



AI²Life

ADVANCED
SCHOOL AI

Mathematical Foundations for AI

Samuele Carli

Who Am I?

Technologist & Computational Scientist *Bridging AI, Research, and Entrepreneurship*

Current Focus

- ▶ **Founder & CTO, Entersys** – Empowering SMEs with open-source and AI-driven solutions.
- ▶ **AI Consultant, National Research Council (CNR)** – Applying ML for early detection of neurodegenerative diseases.
- ▶ **Teacher, Advanced School in Artificial Intelligence (AS-AI)** – Programming, Math and Numerical Analysis courses.

Core Expertise

Computational Science

Neuroscience Modeling
Numerical Methods
High-Performance Computing

Software Architecture

Full-Stack & Distributed Systems
Python, Rust, C++
Linux & Container Orchestration

Applied AI & ML

Data Analysis & Simulation
Knowledge Bases & Expert Sys
Process Automation

Key Highlights

- ▶ **CERN Fellow:** Led the development of a large-scale Machine Learning author disambiguation system for the INSPIRE-HEP digital library.
- ▶ **Published Researcher:** Author of multiple papers on computational models for brain disorders, including Parkinson's Disease.
- ▶ **Open-Source Advocate:** Long-standing passion for building and promoting sustainable, community-driven technology.

Contact: carlisamuele@csspace.net - www.csspace.net

Mathematical Foundations for AI

Samuele Carli

AI2Life - AS-AI - Entersys s.r.l.
carlisamuele@csspace.net - www.csspace.net

Outline I

1. Introduction
2. Linear Algebra
 - Vectors
 - Matrices
 - Advanced Algebra
3. Differential Calculus
 - Limits
 - Differentiability
 - Higher order
 - Partial Derivatives
 - Automatic Differentiation
 - Applications
4. Cost Functions in Machine Learning
5. Numerical aspects
 - Errors
 - Finite Arithmetic
 - Finding Roots
6. Linear Algebra Recap
7. Function Approximation
 - Polynomial interpolation
 - Spline interpolation

Outline II

- Least squares approximation

8. Optimization

- Definitions
- Optimization kinds
- Convex problems
- Multivariate Calculus Recap
- Optimization strategies overview
- Gradient Descent
- Newton method
- Trust region
- Genetic algorithms
- Differential Evolution

9. Literature

Course Overview

1. **Linear Algebra** - The language of data representation and transformation
2. **Differential Calculus** - The engine of optimization and learning
3. **Cost Functions** - The mathematics of measuring model performance
4. **Optimization** - The algorithms that make learning possible
5. **Numerical Analysis** - The practical aspects of running Optimization algorithms using discrete algebra

Why Mathematics Matters for AI

The AI-Mathematics Connection

Every AI system is fundamentally a mathematical system:

- ▶ **Data Representation:** Images, text, and sensor data become vectors and matrices
- ▶ **Learning Algorithms:** Neural networks, SVMs, and clustering rely on linear algebra
- ▶ **Training Process:** Backpropagation and optimization use calculus
- ▶ **Performance Evaluation:** Loss functions and metrics are mathematical functions
- ▶ **Model Interpretation:** Understanding AI decisions requires mathematical analysis

Without Mathematics, AI Would Be Impossible

Mathematics provides the language, tools, and framework that make artificial intelligence possible.

Linear Algebra: The Language of AI

Vectors and Matrices

- ▶ Data representation in high-dimensional space
- ▶ Neural network weights and activations
- ▶ Feature vectors in machine learning
- ▶ Word embeddings in natural language processing

Real AI Applications

- ▶ Google PageRank (eigenvectors)
- ▶ Netflix recommendations (SVD)
- ▶ Face recognition (PCA)
- ▶ Transformer architectures (attention matrices)

Advanced Operations

- ▶ **Eigenvalues:** Principal Component Analysis (PCA)
- ▶ **SVD:** Dimensionality reduction, recommendation systems
- ▶ **Matrix operations:** Deep learning computations
- ▶ **Decompositions:** Understanding data structure

Calculus and Optimization: The Engine of Learning

Differential Calculus

- ▶ **Gradients:** Direction of steepest descent in loss functions
- ▶ **Chain Rule:** Backpropagation algorithm
- ▶ **Hessian:** Second-order optimization methods
- ▶ **Partial Derivatives:** Sensitivity analysis

Optimization Methods

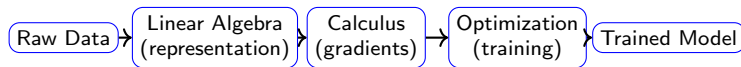
- ▶ **Gradient Descent:** Core training algorithm
- ▶ **Newton's Method:** Second-order optimization
- ▶ **BFGS:** Quasi-Newton methods
- ▶ **Stochastic Methods:** Handling large datasets

AI Training Process

1. Define loss function (calculus)
2. Compute gradients (differential calculus)
3. Update parameters (optimization)
4. Repeat until convergence

From Theory to Practice: The Complete AI Pipeline

How All Mathematics Works Together



Numerical Considerations

- ▶ **Floating-point precision:** Affects training stability
- ▶ **Numerical optimization:** Avoiding computational pitfalls
- ▶ **Function approximation:** Handling real-world complexity
- ▶ **Error analysis:** Understanding model limitations

The Goal

Transform mathematical understanding into working AI systems that solve real-world problems.

Linear Algebra

Why Linear Algebra Matters for AI

- ▶ **Data Representation:** Everything becomes vectors and matrices
- ▶ **Feature Engineering:** Linear transformations create new features
- ▶ **Dimensionality Reduction:** PCA, SVD for compression
- ▶ **Neural Networks:** Weight matrices and transformations
- ▶ **Computer Vision:** Image processing as matrix operations
- ▶ **Natural Language:** Word embeddings as vectors

Vector Definition and Notation

A vector $v \in \mathbb{R}^n$ is an ordered n-tuple:

$$v = [v_1, v_2, \dots, v_n]^T$$

where $v_i \in \mathbb{R}$ for $i = 1, 2, \dots, n$

Geometric interpretation:

- ▶ A point in n-dimensional space
- ▶ A directed line from origin to that point

Formal Vector Properties

A vector space V over field \mathbb{R} satisfies:

Closure axioms:

- ▶ $\forall u, v \in V : u + v \in V$
- ▶ $\forall \alpha \in \mathbb{R}, v \in V : \alpha v \in V$

Addition axioms:

- ▶ Commutative: $u + v = v + u$
- ▶ Associative: $(u + v) + w = u + (v + w)$
- ▶ Identity: $\exists 0 \in V$ such that $v + 0 = v$
- ▶ Inverse: $\forall v \in V, \exists -v \in V$ such that $v + (-v) = 0$

Scalar Multiplication Axioms

Distributive properties:

▶ $\alpha(u + v) = \alpha u + \alpha v$

▶ $(\alpha + \beta)v = \alpha v + \beta v$

Associative property: $\alpha(\beta v) = (\alpha\beta)v$

Identity property: $1 \cdot v = v$

Vector Representation Forms

Row vector: $v = [v_1, v_2, \dots, v_n]$

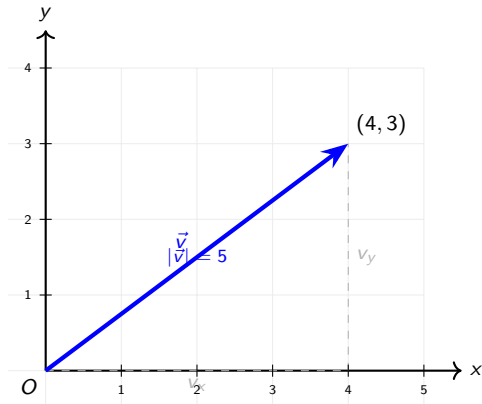
Column vector: $v = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$

Transpose operation: $[v_1, v_2, \dots, v_n]^T = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$

In AI applications, we typically use column vectors for mathematical operations.

Visualizing Vectors in Different Dimensions

2D Vector: $[x, y]$ - Point on plane, arrow from origin



3D Vector: $[x, y, z]$ - Point in space, arrow in 3D

n-D Vector: $[x_1, x_2, \dots, x_n]$ - Abstract point in feature space

Higher-Dimensional Intuition

While we can't visualize dimensions beyond 3D, we can reason about them:

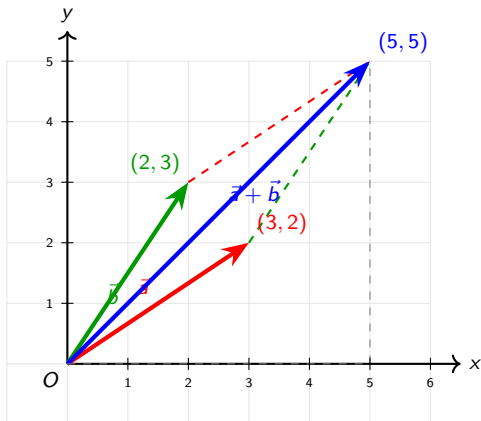
- ▶ **Each dimension represents a feature:** In ML, dimensions might represent age, income, height, etc.
- ▶ **Distance and similarity still work:** The Pythagorean theorem extends to n -dimensions
- ▶ **Geometric intuition transfers:** Lines, planes, and angles generalize to hyperplanes and hyperangles

Vector Addition

For vectors $v = [v_1, v_2, \dots, v_n]^T$ and $w = [w_1, w_2, \dots, w_n]^T$:

$$v + w = [v_1 + w_1, v_2 + w_2, \dots, v_n + w_n]^T$$

Geometric interpretation: Tip-to-tail addition (parallelogram rule)



Vector Addition Properties

Commutative: $v + w = w + v$ - Order doesn't matter for vector addition

Associative: $(v + w) + u = v + (w + u)$ - Grouping doesn't affect the result

Zero vector: $v + 0 = v$ where $0 = [0, 0, \dots, 0]^T$

Additive inverse: $v + (-v) = 0$

Scalar Multiplication

For scalar $\alpha \in \mathbb{R}$ and vector v :

$$\alpha v = [\alpha v_1, \alpha v_2, \dots, \alpha v_n]^T$$

Geometric interpretation: - Scales vector length by $|\alpha|$ - Reverses direction if $\alpha < 0$

Special cases: - $\alpha = 0$: Results in zero vector - $\alpha = 1$: Identity operation - $\alpha = -1$: Vector reversal

Scalar Multiplication Properties

Distributive over vector addition: $\alpha(v + w) = \alpha v + \alpha w$

Distributive over scalar addition: $(\alpha + \beta)v = \alpha v + \beta v$

Associative: $\alpha(\beta v) = (\alpha\beta)v$

Identity: $1 \cdot v = v$

These properties ensure linearity in operations.

Linear Combinations

Given vectors v_1, v_2, \dots, v_k and scalars $\alpha_1, \alpha_2, \dots, \alpha_k$:

Linear combination: $\alpha_1 v_1 + \alpha_2 v_2 + \dots + \alpha_k v_k$

Example: In \mathbb{R}^3 , any vector can be written as:

$$v = \alpha \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + \beta \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} + \gamma \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Span: The Reachable Space

Definition: The span of vectors v_1, v_2, \dots, v_k is the set of all possible linear combinations:

$$\text{Span}\{v_1, v_2, \dots, v_k\} = \{\alpha_1 v_1 + \alpha_2 v_2 + \dots + \alpha_k v_k \mid \alpha_i \in \mathbb{R}\}$$

Geometric meaning: All points reachable by combining the basis vectors

Examples:

- ▶ Span of single non-zero vector: a line through origin
- ▶ Span of two independent vectors in \mathbb{R}^3 : a plane through origin

Linear Independence

Vectors v_1, v_2, \dots, v_k are linearly independent if:

$$\alpha_1 v_1 + \alpha_2 v_2 + \dots + \alpha_k v_k = 0 \iff \alpha_1 = \alpha_2 = \dots = \alpha_k = 0$$

Alternative formulation: No vector can be written as a linear combination of the others

Geometric interpretation: Each vector adds a new “direction” to the space

Testing Linear Independence

Method 1: Solve the system

- ▶ Set up equation: $\alpha_1 v_1 + \alpha_2 v_2 + \dots + \alpha_k v_k = 0$
- ▶ If only solution is $\alpha_1 = \alpha_2 = \dots + \alpha_k = 0$, vectors are independent

Method 2: Matrix rank

- ▶ Form matrix $A = [v_1 \ v_2 \ \dots \ v_k]$
- ▶ If $\text{rank}(A) = k$, vectors are independent - If $\text{rank}(A) < k$, vectors are dependent

Linear Dependence Examples

Dependent vectors: $[1, 2, 3]$, $[2, 4, 6]$, $[1, 0, 0]$ - Second vector is $2 \times$ first, so dependent

Independent vectors: $[1, 0, 0]$, $[0, 1, 0]$, $[0, 0, 1]$ - No vector can be made from others

Test case: $[1, 1, 0]$, $[0, 1, 1]$, $[1, 0, 1]$ - Try to find non-trivial solution to $\alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 = 0$

Vector Space Examples

\mathbb{R}^n : The canonical n -dimensional vector space

Function spaces: All polynomials of degree $\leq n$ form a vector space

Solution spaces: Solutions to homogeneous linear equations

Non-examples: The set of all vectors with first component $= 1$ (not closed under addition)

Dot Product Definition

For vectors $v = [v_1, v_2, \dots, v_n]^T$ and $w = [w_1, w_2, \dots, w_n]^T$:

$$v \cdot w = \sum_{i=1}^n v_i w_i = v^T w$$

Matrix formulation: $v \cdot w = v^T w$ (treating vectors as column matrices)

Example: $[1, 2, 3] \cdot [4, 5, 6] = 1 \cdot 4 + 2 \cdot 5 + 3 \cdot 6 = 32$

Dot Product Properties

Commutative: $v \cdot w = w \cdot v$

Distributive: $v \cdot (w + u) = v \cdot w + v \cdot u$

Linear: $\alpha(v \cdot w) = (\alpha v) \cdot w = v \cdot (\alpha w)$

Positive definite:

► $v \cdot v \geq 0$ for all v

► $v \cdot v = 0 \iff v = 0$

Geometric Interpretation of Dot Product

Relation to angle:

$$v \cdot w = \|v\| \|w\| \cos(\theta)$$

where θ is the angle between vectors

Special cases:

- ▶ $\theta = 0^\circ$: $v \cdot w = \|v\| \|w\|$ (parallel)
- ▶ $\theta = 90^\circ$: $v \cdot w = 0$ (orthogonal)
- ▶ $\theta = 180^\circ$: $v \cdot w = -\|v\| \|w\|$ (anti-parallel)

Vector Norms: Measuring Length

L2 norm (Euclidean):

$$\|v\|_2 = \sqrt{v \cdot v} = \sqrt{\sum_{i=1}^n v_i^2}$$

L1 norm (Manhattan):

$$\|v\|_1 = \sum_{i=1}^n |v_i|$$

L ∞ norm (Maximum):

$$\|v\|_\infty = \max_i |v_i|$$

L p norm (General):

$$\|v\|_p = \left(\sum_{i=1}^n |v_i|^p \right)^{1/p}$$

Norm Properties and Unit Vectors

Properties of all norms:

- ▶ $\|v\| \geq 0$ and $\|v\| = 0 \iff v = 0$
- ▶ $\|\alpha v\| = |\alpha| \cdot \|v\|$ (homogeneity)
- ▶ $\|v + w\| \leq \|v\| + \|w\|$ (triangle inequality)

Unit vector: A vector with norm = 1

Can normalize any non-zero vector: $\hat{v} = \frac{v}{\|v\|}$

Cauchy-Schwarz Inequality

For any vectors v and w : $\|v \cdot w\| \leq \|v\| \cdot \|w\|$

Equality cases:

- ▶ Equality holds when vectors are linearly dependent
- ▶ $v = 0$ or $w = 0$ (trivial case)
- ▶ $w = \alpha v$ for some scalar α

Orthogonality and Projections

Orthogonal vectors: $v \cdot w = 0$

Orthogonal projection: Component of v onto w :

$$\text{proj}_w(v) = \frac{v \cdot w}{\|w\|^2} w$$

Projection length:

$$\|\text{proj}_w(v)\| = \frac{|v \cdot w|}{\|w\|}$$

Residual component: $v - \text{proj}_w(v)$ (orthogonal to w)

Universal Address Analogy

Think of a vector as a precise GPS coordinate in feature space:

- ▶ **2D vector** $[x, y]$ = street address on city map
- ▶ **3D vector** $[x, y, z]$ = location with building floor
- ▶ **n-D vector** = address in a city with n dimensions

Each dimension represents a different feature or characteristic

Operations as navigation:

- ▶ Vector addition = moving to a new location
- ▶ Scalar multiplication = scaling your movement
- ▶ Dot product = checking if you're going in the same direction

Feature Vector Example: Music

A song represented as: [danceability, energy, valence, tempo, key]

Example: Bohemian Rhapsody \approx [0.3, 0.8, 0.4, 0.7, 0.5]

Operations:

- ▶ **Adding vectors** = creating a “playlist average”
- ▶ **Scalar multiplication** = making song “more intense”
- ▶ **Dot product** = measuring similarity between songs

Real application: Spotify’s recommendation system uses such feature vectors to find similar music.

LEGO Analogy for Span

If you have two LEGO pieces (vectors):

- ▶ **Parallel pieces:** You can only build in one direction (1D span)
- ▶ **Different directions:** You can build any flat structure (2D span)
- ▶ **Three non-parallel pieces:** You can build 3D structures (3D span)

Linear independence = no piece is a combination of others

Basis = the minimum set of pieces needed to build everything in your space

Concrete AI Applications: Word Embeddings (NLP)

Mathematical foundation: Words mapped to vectors in high-dimensional space (typically 50-300 dimensions)

Distance metrics measure semantic similarity:

▶ Cosine similarity: $\frac{v \cdot w}{\|v\| \|w\|}$

▶ Euclidean distance: $\|v - w\|_2$

Classic example: $\text{vector}(\text{'king'}) - \text{vector}(\text{'man'}) + \text{vector}(\text{'woman'}) \approx \text{vector}(\text{'queen'})$

This works because the embedding space captures linguistic relationships through geometric relationships between vectors.

Concrete AI Applications: Tabular Data Representation

Customer data row: [age, income, spending_score, last_purchase_days] \rightarrow Vector in 4-dimensional customer space

Applications:

- ▶ **Clustering algorithms:** Find similar customers using distance metrics
- ▶ **Feature engineering:** Becomes vector transformations
- ▶ **Classification:** Decision boundaries in feature space
- ▶ **Anomaly detection:** Identify outliers as vectors far from cluster centers

Concrete AI Applications: Regularization in Machine Learning

L1 regularization (Lasso): Adds $\lambda \|w\|_1$ to loss function

- ▶ Creates sparse solutions, automatic feature selection
- ▶ Useful when you suspect many features are irrelevant

L2 regularization (Ridge): Adds $\lambda \|w\|_2^2$ to loss function

- ▶ Distributes weight across features, prevents overfitting
- ▶ Useful when all features are potentially relevant

Elastic Net: Combines L1 and L2 regularization

Different norms lead to different optimization landscapes and solution properties

Formal Definition of Vector Spaces

A **vector space** over a field \mathbb{F} (typically \mathbb{R}) is a set V equipped with two operations:

► **Vector addition:** $+: V \times V \rightarrow V$

1. **Closure:** $\forall u, v \in V, u + v \in V$
2. **Associativity:** $\forall u, v, w \in V, (u + v) + w = u + (v + w)$
3. **Commutativity:** $\forall u, v \in V, u + v = v + u$
4. **Identity:** $\exists 0 \in V$ such that $\forall v \in V, v + 0 = v$
5. **Inverse:** $\forall v \in V, \exists -v \in V$ such that $v + (-v) = 0$

► **Scalar multiplication:** $\cdot: \mathbb{F} \times V \rightarrow V$

1. **Closure:** $\forall \alpha \in \mathbb{F}, \forall v \in V, \alpha v \in V$
2. **Distributivity of scalar over vector addition:** $\forall \alpha \in \mathbb{F}, \forall u, v \in V, \alpha(u + v) = \alpha u + \alpha v$
3. **Distributivity of scalar addition over scalar multiplication:** $\forall \alpha, \beta \in \mathbb{F}, \forall v \in V, (\alpha + \beta)v = \alpha v + \beta v$
4. **Compatibility of scalar multiplication:** $\forall \alpha, \beta \in \mathbb{F}, \forall v \in V, \alpha(\beta v) = (\alpha\beta)v$
5. **Identity:** $\forall v \in V, 1 \cdot v = v$ where $1 \in \mathbb{F}$

Proof: Uniqueness of Zero Vector

Theorem: The zero vector in a vector space is unique.

Proof: Suppose 0 and $0'$ are both zero vectors. Then:

▶ Since 0 is a zero vector: $0' + 0 = 0'$

▶ Since $0'$ is a zero vector: $0' + 0 = 0$

\implies Therefore: $0' = 0' + 0 = 0$ This proves uniqueness.

Corollary: The additive inverse of each vector is also unique. **Proof:** Suppose v has two inverses w and w' . Then: $w = w + 0 = w + (v + w') = (w + v) + w' = 0 + w' = w'$

Subspaces: Vector Spaces Within Vector Spaces

A **subspace** W of a vector space V is a subset $W \subseteq V$ that is itself a vector space under the same operations.

Subspace Test: $W \subseteq V$ is a subspace if and only if:

1. $0 \in W$ (contains zero vector)
2. **Closure under addition:** $\forall u, v \in W, u + v \in W$
3. **Closure under scalar multiplication:** $\forall \alpha \in \mathbb{F}, \forall v \in W, \alpha v \in W$

Examples of Subspaces:

- ▶ The set of all vectors in \mathbb{R}^3 with $z = 0$ (the xy-plane)
- ▶ The solution space of homogeneous linear equations $Ax = 0$
- ▶ The span of any set of vectors

Linear Independence: Formal Characterization

Theorem: Vectors v_1, v_2, \dots, v_k are linearly dependent if and only if at least one vector can be written as a linear combination of the others.

Proof (\Rightarrow): If dependent, then $\exists \alpha_1, \dots, \alpha_k$ not all zero such that $\sum \alpha_i v_i = 0$. Let $\alpha_j \neq 0$.
Then: $v_j = -\frac{1}{\alpha_j} \sum_{i \neq j} \alpha_i v_i$

Proof (\Leftarrow): If $v_j = \sum_{i \neq j} \beta_i v_i$, then: $v_j - \sum_{i \neq j} \beta_i v_i = 0$, a non-trivial linear combination.

Corollary: In \mathbb{R}^n , any set of more than n vectors must be linearly dependent.

Basis and Dimension: Formal Definitions

A **basis** for a vector space V is a set of vectors that is:

1. **Linearly independent**
2. **Spans** V (every vector in V can be written as a linear combination)

Theorem: Every vector space has a basis (assuming Axiom of Choice).

Dimension: If a vector space has a finite basis, then all bases have the same cardinality. This common size is called the **dimension** of V . **Examples:**

- ▶ \mathbb{R}^n has dimension n (standard basis: e_1, e_2, \dots, e_n)
- ▶ The set of $m \times n$ matrices has dimension mn
- ▶ The space of polynomials of degree $\leq n$ has dimension $n + 1$

Coordinate Systems and Change of Basis

Every vector $v \in V$ can be uniquely represented relative to a basis $B = \{b_1, \dots, b_n\}$:
$$v = \alpha_1 b_1 + \alpha_2 b_2 + \dots + \alpha_n b_n$$

The coefficients $[\alpha_1, \alpha_2, \dots, \alpha_n]^T$ are called the **coordinates** of v relative to basis B .

Change of Basis: If we have two bases B and B' , there exists an invertible matrix P (the change-of-basis matrix) such that: $[v]_{B'} = P[v]_B$

This is fundamental in many AI applications, particularly in dimensionality reduction.

Dot Product: Algebraic Properties Proofs

Theorem: The dot product is bilinear:

1. $(u + v) \cdot w = u \cdot w + v \cdot w$
2. $u \cdot (v + w) = u \cdot v + u \cdot w$
3. $(\alpha u) \cdot v = \alpha(u \cdot v) = u \cdot (\alpha v)$

Proof of (1): $(u + v) \cdot w = \sum (u_i + v_i)w_i = \sum u_i w_i + \sum v_i w_i = u \cdot w + v \cdot w$

Theorem: The dot product is positive definite:

- ▶ $v \cdot v \geq 0$
- ▶ $v \cdot v = 0 \iff v = 0$

Proof: $v \cdot v = \sum v_i^2 \geq 0$, and equals 0 only if each $v_i = 0$.

Cauchy-Schwarz Inequality: Detailed Proof

Theorem: For any vectors $u, v \in \mathbb{R}^n$: $|u \cdot v| \leq \|u\| \|v\|$

Proof: Consider the quadratic function in t :

$$f(t) = \|u + tv\|^2 = (u + tv) \cdot (u + tv) = \|u\|^2 + 2t(u \cdot v) + t^2\|v\|^2$$

Since $f(t) \geq 0$ for all t , the discriminant must be ≤ 0 : $(2u \cdot v)^2 - 4\|u\|^2\|v\|^2 \leq 0$

Thus: $(u \cdot v)^2 \leq \|u\|^2\|v\|^2$, so $|u \cdot v| \leq \|u\| \|v\|$

Equality case: Occurs when u and v are linearly dependent.

Triangle Inequality: Proof and Geometric Interpretation

Theorem: For any vectors $u, v \in \mathbb{R}^n$: $\|u + v\| \leq \|u\| + \|v\|$

Proof: Using Cauchy-Schwarz: $\|u + v\|^2 = (u + v) \cdot (u + v) = \|u\|^2 + 2(u \cdot v) + \|v\|^2$
 $\leq \|u\|^2 + 2\|u\|\|v\| + \|v\|^2 = (\|u\| + \|v\|)^2$

Taking square roots gives the result.

Geometric Interpretation: In any triangle, the length of one side is less than or equal to the sum of the lengths of the other two sides.

Orthogonal Projection: Theoretical Foundation

Theorem: For any vector v and non-zero vector w , the orthogonal projection of v onto w is given by:

$$\text{proj}_w(v) = \frac{v \cdot w}{\|w\|^2} w$$

Proof: We want to find α such that $(v - \alpha w) \cdot w = 0$ (orthogonality condition).

Solving: $v \cdot w - \alpha(w \cdot w) = 0 \Rightarrow \alpha = \frac{v \cdot w}{\|w\|^2}$

Properties:

- ▶ $\text{proj}_w(v)$ is parallel to w
- ▶ $v - \text{proj}_w(v)$ is orthogonal to w
- ▶ The projection minimizes the distance from v to the line spanned by w

Gram-Schmidt Orthogonalization Process

Given linearly independent vectors v_1, v_2, \dots, v_k , we can construct an orthogonal set u_1, u_2, \dots, u_k :

1. $u_1 = v_1$
2. $u_2 = v_2 - \text{proj}_{u_1}(v_2)$
3. $u_3 = v_3 - \text{proj}_{u_1}(v_3) - \text{proj}_{u_2}(v_3)$
4. Continue similarly: $u_k = v_k - \sum_{i=1}^{k-1} \text{proj}_{u_i}(v_k)$

Then normalize: $e_i = \frac{u_i}{\|u_i\|}$ to get an orthonormal basis.

This process is fundamental in QR decomposition and many numerical algorithms.

Advanced: Dual Spaces and Linear Functionals

For any vector space V , the **dual space** V^* consists of all linear functionals $f : V \rightarrow \mathbb{R}$.

Theorem: For finite-dimensional V , V^* is isomorphic to V .

Construction: Given a basis $\{e_1, \dots, e_n\}$ of V , the dual basis $\{e_1^*, \dots, e_n^*\}$ of V^* is defined by:

$e_j^*(e_i) = \delta_{ij}$ (Kronecker delta $\delta_{ij} = [i = j]$)

Connection to Dot Product: In \mathbb{R}^n , every linear functional f can be represented as $f(v) = w \cdot v$ for some unique $w \in \mathbb{R}^n$.

This concept is crucial in optimization and machine learning.

Advanced: Norm Equivalence in Finite Dimensions

Theorem: All norms on a finite-dimensional vector space are equivalent.

Specifically, for any two norms $\|\cdot\|_a$ and $\|\cdot\|_b$ on \mathbb{R}^n , there exist constants $c, C > 0$ such that:

$$c\|v\|_a \leq \|v\|_b \leq C\|v\|_a \quad \forall v \in \mathbb{R}^n$$

Proof Sketch: Show equivalence to a fixed norm (e.g., $\|\cdot\|_2$) using compactness of the unit sphere.

Implication for AI: The choice of norm often doesn't affect theoretical convergence results, though it may affect practical performance.

Advanced: Hölder's Inequality and General Norms

For vectors $x, y \in \mathbb{R}^n$ and $p, q > 1$ with $\frac{1}{p} + \frac{1}{q} = 1$: $\sum_{i=1}^n |x_i y_i| \leq (\sum_{i=1}^n |x_i|^p)^{1/p} (\sum_{i=1}^n |y_i|^q)^{1/q}$

Or more compactly: $\|xy\|_1 \leq \|x\|_p \|y\|_q$

Special cases:

- ▶ $p = q = 2$: Cauchy-Schwarz inequality
- ▶ $p = 1, q = \infty$: $\|xy\|_1 \leq \|x\|_1 \|y\|_\infty$

This general inequality underpins many results in functional analysis and optimization.

Theoretical Exercises with Solutions

Exercise 1: Prove that the intersection of two subspaces is also a subspace.

Solution: Let U, W be subspaces of V . Then: 1. $0 \in U$ and $0 \in W$, so $0 \in U \cap W$ 2. If $u, v \in U \cap W$, then $u + v \in U$ and $u + v \in W$, so $u + v \in U \cap W$ 3. If $u \in U \cap W$ and $\alpha \in \mathbb{F}$, then $\alpha u \in U$ and $\alpha u \in W$, so $\alpha u \in U \cap W$

Exercise 2: Show that $\|v\|_\infty \leq \|v\|_2 \leq \sqrt{n}\|v\|_\infty$ for any $v \in \mathbb{R}^n$.

Solution:

- $\|v\|_\infty = \max |v_i| \leq \sqrt{\sum v_i^2} = \|v\|_2$ since the maximum component is \leq the RMS.
- $\|v\|_2 = \sqrt{\sum v_i^2} \leq \sqrt{n(\max |v_i|)^2} = \sqrt{n}\|v\|_\infty$ since each $|v_i| \leq \max |v_i|$.

Connection to AI: Theoretical Foundations

Representer Theorem: In kernel-based methods, the optimal solution lies in the span of the training data.

Hahn-Banach Theorem: (Infinite-dimensional) Guarantees the existence of supporting hyperplanes, fundamental in optimization.

Riesz Representation Theorem: In Hilbert spaces, every continuous linear functional can be represented via an inner product.

These theoretical results ensure that many ML algorithms have well-defined solutions and desirable properties.

Summary of Key Theoretical Concepts

1. **Vector Space Axioms:** Provide the foundation for linear algebra
2. **Subspaces:** Allow working with lower-dimensional structures within high-dimensional spaces
3. **Basis and Dimension:** Characterize the “size” and coordinate systems of vector spaces
4. **Norms and Inner Products:** Provide geometric structure and measurement tools
5. **Orthogonality:** Enables decomposition and simplification of problems

These concepts form the rigorous mathematical foundation upon which all linear algebra-based AI algorithms are built.

