$ is solved to find a search direction Δc^q
- This requires computing both the Hessian matrix of second derivatives $H_{\hat{f}}^T(c^q)$ and the Jacobian vector of first derivatives $J_{\hat{f}}^T(c^q)$ for a given iterate c^q
- **Second derivatives pose even more issues than first derivatives**
- This makes Quasi-Newton approaches quite popular for optimization
- When c is large, the $O(n^2)$ Hessian $H_{\hat{f}}^T$ can be unwieldy, so some approaches work with $H_{\hat{f}}^{-T}$ (or the pseudo-inverse) directly
- Other approaches optimize further and work with the action of $H_{\hat{f}}^{-T}$ (or the pseudo-inverse) on a vector (in particular, on the right-hand side vector)

Broyden's Method (for Optimization)

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

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 next search direction as $\Delta c^{q+1} = -(H_{\hat{f}}^{-T})^{q+1} J_{\hat{f}}^T(c^{q+1})$

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 $(H_{\hat{f}}^{-T})^{q+1} = (H_{\hat{f}}^{-T})^q + \frac{\left(\Delta c^q - (H_{\hat{f}}^{-T})^q \Delta J_{\hat{f}}^T\right) \left(\Delta c^q - (H_{\hat{f}}^{-T})^q \Delta J_{\hat{f}}^T\right)^T}{\left(\Delta c^q - (H_{\hat{f}}^{-T})^q \Delta J_{\hat{f}}^T\right)^T \Delta J_{\hat{f}}^T}$
- 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 next search direction as $\Delta c^{q+1} = -(H_{\hat{f}}^{-T})^{q+1} J_{\hat{f}}^T(c^{q+1})$

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 $(H_{\hat{f}}^{-T})^{q+1} = (H_{\hat{f}}^{-T})^q - \frac{(H_{\hat{f}}^{-T})^q \Delta J_{\hat{f}}^T \Delta J_{\hat{f}}^T (H_{\hat{f}}^{-T})^q}{\Delta J_{\hat{f}}^T (H_{\hat{f}}^{-T})^q \Delta J_{\hat{f}}^T} + \frac{\Delta c^q (\Delta c^q)^T}{(\Delta c^q)^T \Delta J_{\hat{f}}^T}$
- 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 next search direction as $\Delta c^{q+1} = -(H_{\hat{f}}^{-T})^{q+1} J_{\hat{f}}^T(c^{q+1})$

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 $(H_{\hat{f}}^{-T})^{q+1} = \left(I - \frac{\Delta c^q \Delta J_{\hat{f}}^T}{(\Delta c^q)^T \Delta J_{\hat{f}}^T} \right) (H_{\hat{f}}^{-T})^q \left(I - \frac{\Delta J_{\hat{f}}^T (\Delta c^q)^T}{(\Delta c^q)^T \Delta J_{\hat{f}}^T} \right) + \frac{\Delta c^q (\Delta c^q)^T}{(\Delta c^q)^T \Delta J_{\hat{f}}^T}$
- 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 next search direction as $\Delta c^{q+1} = -(H_{\hat{f}}^{-T})^{q+1} J_{\hat{f}}^T(c^{q+1})$

L-BFGS (Limited Memory BFGS)

- Storing an $n \times n$ approximation to the inverse (or pseudo-inverse) Hessian can become unwieldy for large problems
- More efficient to instead store the vectors (i.e. Δc and $\Delta J_{\hat{f}}^T$) that describe the outer products (**vectors, vector spaces, not matrices**)
 - However, the number of vectors grows with the iteration count q
- L-BFGS estimates the inverse Hessian using only a few of the prior vectors
 - often less than 10 of the prior search directions are required
 - 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 Broyden-style 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, Δc^q is set to the various coordinate directions \hat{e}_k

Other Methods?

Computer Methods in Applied Mechanics and Engineering
Volume 434, 1 February 2025, 117588

Integrating Differential Evolution into Gazelle Optimization for advanced global optimization and engineering applications

Saptadeep Biswas^a, Gyan Singh^a, Binanda Maiti^a, Absalom El-Shamir Ezugwu^b, Kashif Saleem^c, Aseel Smerat^d, Laith Abualigah^a, Uttam Kumar Bera^a

Show more

+ Add to Mendeley Share Cite

<https://doi.org/10.1016/j.cma.2024.117588> Get rights and content

Under a Creative Commons license open access

Highlights

- Proposed a hybrid optimizer merging Gazelle Algorithm with Differential Evolution.
- Improved global and local search for complex optimization problems.
- Outperformed ten optimization algorithms in benchmark tests.
- Validated on engineering problems, proving superior feasibility and efficiency.

Abstract

The Gazelle Optimization Algorithm (GOA) is an innovative metaheuristic inspired by the survival tactics of gazelles in predator-rich environments. While GOA demonstrates

Boosting crayfish algorithm based on halton adaptive quadratic interpolation and piecewise neighborhood for complex optimization problems

Mahmoud Abdel-Salam, Laith Abualigah, Ahmed Ibrahim Alzahrani, Fahad Alblehai, Heming Jia
Article 117429

Article preview

Abstract

Abstract

The Crayfish Optimization Algorithm (CFish) is an innovative meta-heuristic approach that draws inspiration from the movements and behaviors of crayfish. CFish exhibits strong performance across many test sets and optimization issues,

Computer Methods in Applied Mechanics and Engineering
Volume 415, 1 October 2023, 116200

Mantis Search Algorithm: A novel bio-inspired algorithm for global optimization and engineering design problems

Mohamed Abdel-Basset^a, Reda Mohamed^a, Mahinda Zidan^a, Mohammed Jameel^b, Mohamed Abouhawwash^b

Show more

+ Add to Mendeley Share Cite

<https://doi.org/10.1016/j.cma.2023.116200> Get rights and content

Highlights

- A nature-inspired algorithm named mantis search algorithm (MSA) was proposed.
- MSA simulates the hunting behavior and sexual cannibalism of praying mantises.
- MSA was assessed using 52 test problems and 6 engineering design problems.
- MSA is compared with nine published over different benchmarks problems.

Abstract

This study presents a new nature-inspired optimization algorithm, namely the Mantis Search Algorithm (MSA), inspired by the unique hunting behavior and sexual cannibalism of praying mantises. In brief, MSA consists of three optimization stages,

Computer Methods in Applied Mechanics and Engineering
Volume 437, 15 March 2025, 117825

Fungal growth optimizer: A novel nature-inspired metaheuristic algorithm for stochastic optimization

Mohamed Abdel-Basset^a, Reda Mohamed^a, Mohamed Abouhawwash^b

Show more

+ Add to Mendeley Share Cite

<https://doi.org/10.1016/j.cma.2025.117825> Get rights and content

Highlights

- Introduces Fungal Growth Optimizer (FGO), inspired by fungal growth behaviors.
- Balances exploration and exploitation to address optimization challenges.
- Evaluated on four CEC benchmarks and eleven engineering design problems.
- Outperforms sixteen state-of-the-art algorithms in majority of test functions.
- Offers a robust alternative for solving complex stochastic optimization problems.

Abstract

This study presents a new nature-inspired metaheuristic algorithm, known as the fungal growth optimizer (FGO), which is inspired by fungal growth behavior in nature. Fungal growth behavior includes hyphal growth, branching, and spore germination. Hyphal growth behavior replicates hyphal extension and chemotropism to precisely explore the search space, and thus, reach and exploit nutrient-rich regions. This behavior provides a

Computer Methods in Applied Mechanics and Engineering
Volume 440, 15 May 2025, 117908

Harnessing dynamic turbulent dynamics in parrot optimization algorithm for complex high-dimensional engineering problems

Mahmoud Abdel-Salam^a, Saleh Ali Alomari^b, Jing Yang^c, Sangkeum Lee^d, Kashif Saleem^e, Aseel Smerat^f, Vaclav Snasel^h, Laith Abualigah^l

Show more

+ Add to Mendeley Share Cite

<https://doi.org/10.1016/j.cma.2025.117908> Get rights and content

Abstract

The Parrot Optimization Algorithm (PO) is a nature-inspired metaheuristic algorithm developed based on the social and adaptive behaviors of *Pyrrhura molinae* parrots. PO demonstrates robust optimization performance by balancing exploration and exploitation, mimicking foraging and cooperative activities. However, as the algorithm

Computer Methods in Applied Mechanics and Engineering
Volume 436, 1 March 2025, 117718

Dream Optimization Algorithm (DOA): A novel metaheuristic optimization algorithm inspired by human dreams and its applications to real-world engineering problems

Yifan Lang^a, Yuelin Gao^a

Show more

+ Add to Mendeley Share Cite

<https://doi.org/10.1016/j.cma.2024.117718> Get rights and content

Abstract

As optimization problems grow increasingly complex, traditional deterministic algorithms often struggle to address these challenges. Metaheuristic algorithms, with their flexibility and low problem dependency, have emerged as a competitive alternative. This paper introduces the Dream Optimization Algorithm (DOA), inspired by human dreams, which exhibit partial memory retention, forgetting, and logical self-organization characteristics that bear strong similarities to the optimization process in metaheuristic algorithms. DOA incorporates a foundational memory strategy, a forgetting and

Nonlinear Least Squares

- 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

- Minimize $\hat{f}(c) = \frac{1}{2} \tilde{f}^T(c) \tilde{f}(c)$

- Critical points of $\hat{f}(c)$ have $J_{\hat{f}}^T(c) = \begin{pmatrix} \tilde{f}^T(c) \frac{\partial \tilde{f}}{\partial c_1}(c) \\ \tilde{f}^T(c) \frac{\partial \tilde{f}}{\partial c_2}(c) \\ \vdots \\ \tilde{f}^T(c) \frac{\partial \tilde{f}}{\partial c_n}(c) \end{pmatrix} = J_{\tilde{f}}^T(c) \tilde{f}(c) = 0$

- Jacobian matrix of \tilde{f} is $J_{\tilde{f}}(c) = \begin{pmatrix} \frac{\partial \tilde{f}}{\partial c_1}(c) & \frac{\partial \tilde{f}}{\partial c_2}(c) & \dots & \frac{\partial \tilde{f}}{\partial c_n}(c) \end{pmatrix}$

Gauss Newton

Derivation:

- $J_{\tilde{f}}^T(c)\tilde{f}(c) = 0$ becomes $J_{\tilde{f}}^T(c)(\tilde{f}(c^q) + J_{\tilde{f}}(c^q)\Delta c^q + \dots) = 0$
 - By using the Taylor series: $\tilde{f}(c) = \tilde{f}(c^q) + J_{\tilde{f}}(c^q)\Delta c^q + \dots$
- Eliminating high order terms: $J_{\tilde{f}}^T(c)(\tilde{f}(c^q) + J_{\tilde{f}}(c^q)\Delta c^q) \approx 0$
- Finally, evaluating $J_{\tilde{f}}^T$ at c^q gives $J_{\tilde{f}}^T(c^q)J_{\tilde{f}}(c^q)\Delta c^q \approx -J_{\tilde{f}}^T(c^q)\tilde{f}(c^q)$

Analysis:

- Compare to $H_{\hat{f}}^T(c^q)\Delta c^q = -J_{\hat{f}}^T(c^q)$, where $J_{\hat{f}}^T(c) = J_{\tilde{f}}^T(c)\tilde{f}(c)$ is on the previous slide
- Thus, Gauss Newton uses the approximation: $H_{\hat{f}}^T(c^q) \approx J_{\tilde{f}}^T(c^q)J_{\tilde{f}}(c^q)$
 - Removes the second derivatives!