Learning Objectives - Matrices

- ▶ Master matrix operations and their geometric interpretations
- ▶ Understand matrix multiplication and transformations
- ▶ Apply determinants and inverses to solve systems
- ▶ Implement matrix operations using NumPy
- ▶ Connect matrix concepts to AI applications

Matrix Definition and Notation

A matrix $A \in \mathbb{R}^{m \times n}$ is a rectangular array of numbers:

$$A = [a_{ij}] = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

where $i = 1, \dots, m$ (rows) and $j = 1, \dots, n$ (columns)

- ▶ as set of columns vectors: $A = [c_1^T, \dots, c_n^T]$
- ▶ as a set of row vectors: $A = [r_1, \dots, r_n]^T$

Matrix Terminology and Indexing

Key terms:

- ▶ **Order/Dimensions:** $m \times n$ (rows \times columns)
- ▶ **Main diagonal:** Elements a_{ii} where $i = j$
- ▶ **Trace:** Sum of diagonal elements $\text{tr}(A) = \sum_{i=1}^n a_{ii}$
- ▶ **Zero matrix:** All elements are 0

Indexing example: For $A = \begin{bmatrix} 2 & 5 & 8 \\ 3 & 1 & 9 \end{bmatrix}$:

- ▶ $a_{12} = 5$, $a_{21} = 3$, $a_{23} = 9$

Matrix Special Cases

- ▶ **Square matrix:** $m = n$ (e.g., $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$)
- ▶ **Row vector:** $1 \times n$ (e.g., $[1 \ 2 \ 3]$)
- ▶ **Column vector:** $m \times 1$ (e.g., $\begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix}$)
- ▶ **Diagonal matrix:** Non-zero elements only on main diagonal
- ▶ **Identity matrix:** Diagonal matrix with 1s on diagonal
- ▶ **Symmetric matrix:** $A^T = A$ (square matrix)

Types of Square Matrices

Important special cases:

- ▶ **Upper triangular:** All elements below diagonal are 0
- ▶ **Lower triangular:** All elements above diagonal are 0
- ▶ **Orthogonal matrix:** $A^T A = I$ (columns are orthonormal)
- ▶ **Idempotent matrix:** $A^2 = A$
- ▶ **Nilpotent matrix:** $A^k = 0$ for some k

Matrix Addition

For matrices $A, B \in \mathbb{R}^{m \times n}$:

$$(A + B)_{ij} = a_{ij} + b_{ij}$$

Requirements: Same dimensions

Properties:

- ▶ Commutative: $A + B = B + A$
- ▶ Associative: $(A + B) + C = A + (B + C)$
- ▶ Distributive over scalar multiplication: $\alpha(A + B) = \alpha A + \alpha B$
- ▶ Additive identity: $A + 0 = A$ (where 0 is zero matrix)
- ▶ Additive inverse: $A + (-A) = 0$

Scalar Multiplication

For scalar $\alpha \in \mathbb{R}$ and matrix A :

$$(\alpha A)_{ij} = \alpha \cdot a_{ij}$$

Geometric interpretation:

- ▶ Scales all matrix elements uniformly
- ▶ Changes transformation intensity

Properties:

- ▶ Associative: $\alpha(\beta A) = (\alpha\beta)A$
- ▶ Distributive over matrix addition: $\alpha(A + B) = \alpha A + \alpha B$
- ▶ Distributive over scalar addition: $(\alpha + \beta)A = \alpha A + \beta A$
- ▶ Identity: $1 \cdot A = A$

Matrix-Vector Multiplication: Detailed View

For $A \in \mathbb{R}^{m \times n}$ and $v \in \mathbb{R}^n$:

$$(Av)_i = \sum_{j=1}^n a_{ij} \cdot v_j$$

Alternative interpretation: Linear combination of columns

$$Av = v_1 \cdot \text{col}_1(A) + v_2 \cdot \text{col}_2(A) + \cdots + v_n \cdot \text{col}_n(A)$$

Example: $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 \\ 6 \end{bmatrix} = 5 \begin{bmatrix} 1 \\ 3 \end{bmatrix} + 6 \begin{bmatrix} 2 \\ 4 \end{bmatrix} = \begin{bmatrix} 17 \\ 39 \end{bmatrix}$

Matrix-Matrix Multiplication: Deep Dive

For $A \in \mathbb{R}^{m \times p}$ and $B \in \mathbb{R}^{p \times n}$:

$$(AB)_{ij} = \sum_{k=1}^p a_{ik} \cdot b_{kj}$$

Interpretations:

1. **Dot product:** $(AB)_{ij}$ = dot product of row i of A and column j of B
2. **Column perspective:** Column j of $AB = A \times$ column j of B
3. **Row perspective:** Row i of $AB =$ row i of $A \times B$
4. **Sum of outer products:** $AB = \sum_{k=1}^p \text{col}_k(A) \cdot \text{row}_k(B)$

Matrix Multiplication Properties

Properties:

- ▶ Associative: $(AB)C = A(BC)$
- ▶ Distributive: $A(B + C) = AB + AC$ and $(A + B)C = AC + BC$
- ▶ **NOT commutative**: $AB \neq BA$ (generally)
- ▶ Identity: $AI = IA = A$
- ▶ Zero multiplication: $A \cdot 0 = 0 \cdot A = 0$

Important note: $AB = 0$ does NOT imply $A = 0$ or $B = 0$

Identity Matrix: The Do-Nothing Transformation

Identity matrix $I_n \in \mathbb{R}^{n \times n}$:

$$(I_n)_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

Properties:

- ▶ $AI_n = A$ and $I_mA = A$ (for $A \in \mathbb{R}^{m \times n}$)
- ▶ $I_n I_n = I_n$ (idempotent)
- ▶ $I_n^{-1} = I_n$

Geometric meaning: Identity transformation - leaves vectors unchanged

Matrix Inverse: Existence and Computation

Matrix inverse A^{-1} (if it exists): $AA^{-1} = A^{-1}A = I$

Inverse exists iff:

- ▶ A is square ($m = n$)
- ▶ $\det(A) \neq 0$ (non-singular)
- ▶ Rows/columns are linearly independent
- ▶ A has full rank ($\text{rank}(A) = n$)

Properties:

- ▶ $(A^{-1})^{-1} = A$
- ▶ $(AB)^{-1} = B^{-1}A^{-1}$
- ▶ $(kA)^{-1} = k^{-1}A^{-1}$ for scalar $k \neq 0$

Matrix Inverse: Computational Methods

For 2×2 matrix $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$:

$$A^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

For larger matrices:

- ▶ Gaussian elimination (row operations)
- ▶ Adjugate method: $A^{-1} = \frac{1}{\det(A)} \text{adj}(A)$
- ▶ LU decomposition
- ▶ Numerical methods (important for AI applications)

Determinant: Geometric Interpretation

For 2×2 matrix $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$:

$$\det(A) = ad - bc$$

Geometric interpretation:

- ▶ **2D:** Area scaling factor of transformation
- ▶ **3D:** Volume scaling factor
- ▶ **Sign:** Orientation preservation (positive) or reversal (negative)

Example: $\det \left(\begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix} \right) = 6 \rightarrow \text{area increases } 6\times$

Determinant: General $n \times n$ Case

Cofactor expansion along row i:

$$\det(A) = \sum_{j=1}^n (-1)^{i+j} a_{ij} \det(M_{ij})$$

Where:

- ▶ M_{ij} = minor matrix (remove row i, column j)
- ▶ $C_{ij} = (-1)^{i+j} \det(M_{ij})$ = cofactor

Laplace expansion: Can expand along any row or column

Computational complexity: $O(n!)$ for naive implementation, $O(n^3)$ for better methods

Determinant Properties and Applications

Properties:

- ▶ $\det(AB) = \det(A) \cdot \det(B)$
- ▶ $\det(A^{-1}) = 1/\det(A)$
- ▶ $\det(cA) = c^n \det(A)$ for $n \times n$ matrix
- ▶ $\det(A^T) = \det(A)$
- ▶ Swapping rows changes sign of determinant
- ▶ Adding multiple of one row to another doesn't change determinant

Applications: Testing invertibility, solving systems, change of variables in integration

Linear Transformations: Formal Definition

Every matrix $A \in \mathbb{R}^{m \times n}$ represents a linear transformation:

$$T : \mathbb{R}^n \rightarrow \mathbb{R}^m, T(v) = Av$$

Properties of linear transformations:

- ▶ **Additivity:** $T(u + v) = T(u) + T(v)$
- ▶ **Homogeneity:** $T(\alpha v) = \alpha T(v)$
- ▶ **Preserves zero:** $T(0) = 0$

Kernel (null space): $\{v \in \mathbb{R}^n : T(v) = 0\}$ **Image (range):** $\{T(v) : v \in \mathbb{R}^n\}$

Matrix as Basis Transformation

Columns of A = images of standard basis vectors under transformation

If $e_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, $e_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ are standard basis vectors, and $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$, then:

- ▶ $T(e_1) = \begin{bmatrix} a \\ c \end{bmatrix}$ (first column)
- ▶ $T(e_2) = \begin{bmatrix} b \\ d \end{bmatrix}$ (second column)

The matrix tells us where each basis vector goes!

Rank of a Matrix

Definition: The rank of a matrix is:

- ▶ The dimension of the column space (number of linearly independent columns)
- ▶ The dimension of the row space (number of linearly independent rows)
- ▶ The maximum number of linearly independent rows/columns

Properties:

- ▶ $\text{rank}(A) \leq \min(m, n)$
- ▶ $\text{rank}(A) = \text{rank}(A^T)$
- ▶ $\text{rank}(AB) \leq \min(\text{rank}(A), \text{rank}(B))$
- ▶ Full rank: $\text{rank}(A) = n$ for $n \times n$ matrix \rightarrow invertible

Matrix Decompositions Preview

Eigen decomposition: $A = PDP^{-1}$

- ▶ P = eigenvectors, D = eigenvalues
- ▶ Applications: PCA, stability analysis

Singular Value Decomposition (SVD): $A = U\Sigma V^T$

- ▶ U, V orthogonal, Σ diagonal with singular values
- ▶ Applications: Dimensionality reduction, recommendation systems

LU decomposition: $A = LU$

- ▶ L lower triangular, U upper triangular
- ▶ Applications: Solving linear systems efficiently

“Columns as Destination” Analogy: Extended

Imagine you have a map and want to transform it:

- ▶ **Original basis vectors:** $e_1 = [1, 0]$ (east), $e_2 = [0, 1]$ (north)
- ▶ **Matrix** $A = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix}$
- ▶ e_1 moves to $[2, 1]$ (northeast)
- ▶ e_2 moves to $[1, 3]$ (north-northeast)
- ▶ Any point $[x, y] = x \cdot e_1 + y \cdot e_2$ moves to $x \cdot [2, 1] + y \cdot [1, 3]$

The grid lines deform accordingly

- ▶ parallel lines remain parallel, but distances and angles change.

“Recipe Book” Analogy: Mathematical Formulation

A matrix is like a recipe book with instructions:

- ▶ **Input vector** = ingredients $[x_1, x_2, \dots, x_n]$
- ▶ **Matrix rows** = different output components (dishes)
- ▶ **Matrix columns** = how much each ingredient contributes to each output

Mathematically: If A is $m \times n$ and x is $n \times 1$, then:

$$y = Ax \Rightarrow y_i = \sum_{j=1}^n a_{ij}x_j$$

Each output component y_i is a weighted combination of inputs!

“Transformation Machine” Visualization: Mathematical View

Think of a matrix as a machine:

Input (vector) \rightarrow [MATRIX MACHINE] \rightarrow Output (transformed vector)

Mathematical properties:

- ▶ **Linearity:** Machine respects superposition: $A(\alpha u + \beta v) = \alpha Au + \beta Av$
- ▶ **Fixed behavior:** Same input always gives same output
- ▶ **Composition:** Machines can be chained: $C = BA$ means apply A then B

Special machines:

- ▶ **Invertible:** Has “undo” button (A^{-1} exists)
- ▶ **Singular:** Collapses space ($\det(A) = 0$)
- ▶ **Orthogonal:** Preserves lengths and angles ($A^T A = I$)

Summary: Key Takeaways

1. **Matrices represent linear transformations** - they map vectors to vectors
2. **Matrix multiplication** = composition of transformations
3. **Determinant** measures volume scaling and invertibility
4. **Rank** reveals the true dimension of the transformation
5. **These concepts directly power modern AI systems**

Remember: Every neural network forward pass is essentially sophisticated matrix multiplication!

Why Matrices Matter in AI

Matrices are fundamental to AI because they:

- ▶ Represent **neural network layers** (weights and biases)
- ▶ Enable **batch processing** of data
- ▶ Facilitate **dimensionality reduction** (PCA, SVD)
- ▶ Power **image processing** and **computer vision**
- ▶ Support **graph algorithms** and **recommender systems**

Every forward pass in a neural network is essentially matrix multiplication!

Why Advanced Matrix Concepts Matter

Real-world significance:

- ▶ **Eigenvectors:** Google's PageRank algorithm, vibration analysis, quantum mechanics
- ▶ **SVD:** Image compression (JPEG), recommendation systems, natural language processing
- ▶ **PCA:** Face recognition, data visualization, feature engineering

AI applications:

- ▶ Neural network optimization
- ▶ Dimensionality reduction
- ▶ Data preprocessing and feature extraction

Eigenvectors and Eigenvalues: Formal Definition

For a square matrix $A \in \mathbb{R}^{n \times n}$, a non-zero vector v is an eigenvector if:

$$Av = \lambda v$$

Where:

- ▶ v is the eigenvector (direction that doesn't change under transformation)
- ▶ λ is the eigenvalue (scaling factor along that direction)

Characteristic equation: $\det(A - \lambda I) = 0$

Eigenvalue Properties and Theorems

Fundamental properties:

- ▶ Sum of eigenvalues = trace of matrix: $\sum \lambda_i = \text{trace}(A)$
- ▶ Product of eigenvalues = determinant: $\prod \lambda_i = \det(A)$
- ▶ Eigenvalues of A^{-1} are $1/\lambda_i$ (if A is invertible)

Important theorems:

- ▶ **Spectral Theorem:** Symmetric matrices have real eigenvalues and orthogonal eigenvectors
- ▶ **Perron-Frobenius Theorem:** Positive matrices have unique largest eigenvalue

Eigenvalue Calculation for 2×2 Matrices

Given $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$:

Step 1: Compute trace and determinant:

► $\text{trace} = a + d$

► $\det = ad - bc$

Step 2: Solve quadratic equation $\lambda^2 - \text{trace} \cdot \lambda + \det = 0$:

$$\lambda = \frac{\text{trace} \pm \sqrt{\text{trace}^2 - 4 \cdot \det}}{2}$$

Step 3: For each λ , solve $(A - \lambda I)v = 0$

Geometric Interpretation of Eigenvectors

- ▶ **Eigenvectors:** Directions that remain unchanged under transformation
- ▶ **Eigenvalues:** How much stretching/compression occurs along those directions
- ▶ **Visualization:** Arrows that stay pointing same direction after transformation

Types:

- ▶ $\lambda > 1$: Stretching
- ▶ $0 < \lambda < 1$: Compression
- ▶ $\lambda < 0$: Reflection + stretching/compression
- ▶ $\lambda = 0$: Collapse to zero

“Spinning Top” Analogy for Eigenvectors

Imagine a spinning top:

- ▶ Most points on the top trace circles as it spins
- ▶ The axis of rotation stays fixed in space
- ▶ The axis direction is the eigenvector
- ▶ The speed of rotation affects how fast points move along the axis

Mathematical connection:

- ▶ Rotation matrix has eigenvector along rotation axis
- ▶ Eigenvalue = 1 (axis doesn't stretch, only rotates around it)
- ▶ All other eigenvectors are complex (representing rotation)

“Weather Patterns” Analogy

Think of eigenvectors as dominant weather patterns:

- ▶ **Eigenvector 1:** “Summer pattern” (high temp, low pressure)
- ▶ **Eigenvector 2:** “Winter pattern” (low temp, high pressure)
- ▶ **Any day’s weather** = combination of these patterns
- ▶ **Eigenvalue** = how much this pattern dominates overall variation

Data transformation with eigenvectors:

- ▶ Rotate coordinate system to align with weather patterns
- ▶ Scale by eigenvalues (importance of each pattern)

Diagonalization: Representing Matrices in Their “Natural” Coordinates

If A has n linearly independent eigenvectors, then:

$$A = PDP^{-1}$$

Where:

- ▶ P : Matrix whose columns are eigenvectors
- ▶ D : Diagonal matrix of eigenvalues
- ▶ P^{-1} : Inverse of eigenvector matrix

Geometric interpretation: Change to coordinate system where transformation acts independently on each axis

Singular Value Decomposition (SVD): Formal Definition

Any matrix $A \in \mathbb{R}^{m \times n}$ can be decomposed as:

$$A = U\Sigma V^T$$

Where:

- ▶ $U \in \mathbb{R}^{m \times m}$: Left singular vectors (orthogonal matrix)
- ▶ $\Sigma \in \mathbb{R}^{m \times n}$: Diagonal matrix of singular values $\sigma_1 \geq \sigma_2 \geq \dots \geq 0$
- ▶ $V \in \mathbb{R}^{n \times n}$: Right singular vectors (orthogonal matrix)

SVD Geometric Interpretation

A transforms any vector by:

1. **Rotating by V^T** (input space rotation)
2. **Scaling by Σ** (along principal axes)
3. **Rotating by U** (output space rotation)

Relationship to eigenvalues:

- ▶ Singular values = $\sqrt{\text{eigenvalues of } A^T A}$
- ▶ Right singular vectors = eigenvectors of $A^T A$
- ▶ Left singular vectors = eigenvectors of AA^T

“Recipe Decomposition” Analogy for SVD

Any complex recipe (matrix A) can be broken down:

1. **Reorganize ingredients** (V^T rotation)
 - ▶ Put similar ingredients together
 - ▶ Create logical grouping
2. **Adjust quantities** (Σ scaling)
 - ▶ Some ingredients are more important than others
 - ▶ Singular values tell us relative importance
3. **Apply cooking techniques** (U rotation)
 - ▶ Transform the ingredient combinations
 - ▶ Create final dish structure

Low-Rank Approximation with SVD

Key insight: We can approximate A using only the top k singular values:

$$A \approx U[:, 1 : k] \cdot \Sigma[1 : k, 1 : k] \cdot V^T[1 : k, :]$$

Benefits:

- ▶ **Compression:** Store only $k(m + n + 1)$ numbers vs mn
- ▶ **Denoising:** Small singular values often represent noise
- ▶ **Efficiency:** Faster computations with lower-rank approximations

Error bound: Eckart-Young theorem guarantees this is the best rank- k approximation

Principal Component Analysis (PCA): Formal Derivation

Given data matrix $X \in \mathbb{R}^{m \times n}$ (m samples, n features):

Step 1: Center the data $X_{centered} = X - \mu$ (where μ is mean vector)

Step 2: Compute covariance matrix $C = \frac{1}{m} \cdot X_{centered}^T \cdot X_{centered}$

Step 3: Eigendecomposition of covariance matrix $C = Q \Lambda Q^T$ Where:

- ▶ Q : Matrix of eigenvectors (principal components)
- ▶ Λ : Diagonal matrix of eigenvalues (variance explained)

Step 4: Project onto top k components $X_{reduced} = X_{centered} \cdot Q[:, : k]$

Variance explained by component i :

$$\text{Var}_i = \frac{\lambda_i}{\sum_j \lambda_j}$$

“Shadow Projection” Analogy for PCA

Imagine 3D objects casting shadows:

- ▶ **Original data:** 3D objects in space
- ▶ **PCA:** Find best angle to shine light
- ▶ **Shadow:** 2D projection that captures most shape information

Key insights:

- ▶ First principal component = direction of longest shadow
- ▶ Second component = perpendicular direction for next best view
- ▶ Can reconstruct approximate 3D shape from 2D shadows

Relationship Between PCA and SVD

Mathematical equivalence:

- ▶ PCA on centered data $X = \text{SVD on } X_{centered}$
- ▶ Principal components = right singular vectors of $X_{centered}$
- ▶ Explained variance proportional to squared singular values

Practical implications:

- ▶ Can compute PCA using SVD (more numerically stable)
- ▶ SVD provides additional decomposition (U matrix)
- ▶ Both reveal low-dimensional structure in high-dimensional data

Matrix Norms and Condition Number

Frobenius norm: $\|A\|_F = \sqrt{\sum_{ij} a_{ij}^2} = \sqrt{\text{trace}(A^T A)}$

Spectral norm: $\|A\|_2 = \sigma_1$ (largest singular value)

Condition number: $\kappa(A) = \sigma_1 / \sigma_n$ (ratio of largest to smallest singular value)

Interpretation:

- ▶ Large condition number \rightarrow ill-conditioned matrix
- ▶ Small changes in input cause large changes in output
- ▶ Numerical instability in computations

Condition Number and Numerical Stability

Example: Solving $Ax = b$ when A is ill-conditioned

Effects:

- ▶ Small errors in b lead to large errors in x
- ▶ Rounding errors amplified during computation
- ▶ Solutions may be numerically meaningless

Remedies:

- ▶ Regularization: Solve $(A + \lambda I)x = b$
- ▶ Use SVD-based pseudoinverse
- ▶ Precondition the matrix

Advanced Topics for Further Study

Eigenvalue algorithms:

- ▶ Power iteration method
- ▶ QR algorithm for eigenvalues
- ▶ Lanczos algorithm for large sparse matrices

Extensions of PCA:

- ▶ Kernel PCA for non-linear data
- ▶ Sparse PCA for feature selection
- ▶ Robust PCA for outlier detection

Matrix factorization methods:

- ▶ Non-negative matrix factorization (NMF)
- ▶ CUR decomposition for interpretability
- ▶ Tensor decompositions for multi-dimensional data

Concrete AI Applications: PCA for Dimensionality Reduction

Mathematical foundation: Find orthogonal directions that capture maximum variance

Process:

1. Center data by subtracting mean
2. Compute covariance matrix: $C = \frac{1}{m} X^T X$
3. Find eigenvectors of C (principal components)
4. Project data onto top k eigenvectors

Benefits:

- ▶ Reduces computational complexity
- ▶ Removes noise (low-variance components often = noise)
- ▶ Enables visualization of high-dimensional data
- ▶ Regularizes model (fewer parameters)

Concrete AI Applications: Image Compression with SVD

Core idea: Any image can be stored as sum of rank-1 images

Mathematical formulation: $I = U\Sigma V^T = \sigma_1 u_1 v_1^T + \sigma_2 u_2 v_2^T + \cdots + \sigma_r u_r v_r^T$

Compression strategy:

- ▶ Keep only top k singular values (largest σ_i)
- ▶ Approximate: $I \approx \sum_{i=1}^k \sigma_i u_i v_i^T$
- ▶ Storage: $k(m + n + 1)$ vs mn for full image

Concrete AI Applications: Latent Semantic Analysis (LSA)

Problem: Words have multiple meanings, documents use related terms

SVD solution to term-document matrix $A = U\Sigma V^T$:

- ▶ U : Term-topic relationships (what terms define topics)
- ▶ Σ : Topic importance (how much each topic matters)
- ▶ V^T : Document-topic relationships (what topics documents contain)

Benefits:

- ▶ Handles synonymy (related terms map to same topic)
- ▶ Handles polysemy (word meaning depends on document context)
- ▶ Reduces dimensionality dramatically

Concrete AI Applications: Recommender Systems with SVD

Matrix factorization approach:

- ▶ User-item matrix R decomposed as $R \approx U\Sigma V^T$
- ▶ U : User preferences in latent space
- ▶ V : Item characteristics in latent space
- ▶ Σ : Importance of latent factors

Prediction: $\hat{r}_{ui} = u_u^T \Sigma v_i$

Benefits:

- ▶ Captures latent user preferences
- ▶ Handles sparse data effectively
- ▶ Provides interpretable factors

Fundamental Theorem of Linear Algebra

Key insight: Every matrix transformation can be understood through four fundamental subspaces:

- ▶ **Column space** $C(A)$: All possible outputs Ax
- ▶ **Null space** $N(A)$: Solutions to $Ax = 0$
- ▶ **Row space** $C(A^T)$: All possible outputs $A^T x$
- ▶ **Left null space** $N(A^T)$: Solutions to $A^T x = 0$

Orthogonal complements:

- ▶ $C(A) \perp N(A^T)$
- ▶ $C(A^T) \perp N(A)$

Dimension theorem: $\dim(C(A)) + \dim(N(A)) = n$

Eigenvalue Decomposition: Existence and Uniqueness

Theorem: A square matrix $A \in \mathbb{R}^{n \times n}$ is diagonalizable if and only if it has n linearly independent eigenvectors.

Proof sketch:

1. If A has n linearly independent eigenvectors v_1, \dots, v_n with eigenvalues $\lambda_1, \dots, \lambda_n$
2. Let $P = [v_1 \mid \dots \mid v_n]$ and $D = \text{diag}(\lambda_1, \dots, \lambda_n)$
3. Then $AP = PD$, so $A = PDP^{-1}$

Special cases:

- ▶ Symmetric matrices are always diagonalizable with real eigenvalues
- ▶ Normal matrices ($AA^T = A^T A$) are unitarily diagonalizable

Spectral Theorem: Formal Statement and Proof

Theorem: If A is a symmetric matrix ($A = A^T$), then:

1. All eigenvalues are real
2. Eigenvectors corresponding to distinct eigenvalues are orthogonal
3. A is orthogonally diagonalizable: $A = Q\Lambda Q^T$ where Q is orthogonal

Proof sketch for real eigenvalues: Let $Av = \lambda v$ with $v \neq 0$. Then:

$$\lambda \|v\|^2 = v^T(\lambda v) = v^T(Av) = v^T A^T v = (Av)^T v = \lambda \|v\|^2$$

Since $\|v\|^2 > 0$, λ must be real.

Jordan Canonical Form: Beyond Diagonalization

When diagonalization fails: Not all matrices have n linearly independent eigenvectors

Jordan form: Every square matrix can be written as $A = PJP^{-1}$ where J is block diagonal:

$$J = \begin{bmatrix} J_1 & & \\ & \ddots & \\ & & J_k \end{bmatrix}, \quad J_i = \begin{bmatrix} \lambda_i & 1 & & \\ & \lambda_i & \ddots & \\ & & \ddots & 1 \\ & & & \lambda_i \end{bmatrix}$$

Geometric interpretation: When eigenvectors are insufficient, we use generalized eigenvectors

SVD: Existence Proof Sketch

Theorem: Every matrix $A \in \mathbb{R}^{m \times n}$ has a singular value decomposition.

Proof outline:

1. Consider $A^T A$, which is symmetric positive semidefinite
2. By spectral theorem, $A^T A = V \Lambda V^T$ with Λ diagonal, entries ≥ 0
3. Let $\sigma_i = \sqrt{\lambda_i}$ (singular values)
4. Define U columns: $u_i = \frac{1}{\sigma_i} A v_i$ for $\sigma_i > 0$
5. Extend to orthonormal basis for remaining dimensions
6. Verify $A = U \Sigma V^T$

Matrix Norms: Formal Definitions and Properties

Operator norm (induced 2-norm):

$$\|A\|_2 = \max_{x \neq 0} \frac{\|Ax\|_2}{\|x\|_2} = \sigma_1$$

Frobenius norm:

$$\|A\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n |a_{ij}|^2} = \sqrt{\text{trace}(A^T A)} = \sqrt{\sum_{i=1}^{\min(m,n)} \sigma_i^2}$$

Properties:

- ▶ Submultiplicative: $\|AB\| \leq \|A\| \cdot \|B\|$
- ▶ Triangle inequality: $\|A + B\| \leq \|A\| + \|B\|$
- ▶ Compatible with vector norms

Pseudoinverse: Moore-Penrose Inverse

Definition: For $A \in \mathbb{R}^{m \times n}$, the pseudoinverse $A^+ \in \mathbb{R}^{n \times m}$ satisfies:

1. $AA^+A = A$
2. $A^+AA^+ = A^+$
3. $(AA^+)^T = AA^+$
4. $(A^+A)^T = A^+A$

SVD construction: If $A = U\Sigma V^T$, then $A^+ = V\Sigma^+U^T$ where Σ^+ replaces non-zero σ_i with $1/\sigma_i$

Applications: Least squares solutions, rank-deficient systems

Eckart-Young Theorem: Formal Statement

Theorem: For a matrix A with SVD $A = U\Sigma V^T$, the best rank- k approximation in Frobenius norm is:

$$A_k = \sum_{i=1}^k \sigma_i u_i v_i^T$$

Formally: $\min_{\text{rank}(B)=k} \|A - B\|_F = \|A - A_k\|_F = \sqrt{\sum_{i=k+1}^r \sigma_i^2}$

Corollary: Also optimal in spectral norm: $\min_{\text{rank}(B)=k} \|A - B\|_2 = \sigma_{k+1}$

Rayleigh Quotient and Variational Characterization

Definition: For symmetric A , the Rayleigh quotient is:

$$R_A(x) = \frac{x^T A x}{x^T x}$$

Max-min principle:

$$\lambda_1 = \max_{x \neq 0} R_A(x) = \max_{||x||=1} x^T A x$$

$$\lambda_k = \max_{\dim(S)=k} \min_{x \in S, x \neq 0} R_A(x)$$

Connection to PCA: Principal components maximize variance, which equals the Rayleigh quotient of the covariance matrix

PCA: Statistical Interpretation and Maximum Variance

Theorem: The first principal component w_1 maximizes the variance of the projected data:

$$w_1 = \arg \max_{||w||=1} \text{Var}(w^T X) = \arg \max_{||w||=1} w^T C w$$

Where C is the covariance matrix.

Proof: By Rayleigh quotient, maximum is achieved by eigenvector corresponding to largest eigenvalue.

Sequential formulation: The k -th principal component maximizes variance subject to orthogonality to previous components.

Generalized Eigenvalue Problem

Definition: For matrices A, B , find λ and v such that: $Av = \lambda Bv$

Applications:

- ▶ Fisher's linear discriminant analysis (LDA)
- ▶ Vibration analysis with mass matrices
- ▶ Generalized PCA

Solution: Equivalent to standard eigenvalue problem for $B^{-1}A$ (if B invertible) or through generalized SVD

Courant-Fischer Theorem: Spectral Theory Foundation

Theorem: For a symmetric matrix A with eigenvalues $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_n$:

$$\lambda_k = \max_{\dim(S)=k} \min_{x \in S, \|x\|=1} x^T A x = \min_{\dim(S)=n-k+1} \max_{x \in S, \|x\|=1} x^T A x$$

Interpretation: Eigenvalues represent optimal “stretching factors” in different dimensional subspaces

Application: Provides theoretical foundation for PCA and low-rank approximation error bounds

Weyl's Inequality: Eigenvalue Perturbation Theory

Theorem: For symmetric matrices A and E , with eigenvalues $\lambda_i(A)$ and $\lambda_i(A + E)$:

$$|\lambda_i(A + E) - \lambda_i(A)| \leq \|E\|_2$$

Corollary: For singular values $\sigma_i(A)$ and $\sigma_i(A + E)$: $|\sigma_i(A + E) - \sigma_i(A)| \leq \|E\|_2$

Importance: Explains stability of SVD and PCA under small perturbations

Perron-Frobenius Theory: Non-negative Matrices

Theorem: If A is a positive matrix (all entries > 0), then:

1. There is a unique largest eigenvalue $\lambda_1 > 0$ (Perron root)
2. The corresponding eigenvector has all positive entries
3. $|\lambda_i| < \lambda_1$ for all other eigenvalues

Applications: Google's PageRank, Markov chains, population models

Schur Decomposition: General Matrix Factorization

Theorem: Every square matrix A can be written as:

$$A = QUQ^{-1}$$

where Q is unitary ($U^{-1} = U^*$ conjugate transpose) and U is upper triangular.

Properties:

- ▶ Diagonal entries of U are eigenvalues of A
- ▶ Always exists (unlike diagonalization)
- ▶ Foundation for QR algorithm for eigenvalue computation

QR Algorithm for Eigenvalue Computation

Basic iteration:

1. $A_0 = A$
2. For $k = 0, 1, 2, \dots$:
 - ▶ Compute QR decomposition: $A_k = Q_k R_k$
 - ▶ Update: $A_{k+1} = R_k Q_k$

Convergence: Under suitable conditions, A_k converges to upper triangular form with eigenvalues on diagonal

Practical implementation: Includes shifts for acceleration and deflation for efficiency

Power Method: Computing Dominant Eigenvalue

Algorithm:

1. Start with random vector x_0
2. Iterate: $x_{k+1} = \frac{Ax_k}{\|Ax_k\|}$
3. Rayleigh quotient: $\lambda_{k+1} = \frac{x_k^T Ax_k}{x_k^T x_k}$

Convergence: Rate depends on ratio $|\lambda_2/\lambda_1|$

Extensions: Inverse power method (for smallest eigenvalue), subspace iteration

Generalized SVD (GSVD)

Definition: For matrices $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{p \times n}$, there exist:

- ▶ Orthogonal matrices U, V
- ▶ Matrix Q
- ▶ Diagonal matrices C, S

Such that: $A = UCQ^T$, $B = VSQ^T$

Applications: Generalized eigenvalue problems, constrained optimization, weighted least squares

Randomized SVD: Scalable Computation

Motivation: Traditional SVD is $O(\min(mn^2, m^2n))$ - expensive for large matrices

Randomized algorithm:

1. Generate random matrix Ω
2. Compute sketch: $Y = A\Omega$
3. Orthogonalize: $Q = \text{orth}(Y)$
4. Form small matrix: $B = Q^T A$
5. Compute SVD of B

Advantages: Near-optimal accuracy, significant speedup for low-rank matrices

Tensor Decompositions: Beyond Matrices

CP decomposition: Generalization of SVD to higher dimensions

$$\mathcal{X} \approx \sum_{r=1}^R \lambda_r \mathbf{a}_r \otimes \mathbf{b}_r \otimes \mathbf{c}_r$$

Tucker decomposition: Higher-order PCA

$$\mathcal{X} \approx \mathcal{G} \times_1 \mathbf{A} \times_2 \mathbf{B} \times_3 \mathbf{C}$$

Applications: Multi-way data analysis, neural network compression, recommender systems

Nonlinear Dimensionality Reduction: Beyond PCA

Kernel PCA: Map data to higher-dimensional feature space, then apply PCA

- ▶ Uses kernel trick: $K(x, y) = \langle \phi(x), \phi(y) \rangle$
- ▶ Can capture nonlinear relationships

t-SNE: Preserves local neighborhoods for visualization - Probabilistic approach based on Student-t distribution

UMAP: Based on Riemannian geometry and algebraic topology

Advanced PCA Variants

Sparse PCA: Adds L1 regularization to obtain sparse components

$$\max_w w^T C w - \lambda \|w\|_1 \quad \text{subject to } \|w\|_2 = 1$$

Robust PCA: Decomposes matrix into low-rank + sparse components

$$\min_{L,S} \text{rank}(L) + \lambda \|S\|_0 \quad \text{subject to } L + S = A$$

Kernel PCA: Nonlinear extension using kernel trick

Applications in Deep Learning

Singular values and training dynamics:

- ▶ Gradient descent dynamics relate to singular values of weight matrices
- ▶ Training speed depends on condition number
- ▶ Singular values indicate effective capacity of layers

Neural tangent kernel (NTK):

- ▶ Infinite-width limit of neural networks
- ▶ Training dynamics governed by eigenvalue spectrum of NTK
- ▶ Connection to PCA of feature space

Theoretical Limits and Open Problems

Open problems in matrix computations:

- ▶ Fast matrix multiplication: Can we achieve $O(n^{2+\epsilon})$?
- ▶ Optimal complexity for SVD of structured matrices
- ▶ Quantum algorithms for linear algebra

Statistical limits of PCA:

- ▶ High-dimensional asymptotics ($n, p \rightarrow \infty$ with $n/p \rightarrow \gamma$)
- ▶ Spiked covariance model and phase transitions
- ▶ Detection thresholds for signal in noise

Differential Calculus

Overview of Differential Calculus

- ▶ Calculus studies continuous change
- ▶ Derivatives measure instantaneous rates of change
- ▶ Essential for optimization and machine learning
- ▶ Key concepts we'll cover:
 - ▶ Derivatives: The instantaneous speedometer
 - ▶ Gradients: Multi-dimensional rates of change
 - ▶ Chain rule: Composition of functions

Why Calculus Matters for AI

- ▶ **Optimization:** Finding minima/maxima of loss functions
- ▶ **Gradient Descent:** Core training algorithm for neural networks
- ▶ **Backpropagation:** Chain rule for computing gradients
- ▶ **Feature Engineering:** Understanding sensitivity
- ▶ **Regularization:** Penalty functions and their derivatives
- ▶ **Model Interpretation:** Understanding model behavior

Formal Definition of Limits

A function $f(x)$ has limit L as x approaches a if:

$$\lim_{x \rightarrow a} f(x) = L$$

Formal definition (ϵ - δ): For every $\epsilon > 0$, there exists $\delta > 0$ such that if $0 < |x - a| < \delta$, then $|f(x) - L| < \epsilon$

Continuity: f is continuous at a if $\lim_{x \rightarrow a} f(x) = f(a)$

Limit Definition of Derivative

The derivative of $f(x)$ at point x_0 is:

$$f'(x_0) = \lim_{h \rightarrow 0} \frac{f(x_0 + h) - f(x_0)}{h}$$

Geometric interpretation: Slope of tangent line at $(x_0, f(x_0))$

Alternative form:

$$f'(x_0) = \lim_{x \rightarrow x_0} \frac{f(x) - f(x_0)}{x - x_0}$$

Derivative as a Function

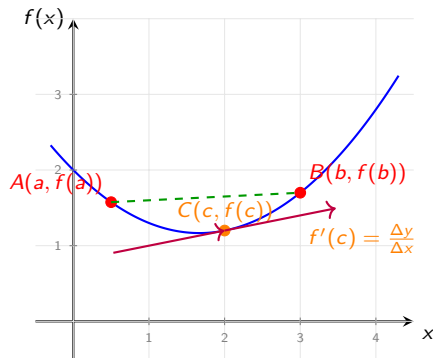
If $f'(x)$ exists for all x in an interval, we can define the **derivative function**:

Notation: $f'(x)$, $\frac{df}{dx}$, $Df(x)$, $\dot{f}(x)$

Differentiability implies continuity: If f is differentiable at x_0 , then f is continuous at x_0

Example: For $f(x) = x^2$, $f'(x) = 2x$ for all $x \in \mathbb{R}$

Visualizing the Derivative



- ▶ **Secant line:** Average rate of change between two points
- ▶ **Tangent line:** Instantaneous rate of change at one point
- ▶ **Derivative:** Slope of tangent line

Differentiability and Smoothness

A function is **differentiable** at x_0 if:

- ▶ The derivative exists at x_0
- ▶ The function is “smooth” at x_0

Cases where derivative may not exist:

- ▶ Corner points: $f(x) = |x|$ at $x = 0$
- ▶ Cusps: $f(x) = x^{2/3}$ at $x = 0$
- ▶ Vertical tangents: $f(x) = \sqrt[3]{x}$ at $x = 0$
- ▶ Discontinuities

Power Rule

For $f(x) = x^n$: $\frac{d}{dx}[x^n] = nx^{n-1}$

Proof using limit definition:

$$\lim_{h \rightarrow 0} \frac{(x+h)^n - x^n}{h} = \lim_{h \rightarrow 0} \frac{nx^{n-1}h + \binom{n}{2}x^{n-2}h^2 + \dots + h^n}{h} = nx^{n-1}$$

Examples:

- ▶ $\frac{d}{dx}[x^2] = 2x$
- ▶ $\frac{d}{dx}[x^3] = 3x^2$
- ▶ $\frac{d}{dx}[x^{-1}] = -x^{-2}$

Special case: $\frac{d}{dx}[\sqrt{x}] = \frac{1}{2\sqrt{x}}$

Constant Multiple Rule

If c is a constant and f is differentiable:

$$\frac{d}{dx}[cf(x)] = c \frac{d}{dx}[f(x)]$$

Example: $\frac{d}{dx}[5x^3] = 5 \cdot 3x^2 = 15x^2$

Sum Rule: $\frac{d}{dx}[f(x) + g(x)] = f'(x) + g'(x)$

Example: $\frac{d}{dx}[x^2 + \sin(x)] = 2x + \cos(x)$

Product Rule

For $f(x) = u(x) \cdot v(x)$: $\frac{d}{dx}[uv] = u'v + uv'$

Proof using limit definition:

$$\begin{aligned}\frac{d}{dx}[uv] &= \lim_{h \rightarrow 0} \frac{u(x+h)v(x+h) - u(x)v(x)}{h} \\ &= \lim_{h \rightarrow 0} \frac{u(x+h)v(x+h) - u(x)v(x+h) + u(x)v(x+h) - u(x)v(x)}{h} \\ &= \lim_{h \rightarrow 0} \left[v(x+h) \frac{u(x+h) - u(x)}{h} + u(x) \frac{v(x+h) - v(x)}{h} \right] = u'v + uv'\end{aligned}$$

Example: $f(x) = x^2 \cdot \sin(x)$

- ▶ $u(x) = x^2$, $u'(x) = 2x$
- ▶ $v(x) = \sin(x)$, $v'(x) = \cos(x)$
- ▶ $f'(x) = 2x \cdot \sin(x) + x^2 \cdot \cos(x)$

Quotient Rule

For $f(x) = \frac{u(x)}{v(x)}$:

$$\frac{d}{dx} \left[\frac{u}{v} \right] = \frac{u'v - uv'}{v^2}$$

Proof: Write $f(x) = u(x) \cdot [v(x)]^{-1}$ and apply product rule + chain rule

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

- ▶ $u(x) = x, u'(x) = 1$
- ▶ $v(x) = x^2 + 1, v'(x) = 2x$
- ▶ $f'(x) = \frac{1 \cdot (x^2+1) - x \cdot 2x}{(x^2+1)^2} = \frac{1-x^2}{(x^2+1)^2}$

Chain Rule

For $f(x) = g(h(x))$: $\frac{d}{dx}[g(h(x))] = g'(h(x)) \cdot h'(x)$

Alternative notation: If $y = g(u)$ and $u = h(x)$, then $\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}$

Example: $f(x) = \sin(x^2)$

- ▶ $g(x) = \sin(x)$, $g'(x) = \cos(x)$
- ▶ $h(x) = x^2$, $h'(x) = 2x$
- ▶ $f'(x) = \cos(x^2) \cdot 2x = 2x \cos(x^2)$

Chain Rule for Multiple Compositions

For $f(x) = g(h(k(x)))$: $\frac{d}{dx}[g(h(k(x)))] = g'(h(k(x))) \cdot h'(k(x)) \cdot k'(x)$

Example: $f(x) = \sin(\cos(x^2))$

- ▶ $g(x) = \sin(x)$, $g'(x) = \cos(x)$
- ▶ $h(x) = \cos(x)$, $h'(x) = -\sin(x)$
- ▶ $k(x) = x^2$, $k'(x) = 2x$
- ▶ $f'(x) = \cos(\cos(x^2)) \cdot (-\sin(x^2)) \cdot 2x = -2x \cos(\cos(x^2)) \sin(x^2)$

Exponential and Logarithmic Derivatives

Exponential: $\frac{d}{dx}[e^x] = e^x$

Natural logarithm: $\frac{d}{dx}[\ln(x)] = \frac{1}{x}$

Logarithm with base a : $\frac{d}{dx}[\log_a(x)] = \frac{1}{x \ln(a)}$

General exponential: $\frac{d}{dx}[a^x] = a^x \ln(a)$

Proof for e^x : Use limit definition and $\lim_{h \rightarrow 0} \frac{e^h - 1}{h} = 1$

Trigonometric Derivatives

- ▶ $\frac{d}{dx}[\sin(x)] = \cos(x)$
- ▶ $\frac{d}{dx}[\cos(x)] = -\sin(x)$
- ▶ $\frac{d}{dx}[\tan(x)] = \sec^2(x) = \frac{1}{\cos^2(x)}$
- ▶ $\frac{d}{dx}[\arcsin(x)] = \frac{1}{\sqrt{1-x^2}}$
- ▶ $\frac{d}{dx}[\arccos(x)] = -\frac{1}{\sqrt{1-x^2}}$
- ▶ $\frac{d}{dx}[\arctan(x)] = \frac{1}{1+x^2}$

Proof for $\sin(x)$: Use limit definition and trigonometric identities

Higher Order Derivatives

The **second derivative**: $f''(x) = \frac{d}{dx}[f'(x)]$

Notation: $f''(x)$, $\frac{d^2f}{dx^2}$, $f^{(2)}(x)$

n-th derivative: $f^{(n)}(x) = \frac{d}{dx}[f^{(n-1)}(x)]$

Physical interpretation:

- ▶ First derivative: velocity (rate of change of position)
- ▶ Second derivative: acceleration (rate of change of velocity)
- ▶ Third derivative: jerk (rate of change of acceleration)

Implicit Differentiation

Used when y is defined implicitly as a function of x :

Example: $x^2 + y^2 = 1$ (circle)

- ▶ Differentiate both sides with respect to x : $2x + 2y \frac{dy}{dx} = 0$
- ▶ Solve for $\frac{dy}{dx}$: $\frac{dy}{dx} = -\frac{x}{y}$

Example: $y^2 = x^2 + \sin(xy)$

- ▶ Differentiate: $2y \frac{dy}{dx} = 2x + \cos(xy)(y + x \frac{dy}{dx})$
- ▶ Solve for $\frac{dy}{dx}$

Partial Derivatives

For $f : \mathbb{R}^n \rightarrow \mathbb{R}$, the partial derivative with respect to x_i :

$$\frac{\partial f}{\partial x_i} = \lim_{h \rightarrow 0} \frac{f(x_1, \dots, x_i + h, \dots, x_n) - f(x_1, \dots, x_i, \dots, x_n)}{h}$$

Geometric interpretation: Rate of change when only x_i varies, others constant

Notation: $\frac{\partial f}{\partial x}$, $\frac{\partial f}{\partial y}$, f_x , f_y

Example: $f(x, y) = x^2y + \sin(xy)$

► $\frac{\partial f}{\partial x} = 2xy + y \cos(xy)$

► $\frac{\partial f}{\partial y} = x^2 + x \cos(xy)$

Computing Partial Derivatives

To compute $\frac{\partial f}{\partial x_i}$:

1. Treat all other variables as constants
2. Differentiate with respect to x_i using standard rules

Example: $f(x, y, z) = x^2 y^3 z + e^{xyz}$

- ▶ $\frac{\partial f}{\partial x} = 2xy^3z + yze^{xyz}$
- ▶ $\frac{\partial f}{\partial y} = 3x^2y^2z + xze^{xyz}$
- ▶ $\frac{\partial f}{\partial z} = x^2y^3 + xye^{xyz}$

Mixed partials: $\frac{\partial^2 f}{\partial x \partial y} = \frac{\partial}{\partial x} \left(\frac{\partial f}{\partial y} \right)$

Gradient Vector

The gradient of $f : \mathbb{R}^n \rightarrow \mathbb{R}$ at point x :

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}$$

Key properties:

- ▶ Points in direction of steepest ascent
- ▶ Magnitude = rate of change in steepest direction
- ▶ Orthogonal to level curves/surfaces
- ▶ Negative gradient ($-\nabla f$) points in direction of steepest descent

Geometric Interpretation of Gradient

For a 2D function $f(x, y)$:

- ▶ **Level curves:** $f(x, y) = c$ (constant)
- ▶ **Gradient** ∇f is perpendicular to level curves
- ▶ Direction of ∇f : steepest uphill direction
- ▶ Magnitude $\|\nabla f\|$: steepness of the slope

Example: $f(x, y) = x^2 + y^2$ (paraboloid)

- ▶ Level curves: circles $x^2 + y^2 = c$
- ▶ $\nabla f = [2x, 2y]^T$
- ▶ At $(1, 1)$: $\nabla f = [2, 2]^T$, pointing away from origin

Directional Derivatives

Directional derivative in direction u (where $\|u\| = 1$):

$$D_u f = \nabla f \cdot u = \|\nabla f\| \cos \theta$$

Where θ is the angle between ∇f and u .

Computing directional derivative:

1. Find gradient ∇f
2. Normalize direction vector u
3. Compute dot product

Interpretation: Rate of change in any specified direction

Maximum directional derivative: Occurs when u is parallel to ∇f

Hessian Matrix

The Hessian matrix contains all second partial derivatives:

$$H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$$

For $f(x, y)$: $H = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial y^2} \end{bmatrix}$

Symmetry: $H_{ij} = H_{ji}$ (if second partials are continuous)

Applications:

- ▶ Curvature information (concavity/convexity)
- ▶ Newton's method: $x_{new} = x_{old} - H^{-1} \nabla f$
- ▶ Second-order Taylor expansion approximation

Taylor Series Expansion

For function f around point x_0 :

$$f(x) \approx f(x_0) + \nabla f(x_0)^T (x - x_0) + \frac{1}{2} (x - x_0)^T H(x_0) (x - x_0) + \dots$$

Linear approximation: $f(x) \approx f(x_0) + \nabla f(x_0)^T (x - x_0)$

Quadratic approximation: Includes Hessian term

This is fundamental to understanding optimization landscapes

Remainder term: $R_n(x) = \frac{f^{(n+1)}(c)}{(n+1)!} (x - x_0)^{n+1}$ for some c between x and x_0

“Speedometer” Analogy for Derivatives

Imagine driving a car:

- ▶ **Position function:** $f(t)$ = your location at time t
- ▶ **Average speed:** $\frac{f(t_2) - f(t_1)}{t_2 - t_1}$
- ▶ **Instantaneous speed:** $f'(t)$ = what your speedometer shows RIGHT NOW

Key insight: Derivative tells you the exact rate of change at a specific instant, not averaged over time. This is crucial for real-time systems.

“Hill Climbing” Analogy for Gradients

You're standing on a hill (3D function $f(x, y)$):

- ▶ $\frac{\partial f}{\partial x}$ = slope if you step east (y constant)
- ▶ $\frac{\partial f}{\partial y}$ = slope if you step north (x constant)
- ▶ $\nabla f = [\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}]$ points to steepest uphill
- ▶ $-\nabla f$ points to steepest downhill

Gradient magnitude: How steep the hill is at your location **Gradient direction:** Which way is steepest

“Weather Forecasting” Analogy for Partial Derivatives

Temperature depends on: $T(x, y, z, t)$ - x : longitude, y : latitude, z : altitude, t : time

Partial derivatives:

- ▶ $\frac{\partial T}{\partial x}$: How temperature changes moving east-west
- ▶ $\frac{\partial T}{\partial y}$: How temperature changes moving north-south
- ▶ $\frac{\partial T}{\partial z}$: How temperature changes with altitude
- ▶ $\frac{\partial T}{\partial t}$: How temperature changes over time

Each partial tells you sensitivity to one factor, holding others constant

“Economic Analysis” Analogy for Hessian

Profit function $P(x_1, x_2, \dots, x_n)$:

- ▶ **Gradient:** Which direction increases profit most
- ▶ **Hessian:** How profit growth accelerates/decelerates

Positive definite Hessian: Diminishing returns (convex)

Negative definite Hessian: Increasing returns (concave)

Indefinite Hessian: Saddle point (unstable equilibrium)

Second-order optimization: Use curvature information for smarter steps

Advanced Topic: Automatic Differentiation

Automatic differentiation (autodiff) is the technique used in modern deep learning frameworks:

Two modes:

- ▶ **Forward mode:** Compute derivatives along with function evaluation
- ▶ **Reverse mode:** Efficient for functions with many inputs (neural networks)

Dual numbers: Extend real numbers with an infinitesimal component ϵ where $\epsilon^2 = 0$

Example: For $f(x) = x^2$, if $x = a + b\epsilon$, then $f(x) = a^2 + 2ab\epsilon$

Concrete AI Applications: Backpropagation Mathematics

Neural network training relies on gradients:

Forward pass: Compute activations layer by layer

Loss computation: $L(f(x; \theta))$ where θ are parameters

Backward pass (chain rule):

$$\frac{\partial L}{\partial \theta_l} = \frac{\partial L}{\partial a_l} \cdot \frac{\partial a_l}{\partial z_l} \cdot \frac{\partial z_l}{\partial \theta_l}$$

Where:

- ▶ $\frac{\partial L}{\partial a_l}$: Gradient of loss w.r.t. activation
- ▶ $\frac{\partial a_l}{\partial z_l}$: Derivative of activation function
- ▶ $\frac{\partial z_l}{\partial \theta_l}$: Gradient of linear combination w.r.t. parameters

Concrete AI Applications: Gradient-Based Feature Engineering

Feature importance can be estimated using gradients:

For model $f(x_1, x_2, \dots, x_n)$:

- ▶ $|\frac{\partial f}{\partial x_i}|$: Sensitivity of output to feature i
- ▶ Large magnitude \rightarrow important feature
- ▶ Small magnitude \rightarrow less important feature

Applications:

- ▶ Feature selection in linear models
- ▶ Saliency maps in computer vision
- ▶ Input importance analysis in deep learning
- ▶ Adversarial attack vulnerability assessment

Concrete AI Applications: Second-Order Optimization

Newton's method uses Hessian information:

Update rule: $x_{t+1} = x_t - H^{-1} \nabla f(x_t)$

Advantages over gradient descent:

- ▶ Quadratic convergence near optimum
- ▶ Step size automatically determined by curvature
- ▶ Can handle ill-conditioned problems better

Challenges:

- ▶ Computing and inverting Hessian is expensive
- ▶ Hessian may not be positive definite
- ▶ Memory requirements for large problems

Concrete AI Applications: Natural Gradient Descent

Uses Fisher information matrix instead of Hessian:

Update rule: $\theta_{t+1} = \theta_t - \eta F^{-1} \nabla_{\theta} L$

Where F is the Fisher information matrix.

Advantages:

- ▶ Invariant to parameterization
- ▶ More stable convergence
- ▶ Particularly useful for policy optimization in reinforcement learning

Cost Functions in Machine Learning

Cost Functions: The Foundation of Learning

What are Cost Functions?

Cost functions (also called loss functions or objective functions) measure the discrepancy between predicted and actual values, guiding the learning process.

Key Properties:

- ▶ **Differentiability:** Required for gradient-based optimization
- ▶ **Convexity:** Desirable for global minima
- ▶ **Scale invariance:** Consistent behavior across data ranges
- ▶ **Computational efficiency:** Fast to evaluate

Role in Training:

- ▶ Quantify prediction errors
- ▶ Provide optimization targets
- ▶ Enable gradient computation
- ▶ Drive parameter updates

"The choice of cost function fundamentally shapes what a model learns"

Regression Cost Functions

Mean Squared Error (MSE):

$$L_{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

Properties:

- ▶ Strongly penalizes large errors
- ▶ Convex and differentiable
- ▶ Sensitive to outliers
- ▶ Standard for regression tasks

Mean Absolute Error (MAE):

$$L_{MAE} = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$$

Properties:

- ▶ Linear penalty for errors
- ▶ Robust to outliers
- ▶ Non-differentiable at zero
- ▶ Used in robust regression

Huber Loss: Best of Both Worlds

$$L_{\delta} = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{if } |y - \hat{y}| \leq \delta \\ \delta(|y - \hat{y}| - \frac{1}{2}\delta) & \text{otherwise} \end{cases}$$

Classification Cost Functions

Binary Cross-Entropy (Log Loss)

$$L_{BCE} = -\frac{1}{n} \sum_{i=1}^n [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

Applications:

- ▶ Binary classification (spam detection, medical diagnosis)
- ▶ Neural network output layers with sigmoid activation
- ▶ Probability calibration

Categorical Cross-Entropy

$$L_{CCE} = -\sum_{i=1}^n \sum_{j=1}^k y_{ij} \log(\hat{y}_{ij})$$

where k is the number of classes

Applications:

- ▶ Multi-class classification (image recognition, text classification)
- ▶ Softmax output layers
- ▶ Language modeling token prediction

Modern AI and LLM Cost Functions

Perplexity (Language Models)

$$PP = \exp \left(-\frac{1}{N} \sum_{i=1}^N \log p(w_i | w_1, \dots, w_{i-1}) \right)$$

Measures how well a model predicts a sequence of words

Transformer-Specific Losses:

- ▶ **Next Token Prediction:** Cross-entropy over vocabulary
- ▶ **Masked Language Modeling:** Predict masked tokens (BERT-style)
- ▶ **Contrastive Loss:** Align embeddings in CLIP-style models

Contrastive Loss (Self-Supervised Learning)

$$L_{contrastive} = -\log \frac{\exp(\text{sim}(z_i, z_j)/\tau)}{\sum_{k=1}^N \exp(\text{sim}(z_i, z_k)/\tau)}$$

Applications:

- ▶ GPT, BERT, T5 family models
- ▶ Vision-language models (CLIP, DALL-E)
- ▶ Embedding alignment and retrieval

Advanced and Specialized Cost Functions

Focal Loss:

$$L_{focal} = -(1 - p_t)^\gamma \log(p_t)$$

- ▶ Addresses class imbalance
- ▶ Focuses on hard examples
- ▶ Used in object detection

Triplet Loss:

$$L_{triplet} = \max(0, d(a, p) - d(a, n) + \text{margin})$$

- ▶ Metric learning
- ▶ Face recognition
- ▶ Embedding spaces

KL Divergence:

$$L_{KL} = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

- ▶ Distribution matching
- ▶ Variational autoencoders
- ▶ Knowledge distillation

Earth Mover's Distance:

- ▶ Optimal transport
- ▶ Domain adaptation
- ▶ GAN training stability

Modern Trend: Composite losses combining multiple objectives

Cost Functions in Practice

Choosing the Right Cost Function

Task Type	Recommended Loss	Considerations
Regression	MSE/MAE	Outlier sensitivity
Binary Classification	Binary Cross-Entropy	Class imbalance
Multi-class	Categorical Cross-Entropy	Number of classes
Sequence Modeling	Perplexity	Sequence length
Object Detection	Focal + IoU Loss	Scale variation
Generation	Adversarial + Perceptual	Quality vs diversity

Practical Tips:

- ▶ **Monitor multiple metrics:** Loss alone can be misleading
- ▶ **Consider class imbalance:** Use weighted or focal losses
- ▶ **Regularization:** Add L1/L2 terms to prevent overfitting
- ▶ **Gradient clipping:** Prevent exploding gradients in deep networks
- ▶ **Learning rate scheduling:** Different losses may need different schedules

“The art of machine learning is choosing the right objective function”

Numerical aspects

Problems





- ▶ AI
- ▶ Fitting of simulation parameters
- ▶ Linear and non-linear systems
- ▶ Simulation

—→ any domain where there are measurements and uncertainty!

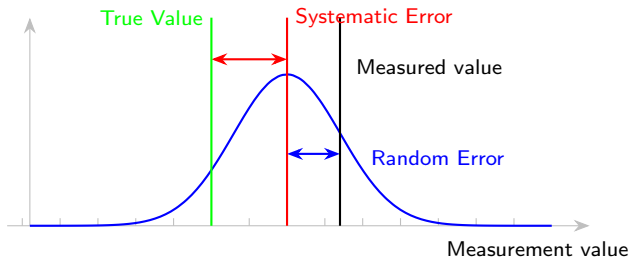
Error sources

- ▶ measurement
- ▶ method / approximation
- ▶ representation

Accuracy vs Precision

	Accurate	Not Accurate (Inaccurate)
Precise		
Not Precise (Imprecise)		

Measurement Error



Approximation / method errors

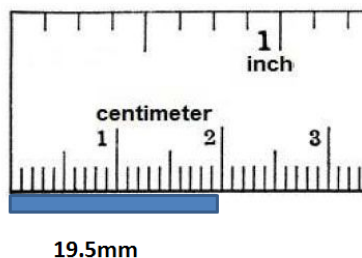
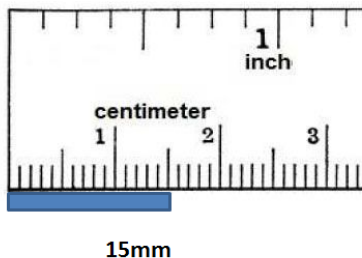
Taylor's approximation:

$$f(a+h) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!} h^n \approx \sum_{n=0}^k \frac{f^{(n)}(a)}{n!} h^n + O(h^{k+1})$$

Newton's method:

$$x \text{ such that } f(x) = 0 \implies x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Representation Error



Absolute - Relative

Absolute:

$$\Delta x \equiv x - \tilde{x}$$

Relative:

$$\epsilon_x \equiv \frac{\Delta x}{x} = \frac{x - \tilde{x}}{x}$$

Think of an error of 1 on $x = 10^{-9}$ or $x = 10^9$

Error analysis

$$y = f(x)$$

$$|\epsilon_y| \approx \kappa |\epsilon_x|$$

- ▶ well conditioned: $\kappa \approx 1$
- ▶ ill conditioned: $\kappa \gg 1$

Finite Arithmetic

Positional representation

$$n_k \dots n_0 . d_1 \dots d_l$$

- ▶ base 10: $n, d \in \{0, \dots, 9\} \rightarrow \sum_{i=0}^k n_i 10^i + \sum_{j=1}^l d_j 10^{-j}$
- ▶ base 2: $n, d \in \{0, 1\} \rightarrow \sum_{i=0}^k n_i 2^i + \sum_{j=1}^l d_j 2^{-j}$

Integers

► base 10: $n, d \in \{0, \dots, 9\} \rightarrow \sum_{i=0}^k n_i 10^i$

► base 2: $n, d \in \{0, 1\} \rightarrow \sum_{i=0}^k n_i 2^i$

16 bit integer:

► Unsigned: 0 to $\sum_{i=0}^{15} 2^i = 2^{16} - 1 = 65535$

► Signed: $-\sum_{i=0}^{15} 2^i = -2^{16} + 1 = -65535$ to
 $\sum_{i=0}^{15} 2^i = 2^{16} - 1 = 65535$

Signed integers

Zero is represented twice! Using the 2-complement:

$$n = \begin{cases} \sum_{i=1}^N n_i 2^{N-i}, & \text{if } n_N = 0 \\ \sum_{i=1}^N n_i 2^{N-i} - 2^N, & \text{if } n_N = 1 \end{cases}$$

we can store one number more so that the interval becomes $[-2^N, 2^N - 1]$

Floating point representation

Much like scientific notation $\pm nEk = n10^k$

- ▶ sign s (1 bit)
- ▶ exponent $e = e_1, \dots, e_m$ (8 bit in float32)
- ▶ fractional part $f = f_1, \dots, f_s$ (23 bit in float32)
- ▶ exponent bias ν

$$x = se_1 \dots e_m f_1 \dots f_s$$

Floating point representation (2)

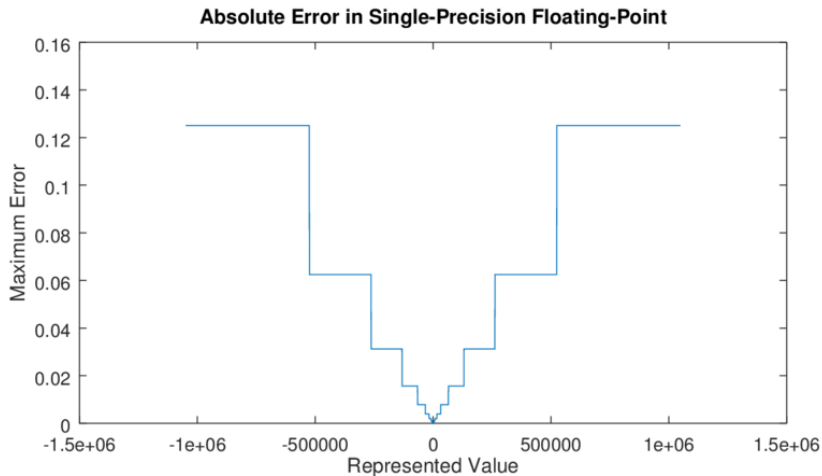
$$x = se_1 \dots e_m f_1 \dots f_s = (-1)^s \left(\sum_{i=1}^s f_i 2^{s-i} \right) 2^{r-\nu}, \quad r = \sum_{j=1}^m e_j 2^{m-j}$$

- ▶ finite set 2^n elements
- ▶ smallest representable number $N_m \approx 2^{-m+1}$
- ▶ biggest representable number $N_M \approx 2^{m+s}$
- ▶ $[N_m, N_M]$ not sampled uniformly: $|[0, 1]| \approx |[1, N_M]|$
- ▶ real implementation has ± 0 , plus combinations reserved for special $\pm\infty$, NaN etc.
- ▶ normalization gives one extra bit

FP limits

- ▶ **Overflow** $|x| > N_M; \pm \text{inf}$
- ▶ **Underflow** $|x| < N_m$

FP error



Implicit base conversions!

Watch out as this always happens! The problem:

$$1|_{10} = 2^0 = 1|_2$$

$$10|_{10} = 10^1 = 2^3 + 2^1 = 1010|_2$$

$$\begin{aligned} 0.1|_{10} &= 2^{-4} + 2^{-5} + 2^{-8} + \dots \\ &= 0.0001|_2 (=0.0625|_{10}) + 0.0000100110\dots|_2 \end{aligned}$$

FP tests

Float16:

- ▶ sign,
- ▶ 5 bit exponent $\approx 2^6 - 1 = 31$, $\nu = 16 \rightarrow$
max exponent $2^6 - 1 - 16 = 15$
- ▶ 10 bit fraction $1.111111111|_2 \approx 1.999|_{10}$
- ▶ expected max $\approx 2 \cdot 2^{15} = 65536$
- ▶ expected normalized min $\approx 2^{-16} \approx 1.52 \cdot 10^{-5}$
- ▶ expected denormalized min $\approx 2^{-10} 2^{-15} \approx 5.96 \cdot 10^{-8}$

Implementation

```
1 import numpy as np
2 import pandas as pd
3 from numpy import float16 as f, float32 as f32
4
5 %precision 100
6 np.set_printoptions(precision=100)
7 pd.options.display.float_format = '{:.100f}'.format
```

```
1 # Expected max: 2^16 = 65536
```

```
2 print(2**15*2)
```

```
3 print(f(65536))
```

```
1 65536
```

```
2 inf
```

```
1 /tmp/ipykernel_1003603/3921099518.py:3: RuntimeWarning: overflow encountered in cast
```

```
2 print(f(65536))
```

```
1 x = 1.999*2**15
2 print(x)
3 print(f(x))
```

```
1 65503.232
2 6.55e+04
```

```
1 #(1).11111 11111 | 2 = (2^0) + 2^(-x) x=1,...,10
2 x = sum([2**-x for x in range(0,11)])
3 print(x)
4 print(x*2**15)
```

```
1 1.9990234375
2 65504.0
```

```
1 # 65500?  
2 print(f(65504))  
3 print(f(65505))  
4  
5 print(f(65519))  
6 print(f(65520))
```

```
1 6.55e+04  
2 6.55e+04  
3 6.55e+04  
4 inf
```

```
1 /tmp/ipykernel_1003603/3696056160.py:6: RuntimeWarning: overflow encountered in cast  
2 print(f(65520))
```

```
1 # There is some approximation done by the display function, different for f32!
2 print(f32(f(65504)))
3 print(f32(f(65505)))
4
5 print(f32(f(65519)))
6 print(f32(f(65520)))
```

```
1 65504.0
2 65504.0
3 65504.0
4 inf
```

```
1 /tmp/ipykernel_1003603/3422968668.py:6: RuntimeWarning: overflow encountered in cast
2 print(f32(f(65520)))
```



```
1 # when the exponent is maximum, the resolution is 32
2 print(2**15 * 2**-10)
3 print(2**16 - 32)
```

```
1 32.0
2 65504
```

```
1 #1.11111 11110 2**-10 * 2**15 = 2**5 = 32 --> approx @ [0-15][16-32]
2 x = sum([2**x for x in range(0,10)])
3 print(x)
4 print(x*2**15)
```

```
1 1.998046875
2 65472.0
```

```
1 #1.11111 11110 2** -10 * 2**15 = 2**5 = 32 --> approx @ [0-15][16-32]
2 print(f32(f(65472)))
3 print(f32(f(65473)))
4 print()
5 #472+16 = 488, 472+32=504
6 print(f32(f(65488)))
7 print(f32(f(65489)))
8 print(f32(f(65490)))
```

```
1 65472.0
2 65472.0
3
4 65472.0
5 65504.0
6 65504.0
```

```
1 #1.11111 11111 + [0-16] -> 65504 + 16 = inf
2 print(f32(f(65504)))
3 #504+16 = 520
4 print(f32(f(65519)))
5 print(f32(f(65520)))
```

```
1 65504.0
2 65504.0
3 inf
```

```
1 /tmp/ipykernel_1003603/1849818127.py:5: RuntimeWarning: overflow encountered in cast
2 print(f32(f(65520)))
```

[illegible]

$s(b)$

```
1 0.1000000000000000055511151231257827021181583404541015625000000000000000000000000000000000000000000
```

0.1

0.10000000000000000055511151231257827021181583404541015625000

$0.1 + 0.1$

```
1 0.2000000000000000001110223024625156540423631668090820312500000000000000000000000000000000000000000
```



```
1 # Attention for stopping conditions!  
2 print(f(0.1)+f(0.1) == 0.2)  
3 print(f(0.1)+f(0.1) == f(0.2))  
4 print(f32(0.1)+f32(0.1) == 0.2)  
5 print(0.1+0.1 == 0.2)
```

```
1 True  
2 True  
3 True  
4 True
```

```
1 #1.00000 00000 * 2^-16 = 1.52 10^-5
2 print(f32(2**-16))
3 print(f(2**-16)) #significant digits
4 print(f32(f(2**-16)))
5 print(1/65536)
```

```
1 1.5258789e-05
2 1.526e-05
3 1.5258789e-05
4 1.52587890625e-05
```

```
1 #0. 00000 001 = 2-16 = 2-24 = 5.9 10-8
2 print(f32(f(2**-24)))
3 print(f32(f(2**-25)))
4 print(f32(2**-25))
```

```
1 5.9604645e-08
2 0.0
3 2.9802322e-08
```

Pay attention to conversions!

- ▶ Integer to FP: always possible within FP precision
- ▶ FP to Integer: may not be representable, or change sign unexpectedly!
- ▶ Some conversions happen without you even knowing! Beware of print functions!

Finding Roots

Bisection method

$$y = f(x) : \quad \bar{x} \in [a, b] \text{ such that } f(\bar{x}) = 0?$$

- ▶ f continuous on $[a, b]$
- ▶ $f(a)f(b) < 0 \Rightarrow \exists \bar{x} | f(\bar{x}) = 0$

Bisection method (2)

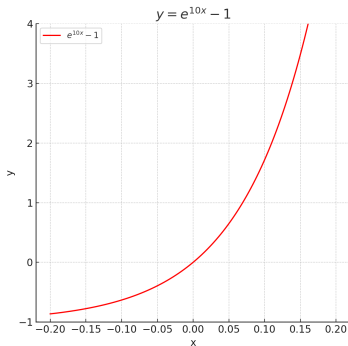
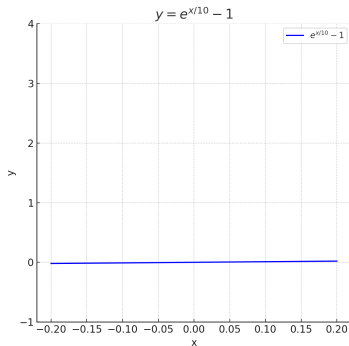
1. let $k = 0$; $[a_0, b_0] = [a, b]$;
2. iteration step: $c_{k+1} = a_k + \frac{1}{2}(b_k - a_k) \implies [a_k, c_{k+1}], [c_{k+1}, b_k]$
3. iteration check: stop if $f(c_{k+1}) = 0$; return c_{k+1}
4. $[a_{k+1}, b_{k+1}] = [a_k, c_{k+1}]$ if $f(a_k)f(c_{k+1}) < 0$ else $[c_{k+1}, b_k]$
5. $k = k + 1$, restart from 2

Bisection method (3)

1. let $k = 0$; $[a_0, b_0] = [a, b]$; let $tolx$; $toly$
2. iteration step: $c_{k+1} = a_k + \frac{1}{2}(b_k - a_k) \implies [a_k, c_{k+1}], [c_{k+1}, b_k]$
3. iteration check: stop if $|b_k - a_k| \leq tolx$; return c_{k+1}
4. iteration check: stop if $f(c_{k+1}) \leq toly$; return c_{k+1}
5. $[a_{k+1}, b_{k+1}] = [a_k, c_{k+1}]$ if $f(a_k)f(c_{k+1}) < 0$ else $[c_{k+1}, b_k]$
6. $k = k + 1$, restart from 2

How to determine *toly*?

think $e^{x/10} - 1$ vs $e^{10x} - 1$



$$tol_y \approx f'(\bar{x})tol_x$$

$$f'(\bar{x}) \approx \frac{f(b_k) - f(a_k)}{b_k - a_k} \implies tol_y = \frac{f(b_k) - f(a_k)}{b_k - a_k} tol_x$$

Conditioning

$f(x)$ can be seen as error on \bar{x} :

$$f(x) = f'(\bar{x})(x - \bar{x}) + \underbrace{f(\bar{x})}_{=0} \implies (x - \bar{x}) \approx \frac{f(x)}{f'(\bar{x})}$$

The factor

$$\kappa \approx \frac{1}{|f'(\bar{x})|}$$

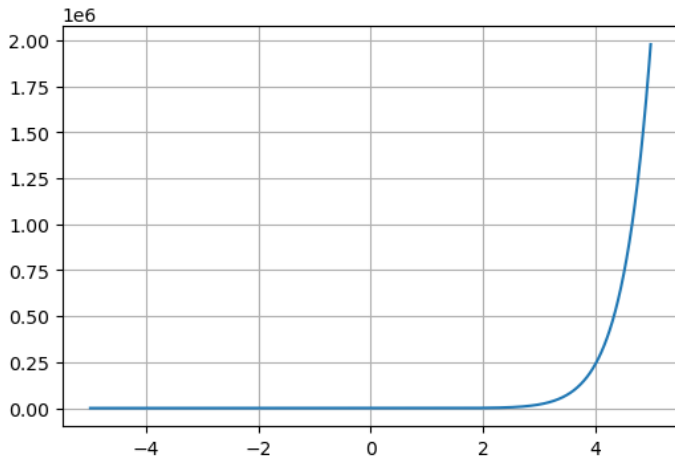
can be seen as an amplification factor of the error on x over f .