Gauss Newton (a QR approach)

- 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 $J_{\tilde{f}}(c^q)\Delta c^q = -\tilde{f}(c^q)$
 - Alternatively, can derive this by 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
- This is just a linearization of the nonlinear system $\tilde{f}(c) = 0$, aiming to solve it directly as opposed to minimizing $\hat{f}(c) = \frac{1}{2}\tilde{f}^T(c)\tilde{f}(c)$
- Can solve $J_{\tilde{f}}(c^q)\Delta c^q = -\tilde{f}(c^q)$ via any least squares (QR) and minimum norm approach

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
 - It also changes the least squares solution for any overdetermined degrees of freedom

Regularized Gauss Newton

- When concerned about small singular values in $J_{\tilde{f}}(c^q)\Delta c^q = -\tilde{f}(c^q)$, can add $\epsilon I = 0$ as extra equations (recall unit 12 regularization)
 - This leads to $(J_{\tilde{f}}^T(c^q)J_{\tilde{f}}(c^q) + \epsilon^2 I)\Delta c^q = -J_{\tilde{f}}^T(c^q)\tilde{f}(c^q)$
- 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) $Ac = b$
 - 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; P1[Part I - Linear Algebra (units 1-12) Ac = b]; P2[Part II - Optimization (units 13-20)]; P2 -- linearize --> P1; P1 -- line search --> P2; P2 -- Theory --> P2_1["(units 13-16) Optimization -> Nonlinear Equations -> 1D roots/minima"]; subgraph P2_2 ["Methods"]; P2 --> P2_2; P2 --> P2_3["(unit 19) Hack 1.0: 'I give up' H = I and J is mostly 0 (descent methods)"]; P2 --> P2_4["(unit 20) Hack 2.0: 'It's an ODE!?' (adaptive learning rate and momentum)"]; end;
```

# Recall: Gradient (Unit 9)

- Consider the scalar (output) function  $f(c)$  with multi-dimensional input  $c$
- The Jacobian of  $f(c)$  is  $J(c) = \left( \frac{\partial f}{\partial c_1}(c) \quad \frac{\partial f}{\partial c_2}(c) \quad \cdots \quad \frac{\partial f}{\partial c_n}(c) \right)$
- 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  $f'(c)$

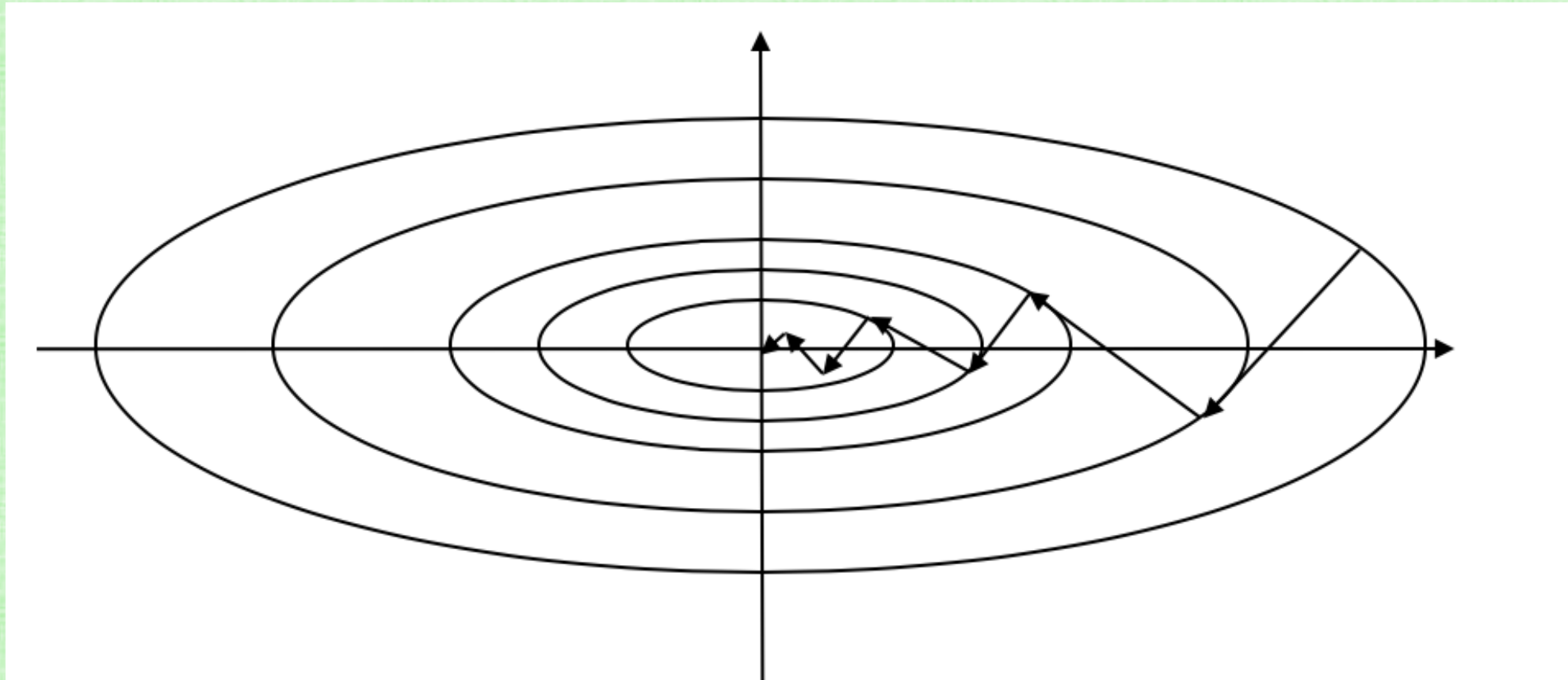
# Gradient/Steepest Descent

- 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):
  - The Quadratic Form of an 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
    - Recall: Since it takes far too long for steepest descent 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)

- 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  $\hat{f}(c) = \frac{1}{2} \tilde{f}^T(c) \tilde{f}(c)$

- Critical points of  $\hat{f}(c)$  have  $J_{\hat{f}}^T(c) = \begin{pmatrix} \tilde{f}^T(c) \frac{\partial \tilde{f}}{\partial c_1}(c) \\ \tilde{f}^T(c) \frac{\partial \tilde{f}}{\partial c_2}(c) \\ \vdots \\ \tilde{f}^T(c) \frac{\partial \tilde{f}}{\partial c_n}(c) \end{pmatrix} = J_{\tilde{f}}^T(c) \tilde{f}(c) = 0$

- Jacobian matrix of  $\tilde{f}$  is  $J_{\tilde{f}}(c) = \begin{pmatrix} \frac{\partial \tilde{f}}{\partial c_1}(c) & \frac{\partial \tilde{f}}{\partial c_2}(c) & \cdots & \frac{\partial \tilde{f}}{\partial c_n}(c) \end{pmatrix}$

# Steepest Descent for Nonlinear Least Squares

- Search direction  $-\nabla \hat{f}(c) = -J_{\hat{f}}^T(c) = -J_{\tilde{f}}^T(c) \tilde{f}(c) = \begin{pmatrix} -\tilde{f}^T(c) \frac{\partial \tilde{f}}{\partial c_1}(c) \\ -\tilde{f}^T(c) \frac{\partial \tilde{f}}{\partial c_2}(c) \\ \vdots \\ -\tilde{f}^T(c) \frac{\partial \tilde{f}}{\partial c_n}(c) \end{pmatrix}$
- Recall that  $\tilde{f}(c)$  was constructed by stacking the  $\hat{m}$  outputs of  $f(x_i, y_i; c)$  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

# Minimizing the Per-Step Cost

- When there is a lot of data,  $m$  can be extremely large
- The cost is exacerbated when the derivatives  $\frac{\partial \tilde{f}}{\partial c_k}$  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)
  - SGD bears some similarity in spirit to Coordinate Descent

# Unit 20

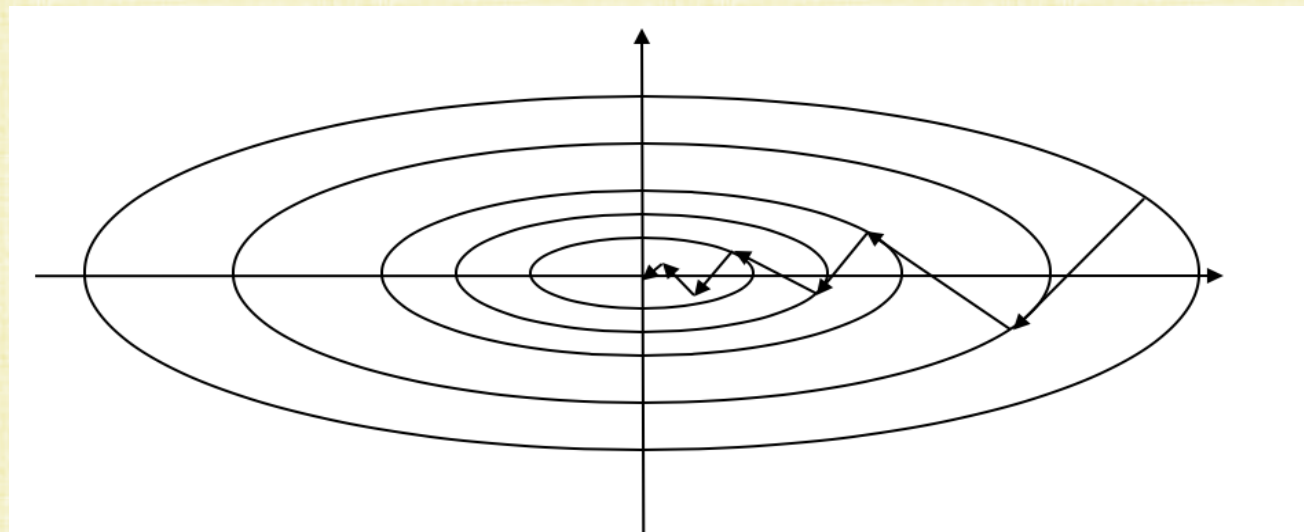
# Momentum Methods

# 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; P1[Part I - Linear Algebra (units 1-12) Ac = b]; P2[Part II - Optimization (units 13-20)]; P2 -- linearize --> P1; P1 -- line search --> P2; P2 -- Theory --> P2_1["(units 13-16) Optimization -> Nonlinear Equations -> 1D roots/minima"]; subgraph P2_2 ["Methods"]; P2_2_1["(units 17-18) Computing/Avoiding Derivatives"]; P2_2_2["(unit 19) Hack 1.0: 'I give up' H = I and J is mostly 0 (descent methods)"]; P2_2_3["(unit 20) Hack 2.0: 'It's an ODE!?' (adaptive learning rate and momentum)"]; end;
```