When $f'(\bar{x}) \approx 0$, very small changes of f have dramatic effects on the error on x .
toly could become too small to be useful!

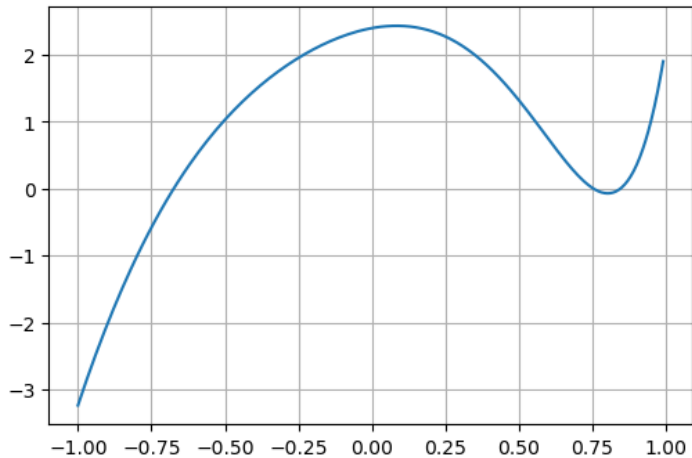
Implementation

```
1 import numpy as np
2 import math
3 from matplotlib import pyplot as plt
4 plt.rcParams['axes.grid'] = True
```

```
1 def f(x):  
2     return 0.8*np.exp(x) * (6*x**5 - 3*x**4 + 2*x**3 - 5*x**2 - 2*x + 3)  
3  
4 x = np.arange(-5,5,0.01)  
5 plt.plot(x, f(x));
```



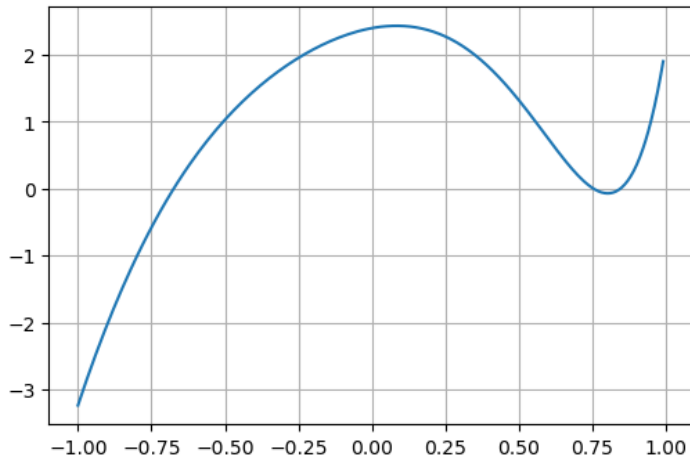
```
1 x = np.arange(-1,1,0.01)
2 plt.plot(x, f(x));
```




```
1 def bisect(a, b, f, counter=0, maxiters=1000):
2     c = a + (b-a)/2
3
4     if counter>maxiters:
5         return c
6
7     if f(c) == 0:
8         return c
9
10    if f(a)*f(c) < 0:
11        return bisect(a,c,f, counter=counter+1, maxiters=maxiters)
12    else:
13        return bisect(c,b,f, counter=counter+1, maxiters=maxiters)
```

```
1 a, b = -1, 1
2 x = bisect(a,b,f)
3 print(x)
4 print(f(x))
5
6 x = np.arange(-1,1,0.01)
7 plt.plot(x, f(x));
```

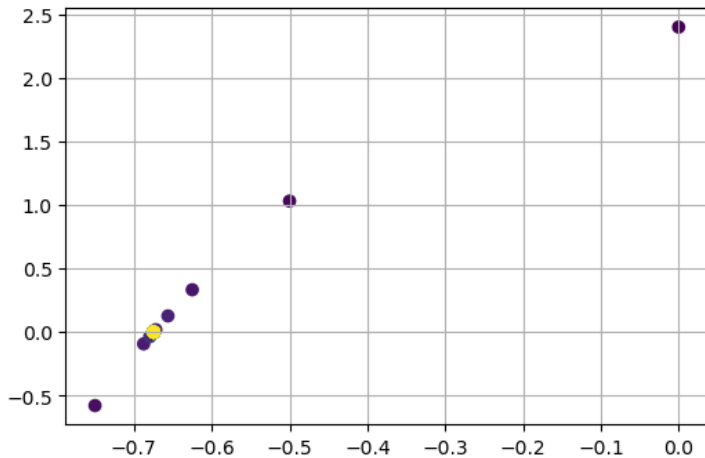
```
1 -0.6746113792482088
2 0.0
```



```
1 def bisection_trace(a, b, f, counter=0, maxiters=1000, trace=None):
2     if trace is None:
3         trace = list()
4
5     c = a + (b-a)/2
6
7     trace.append([counter, a, b, c, f(c)])
8
9     if counter>maxiters:
10        return c, trace
11
12    if f(c) == 0:
13        return c, trace
14    if f(a)*f(c) < 0:
15        return bisection_trace(a,c,f, counter=counter+1, maxiters=maxiters, trace=trace)
16    else:
17        return bisection_trace(c,b,f, counter=counter+1, maxiters=maxiters, trace=trace)
```

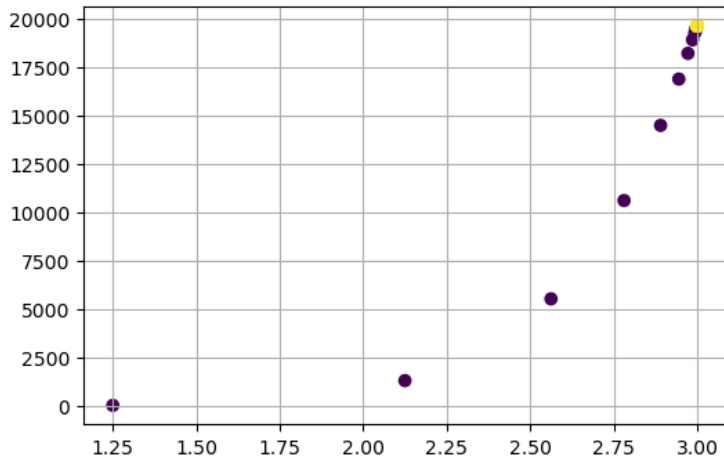
```
1 a, b = -1, 1
2 x, trace = bisect_trace(a,b,f)
3 print(x)
4 print(f(x))
5 print(trace[-1])
6
7 colors = np.linspace(0,1,len(trace))
8 plt.scatter([x[3] for x in trace], [x[4] for x in trace], c=colors, cmap='viridis');
```

```
1 -0.6746113792482088
2 0.0
3 [53, -0.6746113792482089, -0.6746113792482087, -0.6746113792482088, np.float64(0.0)]
```

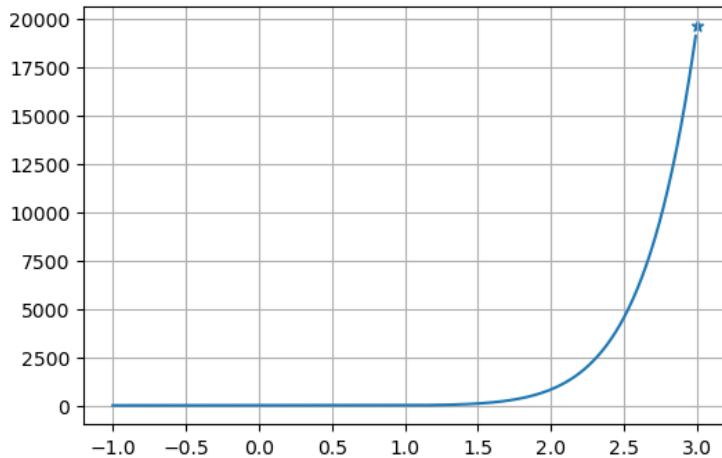


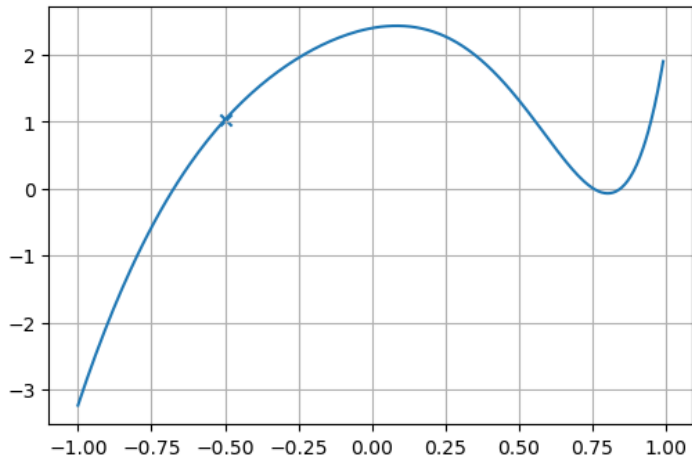
```
1 a, b = -0.5, 3
2 x, trace = bisect_trace(a,b,f)
3 print(x)
4 print(f(x))
5 print(trace[-1])
6
7 colors = np.linspace(0,1,len(trace))
8 plt.scatter([x[3] for x in trace], [x[4] for x in trace], c=colors, cmap='viridis');
```

```
1 3.0
2 19619.552466569716
3 [1001, 3.0, 3, 3.0, np.float64(19619.552466569716)]
```

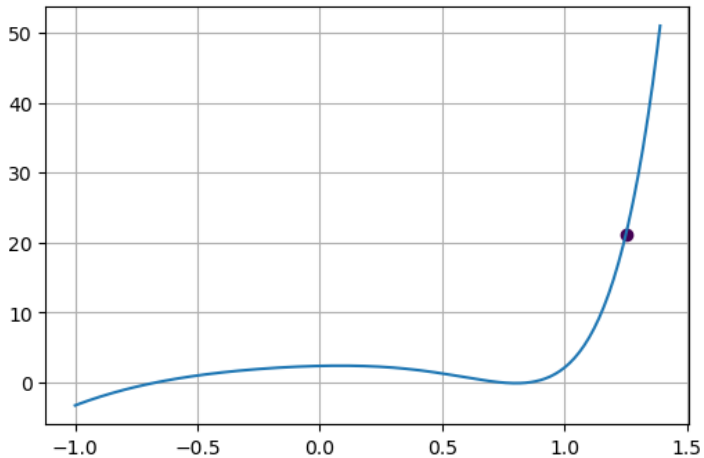



```
1 x = np.arange(-1,3,0.01)
2 plt.plot(x, f(x));
3 plt.scatter(b, f(b), marker='*');
4
5 plt.figure()
6 x = np.arange(-1,1,0.01)
7 plt.plot(x, f(x));
8 plt.scatter(a, f(a), marker='x');
```



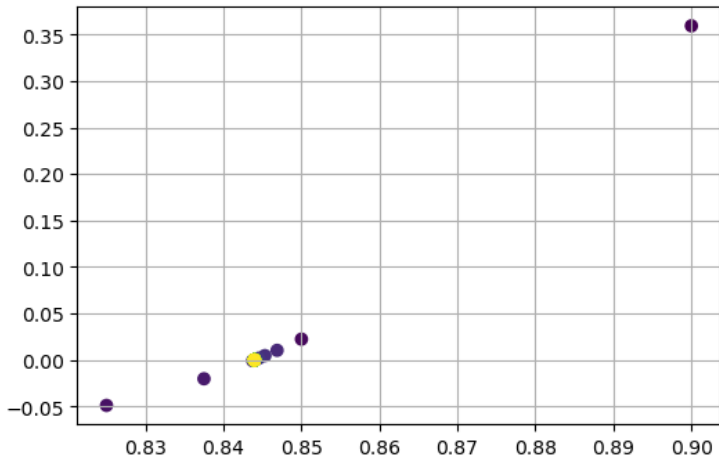


```
1 x = np.arange(-1,1.4,0.01)
2 m = 1
3 plt.plot(x, f(x));
4 colors = np.linspace(0,1,len(trace[:m]))
5 plt.scatter([x[3] for x in trace[:m]], [x[4] for x in trace[:m]], c=colors, cmap='viridis');
```

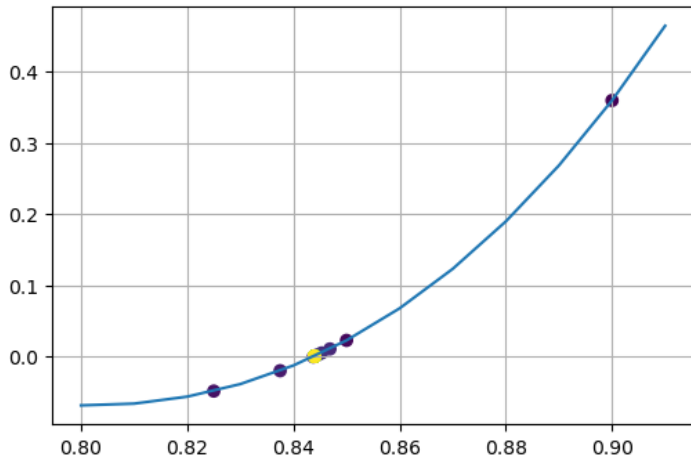


```
1 a, b = 0.8, 1
2 x, trace = bisect_trace(a,b,f)
3 print(x)
4 print(f(x))
5 print(trace[-1])
6
7 colors = np.linspace(0,1,len(trace))
8 plt.scatter([x[3] for x in trace], [x[4] for x in trace], c=colors, cmap='viridis');
```

```
1 0.8439655868551936
2 0.0
3 [44, 0.843965586855188, 0.8439655868551994, 0.8439655868551936, np.float64(0.0)]
```



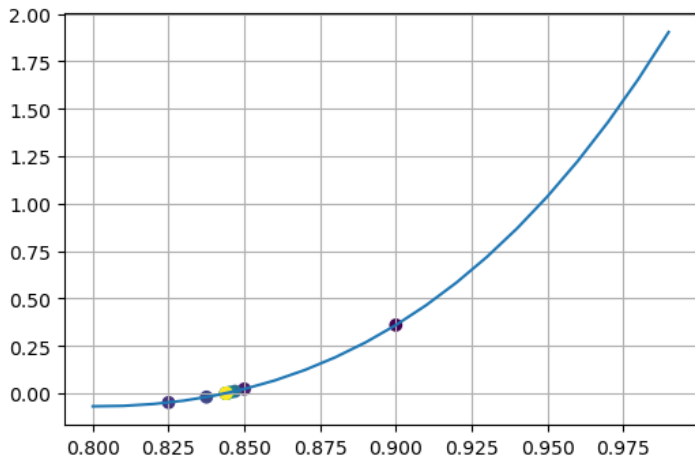
```
1 x = np.arange(0.8,0.92,0.01)
2 plt.plot(x, f(x));
3 colors = np.linspace(0,1,len(trace))
4 plt.scatter([x[3] for x in trace], [x[4] for x in trace], c=colors, cmap='viridis');
```

```
1 def bisect_tol_trace(a, b, f, tol_x=10**-3, toly=None, counter=0, maxiters=1000, trace=None):
2     if trace is None:
3         trace = list()
4
5     if toly is None:
6         toly = abs(tol_x*(f(b)-f(a))/(b-a))
7
8     c = a + (b-a)/2
9
10    trace.append([counter, a, b, c, f(c)])
11
12    if counter>maxiters:
13        return c, trace
14
15    if abs(c-a) <= tol_x or abs(b-c) <= tol_x:
16        return c, trace
17
18    if abs(f(c)) <= toly:
19        return c, trace
20
21    if f(a)*f(c) < 0:
22        return bisect_tol_trace(a,c,f, tol_x=tol_x, toly=toly, counter=counter+1, maxiters=maxiters, trace=trace)
23    else:
24        return bisect_tol_trace(c,b,f, tol_x=tol_x, toly=toly, counter=counter+1, maxiters=maxiters, trace=trace)
```

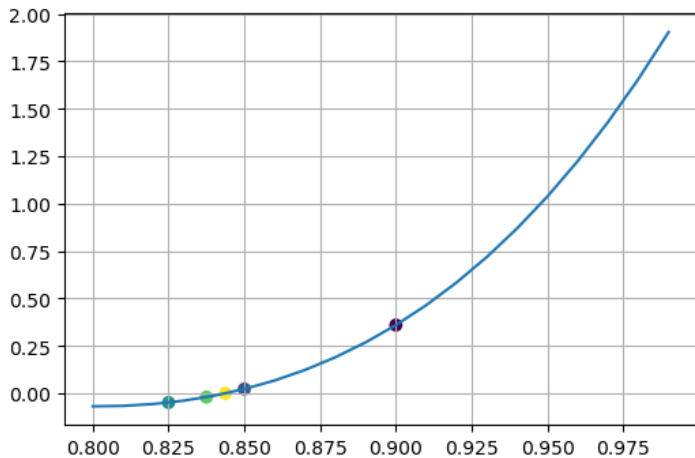
```
1 a, b = 0.8, 1
2 x, trace = bisect_tol_trace(a,b,f, tolx=10**-6)
3 print(x)
4 print(f(x))
5 print(trace[-1])
6
7 colors = np.linspace(0,1,len(trace))
8 plt.scatter([x[3] for x in trace], [x[4] for x in trace], c=colors, cmap='viridis');
9 x = np.arange(0.8,1,0.01)
10 plt.plot(x, f(x));
```

```
1 0.843963623046875
2 -6.724416240101983e-06
3 [14, 0.8439575195312501, 0.8439697265625, 0.843963623046875, np.float64(-6.724416240101983e-06)]
```



```
1 a, b = 0.8, 1
2 x, trace = bisect_tol_trace(a,b,f, tolx=10**-3)
3 print(x)
4 print(f(x))
5 print(trace[-1])
6
7 colors = np.linspace(0,1,len(trace))
8 plt.scatter([x[3] for x in trace], [x[4] for x in trace], c=colors, cmap='viridis');
9 x = np.arange(0.8,1,0.01)
10 plt.plot(x, f(x));
```

```
1 0.8437500000000001
2 -0.0007360523926994878
3 [4, 0.8375000000000001, 0.8500000000000001, 0.8437500000000001, np.float64(-0.0007360523926994878)]
```



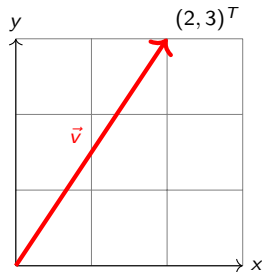
Linear Algebra Recap

Scalar, Vector, Matrix

► scalar: one-dimensional number

► vector: $\mathbf{x} = (x_1, \dots, x_n)^T = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$

► matrix: $A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$



Norm

Length or magnitude of a vector (euclidean norm):

$$\|\mathbf{x}\| = \|\mathbf{x}\|_2 = \sqrt{\sum_i |x_i|^2}$$

In general, a function that holds:

- ▶ Triangle inequality: $f(x + y) \leq f(x) + f(y)$
- ▶ Linearity: $f(kx) = kf(x)$
- ▶ Positive definite: $f(x) = 0 \Rightarrow x = 0$

Other norms example:

- ▶ manhattan norm $\|\mathbf{x}\|_1 = \sum_i |x_i|$
- ▶ p-norm $\|\mathbf{x}\|_p = (\sum_i |x_i|^p)^{1/p}$
- ▶ ∞ -norm: $\max_i (|x_i|)$

Vector Operations

Scale, Sum, Hadamard Product, Scalar Product

$$c\mathbf{x} = \begin{pmatrix} cx_1 \\ \vdots \\ cx_n \end{pmatrix}, \quad \mathbf{x} + \mathbf{y} = \begin{pmatrix} x_1 + y_1 \\ \vdots \\ x_n + y_n \end{pmatrix}, \quad \mathbf{x} \circ \mathbf{y} = \begin{pmatrix} x_1 y_1 \\ \vdots \\ x_n y_n \end{pmatrix}, \quad \mathbf{x} \cdot \mathbf{y} = \mathbf{x}^T \mathbf{y} = \sum_i (x_i y_i)$$

Cross Product (only defined in \mathbb{R}^3):

$$\mathbf{x} \times \mathbf{y} = \|\mathbf{x}\| \|\mathbf{y}\| \sin(\theta) \mathbf{n} = (x_2 y_3 - x_3 y_2) \mathbf{e}_1 + (x_3 y_1 - x_1 y_3) \mathbf{e}_2 + (x_1 y_2 - x_2 y_1) \mathbf{e}_3$$

where \mathbf{n} is the unit vector perpendicular to the plane containing \mathbf{x}, \mathbf{y} , with direction such that $(\mathbf{x}, \mathbf{y}, \mathbf{n})$ is positively oriented (right hand rule)

Matrix Product

$$A \in M^{n \times m}, B \in M^{m \times p} \rightarrow AB \in M^{n \times p}$$

$$AB = \begin{pmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & & \vdots \\ a_{n1} & \cdots & a_{nm} \end{pmatrix} \begin{pmatrix} b_{11} & \cdots & b_{1p} \\ \vdots & & \vdots \\ b_{m1} & \cdots & b_{mp} \end{pmatrix} = \begin{pmatrix} \mathbf{a}_1 \\ \vdots \\ \mathbf{a}_n \end{pmatrix} (\mathbf{b}_1, \dots, \mathbf{b}_p) = \begin{pmatrix} \mathbf{a}_1 \cdot \mathbf{b}_1 & \cdots & \mathbf{a}_1 \cdot \mathbf{b}_p \\ \vdots & & \vdots \\ \mathbf{a}_n \cdot \mathbf{b}_1 & \cdots & \mathbf{a}_n \cdot \mathbf{b}_p \end{pmatrix}$$

Not commutative!

Watch out for bad notation! $\mathbf{a}_i \in \mathbb{R}^{1 \times m}$, $\mathbf{b}_j \in \mathbb{R}^{m \times 1}$

Linear (in)dependency

$$A\mathbf{x} = \mathbf{0} \iff \mathbf{x} = \mathbf{0}$$

$$\rightarrow \exists A^{-1}A = \mathbf{I} = \begin{pmatrix} 1 & & \\ & \ddots & \\ & & 1 \end{pmatrix}$$

$$A\mathbf{x} = \mathbf{b} \rightarrow \mathbf{x} = A^{-1}\mathbf{b}$$

Function Approximation

Function approximation

Given $f : [a, b] \subset \mathbb{R} \rightarrow \mathbb{R}$ define $g : [a, b] \subset \mathbb{R} \rightarrow \mathbb{R}$ such that: $g([a, b]) \approx f([a, b])$

- ▶ f too expensive to compute
- ▶ f unknown, but can be measured on a grid like $a < x_0 < x_1 < \dots < x_n < b$
- ▶ f is noisy or has random components

Polynomial interpolation

Given (x_i, f_i) , $i = 0, 1, \dots, n$, $f_i \equiv x_i$:
find a polynomial $p(x) \in \Pi$ such that $p(x_i) = f_i$

Canonical Base

Theorem ($\exists! p(x) \in \Pi_n$ such that $p(x_i) = f_i, i = 0, \dots, n$)

$$p(x) = \sum_{k=0}^n a_k x^k \implies V\mathbf{a} = \mathbf{f}$$

where:

$$V = \begin{pmatrix} x_0^0 & x_0^1 & \cdots & x_0^n \\ x_1^0 & x_1^1 & \cdots & x_1^n \\ \vdots & \vdots & & \vdots \\ x_n^0 & x_n^1 & \cdots & x_n^n \end{pmatrix}, \mathbf{a} = \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix}, \mathbf{f} = \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_n \end{pmatrix}$$

V is a Vandermonde matrix: $\det(V) = \prod_{i>j} (x_i - x_j) \neq 0 \iff x_i \neq x_j \forall i, j$

$$\mathbf{a} = V^{-1}\mathbf{f}$$

Conditioning

Vandermonde matrices $\in M^{n \times n}$ have bad conditioning already for small n !

$$|p(x) - \tilde{p}(x)| = \kappa \max_i |f_i - \tilde{f}_i|, \quad \kappa \gg 1$$

It's complicated, see <http://arxiv.org/abs/1504.02118v3> and

https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://osnadoes.ub.uni-osnabrueck.de/bitstream/urn:nbn:de:gbv:700-202103194121/1/thesis_nagel.pdf&ved=2ahUKEwjItvWk1M2JAXXB3QIHHQgnD_MQFnoECBUQAQ&usg=A0vVaw30Fier08PagrHVx9c8XiKj to get an idea!

Lagrange form

$$L_{kn}(x) = \prod_{j=0, j \neq k}^{n-1} \frac{x - x_j}{x_k - x_j}, \quad k = 0, \dots, n-1$$

Observation

$$L_{kn}(x_i) = \delta_{ki} : \quad 1 \iff k = i, \quad 0 \text{ otherwise}$$

$$L_{kn} \equiv \Pi_n, \quad \text{linearly independent}$$

$$p(x) = \sum_{k=0}^{n-1} f_k L_{kn}(x)$$

Problem Conditioning

Assumptions:

- ▶ $\{x_i\}$ can be usually chosen arbitrarily
- ▶ error on $\{x_i\}$ usually small
- ▶ $\{x_i\}$ usually common to many functions in real world (quantities measured at regular intervals)
- ▶ error on $\{f_i\}$ usually bigger

Consider:

$$p(x) = \sum_{k=0}^n f_k L_{kn}(x), \quad \tilde{p}(x) = \sum_{k=0}^n \tilde{f}_k L_{kn}(x)$$

Problem Conditioning (2)

$$\begin{aligned}
 |p(x) - \tilde{p}(x)| &= \left| \sum_{k=0}^n f_k L_{kn}(x) - \sum_{k=0}^n \tilde{f}_k L_{kn}(x) \right| \\
 &= \left| \sum_{k=0}^n (f_k - \tilde{f}_k) L_{kn}(x) \right| \leq \sum_{k=0}^n |(f_k - \tilde{f}_k)| \cdot |L_{kn}(x)| \\
 &\leq \left(\sum_{k=0}^n |L_{kn}(x)| \right) \max_k |f_k - \tilde{f}_k| = \Lambda_n \max_k |f_k - \tilde{f}_k|
 \end{aligned}$$

- ▶ Λ_n depends only on $[a, b]$ and $\{x_i\}$
- ▶ $\max_k |f_k - \tilde{f}_k|$ is an input error

Problem Conditioning (3)

- ▶ if $[a, b]$ can be mapped in $[c, d]$ with a linear transformation, Λ_n does not change for the two spaces
- ▶ $\Lambda_n \geq O(\log n) \rightarrow \infty$ for $n \rightarrow \infty$
- ▶ Equidistant $\{x_i\}$ generate an approximately exponential sequence $\{\Lambda_n\}$ for $n \rightarrow \infty$

Piecewise interpolation

Define $q : [a, b] \subset \mathbb{R} \rightarrow \mathbb{R}$ such that:

$$q(x_i) = f(x_i) \tag{1}$$

$$q(x) = q_i(x) \in \Pi_k, \quad x_i \leq x \leq x_{i+1}, \quad i = 0, \dots, n$$

We also impose:

$$q_i^{(s)}(x_{i+1}) = q_{i+1}^{(s)}(x_{i+1}), \quad s = 0, \dots, k-1, \quad i = 1, \dots, n-1$$

- ▶ $q(x) \in \Pi_k$ over $[a, b]$ is called *spline* of degree k , interpolating f when subject to (1)
- ▶ $q(x) \in \mathcal{C}^k$
- ▶ $q'(x) \in \mathcal{C}^{k-1}$

Spline

Observation

$q_i(x) \in \Pi_k$ has $k + 1$ degrees of freedom

Given a partition with $n + 1$ points x_0, \dots, x_n , the spline $q(x)$ has $k + n$ degrees of freedom

Each of the n polynomials q_i has $k + 1$ degrees of freedom, for a total of $n(k + 1)$ free coefficients. The k derivative continuity condition impose $k(n - 1)$ parameters on the $n - 1$ internal points x_1, \dots, x_{n-1} , for a total of $k + n$ free parameters left.

- ▶ A spline of degree 1 (joint line) has $n + 1$ free parameters: exactly the interpolation points!
- ▶ For degree > 1 , one has to impose $k - 1$ additional conditions.

Cubic splines

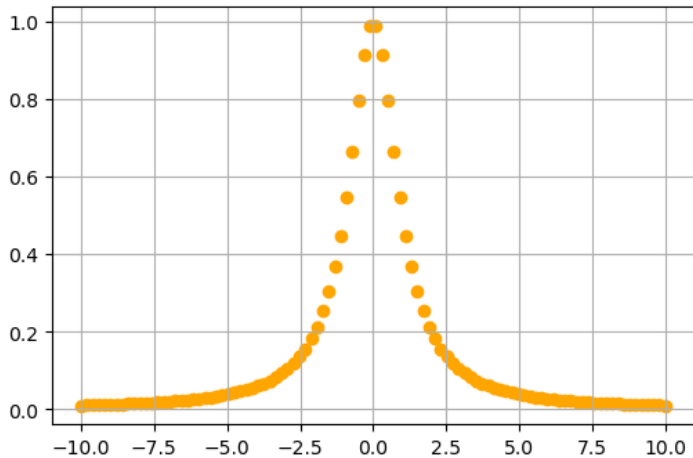
$q_i(x) \in \Pi_3$: $3 - 1 = 2$ additional conditions to impose, on $[a = x_0, x_1, \dots, x_n = b]$

- ▶ Natural spline: $q''(a) = q''(b) = 0$
- ▶ Complete spline: $q'(a) = f'(a)$, $q'(b) = f'(b)$
- ▶ Periodic spline: $q'(a) = q'(b)$, $q''(a) = q''(b)$
- ▶ Not-a-knot: $q_0'''(x_1) = q_1'''(x_1)$, $q_{n-1}'''(x_{n-1}) = q_n'''(x_{n-1})$

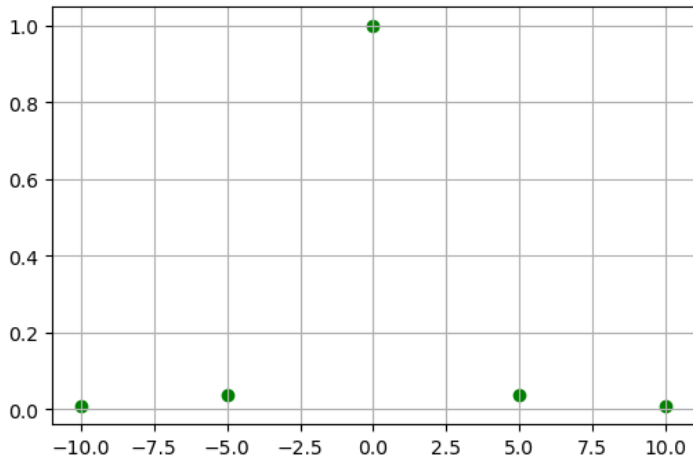
Implementation

```
1 import numpy as np
2 import math
3 from scipy import interpolate
4 from matplotlib import pyplot as plt
5 plt.rcParams['axes.grid'] = True
```

```
1 def f(x):  
2     return 1/(1+x**2)  
3  
4 X = np.linspace(-10,10,100 )  
5 Y = f(X)  
6  
7 plt.scatter(X,Y, color='orange');
```



```
1 X = np.linspace(-10,10,5 )
2 Y = f(X)
3
4 plt.scatter(X,Y, color='green');
```



```
1 def p(x, a):  
2     return sum([a[k]*x**k for k in range(len(a))])
```

```
1 print(X)
2 V = np.array([X**i for i in range(len(X))] ).transpose()
3 print(V)
```

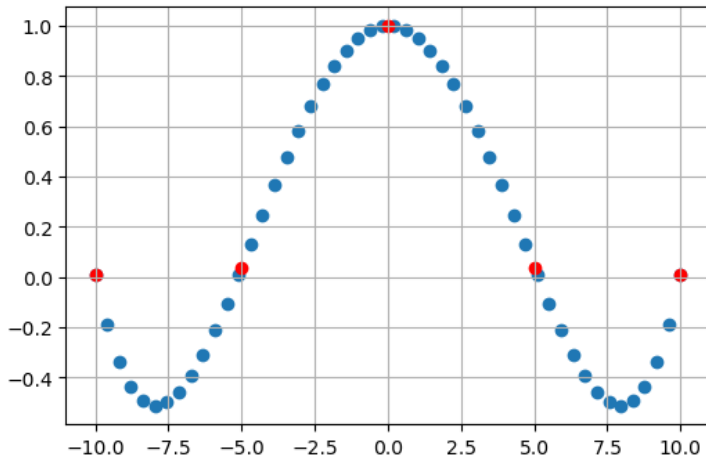
```
1 [-10.  -5.   0.   5.  10.]
2 [[ 1.00e+00 -1.00e+01  1.00e+02 -1.00e+03  1.00e+04]
3  [ 1.00e+00 -5.00e+00  2.50e+01 -1.25e+02  6.25e+02]
4  [ 1.00e+00  0.00e+00  0.00e+00  0.00e+00  0.00e+00]
5  [ 1.00e+00  5.00e+00  2.50e+01  1.25e+02  6.25e+02]
6  [ 1.00e+00  1.00e+01  1.00e+02  1.00e+03  1.00e+04]]
```



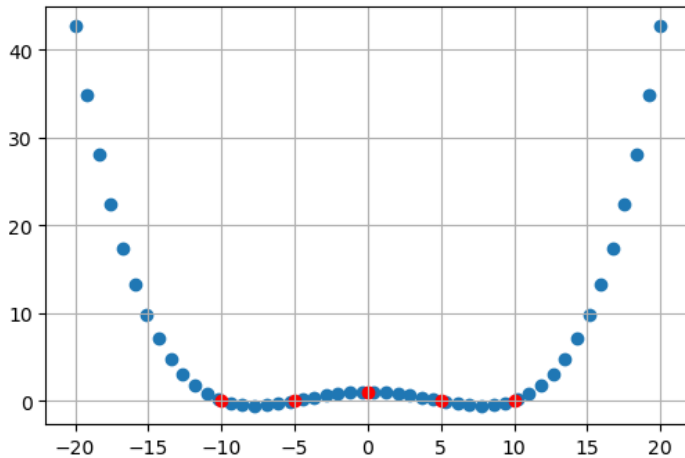
```
1 a = np.linalg.inv(V).dot(Y)
2 print(a)

1 [ 1.00000000e+00  2.29006399e-21 -4.79817212e-02 -3.55943931e-22
2   3.80807312e-04]
```

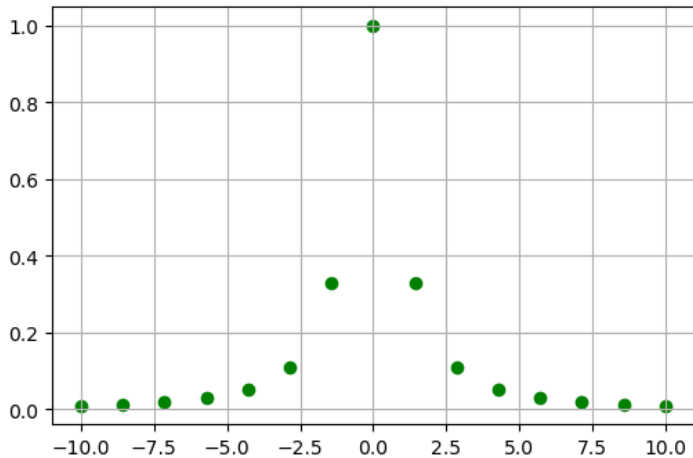
```
1 x = np.linspace(-10,10,50)
2 y = p(x, a)
3 plt.scatter(x,y);
4 plt.scatter(X,Y, color='red');
```



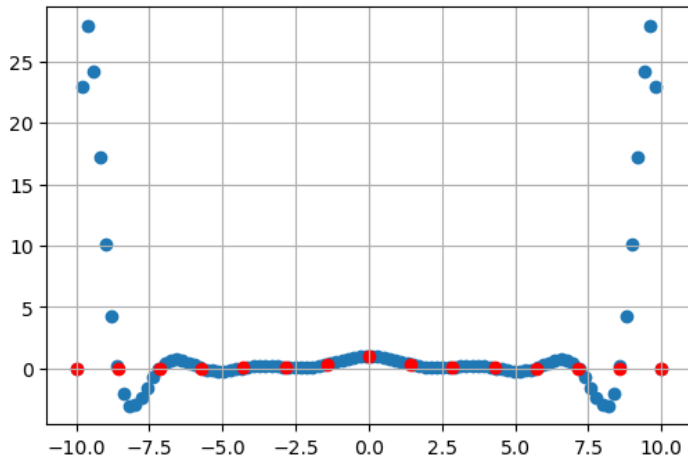
```
1 x = np.linspace(-20,20,50)
2 y = p(x, a)
3 plt.scatter(x,y);
4 plt.scatter(X,Y, color='red');
```



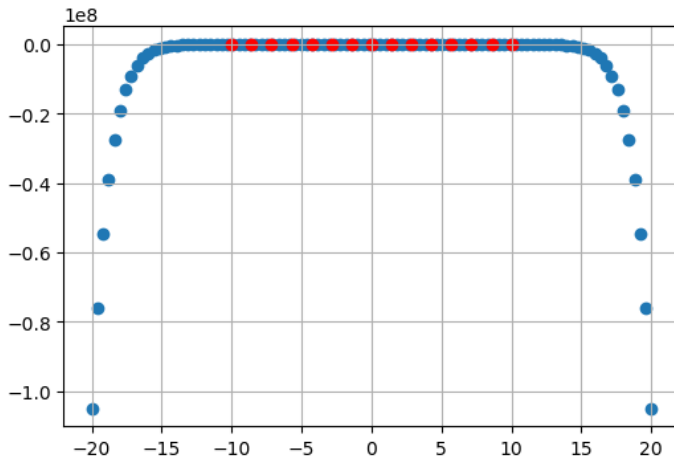
```
1 X = np.linspace(-10,10,15)
2 Y = f(X)
3
4 plt.scatter(X,Y, color='green');
```



```
1 V = np.array([X**i for i in range(len(X))] ).transpose()
2 a = np.linalg.inv(V).dot(Y)
3 x = np.linspace(-10,10,100)
4 y = p(x, a)
5 plt.scatter(x,y);
6 plt.scatter(X,Y, color='red');
```

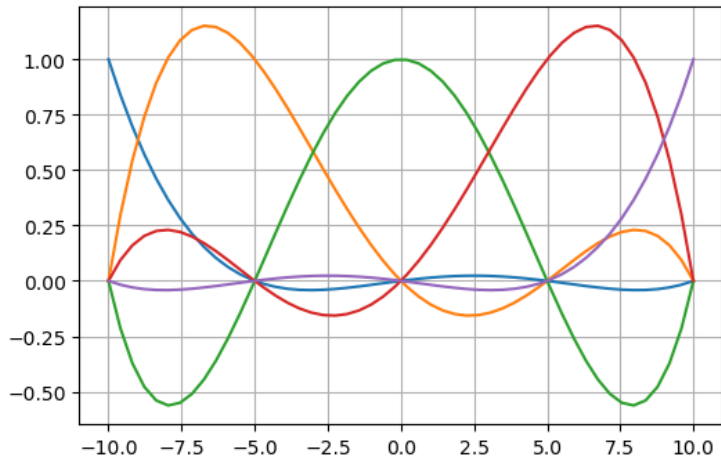



```
1 x = np.linspace(-20,20,100)
2 y = p(x, a)
3 plt.scatter(x,y);
4 plt.scatter(X,Y, color='red');
```



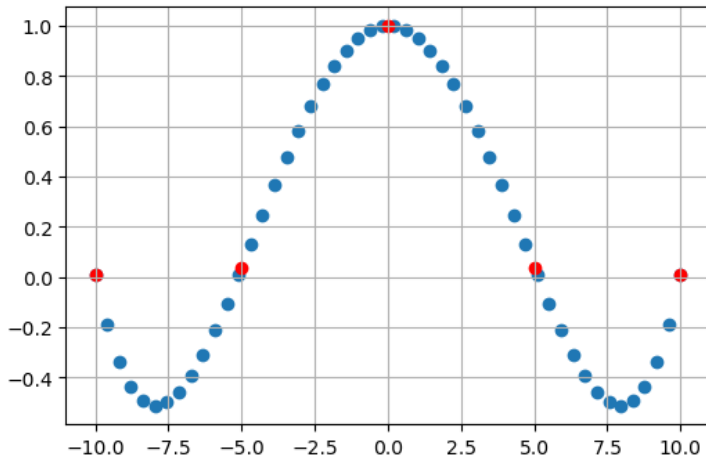
```
1 def Lkn(Xi,k,x):
2     x_k = Xi[k]
3     X = list(Xi[:k])+list(Xi[k+1:])
4     res = 1.
5     for x_j in X:
6         res *= (x-x_j)/(x_k-x_j)
7     return res
8
```

```
1 X = np.linspace(-10, 10, 5)
2 x = np.linspace(-10,10, 50)
3
4 for k in range(0,5):
5     Y = Lkn(X,k,x)
6     plt.plot(x,Y);
```



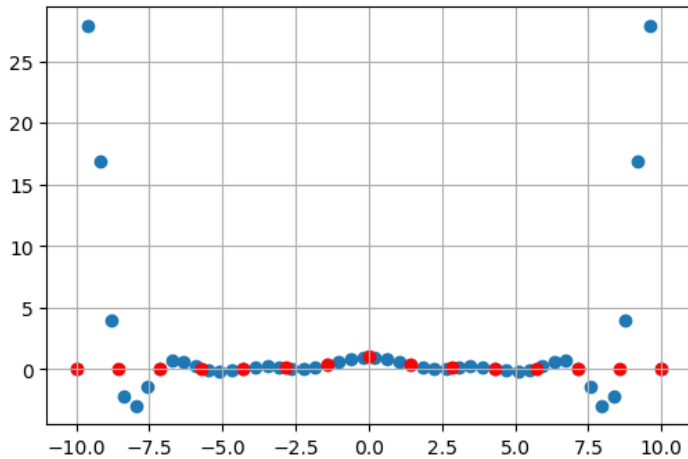
```
1 def pl(x,Xi,f):  
2     return sum([f[k]*Lkn(Xi,k,x) for k in range(len(f))])
```

```
1 X = np.linspace(-10, 10, 5)
2 Y = f(X)
3
4 x = np.linspace(-10,10,50)
5 y = pl(x, X, Y)
6 plt.scatter(x,y);
7 plt.scatter(X,Y, color='red');
```

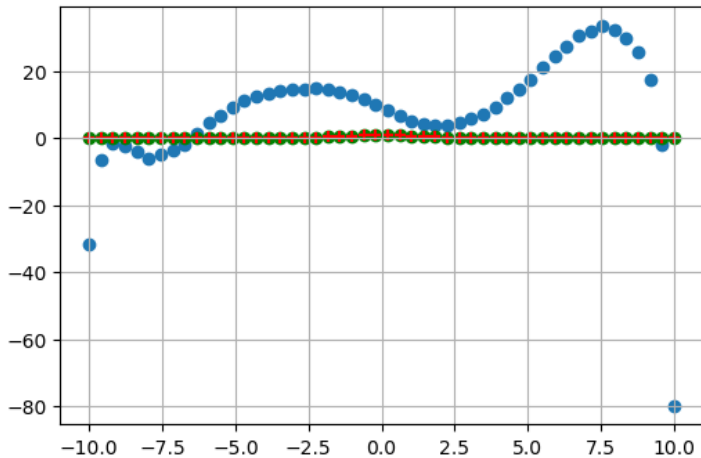



```
1 X = np.linspace(-10, 10, 15)
2 Y = f(X)
```

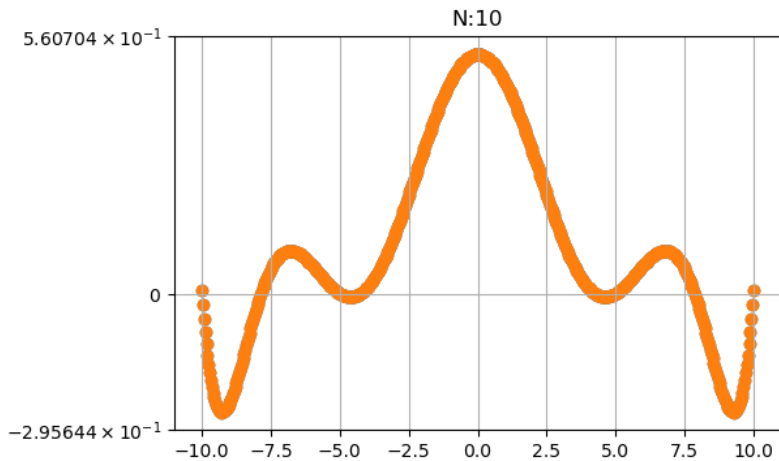
```
1 x = np.linspace(-10,10,50)
2 y = pl(x, X, Y)
3 plt.scatter(x,y);
4 plt.scatter(X,Y, color='red');
```

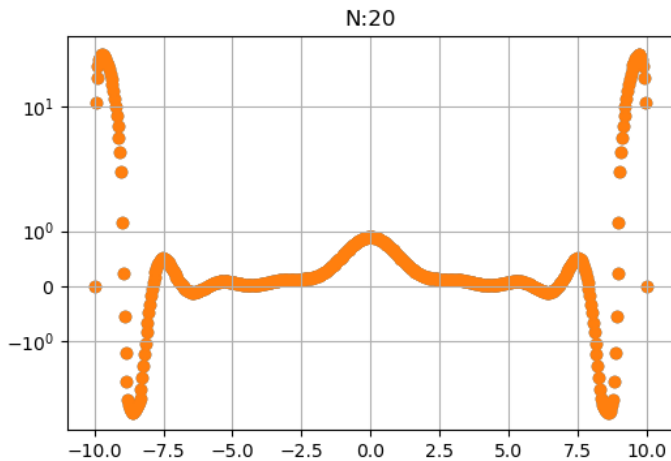


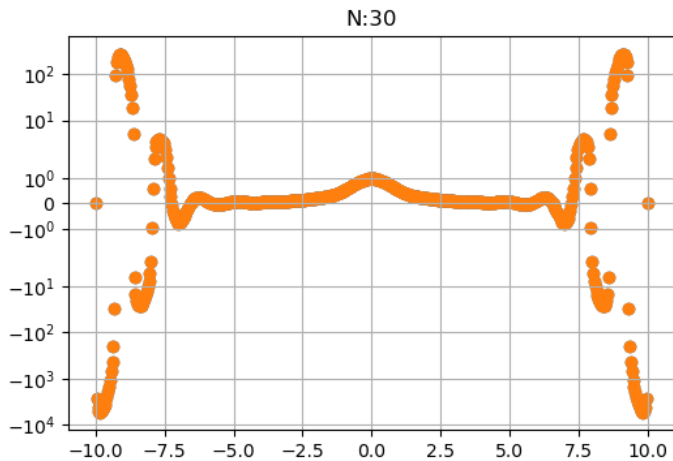
```
1 X = np.linspace(-10, 10, 50)
2 Y = f(X)
3
4 x = np.linspace(-10,10,50)
5
6 V = np.array([X**i for i in range(len(X))] ).transpose()
7 a = np.linalg.inv(V).dot(Y)
8 yp = p(x, a)
9
10 yl = pl(x, X, Y)
11
12 plt.scatter(x,yp);
13 plt.scatter(x,yl, color='green');
14 plt.scatter(X,Y, color='red', marker='+');
```

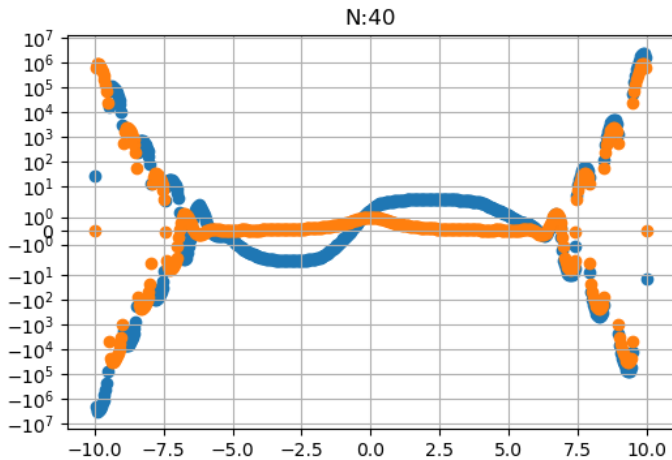


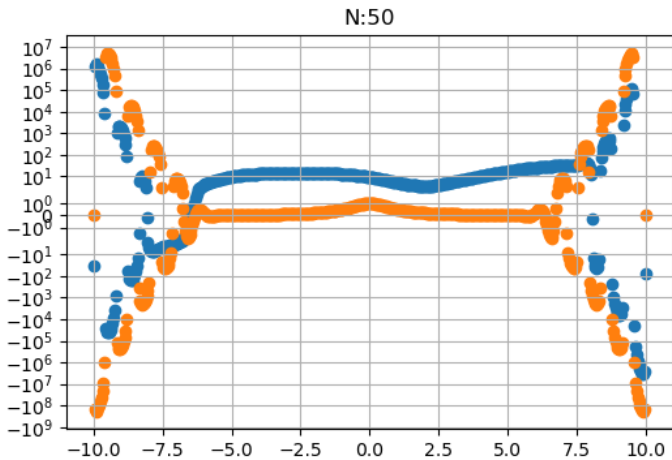
```
1  for n in range(10,100, 10):
2      X = np.linspace(-10, 10, n)
3      Y = f(X)
4
5      x = np.linspace(-10,10,500)
6
7      V = np.array([X**i for i in range(len(X))] ).transpose()
8      a = np.linalg.inv(V).dot(Y)
9      yp = p(x, a)
10
11     yl = pl(x, X, Y)
12     plt.figure()
13
14     plt.scatter(x,yp);
15     plt.scatter(x,yl);
16     plt.yscale('symlog')
17     plt.title('N: '+str(n))
```

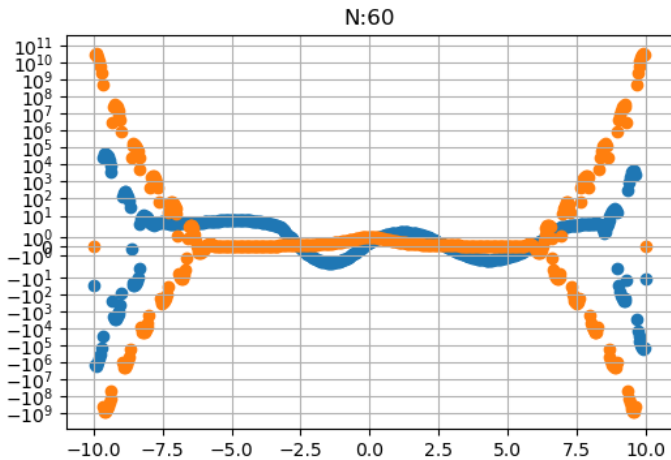


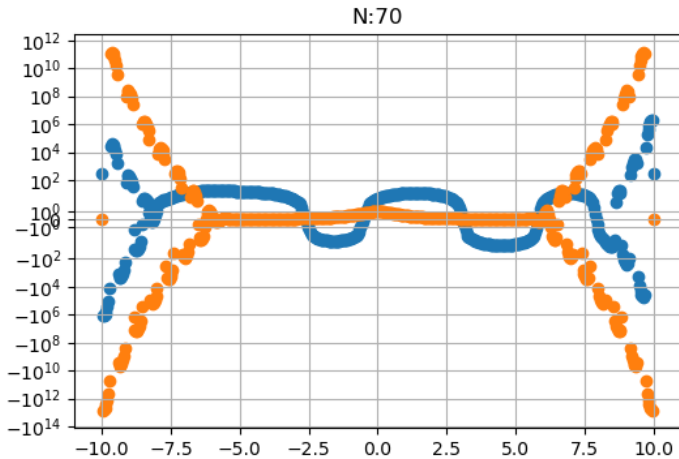


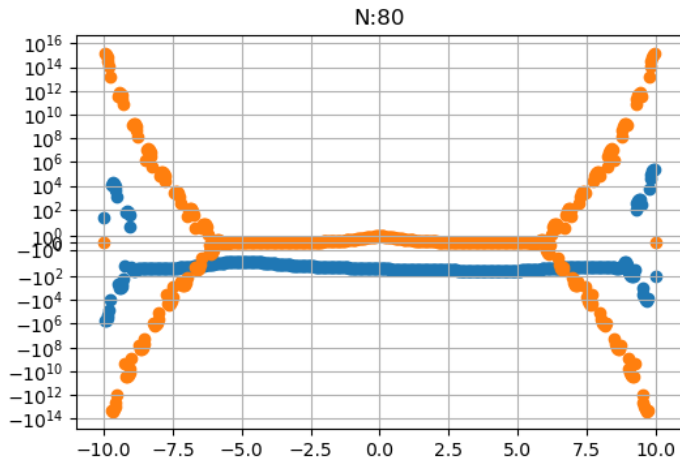


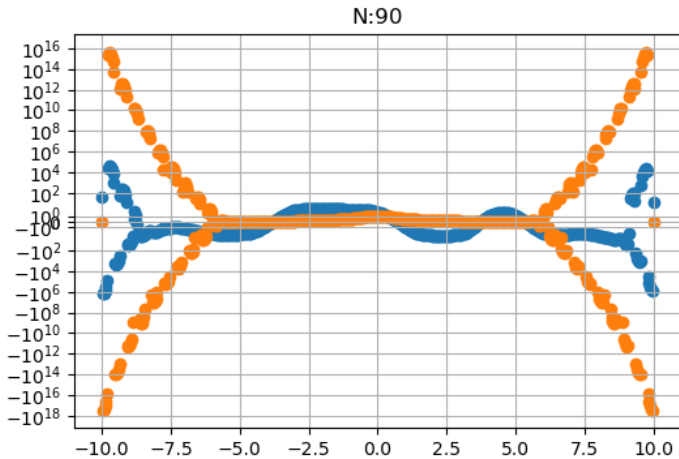




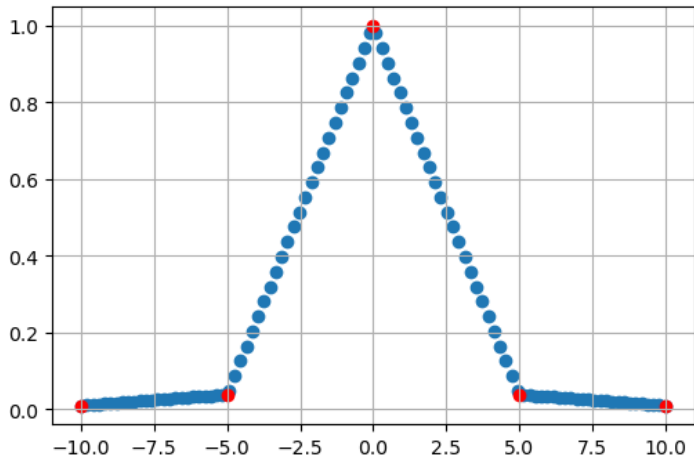




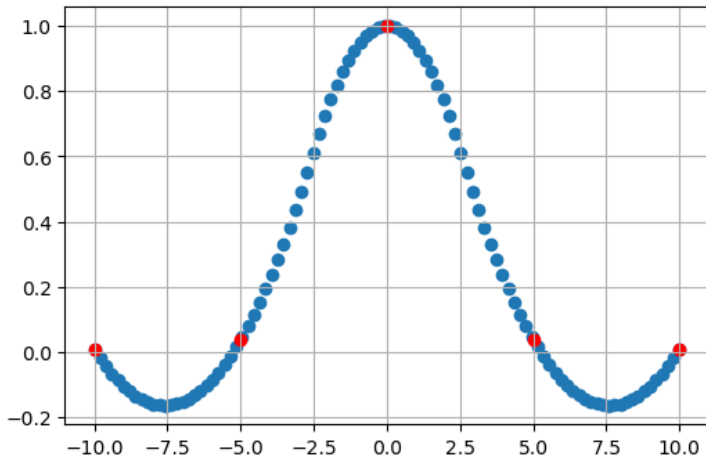




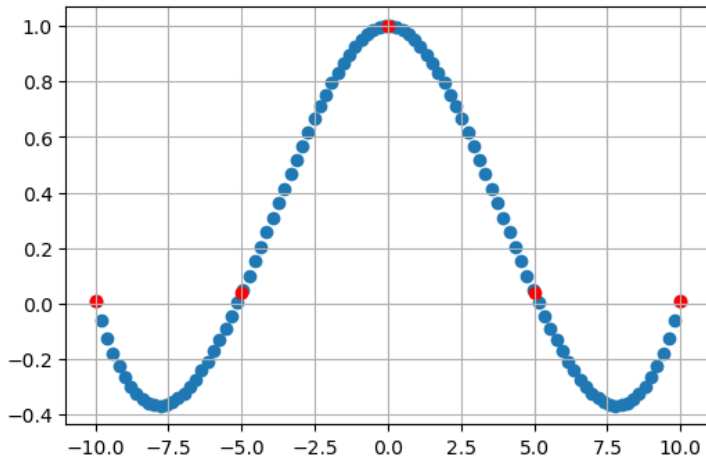

```
1 X = np.linspace(-10, 10, 5)
2 Y = f(X)
3
4 x = np.linspace(-10,10,100)
5 spline = interpolate.interp1d(X,Y, kind='linear')
6 y = spline(x)
7 plt.scatter(x,y);
8 plt.scatter(X,Y, color='red');
```



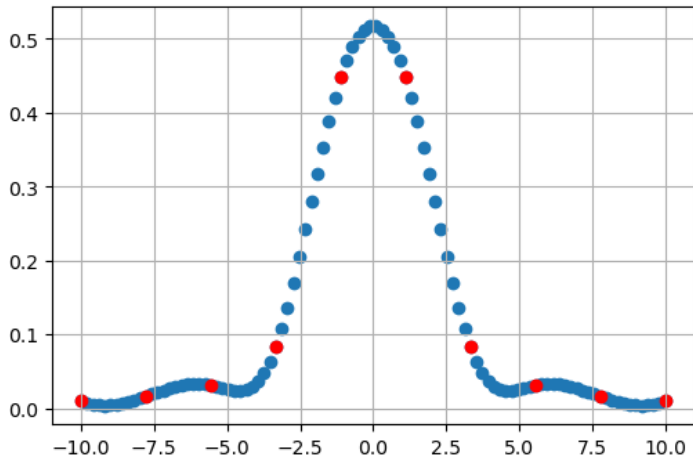
```
1 x = np.linspace(-10,10,100)
2 spline = interpolate.interp1d(X,Y, kind='quadratic')
3 y = spline(x)
4 plt.scatter(x,y);
5 plt.scatter(X,Y, color='red');
```



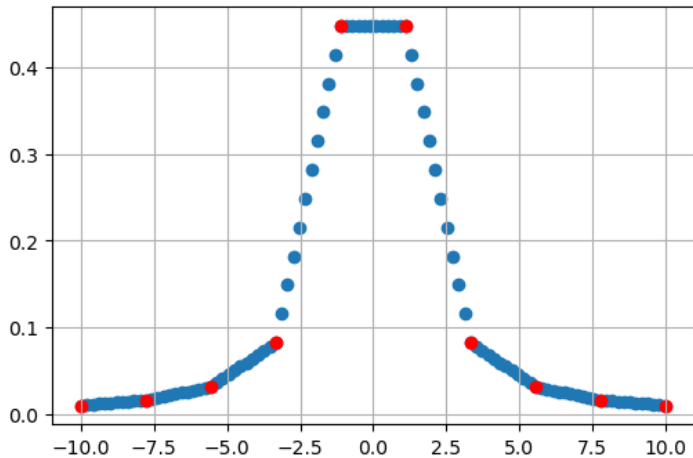
```
1 x = np.linspace(-10,10,100)
2 spline = interpolate.interp1d(X,Y, kind='cubic')
3 y = spline(x)
4 plt.scatter(x,y);
5 plt.scatter(X,Y, color='red');
```



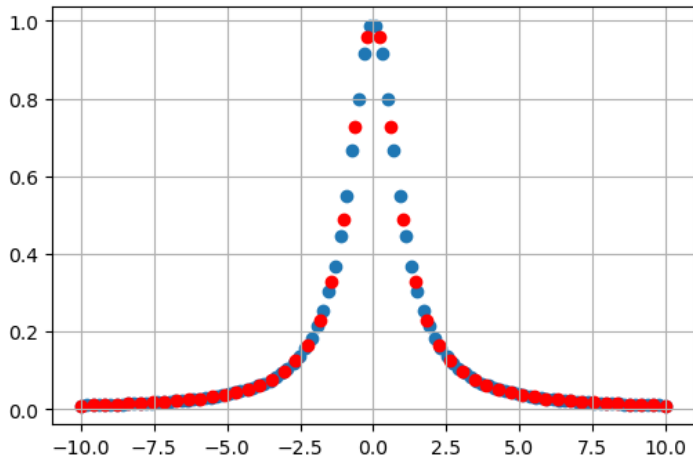
```
1 X = np.linspace(-10, 10, 10)
2 Y = f(X)
3 x = np.linspace(-10,10,100)
4 spline = interpolate.interp1d(X,Y, kind='cubic')
5 y = spline(x)
6 plt.scatter(x,y);
7 plt.scatter(X,Y, color='red');
```



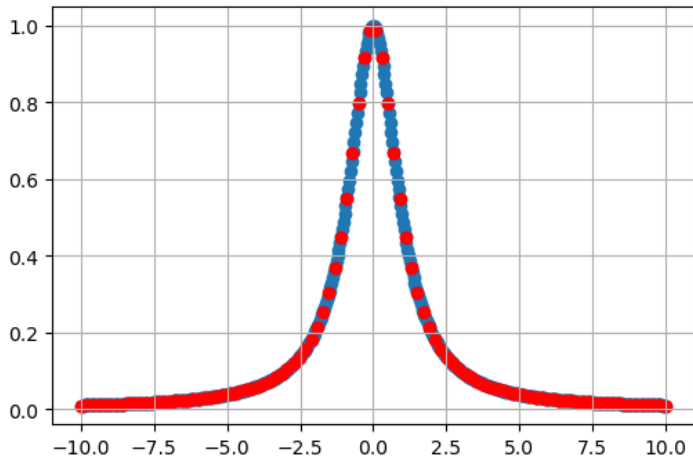

```
1 spline = interpolate.interp1d(X,Y, kind='linear')
2 y = spline(x)
3 plt.scatter(x,y);
4 plt.scatter(X,Y, color='red');
```



```
1 X = np.linspace(-10, 10, 50)
2 Y = f(X)
3 x = np.linspace(-10,10,100)
4 spline = interpolate.interp1d(X,Y, kind='cubic')
5 y = spline(x)
6 plt.scatter(x,y);
7 plt.scatter(X,Y, color='red');
```



```
1 X = np.linspace(-10, 10, 100)
2 Y = f(X)
3 x = np.linspace(-10,10,500)
4 spline = interpolate.interp1d(X,Y, kind='cubic')
5 y = spline(x)
6 plt.scatter(x,y);
7 plt.scatter(X,Y, color='red');
```



Least squares approximation

Least square approximation

Data in the form of noisy measurements:

$$(x_i, y_i), \quad i = 0, \dots, n$$

Let:

$$\mathbf{y} = (y_0, \dots, y_n)^T, \quad \mathbf{x} = (x_0, \dots, x_n)^T, \quad \mathbf{z} = f(\mathbf{x}) = (z_0, \dots, z_n)^T$$

where f is the approximating function (be it polynomial, spline, etc.). We want to minimize:

$$\|\mathbf{y} - \mathbf{z}\|_2^2 = \sum_{i=0}^n |y_i - z_i|^2$$

Repeated measurements

Multiple measurements of the same point (example: $R = V/I$ multiple measures of I for each V)

$$(x_i^{[k]}, y_i^{[k]})$$

define:

$$\mathbf{x} = (x_0, \dots, x_n)^T,$$

where $x_i \neq x_j$, and:

$$\mathbf{y} = \begin{pmatrix} \sum_{j=0}^k y_0^{[j]} / k \\ \vdots \\ \sum_{j=0}^k y_n^{[j]} / k \end{pmatrix}$$

Overdetermined Polynomial Approximation

Given $\mathbf{y} = (y_0, \dots, y_n)^T$, $\mathbf{x} = (x_0, \dots, x_n)^T$

$$p(x) \in \Pi_d = \sum_{k=0}^d a_k x^k \implies V\mathbf{a} = \mathbf{y}$$

where:

$$V = \begin{pmatrix} x_0^0 & x_0^1 & \cdots & x_0^d \\ x_1^0 & x_1^1 & \cdots & x_1^d \\ \vdots & \vdots & & \vdots \\ x_n^0 & x_n^1 & \cdots & x_n^d \end{pmatrix}, \mathbf{a} = \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_d \end{pmatrix}, \mathbf{y} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix}$$

with $n \gg d$. $V \in \mathbb{R}^{n \times d}$ is singular and not invertible!

$$V\mathbf{a} = \mathbf{y} \quad ?$$

Observation

V has maximum rank $(m+1)$: the first $m+1$ rows is a Vandermonde matrix, which is non-singular.

QR Decomposition

$$V\mathbf{a} = \mathbf{y} + \mathbf{r}$$

such that $\|\mathbf{r}\|_2^2 = \|V\mathbf{a} - \mathbf{y}\|_2^2$ is minimized (least squares).

$$V = QR = Q \begin{pmatrix} \hat{R} \\ O \end{pmatrix}, \quad V \in \mathbb{R}^{n \times d}, Q \in \mathbb{R}^{n \times n}, R \in \mathbb{R}^{n \times d}$$

where Q orthogonal ($Q^T Q = Q Q^T = I$), $\hat{R} \in \mathbb{R}^{d \times d}$ upper triangular and nonsingular.