Path through Parameter Space

- Optimization solvers iteratively update the state variable c at each iteration
- This is often 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

- At each iteration, there is a discrete jump from one point to another
- Connecting the points with 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), the path becomes continuous

- Parameterize the continuous path/curve with a scalar t (typically called time)
 - Then, $c(t)$ represents a continuous path in parameter space
 - For each t , the corresponding $c(t)$ represents a position
 - Changing the value of t moves the position $c(t)$ along the path
 - Differentiating gives $\frac{dc}{dt}(t)$ or $c'(t)$ as a time varying velocity

Ordinary Differential Equations (ODEs)

- ODEs describe rates of change
 - For example, $\frac{dc}{dt}(t) = f(t, c(t))$ states that the parameter space velocity is $f(t, c(t))$ at the point c at time t
- “Solving” an ODE means finding a function that has the given rates of change
 - For example, find a curve $c(t)$ that has velocity $\frac{dc}{dt}(t) = f(t, c(t))$
- Consider a greedy path that 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

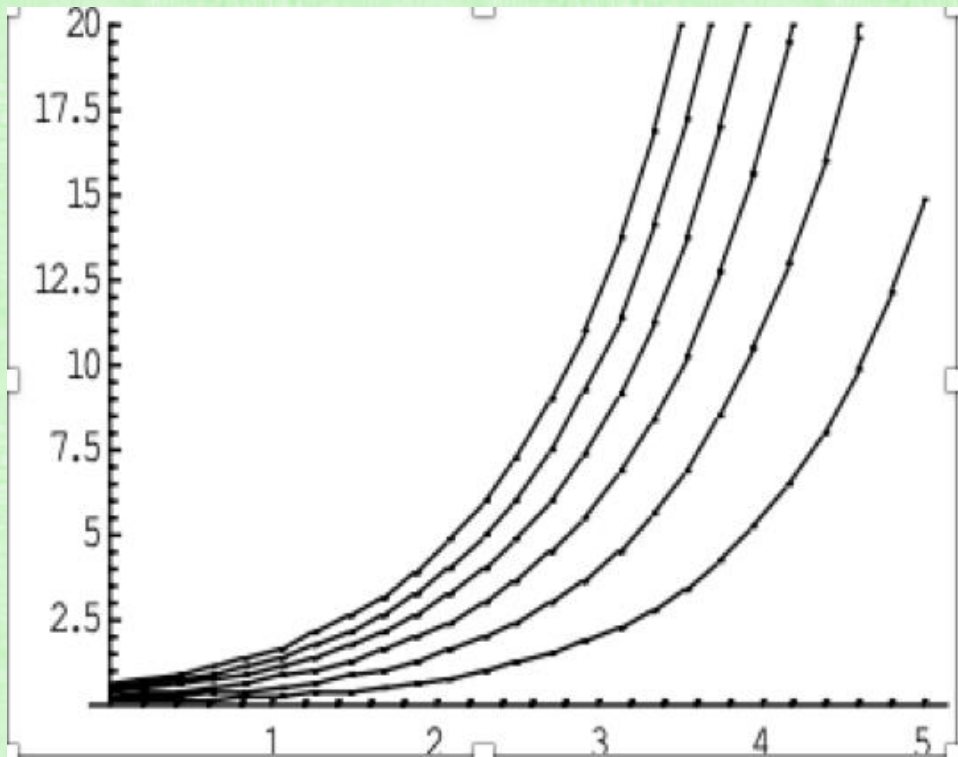
- The ODE for gradient flow is: $\frac{dc}{dt}(t) = -\nabla \hat{f}(c(t))$

- 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, i.e. based on $-\nabla \hat{f}(c(t))$
- The path given by $c(t)$ follows the direction of steepest descent

Families of Solutions

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



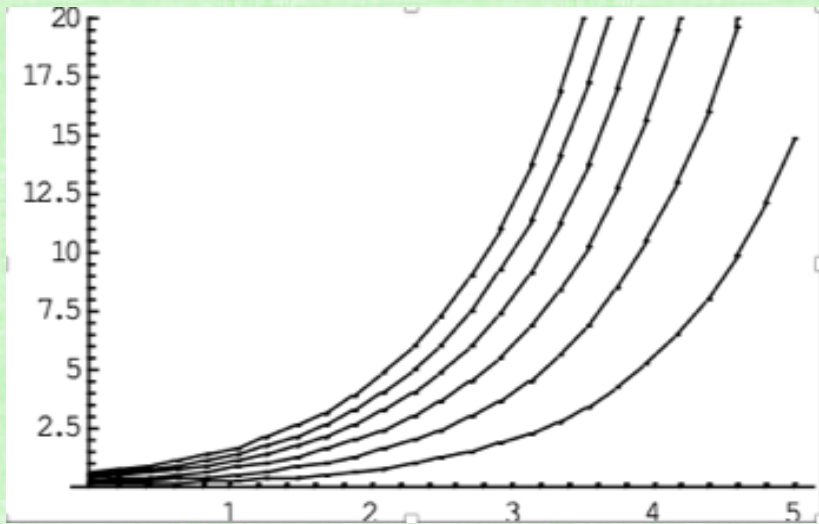
- E.g. $c' = c$ or $\frac{dc}{dt} = c$ or $\frac{dc}{c} = dt$
- $\int_{c_0}^c \frac{1}{c} dc = \int_{t_0}^t dt$ or $\ln c - \ln c_0 = t - t_0$
- $\ln \frac{c}{c_0} = t - t_0$ or $\frac{c}{c_0} = e^{t-t_0}$ or $c = c_0 e^{t-t_0}$
- Solution $c(t) = c_0 e^{t-t_0}$ depends on the initial condition $c(t_0) = c_0$
- The figure shows solutions for various values of c_0 at $t_0 = 0$

Gradient Flow

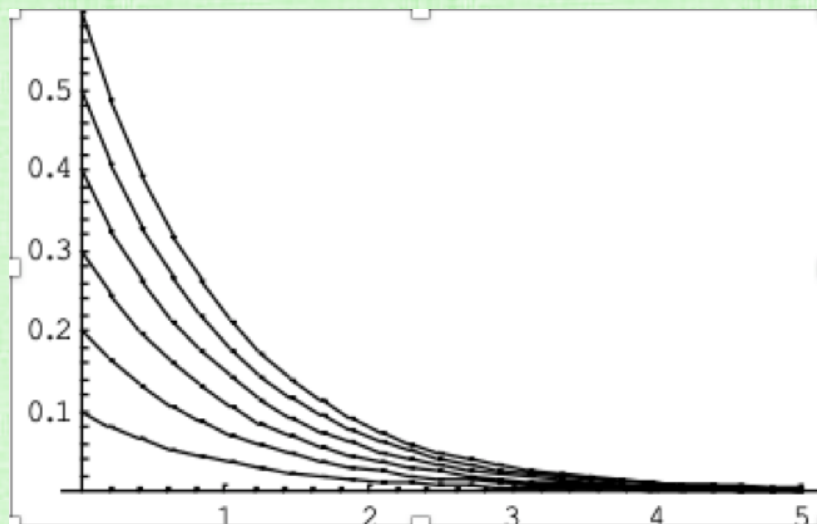
- Ansatz: following the solution trajectory in gradient flow leads to a preferred minimum of $\hat{f}(c)$
- Numerical errors cause perturbations away from this desired trajectory onto nearby trajectories (perhaps in the same family of solutions)
 - Hopefully, the perturbed trajectories stay close to 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 minima (which are presumably less preferred)

Posedness

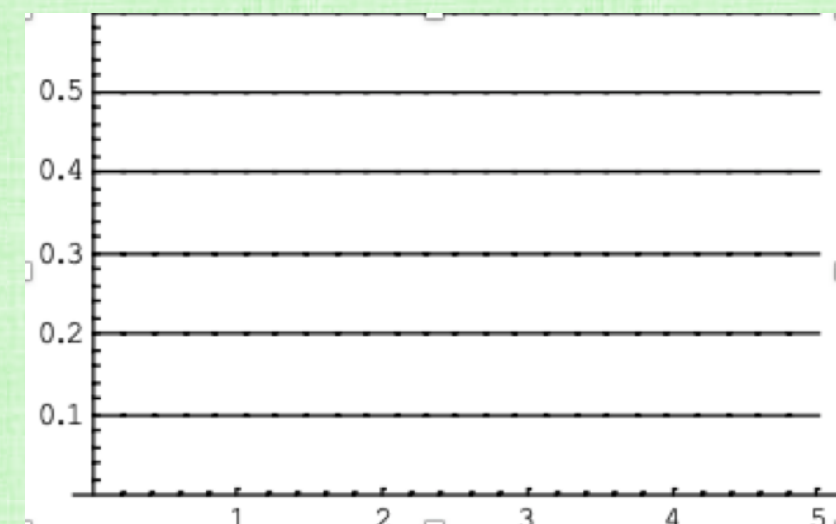
- Consider $c' = \lambda c$ with solution family $c(t) = c_0 e^{\lambda(t-t_0)}$



- $\lambda > 0$, exponential growth, ill-posed
- Small changes in initial conditions (and small numerical errors) result in large changes to the trajectory



- $\lambda < 0$, exponential decay, well-posed
- Small changes in initial conditions (and small numerical errors) are damped by the convergence of trajectories



- $\lambda = 0$, constant solution, linearly stable, mildly ill-posed
- Small changes in initial conditions (and small numerical errors) result in (slow, but cumulative) trajectory drift

Posedness for Systems

- A system of ODEs $c' = F(t, c)$ has a 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 diverge wildly
 - This can (and typically does) pollute the entire solution vector c
- Thus, all eigenvalues of $J_F(t, c(t))$ must be non-positive for all t under consideration, in order for the problem to be considered to be well-posed
 - Moreover, eigenvalues close to zero may be suspect due to numerical errors
- Ill-posedness can rapidly lead to solution family bifurcation, which can result in minima far from what one might otherwise expect

Stability and Accuracy

- Given a well-posed ODE, a solver is considered stable if it does not overflow and produce NaNs (i.e., does not veer off to $\pm\infty$ in some dimension in parameter space)
- Stability is often guaranteed via a restriction on the size of the time step Δt
 - Larger time steps lead to the method going unstable
- Given a well posed ODE, a stable solver can be analyzed for accuracy to see how well it matches known solutions
- Hopefully, stability and reasonable accuracy keep the numerical solution close to an ideal trajectory (leading to a preferred minimum)

Forward Euler Method

- Approximate $c' = f(t, c)$ via $\frac{c^{q+1} - c^q}{\Delta t} = f(t^q, c^q)$
- 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 (i.e., each iteration)
 - Getting through an $O(1)$ interval $[t_o, t_f]$ takes $\frac{t_f - t_o}{\Delta t} = O\left(\frac{1}{\Delta t}\right)$ time steps; so, the total error or global truncation error is $O(\Delta t^2) O\left(\frac{1}{\Delta t}\right) = O(\Delta t)$
 - Thus, the method is 1st order accurate
 - Recall comments on accuracy and Newton-Cotes approaches in Unit 7 Curse of Dimensionality

Runge-Kutta (RK) Methods

Taylor series can be used to (similarly) construct more accurate methods:

- **1st order:** $\frac{c^{q+1}-c^q}{\Delta t} = f(t^q, c^q)$ which is the forward Euler method
- **2nd 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 $k_2 = f(t^{q+1}, c^q + \Delta tk_1)$
- **4th 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 $k_1 = f(t^q, c^q)$, $k_2 = f\left(t^{q+\frac{1}{2}}, c^q + \frac{\Delta t}{2}k_1\right)$, $k_3 = f\left(t^{q+\frac{1}{2}}, c^q + \frac{\Delta t}{2}k_2\right)$, $k_4 = f(t^{q+1}, c^q + \Delta tk_3)$
 - Again, each term builds on the prior term in a predictor style fashion

TVD Runge-Kutta Methods

Combinations of forward Euler and averaging (since both are well-behaved)

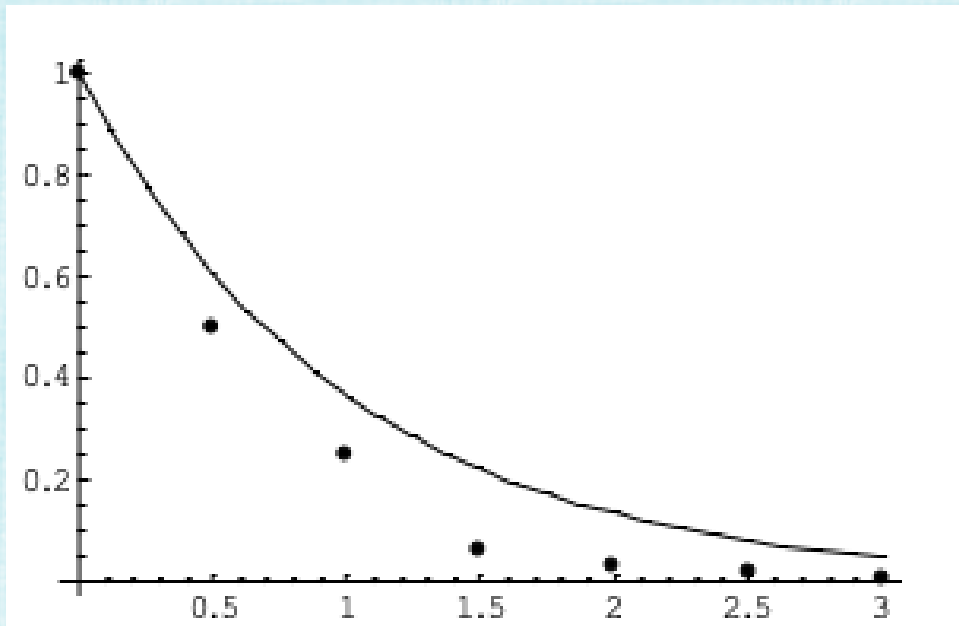
- **1st order**: same as standard RK1 and forward Euler
- **2nd order**: same as standard RK2 (also called the midpoint rule, the modified Euler method, and Heun's predictor-corrector method)
 - Take two forward Euler steps: $\frac{\hat{c}^{q+1} - c^q}{\Delta t} = f(t^q, c^q)$ and $\frac{\hat{c}^{q+2} - \hat{c}^{q+1}}{\Delta t} = f(t^{q+1}, \hat{c}^{q+1})$
 - Then, average the initial and final state: $c^{q+1} = \frac{1}{2} c^q + \frac{1}{2} \hat{c}^{q+2}$
- **3rd order**: different from the standard RK3
 - Take two forward 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: $\frac{\hat{c}^{q+\frac{3}{2}} - \hat{c}^{q+\frac{1}{2}}}{\Delta t} = f(t^{q+\frac{1}{2}}, \hat{c}^{q+\frac{1}{2}})$
 - Finally, average again: $c^{q+1} = \frac{1}{3} c^q + \frac{2}{3} \hat{c}^{q+\frac{3}{2}}$