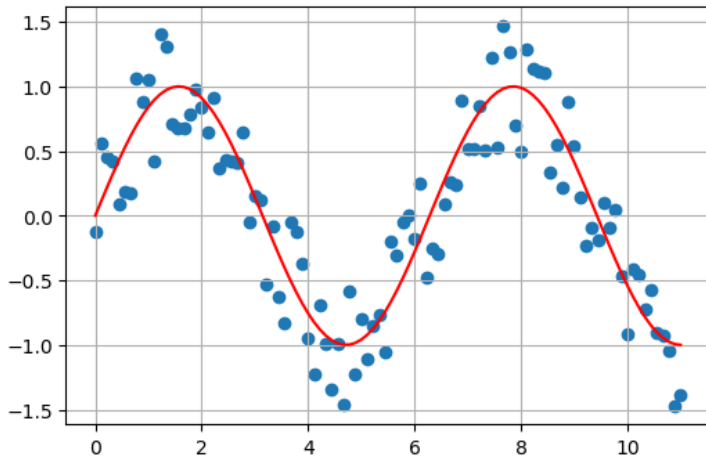
$$\|\mathbf{r}\|_2^2 = \|V\mathbf{a} - \mathbf{y}\| = \|QR\mathbf{a} - \mathbf{y}\| = \|Q(R\mathbf{a} - Q^T \mathbf{y})\| = \|Q(R\mathbf{a} - \mathbf{g})\| = \left\| Q \begin{pmatrix} \hat{R} \\ O \end{pmatrix} \mathbf{a} - \begin{pmatrix} \mathbf{g}_u \\ \mathbf{g}_l \end{pmatrix} \right\|$$

$$\mathbf{a} = \hat{R}^{-1} \mathbf{g}_u$$

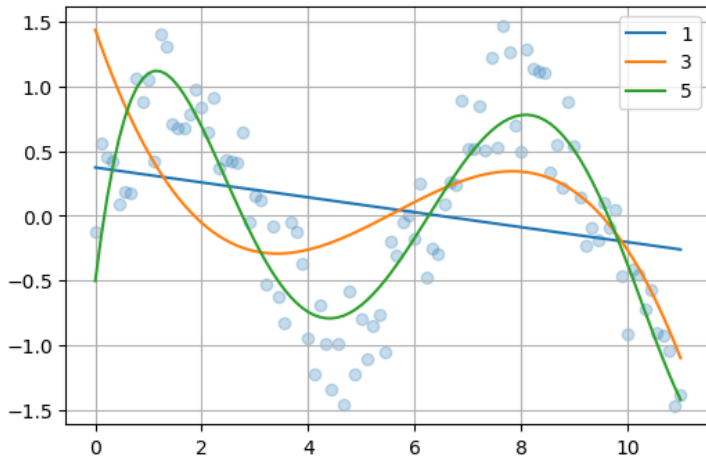
Implementation

```
1 import numpy as np
2 import math
3 from scipy import interpolate, optimize
4 from matplotlib import pyplot as plt
5 plt.rcParams['axes.grid'] = True
```

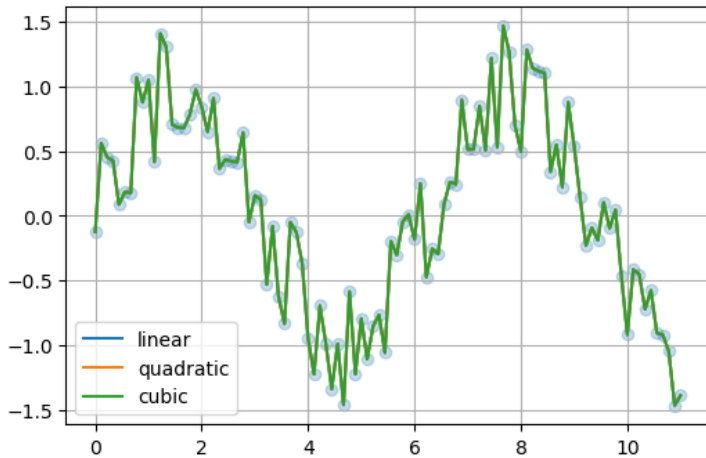
```
1  np.random.seed(42)
2  Xmin, Xmax = 0, 3.5*np.pi
3  X = np.linspace(Xmin, Xmax, 100)
4  Y = np.sin(X) + np.random.random(len(X))-0.5
5  plt.scatter(X,Y);
6  plt.plot(X, np.sin(X), color='red');
```



```
1 f1 = np.polynomial.Polynomial.fit(X,Y,1)
2 f3 = np.polynomial.Polynomial.fit(X,Y,3)
3 f5 = np.polynomial.Polynomial.fit(X,Y,5)
4 plt.scatter(X,Y, alpha=0.25)
5 plt.plot(X,f1(X), label='1')
6 plt.plot(X,f3(X), label='3')
7 plt.plot(X,f5(X), label='5')
8 plt.legend();
```

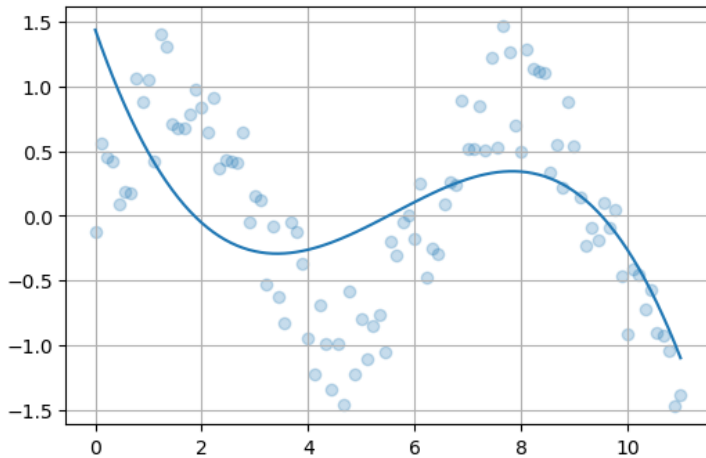



```
1 flin = interpolate.interp1d(X,Y, kind='linear')
2 fquad = interpolate.interp1d(X,Y, kind='quadratic')
3 fcub = interpolate.interp1d(X,Y, kind='cubic')
4
5 plt.scatter(X,Y, alpha=0.25)
6 plt.plot(X,flin(X), label='linear')
7 plt.plot(X,fquad(X), label='quadratic')
8 plt.plot(X,fcub(X), label='cubic')
9 plt.legend();
```



```
1 def curve(x, a,b,c,d):
2     return a*x**3+b*x**2+c*x+d
3 popt, pcov = optimize.curve_fit(curve, X,Y)
4 print(popt)
5 print(pcov)
6
7 plt.scatter(X,Y, alpha=0.25)
8 plt.plot(X, curve(X, *popt));
```

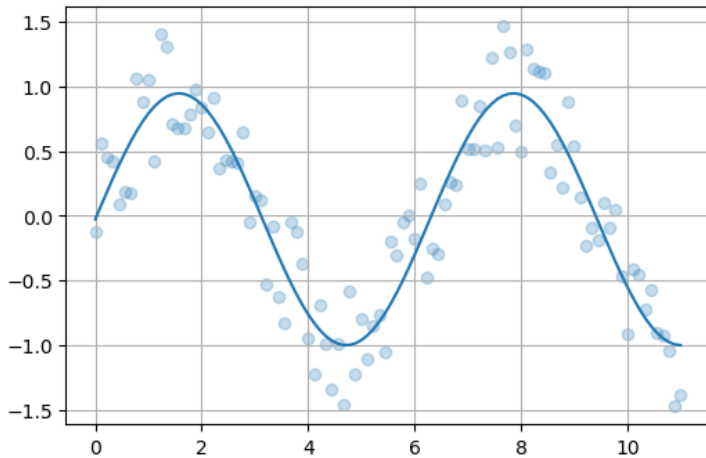
```
1 [-0.01476874  0.24923265 -1.18574687  1.43703208]
2 [[ 5.78890525e-06 -9.54785064e-05  4.17829794e-04 -3.73205142e-04]
3  [-9.54785064e-05  1.62062692e-03 -7.39573436e-03  7.07028021e-03]
4  [ 4.17829794e-04 -7.39573436e-03  3.60802771e-02 -3.90695150e-02]
5  [-3.73205142e-04  7.07028021e-03 -3.90695150e-02  5.75807861e-02]]
```



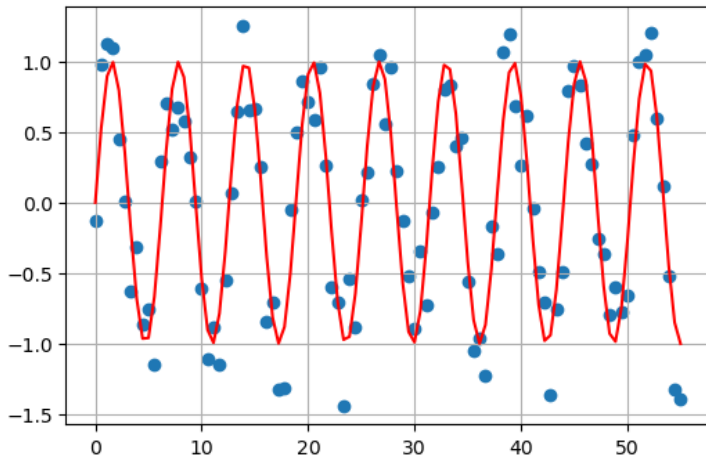
```
1 import inspect
2 inspect.signature(curve).parameters
```

```
1 mappingproxy({'x': <Parameter "x">,
2              'a': <Parameter "a">,
3              'b': <Parameter "b">,
4              'c': <Parameter "c">,
5              'd': <Parameter "d">})
```

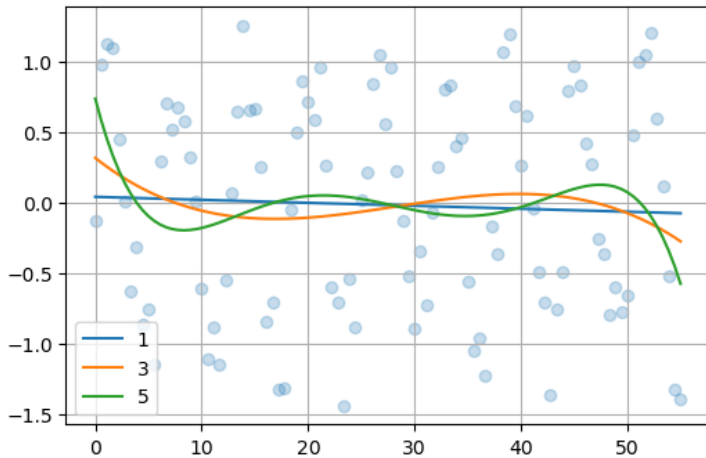
```
1 def curve(x, a,b,c):  
2     return a*np.sin(b*x)+c  
3 popt, pcov = optimize.curve_fit(curve, X,Y)  
4  
5 plt.scatter(X,Y, alpha=0.25)  
6 plt.plot(X, curve(X, *popt));
```



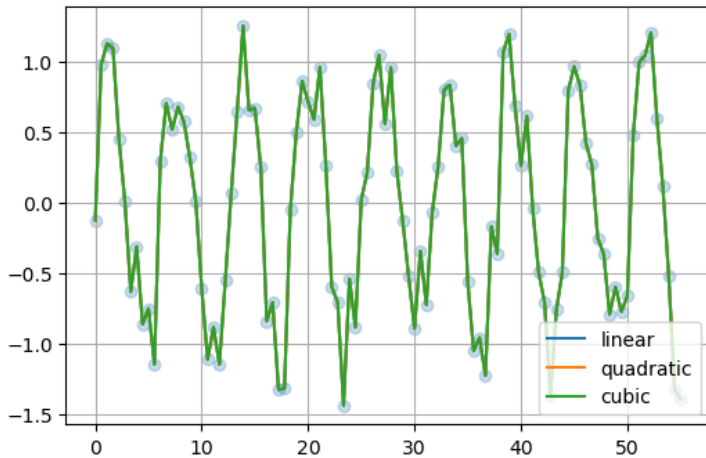

```
1  np.random.seed(42)
2  Xmin, Xmax = 0, 5*3.5*np.pi
3  X = np.linspace(Xmin, Xmax, 100)
4  Y = np.sin(X) + np.random.random(len(X))-0.5
5  plt.scatter(X,Y);
6  plt.plot(X, np.sin(X), color='red');
```



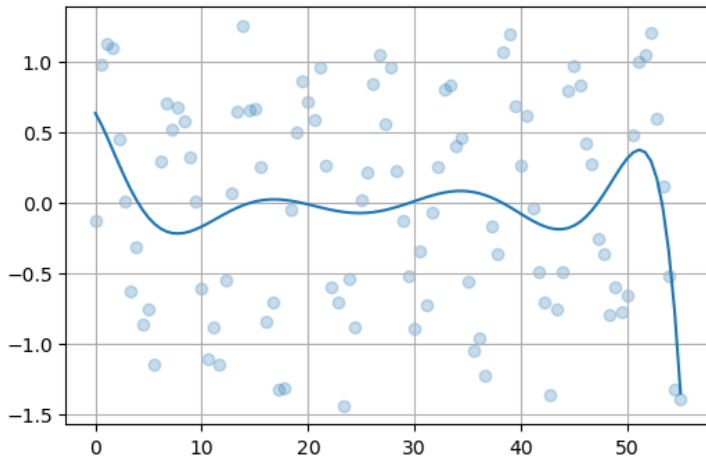
```
1 f1 = np.polynomial.Polynomial.fit(X,Y,1)
2 f3 = np.polynomial.Polynomial.fit(X,Y,3)
3 f5 = np.polynomial.Polynomial.fit(X,Y,5)
4 plt.scatter(X,Y, alpha=0.25)
5 plt.plot(X,f1(X), label='1')
6 plt.plot(X,f3(X), label='3')
7 plt.plot(X,f5(X), label='5')
8 plt.legend();
```



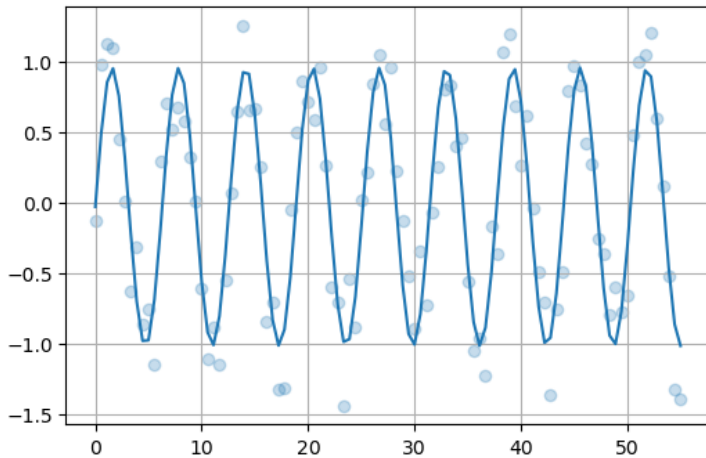
```
1 flin = interpolate.interp1d(X,Y, kind='linear')
2 fquad = interpolate.interp1d(X,Y, kind='quadratic')
3 fcub = interpolate.interp1d(X,Y, kind='cubic')
4
5 plt.scatter(X,Y, alpha=0.25)
6 plt.plot(X,flin(X), label='linear')
7 plt.plot(X,fquad(X), label='quadratic')
8 plt.plot(X,fcub(X), label='cubic')
9 plt.legend();
```



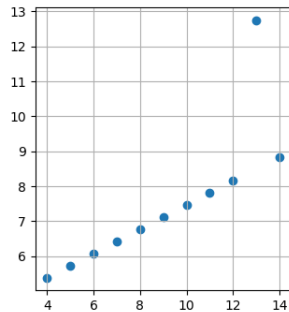
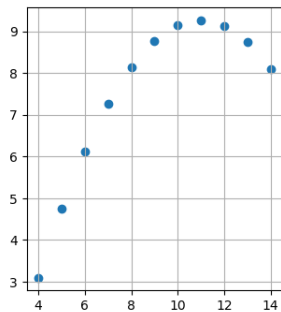
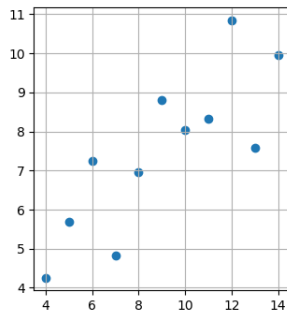
```
1 def curve(x, a,b,c,d,e,f,g,h,i):
2     params = [a,b,c,d,e,f,g,h,i]
3     terms = [params[i]*x**i for i in range(len(params))]
4     return sum(terms)
5 popt, pcov = optimize.curve_fit(curve, X,Y)
6
7 plt.scatter(X,Y, alpha=0.25)
8 plt.plot(X, curve(X, *popt));
```



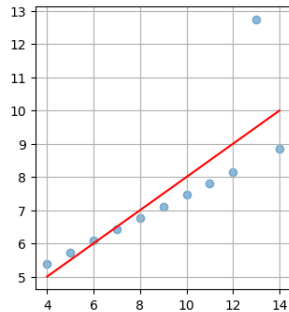
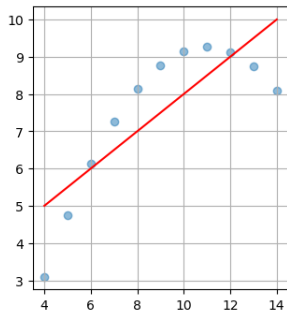
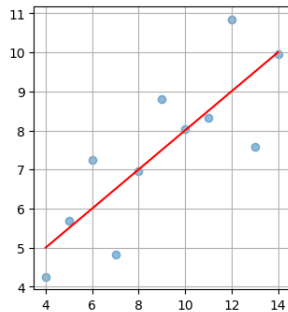

```
1 def curve(x, a,b,c):  
2     return a*np.sin(b*x)+c  
3 popt, pcov = optimize.curve_fit(curve, X,Y)  
4  
5 plt.scatter(X,Y, alpha=0.25)  
6 plt.plot(X, curve(X, *popt));
```



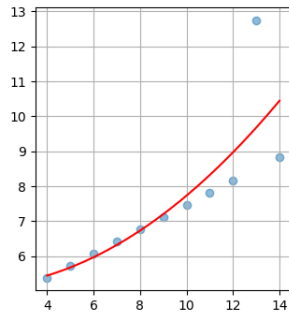
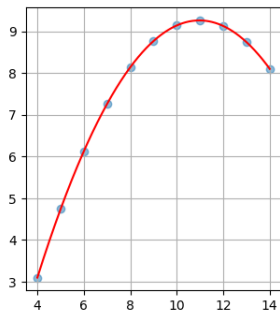
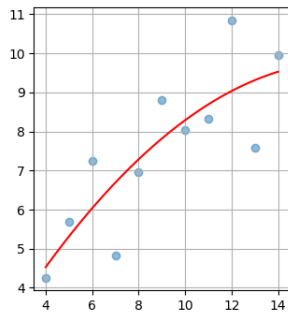
```
1 x = [10, 8, 13, 9, 11, 14, 6, 4, 12, 7, 5]
2 y1 = [8.04, 6.95, 7.58, 8.81, 8.33, 9.96, 7.24, 4.26, 10.84, 4.82, 5.68]
3 y2 = [9.14, 8.14, 8.74, 8.77, 9.26, 8.10, 6.13, 3.10, 9.13, 7.26, 4.74]
4 y3 = [7.46, 6.77, 12.74, 7.11, 7.81, 8.84, 6.08, 5.39, 8.15, 6.42, 5.73]
5
6 fbase = np.linspace(min(x), max(x), 1000)
7 fig, (ax1,ax2,ax3) = plt.subplots(1,3)
8 fig.set_size_inches(12,4)
9 for ax,y in [(ax1,y1),(ax2,y2),(ax3,y3)]:
10     ax.scatter(x,y)
```



```
1 f1 = np.polynomial.Polynomial.fit(x,y1,1)
2 f2 = np.polynomial.Polynomial.fit(x,y2,1)
3 f3 = np.polynomial.Polynomial.fit(x,y3,1)
4 fbase = np.linspace(min(x), max(x), 1000)
5 fig, (ax1,ax2,ax3) = plt.subplots(1,3)
6 fig.set_size_inches(12,4)
7 for ax,f,y in [(ax1,f1,y1),(ax2,f2,y2),(ax3,f3,y3)]:
8     ax.scatter(x,y, alpha=0.5, label='Data')
9     ax.plot(fbase,f(fbase), color='r')
```



```
1 f1 = np.polynomial.Polynomial.fit(x,y1,2)
2 f2 = np.polynomial.Polynomial.fit(x,y2,2)
3 f3 = np.polynomial.Polynomial.fit(x,y3,2)
4 fbase = np.linspace(min(x), max(x), 1000)
5 fig, (ax1,ax2,ax3) = plt.subplots(1,3)
6 fig.set_size_inches(12,4)
7 for ax,f,y in [(ax1,f1,y1),(ax2,f2,y2),(ax3,f3,y3)]:
8     ax.scatter(x,y, alpha=0.5, label='Data')
9     ax.plot(fbase,f(fbase), color='r')
```



Optimization

Definition

Given $\mathbf{x} \in \mathbb{C}^n$, $f : \mathbb{C}^n \rightarrow \mathbb{R}$, $c_i : \mathbb{C} \rightarrow \{T, F\}$

Definition

$$\min_{\mathbf{x} \in \mathbb{C}^n} f(\mathbf{x}), \quad \text{subject to } c_i(x_i) = T$$

Example

$$\mathbf{x} \in \mathbb{R}^2, \quad f(\mathbf{x}) = \left[\mathbf{x} - \begin{pmatrix} k_1 \\ k_2 \end{pmatrix} \right]^{2/3}, \quad \text{subject to } c(\mathbf{x}) = \begin{pmatrix} c_1(\mathbf{x}) \\ \vdots \\ c_k(\mathbf{x}) \end{pmatrix} = \begin{pmatrix} -x_1 + x_2^2 \geq 0 \\ \vdots \\ \|\mathbf{x}\|_2^2 < 0.5 \end{pmatrix}$$

Example: Distribution problem

- ▶ k factories F_i produce each p_i tons of product
- ▶ n users (receivers) R_j need each b_j tons of product
- ▶ the cost of the product from F_i to R_j is c_{ij}
- ▶ the amount of product moved from F_i to R_j is x_{ij}

$$\min \sum_{ij} (c_{ij} x_{ij}) \text{ subject to } \begin{pmatrix} \sum_{j=1}^n x_{ij} \leq p_i, & i = 1, \dots, k \\ \sum_{i=1}^k x_{ij} \geq b_j, & j = 1, \dots, n \\ x_{ij} \geq 0 \end{pmatrix}; \quad \left(\text{or } \min \sum_{ij} (c_{ij} \sqrt{\delta + x_{ij}}), \delta > 0 \right)$$

$$\mathbf{x} \in \mathbb{R}^{kn} = \begin{pmatrix} x_{1,1} \\ x_{1,2} \\ \vdots \\ x_{1,n} \\ x_{2,1} \\ \vdots \\ x_{k,n} \end{pmatrix}, \quad c(\mathbf{x}) = \begin{pmatrix} \sum_{j=1}^n x_{1j} \leq p_1 \\ \sum_{j=1}^n x_{2j} \leq p_2 \\ \vdots \\ \sum_{i=1}^k x_{i1} \geq b_1 \\ \sum_{i=1}^k x_{i2} \geq b_2 \\ \vdots \\ x_{1,1} \geq 0 \\ x_{1,2} \geq 0 \\ \vdots \end{pmatrix}$$

Other Examples

Portfolio optimization:

- ▶ variables: amounts invested in different assets
- ▶ constraints: budget, investment limits per assets, minimum return
- ▶ objectives: overall risk, return variance...

Electronic circuits component placement:

- ▶ variables: components size, routing
- ▶ constraints: manufacturing limits, board size, assembly time, manual labor requirements
- ▶ objectives: uniform heat dissipation, reliability, cost...

Machine Learning:

- ▶ variables: model parameters
- ▶ constraints: parameter limits, data, computational cost, time...
- ▶ objectives: prediction accuracy, classification accuracy, generalization...

Continuous vs Discrete optimization

- ▶ Distribution problem (could be both)
- ▶ ML tasks (usually continuous)
- ▶ Number of power plants to build
- ▶ In which city to place a factory
- ▶ Mixed problems (some integer variables, some continuous)
- ▶ Discrete problems are generally more difficult to solve, there are special techniques
 - When to approximate? During optimization or after? Depends on the problem...

Constrained vs Unconstrained optimization

- ▶ There are *a/ways* constraints!
- ▶ ... but they are costly to check!
- ▶ ... but sometimes they don't affect the solution, so they can be "safely" ignored
- ▶ ... but sometimes they can be reformulated as part of the cost function!
- ▶ linear cost and constraint much easier to deal with
- ▶ nonlinear constraints typical of physical world, costly
- ▶ usually worth understanding which constraints can be dropped or represented in the cost function

Global vs Local optimization

Definition (Global optimization)

$\min f(\mathbf{x})$ where $\mathbf{x} \in \text{Domain}$

Definition (Local optimization)

$\min f(\mathbf{x})$ where $\mathbf{x} \in \text{Neighborhood of starting point within the Domain}$

- ▶ Generally, global optimization requirese infinite checks (depending on properties of f)
- ▶ Global optimization is approximated with multiple local optimizations
- ▶ Convex problems surely admit a global minimum

Stochastic vs Deterministic optimization

Method:

- ▶ Deterministic methods always obtain the same exact result given the same conditions
- ▶ Stochastic methods use a (pseudo)random component (Genetic algorithm, montecarlo etc.) (seed!)

Problem:

- ▶ Some problems have stochastic components (economic model depending on future demand)
- ▶ Instead of “best guess”, model probability into the model!
 - ▶ know scenarios with associated probabilities
 - ▶ produce expected performance with associated probability

Convexity

Definition (Convex set)

$$S \in \mathbb{R}^n \text{ such that } \forall \mathbf{x}, \mathbf{y} \in S, \alpha \mathbf{x} + (1 - \alpha) \mathbf{y} \in S, \forall \alpha \in [0, 1]$$

- ▶ A convex set is such that a straight line segment between any two points in S are also in S .
- ▶ Example: the unit sphere $\{\mathbf{y} \in \mathbb{R}^n \mid \|\mathbf{y}\|_2 \leq 1\}$
- ▶ Example: a polyhedron $\{\mathbf{x} \in \mathbb{R}^n \mid A\mathbf{x} = \mathbf{b}, C\mathbf{x} \leq \mathbf{d}\}$

Definition (Convex function)

$f : S \rightarrow \mathbb{R}$ is convex if S is convex and

$$f(\alpha \mathbf{x} + (1 - \alpha) \mathbf{y}) \leq \alpha f(\mathbf{x}) + (1 - \alpha) f(\mathbf{y}), \quad \forall \alpha \in [0, 1]$$

- ▶ Example: $f(\mathbf{x}) = \mathbf{c}^T \mathbf{x} + k$
- ▶ Example: $f(\mathbf{x}) = \mathbf{x}^T H \mathbf{x}$ with H symmetric positive semidefinite

Multivariate Calculus Recap

Partial Derivatives

$$f : \mathbb{R}^n \rightarrow \mathbb{R}$$

$$\frac{\partial f}{\partial x_k} = \frac{\partial}{\partial x_k} f = f_{x_k}(\mathbf{x}) = D_{x_k} f = \lim_{h \rightarrow 0} \frac{f(\mathbf{x} + h\mathbf{e}_k) - f(\mathbf{x})}{h}$$

when $\exists f_{x_k}(\mathbf{x}) \neq \pm\infty$, and where \mathbf{e}_k is the k -th unit versor.

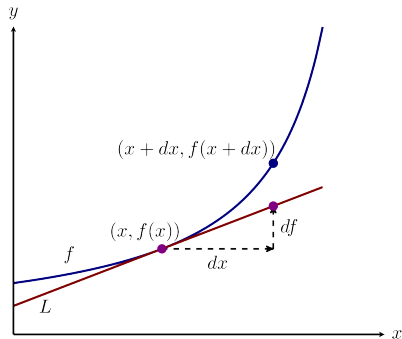
Gradient:

$$\nabla f(\mathbf{x}) = \begin{pmatrix} \frac{\partial f(\mathbf{x})}{\partial x_1} \\ \vdots \\ \frac{\partial f(\mathbf{x})}{\partial x_n} \end{pmatrix}$$

First order approximation

$$f(x + h) = f(x) + f'(x)h + O(h^2)$$

$$f(\mathbf{x} + \mathbf{y}) = f(\mathbf{x}) + \nabla f(\mathbf{x})^T \mathbf{y} + O(\|\mathbf{y}\|)$$



Second Derivative

If $\nabla f(\mathbf{x})$ are still all derivable with respect to all components of \mathbf{x} :

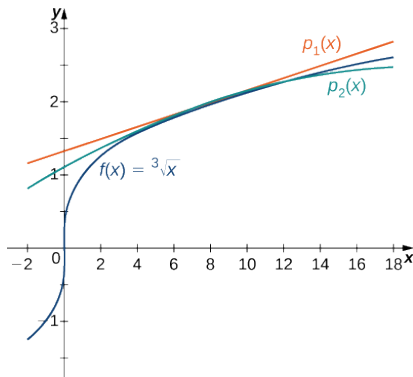
$$\nabla^2 f(\mathbf{x}) = \nabla(\nabla f(\mathbf{x})^T) = \nabla \left(\frac{\partial f(\mathbf{x})}{\partial x_1}, \dots, \frac{\partial f(\mathbf{x})}{\partial x_n} \right) = \begin{pmatrix} \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_1} & \dots & \frac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_n} & \dots & \frac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_n} \end{pmatrix}$$

$\nabla^2 f(\mathbf{x})$ is called *Hessian*, and is a symmetric matrix

Second order approximation

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + O(h^3)$$

$$f(\mathbf{x} + \mathbf{y}) = f(\mathbf{x}) + \nabla f(\mathbf{x})^T \mathbf{y} + \frac{1}{2} \mathbf{y}^T \nabla^2 f(\mathbf{x}) \mathbf{y} + O(\|\mathbf{y}\|^2)$$

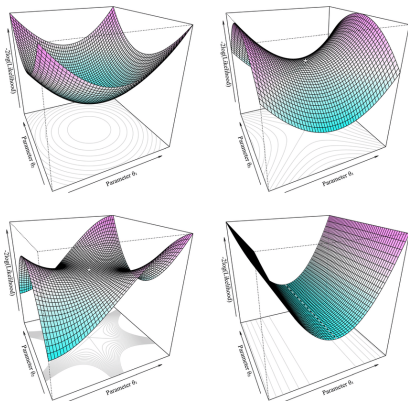
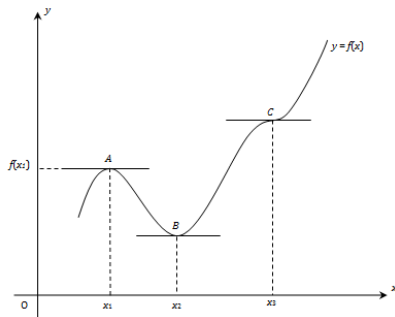


Stationary points

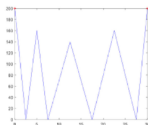
Stationary point \bar{x} of $f : \mathbb{R}^n \rightarrow \mathbb{R}$ is such that:

$$\nabla f(\bar{x}) = 0$$

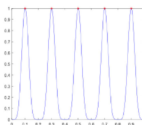
- candidates for local/global max/min!



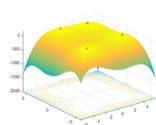
Function Examples, local and global maxima



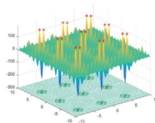
(a) F1



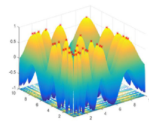
(b) F2



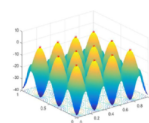
(c) F4



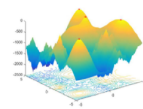
(d) F6



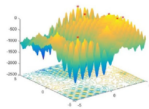
(e) F7



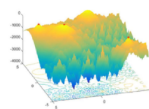
(f) F10



(g) F11



(h) F12



(i) F13

Definitions

$$f : S \rightarrow \mathbb{R}$$

Definition (Global Minimizer)

$$\bar{\mathbf{x}} \text{ for which } f(\bar{\mathbf{x}}) \leq f(\mathbf{x}) \quad \forall \mathbf{x} \in S$$

Definition (Local Minimizer)

$$\bar{\mathbf{x}} \text{ for which } \exists \mathcal{N}(\bar{\mathbf{x}}) \mid f(\bar{\mathbf{x}}) \leq f(\mathbf{x}) \quad \forall \mathbf{x} \in \mathcal{N}(\bar{\mathbf{x}})$$

Definition (Strict Local Minimizer)

$$\bar{\mathbf{x}} \text{ for which } \exists \mathcal{N}(\bar{\mathbf{x}}) \mid f(\bar{\mathbf{x}}) < f(\mathbf{x}) \quad \forall \mathbf{x} \in \mathcal{N}(\bar{\mathbf{x}}) \text{ with } \mathbf{x} \neq \bar{\mathbf{x}}$$

Recognizing local minimum

Theorem (First order necessary conditions)

If $\bar{\mathbf{x}}$ is a local minimizer and f is continuously differentiable in an open neighbour $\mathcal{N}(\bar{\mathbf{x}})$, then $\nabla f(\bar{\mathbf{x}}) = 0$

Theorem (Second order necessary condition)

If $\bar{\mathbf{x}}$ is a local minimizer and $\nabla^2 f$ exists and is continuously differentiable in an open neighbourhood $\mathcal{N}(\bar{\mathbf{x}})$, then $\nabla f(\bar{\mathbf{x}}) = 0$ and $\nabla^2 f(\bar{\mathbf{x}})$ is positive semidefinite

Theorem (Second order sufficient condition)

Suppose $\nabla^2 f$ continuous in an open Neighborhood $\mathcal{N}(\bar{\mathbf{x}})$ and that $\nabla f(\bar{\mathbf{x}})$ and $\nabla^2 f(\bar{\mathbf{x}})$ is positive definite. Then $\bar{\mathbf{x}}$ is a strict local minimizer of f .

Theorem (Convexity)

If f is convex, any local minimizer $\bar{\mathbf{x}}$ is also a global minimizer for f .

If f is convex and differentiable, any stationary point $\bar{\mathbf{x}}$ is also a global minimizer of f .

Examples of convex/concave functions

On \mathbb{R} :

$$ax + b, e^{ax}, |x|^\alpha, x \log x, \log x$$

On \mathbb{R}^n

$$\mathbf{a}^T \mathbf{x} + \mathbf{b}, \|\mathbf{x}\|_p$$

Examples of operations that preserve convexity:

- ▶ nonnegative weighted sum: $\sum \alpha_i f_i$ is convex if f_i are all convex and $\alpha_i > 0$
- ▶ composition: $f(\mathbf{A}\mathbf{x} + \mathbf{b})$ is convex if f is convex
- ▶ ...

The study of convexity can greatly improve optimization performance!

Optimization strategies overview

Line Search vs Trust Region

Line Search:

- ▶ Choose a direction \mathbf{p}_k and look for a point with lower value:

$$\mathbf{x}_{k+1} = \min_{\alpha > 0} f(\mathbf{x}_k + \alpha \mathbf{p}_k)$$

- ▶ exact solution for \mathbf{p} usually too expensive/impossible
- ▶ use a loose approximation of \mathbf{p} after a few trials and try again

Trust Region:

- ▶ build a model function m_k to approximate/simplify f over trust-region $\mathcal{N}_k = \mathcal{N}(\mathbf{x}_k)$
- ▶ solve (approximate):

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{p} \text{ where } \min_{\mathbf{p}} m_k(\mathbf{x}_k + \mathbf{p})$$

- ▶ if f does not decrease enough ($f(\mathbf{x}_{k+1}) - f(\mathbf{x}_k) > -\delta$) shrink the trust region and try again
- ▶ usually the trust region is a sphere of given radius
- ▶ usually the model m is a polynomial approximation of low (2, 3) degree

Derivative-free optimization

And what if the f is not differentiable? Or has stochastic components, noise etc.?

Nelder-Mead simplex-reflection method:

- ▶ Define a simplex trust region where to search for a minimum
- ▶ At each iteration, replace the vertex with the worst f value with a better one
- ▶ If impossible, replace all vertices but the best ones with some closer to the best one
- ▶ Stopping conditions on simplex size

Genetic algorithms:

- ▶ Completely stochastic approach
- ▶ Consider each dimension of \mathbf{R}^n as a “gene”
- ▶ Generate a population of vectors in the search space
- ▶ Combine vectors of the population, choosing the best individuals in each generation
- ▶ Stopping condition on lack of improvement after some generations

Gradient Descent

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \eta \nabla f(\mathbf{x}_k) \quad (2)$$

- ▶ Use derivative (gradient) to move in the correct direction
- ▶ Only works on differentiable and convex functions!