Stability Analysis

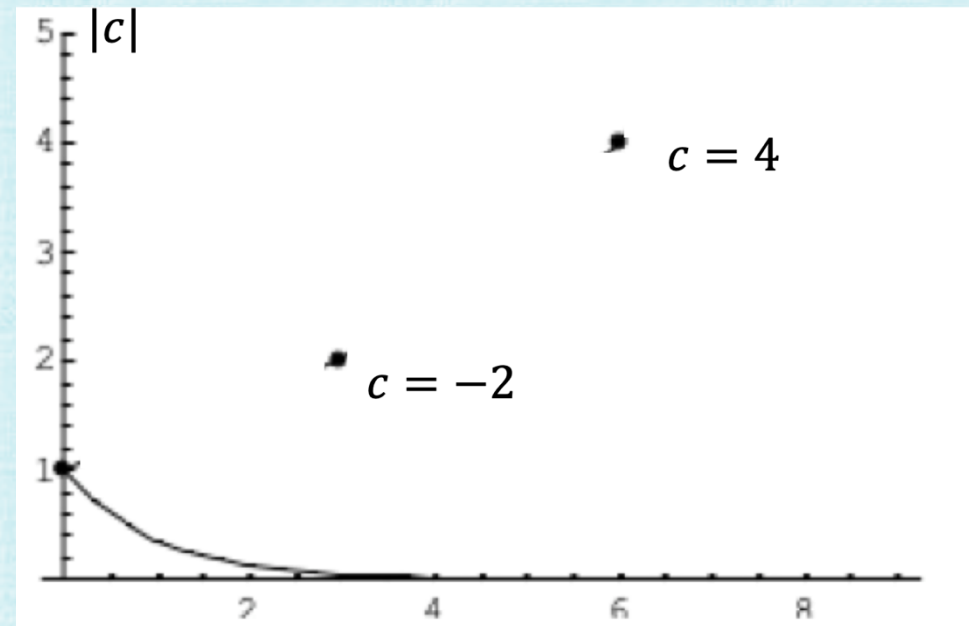
- Consider the model equation $c' = \lambda c$ with a well-posed $\lambda < 0$
 - This model equation is meant to illustrate how an eigenvalue λ of a Jacobian matrix might behave
- Forward Euler gives $c^{q+1} = c^q + \Delta t \lambda c^q = (1 + \Delta t \lambda) c^q = (1 + \Delta t \lambda)^{q+1} c^0$
 - The error shrinks and the solutions 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 $\Delta t > 0$, need $\Delta t < \frac{2}{-\lambda}$ for stability
 - This is called a time step restriction

Stability (an 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 steepest descent search direction $\Delta c^q = -\nabla \hat{f}(c^q)$
- Given this steepest descent search direction, line search would utilize a 1D root/minimization approach to determine the next iterate
- This forward Euler interpretation suggests that one may instead choose Δt according to various ODE (or other similar) considerations

Adaptive Time Stepping

- ODEs utilize either a fixed size Δt or a time varying Δt^q
 - The latter case is referred to as adaptive time stepping
- The ML community refers to the size of Δt as the learning rate
- The ML community refers to each time step as an epoch
 - 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
 - That is, each epoch allows the $-\nabla \hat{f}(c^q)$ estimates to see all the training data one time

Adaptive Learning Rates

- Adagrad maintains a separate adaptive learning rate for each parameter, and modifies them based on the past gradients computed for that parameter
 - Note: 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 (and better behaved, e.g. not getting stuck in local minima)
 - Recall that the Broyden-style methods also use a non-localized approach when they approximate the Hessian, as compared to steepest descent which only uses local information
- Unfortunately, Adagrad's learning rates monotonically decrease and often go to zero, stalling out the algorithm
- Adadelta and RMSprop decrease the effect of prior gradients (similar in spirit to L-BFGS) so that the learning rate is not driven to zero

Implicit Methods

- Allows for larger (but still stable) time steps than forward Euler and RK methods
- Implicit methods typically have either no time step restriction or a very generous one
- However, implicit methods typically require the use of a (potentially expensive) nonlinear solver in order to advance each time step
 - Sometimes, the nonlinear solver requires more computational effort than all the smaller (and simpler) forward Euler and/or RK time steps combined (making it less efficient)
- The large time steps often/typically lead to overly damped solutions (or unwanted oscillations)
 - For some problems, this damping can act as a desired regularization

Backward (Implicit) Euler

- $\frac{c^{q+1} - c^q}{\Delta t} = f(t^{q+1}, c^{q+1})$ is 1st 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 Δt (assuming $\lambda < 0$)
- Typically need to solve a nonlinear equation/system to find c^{q+1}
 - This is often 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 (the $c_t \rightarrow 0$ solution)
 - But, overly damping leads to the steady state too quickly, which is especially undesirable when higher frequencies are important
 - Great for so-called stiff problems, defined as problems with unimportant high frequencies (thus, overly damping them is fine)