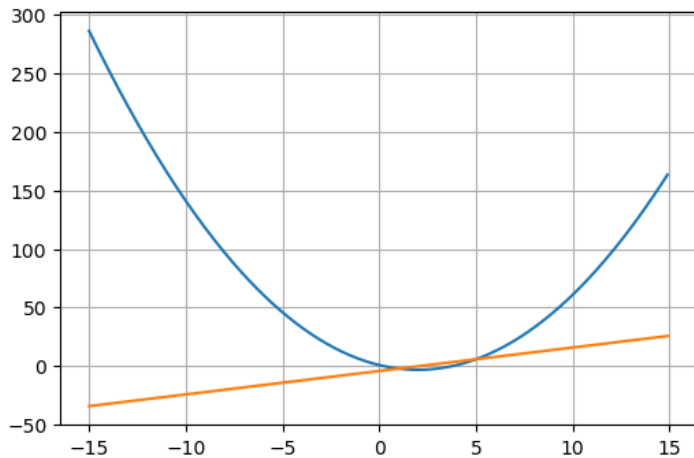
Steps:

1. Check for differentiability and convexity
2. Iterate (2) until:
 - ▶ maximum iterations reached
 - ▶ tol_x (and/or tol_f) reached


```
1 import numpy as np
2 from matplotlib import pyplot as plt
3 plt.rcParams['axes.grid'] = True
```

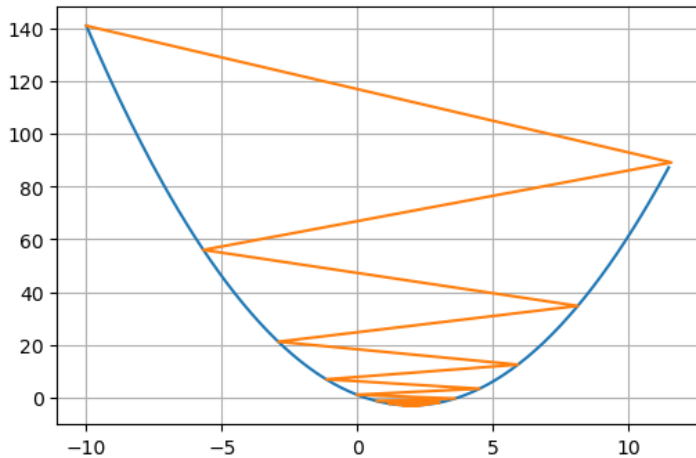
```
1 def gradient_descent(f, f_grad, x_0, maxiter=1000, eta=0.9, tolx=10**-10):
2     counter = 0
3     x_history = [x_0]
4
5     while counter < maxiter:
6         counter += 1
7         x = x_history[-1]
8         new_x = x - eta * f_grad(x)
9         x_history.append(new_x)
10        if abs(np.min(new_x-x)) < tolx:
11            break
12
13    return new_x, np.array(x_history)
```

```
1 def f(x):  
2     return x**2 - 4*x + 1  
3  
4 def f_grad(x):  
5     return 2*x - 4  
6  
7 x_base = np.arange(-15, 15, 0.1)  
8 plt.plot(x_base, f(x_base));  
9 plt.plot(x_base, f_grad(x_base));
```



```
1 x_min, x_history = gradient_descent(f, f_grad, -10, eta=0.9)
2 print(x_min)
3 print(len(x_history))
4 x_base = np.arange(min(x_history), max(x_history), 0.1)
5 plt.plot(x_base, f(x_base));
6 plt.plot(x_history, f(np.array(x_history)));
```

```
1 1.9999999999559648
2 119
```



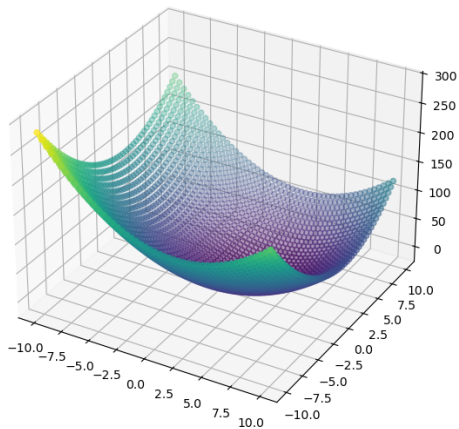
```
1 x = np.linspace(-10, 10, 50)
2 y = np.linspace(-10, 10, 50)
3 X, Y = np.meshgrid(x, y)
4 print(X.shape)
5 print(X.ravel().shape)
6 x_base = np.vstack([X.ravel(), Y.ravel()]).T
7 print(x_base.shape)
```

```
1 (50, 50)
2 (2500,)
3 (2500, 2)
```

```
1 # m points (rows) times n dimensions (columns), same polynomial on all dimensions
2 def f(x):
3     return np.sum(x**2 - 4*x + 1, axis=1)
4
5 # gradient is a column vector! A row per dimension, a column per point
6 def f_grad(x):
7     return (2*x - 4).T
8
```

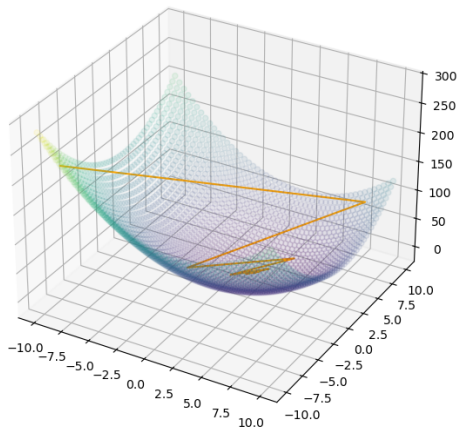


```
1 Z = f(x_base)
2 print(Z.shape)
3
4 fig = plt.figure(figsize=(10, 7))
5 ax = fig.add_subplot(111, projection='3d')
6 surf = ax.scatter3D(X.ravel(), Y.ravel(), Z, c=Z, cmap='viridis')
1 (2500,)
```

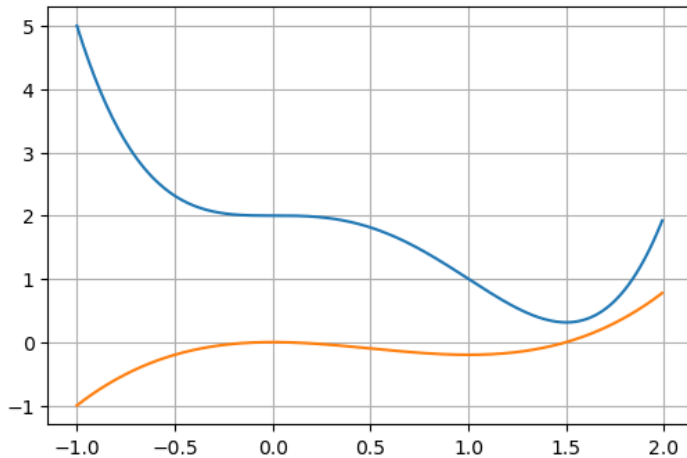


```
1 x_min, x_history = gradient_descent(f, f_grad, np.array([-8,-10]) , eta=0.8)
2 print(x_min)
3 print(len(x_history))
4
5 fig = plt.figure(figsize=(10, 7))
6 ax = fig.add_subplot(111, projection='3d')
7 surf = ax.scatter3D(X.ravel(), Y.ravel(), Z, c=Z, cmap='viridis', alpha=0.1)
8 x,y = x_history[:,0], x_history[:,1]
9 surf = ax.plot3D(x, y, f(x_history), color='orange', alpha=1)

1 [2. 2.]
2 53
```

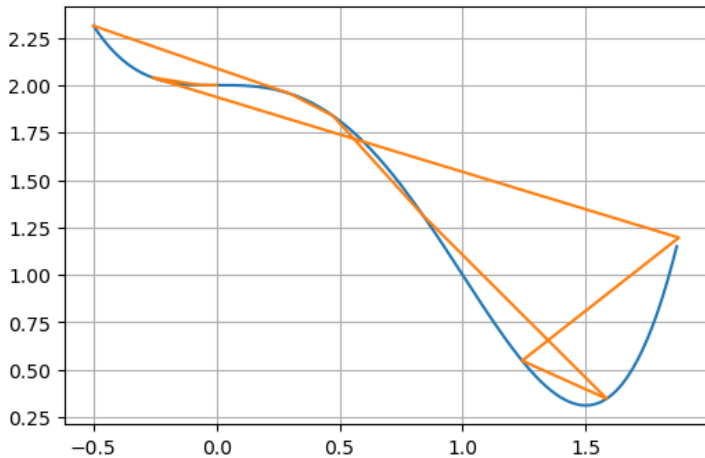


```
1 def f(x):  
2     return x**4 - 2*x**3 + 2  
3  
4 def f_grad(x):  
5     return 4*x**3 - 6*x**2  
6  
7 x_base = np.arange(-1, 2, 0.01)  
8 plt.plot(x_base, f(x_base));  
9 plt.plot(x_base, 0.1*f_grad(x_base));
```



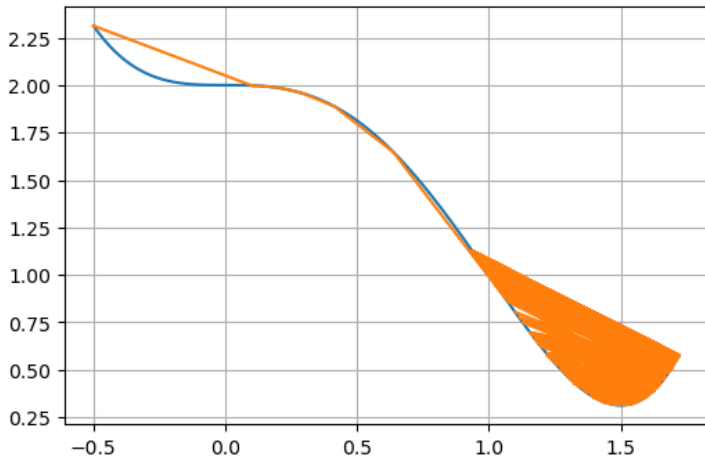
```
1 x_min, x_history = gradient_descent(f, f_grad, -0.5, eta=0.4)
2 print(x_min)
3 print(len(x_history))
4 x_base = np.arange(min(x_history), max(x_history), 0.01)
5 plt.plot(x_base, f(x_base));
6 plt.plot(x_history, f(np.array(x_history)));
```

```
1 -0.00041472877781818996
2 1001
```



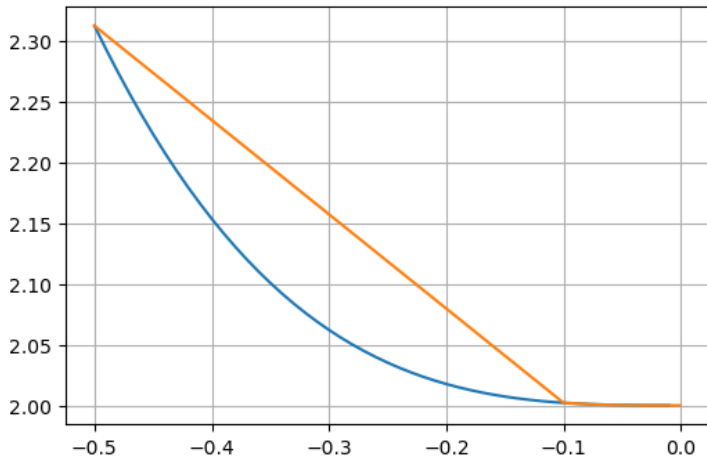

```
1 x_min, x_history = gradient_descent(f, f_grad, -0.5, eta=0.3)
2 print(x_min)
3 print(len(x_history))
4 x_base = np.arange(min(x_history), max(x_history), 0.01)
5 plt.plot(x_base, f(x_base));
6 plt.plot(x_history, f(np.array(x_history)));
```

```
1 1.7202062388426052
2 1001
```



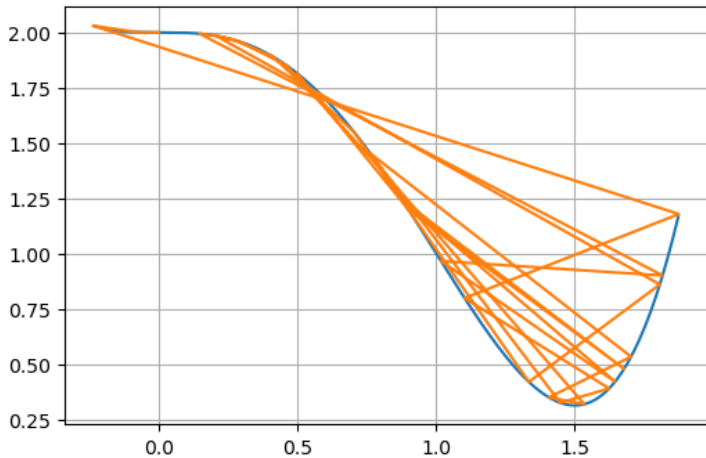
```
1 x_min, x_history = gradient_descent(f, f_grad, -0.5, eta=0.2)
2 print(x_min)
3 print(len(x_history))
4 x_base = np.arange(min(x_history), max(x_history), 0.01)
5 plt.plot(x_base, f(x_base));
6 plt.plot(x_history, f(np.array(x_history)));
```

```
1 -0.0008211225683724356
2 1001
```



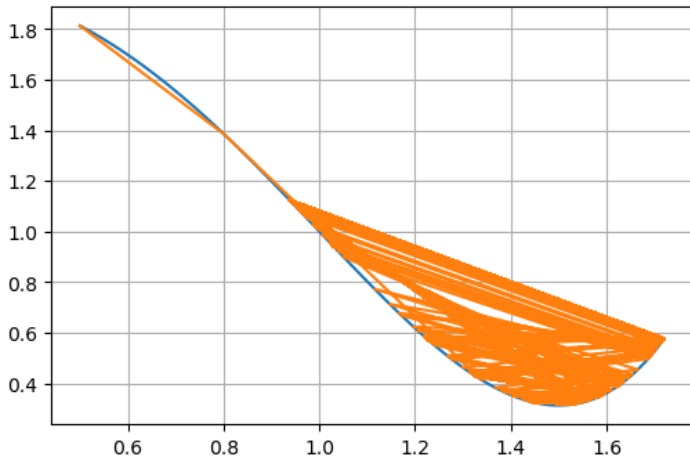
```
1 x_min, x_history = gradient_descent(f, f_grad, 0.5, eta=0.4)
2 print(x_min)
3 print(len(x_history))
4 x_base = np.arange(min(x_history), max(x_history), 0.01)
5 plt.plot(x_base, f(x_base));
6 plt.plot(x_history, f(np.array(x_history)));
```

```
1 -0.0004230136377711251
2 1001
```



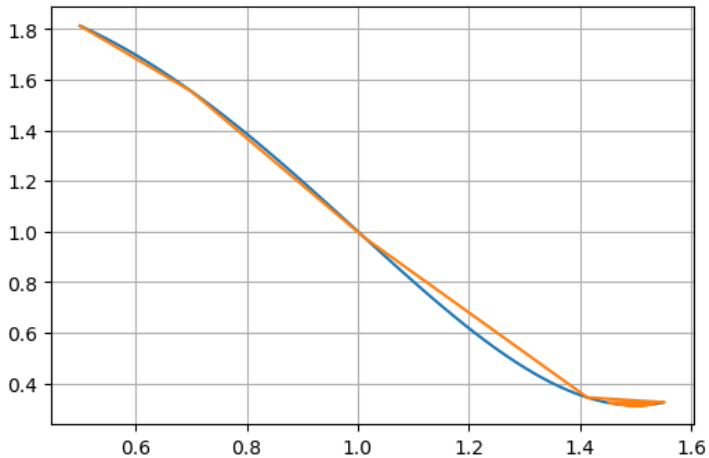
```
1 x_min, x_history = gradient_descent(f, f_grad, 0.5, eta=0.3)
2 print(x_min)
3 print(len(x_history))
4 x_base = np.arange(min(x_history), max(x_history), 0.01)
5 plt.plot(x_base, f(x_base));
6 plt.plot(x_history, f(np.array(x_history)));
```

```
1 0.9382490585492951
2 1001
```




```
1 x_min, x_history = gradient_descent(f, f_grad, 0.5, eta=0.2)
2 print(x_min)
3 print(len(x_history))
4 x_base = np.arange(min(x_history), max(x_history), 0.01)
5 plt.plot(x_base, f(x_base));
6 plt.plot(x_history, f(np.array(x_history)));
```

```
1 1.50000000000412663
2 99
```



Newton's method

$$\mathbf{x}_{k+1} = \mathbf{x}_n - \nabla^2 f(\mathbf{x}_k)^{-1} \nabla f(\mathbf{x}_k) \quad (3)$$

- ▶ Also only works on differentiable and convex functions!
- ▶ Additionally uses curvature information
- ▶ For convex function can converge to the global optimum with quadratic rate
- ▶ hessian (especially the inverse!) very expensive to compute

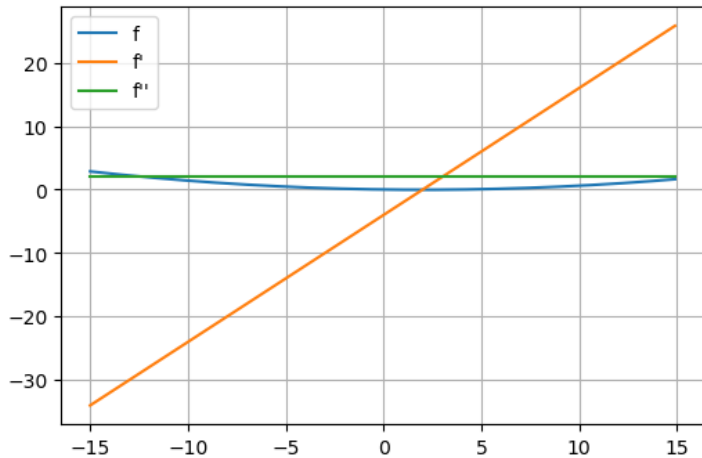
Steps:

1. Check for double differentiability and convexity
2. Iterate (3) until:
 - ▶ maximum iterations reached
 - ▶ tol_x (and/or tol_f) reached

```
1 import numpy as np
2 from scipy import optimize
3 from matplotlib import pyplot as plt
4 plt.rcParams['axes.grid'] = True
```

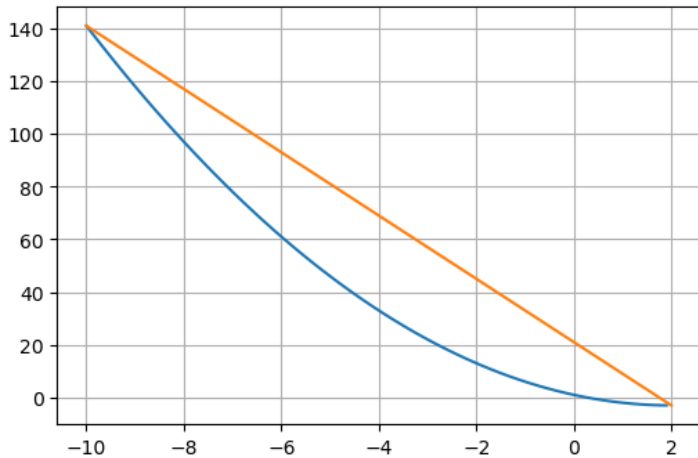
```
1 # 1-dimensional example
2 def newton(f, f_grad, f_hessian, x_0, maxiter=1000, eta=0.9, tolx=10**-10):
3     counter = 0
4     x_history = [x_0]
5
6     while counter < maxiter:
7         counter += 1
8         x = x_history[-1]
9         new_x = x - f_grad(x)/f_hessian(x)
10        x_history.append(new_x)
11        if abs(np.min(new_x-x)) < tolx:
12            break
13
14    return new_x, np.array(x_history)
```

```
1 def f(x):
2     return x**2 - 4*x + 1
3
4 def f_grad(x):
5     return 2*x - 4
6
7 def f_hessian(x):
8     return 2 + 0*x
9
10 x_base = np.arange(-15, 15, 0.1)
11 plt.plot(x_base, 0.01*f(x_base), label='f');
12 plt.plot(x_base, f_grad(x_base), label='f\');
13 plt.plot(x_base, f_hessian(x_base), label='f\'\'');
14 plt.legend();
```

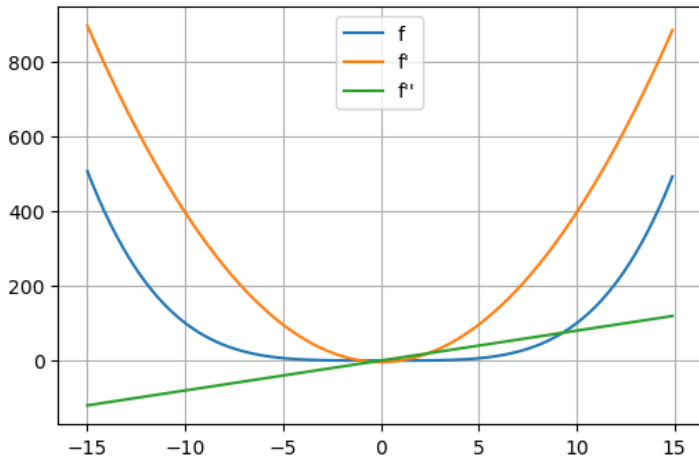


```
1 x_min, x_history = newton(f, f_grad, f_hessian, -10)
2 print(x_min, f(x_min))
3 print(len(x_history))
4 x_base = np.arange(min(x_history), max(x_history), 0.1)
5 plt.plot(x_base, f(x_base));
6 plt.plot(x_history, f(np.array(x_history)));
```

```
1 2.0 -3.0
2 3
```

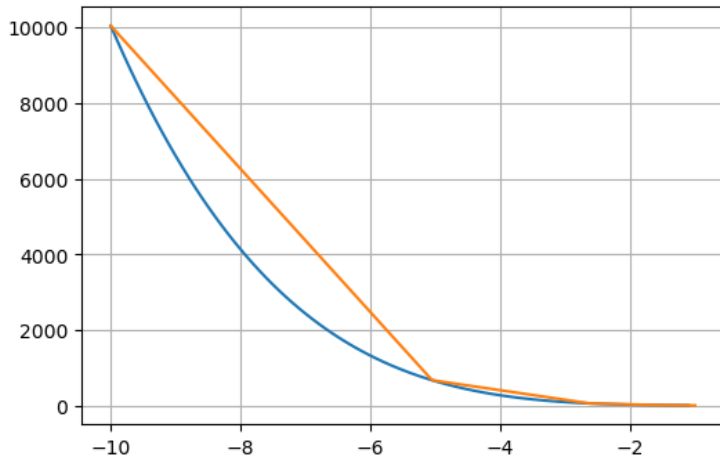



```
1 def f(x):
2     return x**4 - 4*x + 1
3
4 def f_grad(x):
5     return 4*x**3 - 4
6
7 def f_hessian(x):
8     return 12*x**2
9
10 x_base = np.arange(-15, 15, 0.1)
11 plt.plot(x_base, 0.01*f(x_base), label='f');
12 plt.plot(x_base, f_grad(x_base), label='f\');
13 plt.plot(x_base, f_hessian(x_base), label='f\'\'');
14 plt.legend();
```



```
1 x_min, x_history = newton(f, f_grad, f_hessian, -10)
2 print(x_min, f(x_min))
3 print(len(x_history))
4 x_base = np.arange(min(x_history), max(x_history), 0.1)
5 plt.plot(x_base, f(x_base));
6 plt.plot(x_history, f(np.array(x_history)));
```

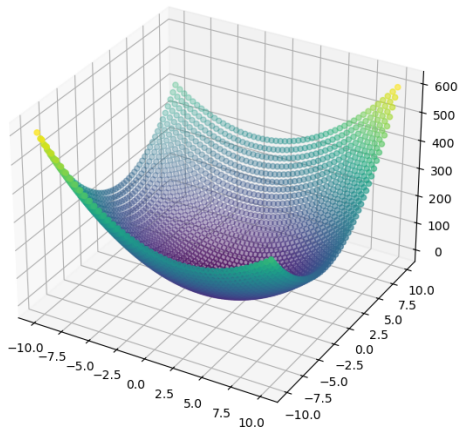
```
1 -1.0 6.0
2 9
```



```
1 # vectorial implementation
2 def newton(f, f_grad, f_hessian, x_0, maxiter=1000, eta=0.9, tolx=10**-10):
3     counter = 0
4     x_history = [x_0]
5
6     while counter < maxiter:
7         counter += 1
8         x = x_history[-1]
9         new_x = x - np.linalg.inv(f_hessian(x)).dot(f_grad(x))
10        x_history.append(new_x)
11        if abs(np.min(new_x-x)) < tolx:
12            break
13
14    return new_x, np.array(x_history)
```

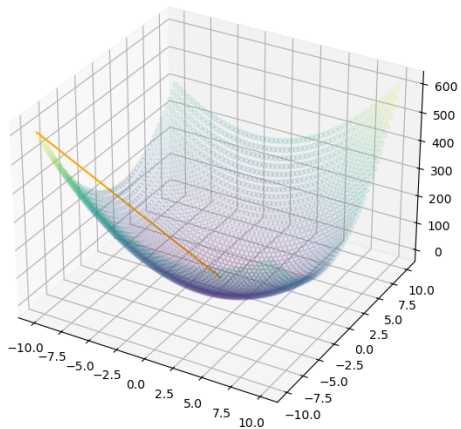
```
1 # m points (rows) times n dimensions (columns)
2 # f = 2 x^2 + xy + 3y^2
3 def f(x: np.array):
4     assert x.shape[1] == 2
5     return 2*x[:,0]**2 + x[:,0]*x[:,1] + 3*x[:,1]**2
6
7 # gradient is a column vector! A row per dimension, a column per point
8 def f_grad(x: np.array):
9     assert x.shape == (2,)
10    return np.array([ 4*x[0] + x[1] , 6*x[1] + x[0] ]).T
11
12
13 # not-vectorized implementation for simplicity
14 # assumes x is a vector size n (dimension)
15 def f_hessian(x):
16     assert x.shape == (2,)
17     gt = f_grad(x).T
18     h = np.array( [[ 4 , 1],
19                   [ 1 , 6] ] )
20
21    return h
```

```
1 x = np.linspace(-10, 10, 50)
2 y = np.linspace(-10, 10, 50)
3 X, Y = np.meshgrid(x, y)
4 x_base = np.vstack([X.ravel(), Y.ravel()]).T
5
6 Z = f(x_base)
7
8 fig = plt.figure(figsize=(10, 7))
9 ax = fig.add_subplot(111, projection='3d')
10 surf = ax.scatter3D(X.ravel(), Y.ravel(), Z, c=Z, cmap='viridis')
```

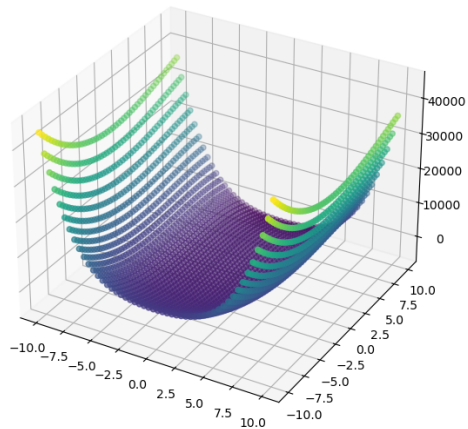
```
1 x_min, x_history = newton(f, f_grad, f_hessian, np.array([-10,-10]))
2 print(x_min, f(np.array([x_min]).reshape((1,2))))
3 print(len(x_history))
4 print(x_history)
5
6 fig = plt.figure(figsize=(10, 7))
7 ax = fig.add_subplot(111, projection='3d')
8 surf = ax.scatter3D(X.ravel(), Y.ravel(), Z, c=Z, cmap='viridis', alpha=0.1)
9 x,y = x_history[:,0], x_history[:,1]
10 surf = ax.plot3D(x, y, f(x_history), color='orange', alpha=1)
```

```
1 [0. 0.] [0.]
2 3
3 [[-1.00000000e+01 -1.00000000e+01]
4  [ 0.00000000e+00 -1.77635684e-15]
5  [ 0.00000000e+00  0.00000000e+00]]
```



```
1 # f = 3x^4 - 4y**3 + x**2y**2
2 def f(x: np.array):
3     assert x.shape[1] == 2 # m points (rows) times n dimensions (columns)
4     return 3*x[:,0]**4 - 4*x[:,1]**3 + x[:,0]**2 * x[:,1]**2
5
6 # gradient is a column vector! A row per dimension, a column per point
7 def f_grad(x: np.array):
8     assert x.shape == (2,) # one point at a time!
9     return np.array([ 12*x[0]**3 + 2*x[0]*x[1]**2 , -12*x[1]**2 + 2*x[0]**2*x[1] ]).T
10
11
12 # not-vectorized implementation for simplicity
13 # assumes x is a vector size n (dimension)
14 def f_hessian(x):
15     assert x.shape == (2,) # one point at a time!
16     h = np.array( [[ 36*x[0]**2 + 2*x[1]**2 , 4*x[0]*x[1] ],
17                   [ 4*x[1]*x[0] , -24*x[1] + 2*x[0]**2] ] )
18     return h
```

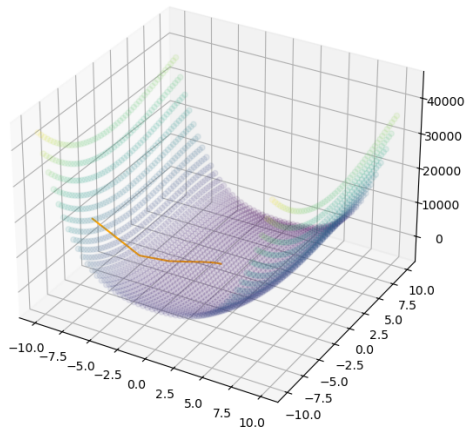
```
1 x = np.linspace(-10, 10, 50)
2 y = np.linspace(-10, 10, 50)
3 X, Y = np.meshgrid(x, y)
4 x_base = np.vstack([X.ravel(), Y.ravel()]).T
5
6 Z = f(x_base)
7
8 fig = plt.figure(figsize=(10, 7))
9 ax = fig.add_subplot(111, projection='3d')
10 surf = ax.scatter3D(X.ravel(), Y.ravel(), Z, c=Z, cmap='viridis')
```



```
1 x_min, x_history = newton(f, f_grad, f_hessian, np.array([-8,-6]), tolx=10**-12)
2 print(x_min, f(np.array([x_min]).reshape((1,2))))
3 print(len(x_history))
4 print(x_history[0::5])
```

```
1 [-9.84218546e-08 -5.29568876e-13] [2.81505855e-28]
2 46
3 [[-8.00000000e+00 -6.00000000e+00]
4  [-1.08231914e+00 -3.23864028e-01]
5  [-1.43234065e-01 -1.34383219e-02]
6  [-1.88717096e-02 -4.84817559e-04]
7  [-2.48526003e-03 -1.63310003e-05]
8  [-3.27277880e-04 -5.31284851e-07]
9  [-4.30983284e-05 -1.69700479e-08]
10 [-5.67550009e-06 -5.36724698e-10]
11 [-7.47390958e-07 -1.68841912e-11]
12 [-9.84218546e-08 -5.29568876e-13]]
```

```
1 fig = plt.figure(figsize=(10, 7))
2 ax = fig.add_subplot(111, projection='3d')
3 surf = ax.scatter3D(X.ravel(), Y.ravel(), Z, c=Z, cmap='viridis', alpha=0.1)
4 x,y = x_history[:,0], x_history[:,1]
5 surf = ax.plot3D(x, y, f(x_history), color='orange', alpha=1)
```

```
1 x_min, x_history = newton(f, f_grad, f_hessian, np.array([-8,-6]), tol=10**-32)
2 print(x_min, f(np.array([x_min]).reshape((1,2))))
3 print(len(x_history))
4 print(x_history[0::5])
```

```
1 [-2.35000316e-19 -7.20977671e-33] [9.14945112e-75]
2 112
3 [[-8.00000000e+00 -6.00000000e+00]
4  [-1.08231914e+00 -3.23864028e-01]
5  [-1.43234065e-01 -1.34383219e-02]
6  [-1.88717096e-02 -4.84817559e-04]
7  [-2.48526003e-03 -1.63310003e-05]
8  [-3.27277880e-04 -5.31284851e-07]
9  [-4.30983284e-05 -1.69700479e-08]
10 [-5.67550009e-06 -5.36724698e-10]
11 [-7.47390958e-07 -1.68841912e-11]
12 [-9.84218546e-08 -5.29568876e-13]
13 [-1.29609027e-08 -1.65826676e-14]
14 [-1.70678554e-09 -5.18792059e-16]
15 [-2.24761881e-10 -1.62223772e-17]
16 [-2.95982724e-11 -5.07124907e-19]
17 [-3.89771489e-12 -1.58506991e-20]
18 [-5.13279327e-13 -4.95387170e-22]
19 [-6.75923393e-14 -1.54817651e-23]
20 [-8.90104880e-15 -4.83821044e-25]
21 [-1.17215457e-15 -1.51196831e-26]
22 [-1.54357804e-16 -4.72494875e-28]
23 [-2.03269536e-17 -1.47655477e-29]
24 [-2.67680048e-18 -4.61424802e-31]
25 [-3.52500474e-19 -1.44195500e-32]]
```

```
1 # This is as proof-test for the next section!
2 def f(x: np.array):
3     assert x.shape == (2,) # one point at a time!
4     return 3*x[0]**4 - 4*x[1]**3 + x[0]**2 * x[1]**2
5
6 x_min, x_history = newton(f, f_grad, f_hessian, np.array([-8,-6]), tolx=10**-12)
7 print(x_min, f(x_min))
8 print(len(x_history))
9 print(x_history[0::5])
```

```
1 [-9.84218546e-08 -5.29568876e-13] 2.815058553998554e-28
2 46
3 [[-8.00000000e+00 -6.00000000e+00]
4  [-1.08231914e+00 -3.23864028e-01]
5  [-1.43234065e-01 -1.34383219e-02]
6  [-1.88717096e-02 -4.84817559e-04]
7  [-2.48526003e-03 -1.63310003e-05]
8  [-3.27277880e-04 -5.31284851e-07]
9  [-4.30983284e-05 -1.69700479e-08]
10 [-5.67550009e-06 -5.36724698e-10]
11 [-7.47390958e-07 -1.68841912e-11]
12 [-9.84218546e-08 -5.29568876e-13]]
```

Newton Method with approximate derivatives

What if f is unknown? If not noisy/stochastic:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}, \quad f''(x) \approx \frac{f'(x+h) - f'(x)}{h} = \frac{f(x+2h) + f(x)}{h^2}$$

$$\nabla f(\mathbf{x}) = \frac{1}{h} \begin{pmatrix} f(\mathbf{x} + h\mathbf{e}_1) - f(\mathbf{x}) \\ \vdots \\ f(\mathbf{x} + h\mathbf{e}_n) - f(\mathbf{x}) \end{pmatrix}, \quad \nabla^2 f(\mathbf{x}) = \frac{1}{h^2} (f(\mathbf{x} + h\mathbf{e}_i + h\mathbf{e}_j) - f(\mathbf{x} + h\mathbf{e}_j) - f(\mathbf{x} + h\mathbf{e}_i) + f(\mathbf{x}))_{ij}$$

- ▶ That's a *lot* of function evaluations at each step! ($2 \times n$ for ∇ , $4 \times n^2$ for ∇^2 minus change)
- ▶ How to choose h ? Watch for numerical issues!
- ▶ What if $\nabla^2 f(\mathbf{x})$ is singular or not defined?