Implicit Stochastic Gradient Descent (ISGD)

- Can be used to overcome instabilities caused by using too large (and thus unstable) 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 (training) data point 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 2nd 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 Δt (assuming $\lambda < 0$)
- Typically need to solve a nonlinear equation/system to find c^{q+1}
 - This is often 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., for $c' = \lambda c$, this is $c^{q+1} = -c^q$ which is oscillatory
 - For the general $c' = f(t, c)$ case, this is $(c')^{q+1} = -(c')^q$ indicating that the derivative is changing sign every iteration (causing oscillations)

Momentum

- Optimization methods often struggle when they are too local
 - Broyden-style methods help alleviate this with their lagged rank one updates
 - Adaptive learning rates help alleviate this by being based on a time history
 - Momentum methods also aim to address this
- Momentum methods derive 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 and velocity

Newton's Second Law

- Kinematics describe position $X(t)$, velocity $V(t)$, acceleration $A(t)$ as functions of time 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
 - $\frac{d^2X}{dt^2}(t) = X''(t) = V'(t) = A(t) = \frac{F(t)}{M}$
- 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 (as we have stated before)**
 - This makes the optimization 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:

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

- 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 is 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 nonlinear 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 Δt and Δt^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: $c^{q+1} = c^q + \alpha v^q - \Delta t \nabla \hat{f}(c^q)$
 - Rewriting 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

- A predictor-corrector approach similar to 2nd order Runge- Kutta
- (Predictor) A forward Euler step $\hat{c}^{q+1} = c^q + \Delta t \hat{v}^{q+1}$ uses velocity $\hat{v}^{q+1} = \frac{\alpha}{\Delta t} v^q$
 - On the last slide, the velocity was $v^{q+1} = \frac{\alpha}{\Delta t} v^q - \nabla \hat{f}(c^q)$
 - The current gradient $-\nabla \hat{f}(c^q)$ is ignored in the predictor step
 - The predictor step simplifies to $\hat{c}^{q+1} = c^q + \alpha v^q$
- (Corrector) Evaluate the gradient at the predicted \hat{c}^{q+1} location, and use it in “The” ML Momentum method: $c^{q+1} = c^q + \Delta t v^{q+1}$ with $v^{q+1} = \frac{\alpha}{\Delta t} v^q - \nabla \hat{f}(\hat{c}^{q+1})$
 - This simplifies to $c^{q+1} = c^q + \alpha v^q - \Delta t \nabla \hat{f}(\hat{c}^{q+1})$
 - Once again, there is an inconsistent velocity update $v^{q+1} = \frac{\alpha}{\Delta t} v^q - \nabla \hat{f}(\hat{c}^{q+1})$

Addressing the Inconsistency

- Numerical ODE theory dictates (via consistency with the Taylor expansion) that the correct solution (or 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} equal to either c^q or \hat{c}^{q+1}) is problematic
- Revert to where liberties were taken 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 $v^{q+1} = \frac{\alpha}{\Delta t} v^q - \Delta t \hat{\beta} \nabla \hat{f}(\tilde{c})$
- Set $\alpha = \Delta t$ to obtain 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 Δt , so that they accumulate to the same $O(1)$ net effect in $O(1)$ time (regardless of Δt)

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^∞ 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 believe that Adam might converge quicker than SGD w/momentum, but sometimes quicker to a worse solution; so, some practitioners prefer SGD
 - **There is still a lot to do!**

Adam: A method for stochastic optimization

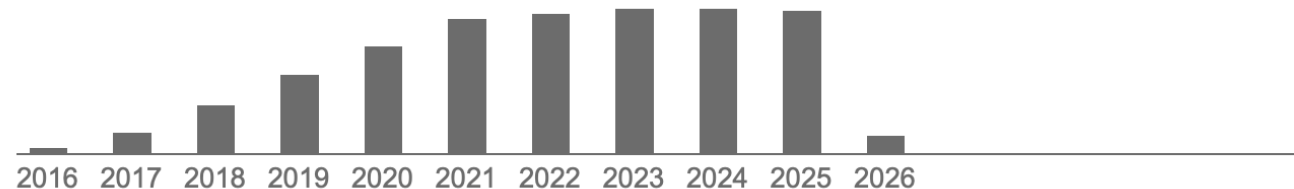
Authors Diederik Kingma, Jimmy Ba

Publication date 2015

Conference International Conference on Learning Representations

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 245004](#)



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 Newton's 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)
 - $X^{q+1} = X^q + \Delta t V^q + \frac{\Delta t^2}{2} (A^q + A^{q+1})$ 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 Differencing

- $X^{q+1} = X^q + \Delta t V^q + \frac{\Delta t^2}{2} A^q$ 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
- Σ_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
- ϕ are basis functions
- θ are pose parameters, φ 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, ψ 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

- λ eigenvalue (scalar)
- v eigenvector, u right eigenvector (both column vectors)
- α 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
- Σ diagonal (not necessarily square, potentially has zeros on the diagonal)
- σ_k singular values (diagonal entries of Σ)

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
- Λ 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)
- ϵ 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
- α, β 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
- ϕ 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 derivative
- J Jacobian matrix of all first partial derivatives
- F' is the Jacobian of F
- ∇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 λ^q are A and λ raised to the q power

Unit 12: Regularization

- ϵ 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
- Θ is the angle between two vectors
- θ are pose parameters, φ 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
- η 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)
- Δ finite size difference
- α, β are scalars with $\beta \in [0,1)$
- g scalar function (that determines the line search parameter α)

Unit 15: Root Finding

- \hat{g} is a modified g
- t is search parameter in 1D, replacing α
- 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
- δ scalar (interval size)
- $\lambda \in (0, .5)$ is a scalar
- $\tau \in (0,1)$ is a scalar
- H_F is a 3rd order tensor of 2nd derivatives of F
- $OMG_{\hat{f}}$ is a 3rd order tensor of 3rd 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
- \hat{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_0, t_f initial and final time
- Δt 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
- λ 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