Gradient descent with approximation (very simplified BFGS)

Idea is still:

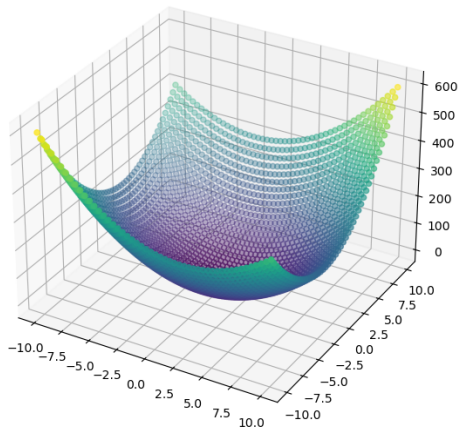
$$\mathbf{x}_{k+1} = \mathbf{x}_k - \eta \nabla f(\mathbf{x}_k)$$

But:

- ▶ the gradient ∇f is an approximation
- ▶ the step η is computed instead of fixed
- ▶ the search for η is done on a polynomial approximation of f

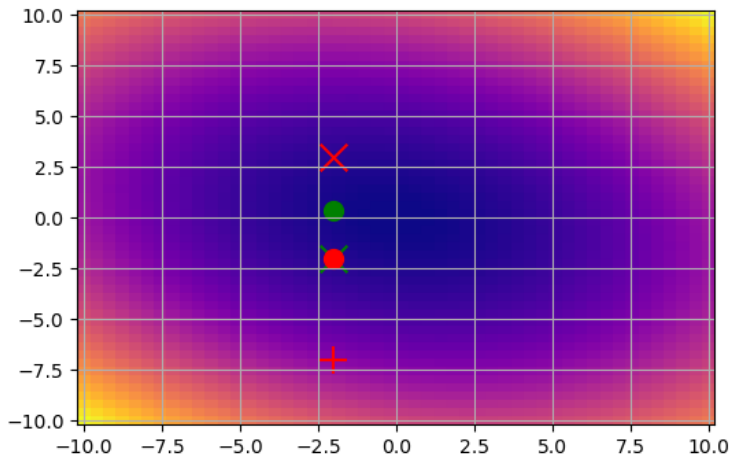
```
1 import numpy as np
2 from scipy import optimize
3 from matplotlib import pyplot as plt
4 plt.rcParams['axes.grid'] = True
```

```
1 def f(x):
2     return 2*x[0]**2 + x[0]*x[1] + 3*x[1]**2
3
4 x = np.linspace(-10, 10, 50)
5 y = np.linspace(-10, 10, 50)
6 X, Y = np.meshgrid(x, y)
7 x_base = np.vstack([X.ravel(), Y.ravel()])
8
9 Z = f(x_base)
10
11 fig = plt.figure(figsize=(10, 7))
12 ax = fig.add_subplot(111, projection='3d')
13 surf = ax.scatter3D(X.ravel(), Y.ravel(), Z, c=Z, cmap='viridis')
```

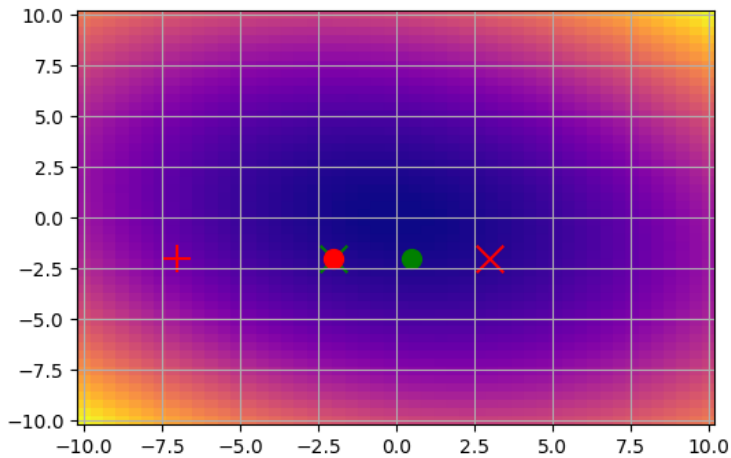



```
1 def find_eta(f, p, x, min_eta=-1, max_eta=1):
2     etas = np.array([min_eta, (max_eta+min_eta)/2, max_eta])
3     x_0 = x + etas[0]*p
4     x_1 = x + etas[1]*p
5     x_2 = x + etas[2]*p
6
7     X = np.vstack([x_0, x_1, x_2]).T
8     Y = f(X)
9
10    coeffs = np.polyfit(etas, Y, 2) # ax^2+bx+c
11    #np.polyval(coeffs, x)
12    # p = ax^2+bx+c
13    # p' = 2ax + b
14    # p'' = 2a
15    if coeffs[0] > 0: #convex parabola
16        eta = -coeffs[1]/(2*coeffs[0]) #-b/2a
17        return max(min_eta, min(max_eta, eta))
18    else: #concave parabola
19        if Y[0] < Y[2]:
20            return min_eta
21        else:
22            return max_eta
```

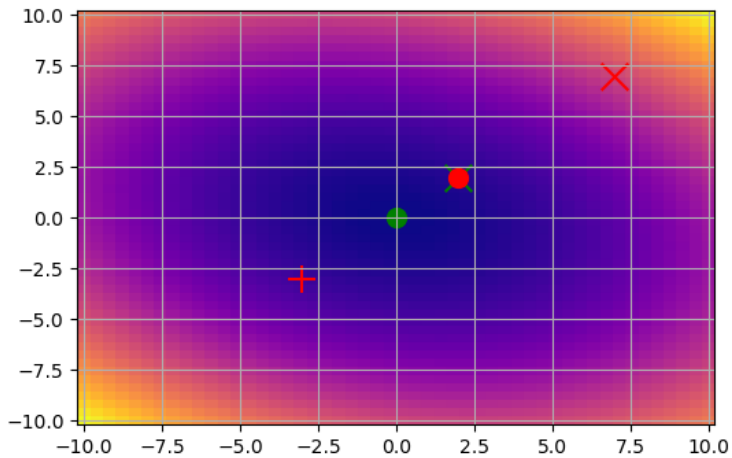
```
1 x0 = np.array([-2,-2])
2 p = np.array([0,1])
3 min_eta, max_eta = -5, 5
4 eta = find_eta(f, p, x0, min_eta=min_eta, max_eta=max_eta)
5 x = x0 + eta*p
6
7 z = Z.reshape((50,50))
8 plt.pcolormesh(X, Y, z, cmap='plasma', shading='auto');
9 plt.scatter(*x0, marker='x', color='green', s=200);
10 plt.scatter(*(x0+min_eta*p), marker='+', color='red', s=200);
11 plt.scatter(*(x0+max_eta*p), marker='x', color='red', s=200);
12 plt.scatter(*(x0+((max_eta+min_eta)/2)*p), marker='o', color='red', s=100);
13 plt.scatter(*x, marker='o', color='green', s=100);
```



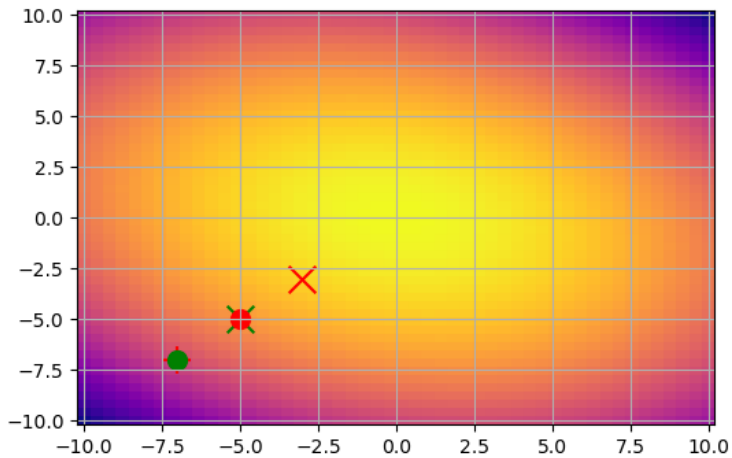
```
1 x0 = np.array([-2,-2])
2 p = np.array([1,0])
3 min_eta, max_eta = -5, 5
4 eta = find_eta(f, p, x0, min_eta=min_eta, max_eta=max_eta)
5 x = x0 + eta*p
6
7 z = Z.reshape((50,50))
8 plt.pcolormesh(X, Y, z, cmap='plasma', shading='auto');
9 plt.scatter(*x0, marker='x', color='green', s=200);
10 plt.scatter(*(x0+min_eta*p), marker='+', color='red', s=200);
11 plt.scatter(*(x0+max_eta*p), marker='x', color='red', s=200);
12 plt.scatter(*(x0+((max_eta+min_eta)/2)*p), marker='o', color='red', s=100);
13 plt.scatter(*x, marker='o', color='green', s=100);
```



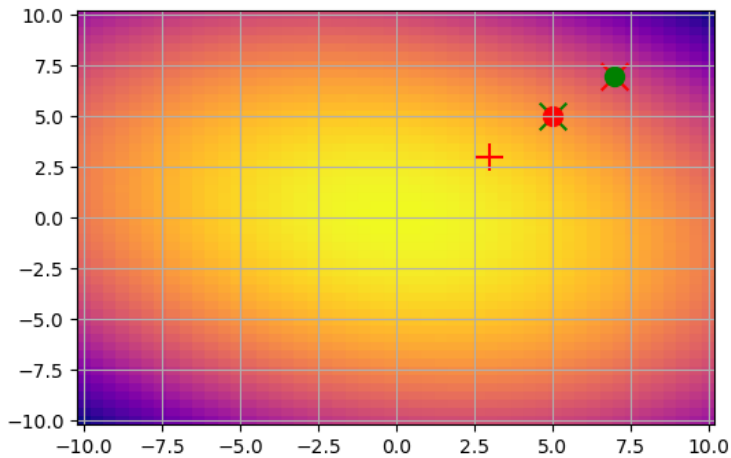
```
1 x0 = np.array([2,2])
2 p = np.array([1,1])
3 min_eta, max_eta = -5,5
4 eta = find_eta(f, p, x0, min_eta=min_eta, max_eta=max_eta)
5 x = x0 + eta*p
6
7 z = Z.reshape((50,50))
8 plt.pcolormesh(X, Y, z, cmap='plasma', shading='auto');
9 plt.scatter(*x0, marker='x', color='green', s=200);
10 plt.scatter(*(x0+min_eta*p), marker='+', color='red', s=200);
11 plt.scatter(*(x0+max_eta*p), marker='x', color='red', s=200);
12 plt.scatter(*(x0+((max_eta+min_eta)/2)*p), marker='o', color='red', s=100);
13 plt.scatter(*x, marker='o', color='green', s=100);
```



```
1 g = lambda x: -f(x)
2 x0 = np.array([-5,-5])
3 p = np.array([1,1])
4 min_eta, max_eta = -2,2
5 eta = find_eta(g, p, x0, min_eta=min_eta, max_eta=max_eta)
6 x = x0 + eta*p
7
8 plt.pcolormesh(X, Y, -z, cmap='plasma', shading='auto');
9 plt.scatter(*x0, marker='x', color='green', s=200);
10 plt.scatter(*(x0+min_eta*p), marker='+', color='red', s=200);
11 plt.scatter(*(x0+max_eta*p), marker='x', color='red', s=200);
12 plt.scatter(*(x0+((max_eta+min_eta)/2)*p), marker='o', color='red', s=100);
13 plt.scatter(*x, marker='o', color='green', s=100);
```

```
1 x0 = np.array([5,5])
2 eta = find_eta(g, p, x0, min_eta=min_eta, max_eta=max_eta)
3 x = x0 + eta*p
4
5 plt.pcolormesh(X, Y, -z, cmap='plasma', shading='auto');
6 plt.scatter(*x0, marker='x', color='green', s=200);
7 plt.scatter(*(x0+min_eta*p), marker='+', color='red', s=200);
8 plt.scatter(*(x0+max_eta*p), marker='x', color='red', s=200);
9 plt.scatter(*(x0+((max_eta+min_eta)/2)*p), marker='o', color='red', s=100);
10 plt.scatter(*x, marker='o', color='green', s=100);
```



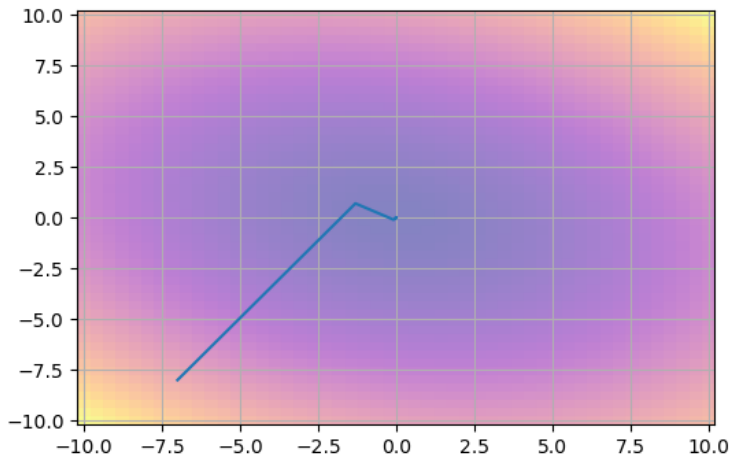
```
1 def f_grad_approx(f, x_0, h=1e-1):
2     n = len(x_0)
3     grad = np.zeros(n)
4
5     for i in range(n):
6         h_i = np.eye(1,n,i).flatten() * h
7         x_l, x_r = x_0-h_i, x_0+h_i
8         grad[i] = (f(x_r)-f(x_l))/(2*h)
9
10    return grad
```

```
1 def simplified_BFGS(f, x_0, maxiter=1000, tol=1e-10, eta=1):
2     counter = 0
3     x_history = [x_0]
4     f_grad = f_grad_approx(f, x_0)
5
6     while counter < maxiter:
7         counter += 1
8         x = x_history[-1]
9         p = f_grad_approx(f, x)
10        neta = find_eta(f, p, x, -eta, eta)
11        new_x = x + neta*p
12        x_history.append(new_x)
13        if abs(np.min(new_x-x)) < tol:
14            break
15
16    return new_x, np.array(x_history)
```

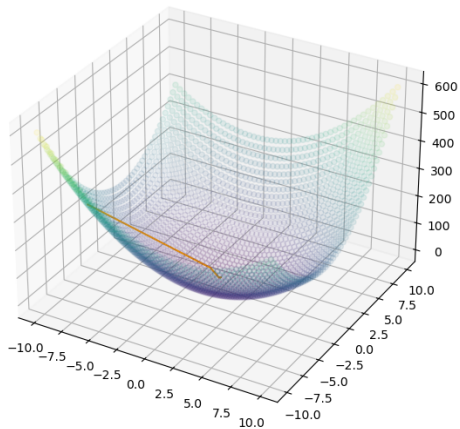
```
1 x_0 = np.array([-7, -8])
2 sol, x_history = simplified_BFGS(f, x_0)
3 print(sol, len(x_history))

1 [-2.94289276e-12  1.60009673e-12] 14
```

```
1 plt.pcolormesh(X, Y, z, cmap='plasma', shading='auto', alpha=0.5);  
2 plt.plot(x_history[:,0], x_history[:,1]);
```

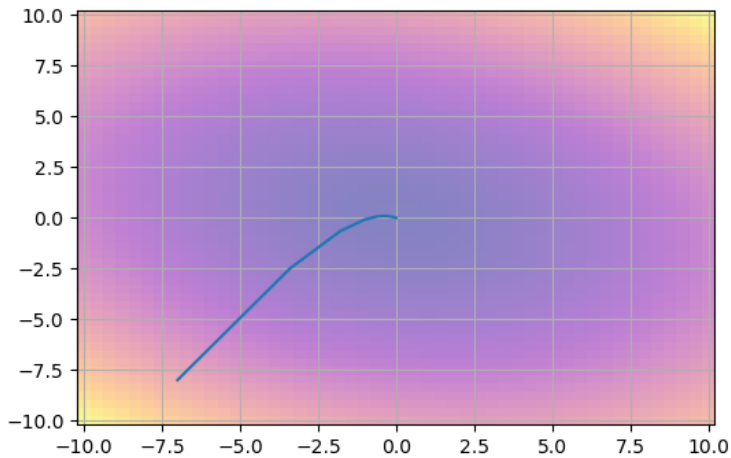



```
1 fig = plt.figure(figsize=(10, 7))
2 ax = fig.add_subplot(111, projection='3d')
3 surf = ax.scatter3D(X.ravel(), Y.ravel(), Z, c=Z, cmap='viridis', alpha=0.1)
4 x,y = x_history[:,0], x_history[:,1]
5 surf = ax.plot3D(x, y, f(x_history.T), color='orange', alpha=1)
```

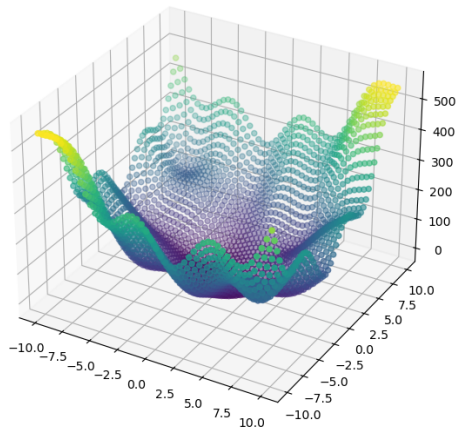


```
1 x_0 = np.array([-7, -8])
2 sol, x_history = simplified_BFGS(f, x_0, eta=0.1)
3 print(sol, len(x_history))
4 plt.pcolormesh(X, Y, z, cmap='plasma', shading='auto', alpha=0.5);
5 plt.plot(x_history[:,0], x_history[:,1]);
```

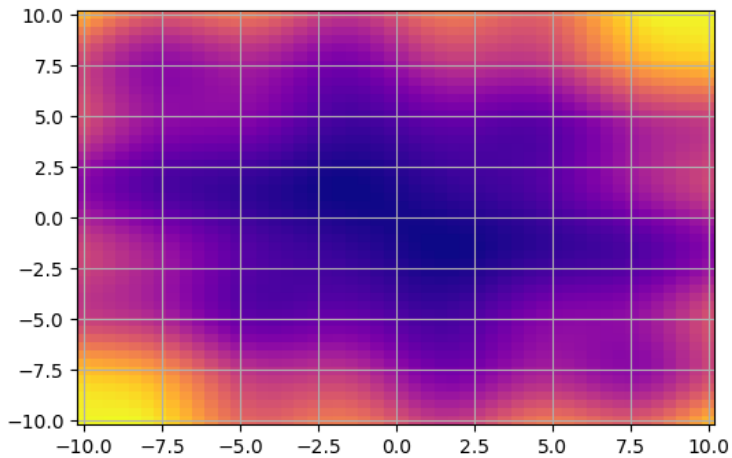
```
1 [-2.94630697e-10  1.22040026e-10] 53
```



```
1 def f(x):
2     return 2*x[0]**2 + x[0]*x[1] + 3*x[1]**2 + 5*x[0]*np.sin(x[1]) + 5*x[1]*np.sin(x[0])
3
4 x = np.linspace(-10, 10, 50)
5 y = np.linspace(-10, 10, 50)
6 X, Y = np.meshgrid(x, y)
7 x_base = np.vstack([X.ravel(), Y.ravel()])
8
9 Z = f(x_base)
10
11 fig = plt.figure(figsize=(10, 7))
12 ax = fig.add_subplot(111, projection='3d')
13 surf = ax.scatter3D(X.ravel(), Y.ravel(), Z, c=Z, cmap='viridis')
```

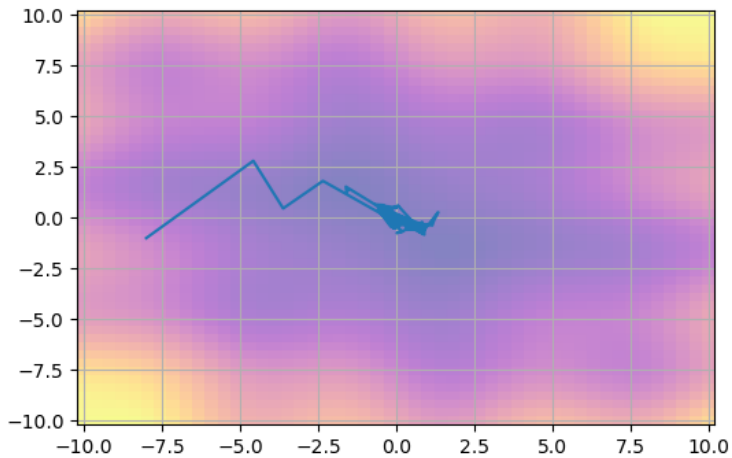


```
1 z = Z.reshape((50,50))  
2 plt.pcolormesh(X, Y, z, cmap='plasma', shading='auto');
```

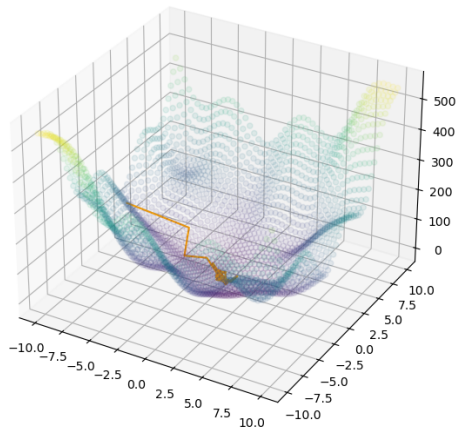



```
1 x_0 = np.array([-8, -1])
2 sol, x_history = simplified_BFGS(f, x_0, eta=1)
3 print(sol, len(x_history))
4
5 plt.pcolormesh(X, Y, z, cmap='plasma', shading='auto', alpha=0.5);
6 plt.plot(x_history[:,0], x_history[:,1]);
```

```
1 [-1.56837405  1.36341575] 74
```

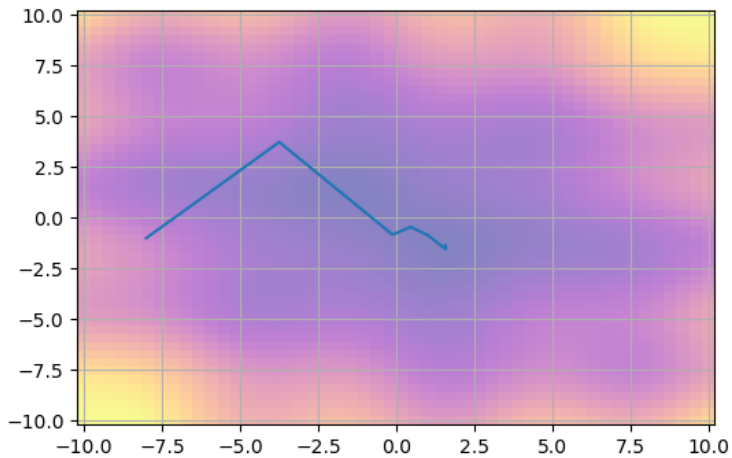


```
1 fig = plt.figure(figsize=(10, 7))
2 ax = fig.add_subplot(111, projection='3d')
3 surf = ax.scatter3D(X.ravel(), Y.ravel(), Z, c=Z, cmap='viridis', alpha=0.1)
4 x,y = x_history[:,0], x_history[:,1]
5 surf = ax.plot3D(x, y, f(x_history.T), color='orange', alpha=1)
```

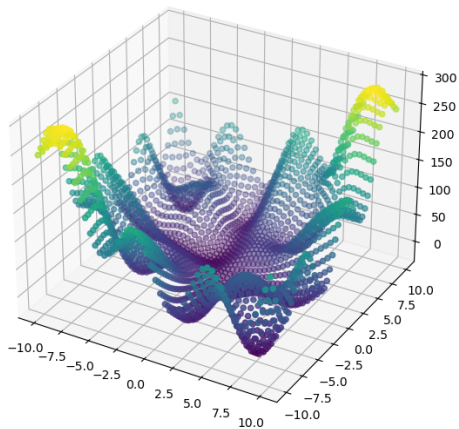


```
1 x_0 = np.array([-8, -1])
2 sol, x_history = simplified_BFGS(f, x_0, eta=0.2)
3 print(sol, len(x_history))
4
5 plt.pcolormesh(X, Y, z, cmap='plasma', shading='auto', alpha=0.5);
6 plt.plot(x_history[:,0], x_history[:,1]);

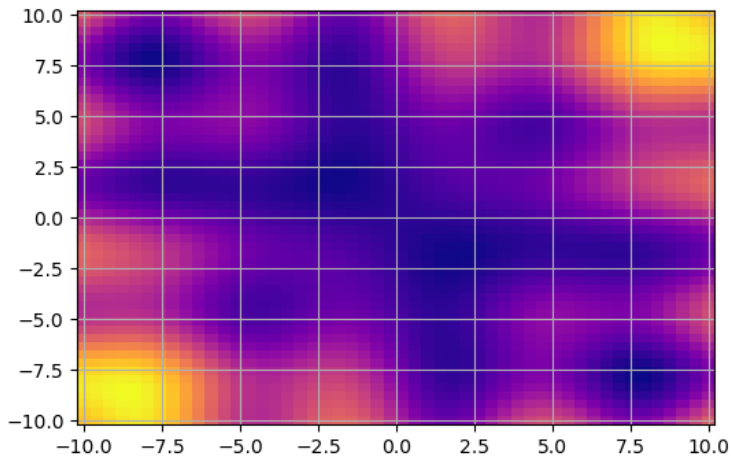
1 [ 1.56843658 -1.3635669 ] 18
```



```
1 def f(x):
2     return x[0]**2 + x[0]*x[1] + x[1]**2 + 5*x[0]*np.sin(x[1]) + 5*x[1]*np.sin(x[0])
3
4 x = np.linspace(-10, 10, 50)
5 y = np.linspace(-10, 10, 50)
6 X, Y = np.meshgrid(x, y)
7 x_base = np.vstack([X.ravel(), Y.ravel()])
8
9 Z = f(x_base)
10
11 fig = plt.figure(figsize=(10, 7))
12 ax = fig.add_subplot(111, projection='3d')
13 surf = ax.scatter3D(X.ravel(), Y.ravel(), Z, c=Z, cmap='viridis')
```

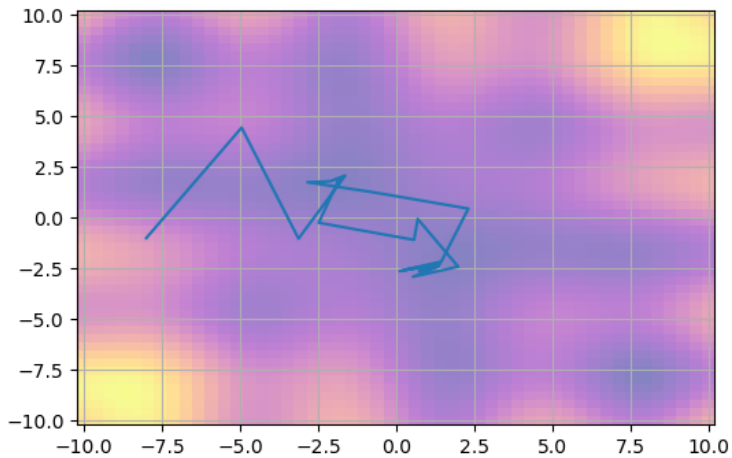



```
1 z = Z.reshape((50,50))  
2 plt.pcolormesh(X, Y, z, cmap='plasma', shading='auto');
```

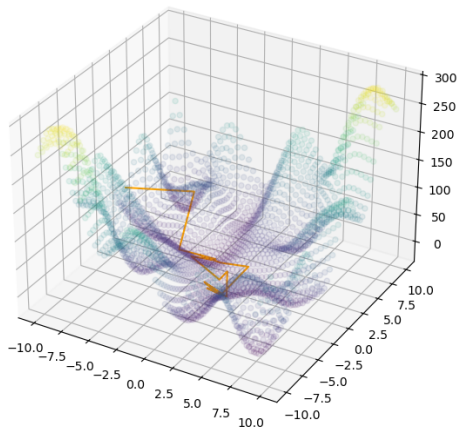


```
1 x_0 = np.array([-8, -1])
2 sol, x_history = simplified_BFGS(f, x_0, eta=1)
3 print(sol, len(x_history))
4
5 plt.pcolormesh(X, Y, z, cmap='plasma', shading='auto', alpha=0.5);
6 plt.plot(x_history[:,0], x_history[:,1]);
```

```
1 [-1.88223468  1.88175382] 41
```

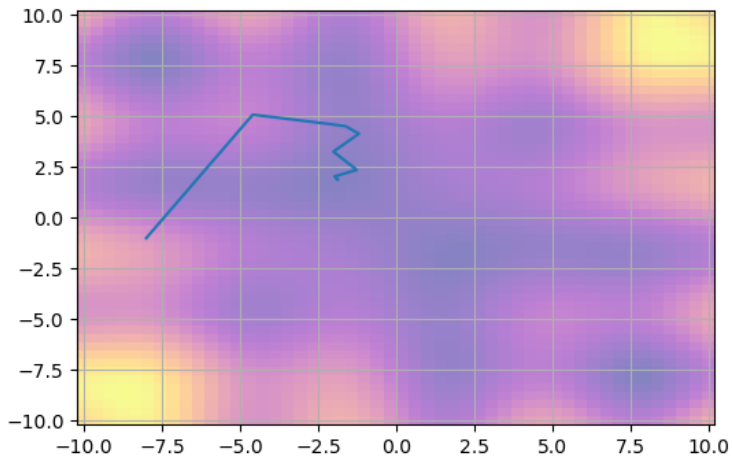


```
1 fig = plt.figure(figsize=(10, 7))
2 ax = fig.add_subplot(111, projection='3d')
3 surf = ax.scatter3D(X.ravel(), Y.ravel(), Z, c=Z, cmap='viridis', alpha=0.1)
4 x,y = x_history[:,0], x_history[:,1]
5 surf = ax.plot3D(x, y, f(x_history.T), color='orange', alpha=1)
```



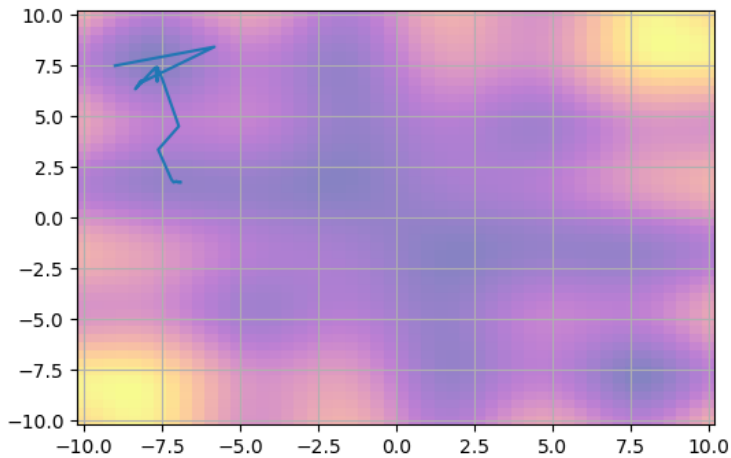
```
1 x_0 = np.array([-8, -1])
2 sol, x_history = simplified_BFGS(f, x_0, eta=0.2)
3 print(sol, len(x_history))
4
5 plt.pcolormesh(X, Y, z, cmap='plasma', shading='auto', alpha=0.5);
6 plt.plot(x_history[:,0], x_history[:,1]);

1 [-1.88162442  1.88186874] 16
```



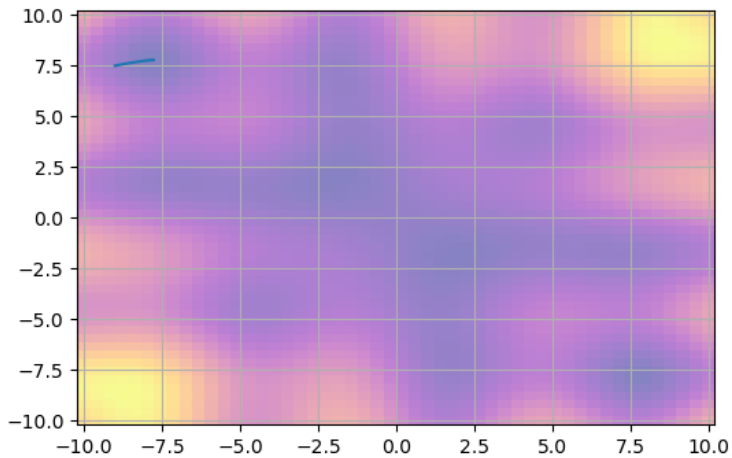

```
1 x_0 = np.array([-9, 7.5])
2 sol, x_history = simplified_BFGS(f, x_0, eta=0.2)
3 print(sol, len(x_history))
4
5 plt.pcolormesh(X, Y, z, cmap='plasma', shading='auto', alpha=0.5);
6 plt.plot(x_history[:,0], x_history[:,1]);

1 [-6.90361457  1.75472133] 35
```



```
1 x_0 = np.array([-9, 7.5])
2 sol, x_history = simplified_BFGS(f, x_0, eta=0.1)
3 print(sol, len(x_history))
4
5 plt.pcolormesh(X, Y, z, cmap='plasma', shading='auto', alpha=0.5);
6 plt.plot(x_history[:,0], x_history[:,1]);

1 [-7.78210924  7.78203928] 10
```



Check existing implementations!



BFGS: quasi-Newton method of Broyden, Fletcher, Goldfarb, and Shannon

- Only uses first derivative if available, otherwise approximates also hessian

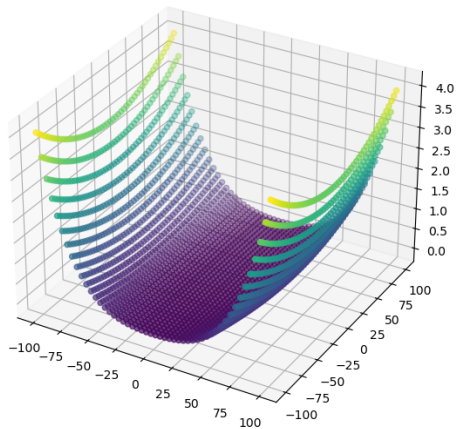
`scipy.optimize.minimize(method='BFGS')`

Newton-CG, trust-constr (previous slide)...

```
1 import numpy as np
2 from scipy import optimize
3 from matplotlib import pyplot as plt
4 plt.rcParams['axes.grid'] = True
```

```
1 def f(x: np.array):  
2     return 3*x[0]**4 - 4*x[1]**3 + x[0]**2 * x[1]**2
```

```
1 x = np.linspace(-100, 100, 50)
2 y = np.linspace(-100, 100, 50)
3 X, Y = np.meshgrid(x, y)
4 X = X.ravel()
5 Y = Y.ravel()
6
7 Z = [f([X[i],Y[i]]) for i in range(len(X))]
8
9 fig = plt.figure(figsize=(10, 7))
10 ax = fig.add_subplot(111, projection='3d')
11 surf = ax.scatter3D(X, Y, Z, c=Z, cmap='viridis')
```

```
1 test = f(np.vstack([X,Y]))  
2 all(test == Z)
```

```
1 False
```

```
1 def callback_generator():  
2     history = list()  
3     def cb(*args, **kwargs):  
4         history.append([args, kwargs])  
5     return cb, history
```

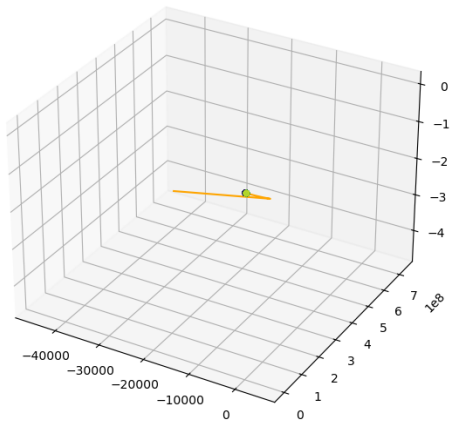
```
1 cb, history = callback_generator()
2 optimize.minimize(f, np.array([-8, -6]), method='BFGS', tol=10**-10, callback=cb)
```

```
1 message: Desired error not necessarily achieved due to precision loss.
2 success: False
3 status: 2
4 fun: -4.547420283313538e+26
5 x: [-4.557e+04  7.315e+08]
6 nit: 89
7 jac: [-4.877e+22 -3.384e+18]
8 hess_inv: [[ 4.077e-05 -5.877e-01]
9            [-5.877e-01  8.471e+03]]
10 nfev: 522
11 njev: 170
```

```
1 print(history[0])
2
3 def read_history(h):
4     return np.vstack([x[0][0] for x in h])
5
6
7 x_history = read_history(history)
8 print(x_history.shape)
9 print(x_history[:,5])
```

```
1 [(array([-7.00572815, -5.82245146]),), {}]
2 (89, 2)
3 [[-7.00572815e+00 -5.82245146e+00]
4  [-3.83689037e+00 -1.42354370e+00]
5  [-9.66002693e-01 -5.24331788e-02]
6  [ 4.98245810e-01  6.00531026e-01]
7  [ 3.78209109e+00  7.20883987e+00]
8  [ 1.22295762e+01  4.69636073e+01]
9  [ 2.17991742e+01  1.33925099e+02]
10 [ 3.55090684e+01  3.38858263e+02]
11 [ 5.67989989e+01  8.48074334e+02]
12 [ 8.97365022e+01  2.09705217e+03]
13 [ 1.40907762e+02  5.15029677e+03]
14 [ 2.20686314e+02  1.26123486e+04]
15 [ 3.45024227e+02  3.08059402e+04]
16 [ 5.38949967e+02  7.51581209e+04]
17 [ 8.46514832e+02  1.85342183e+05]
18 [ 1.31563785e+03  4.47284558e+05]
19 [ 2.20002599e+03  1.24853145e+06]
20 [ 4.10019768e+03  4.39203073e+06]]
```

```
1 fig = plt.figure(figsize=(10, 7))
2 ax = fig.add_subplot(111, projection='3d')
3 surf = ax.scatter3D(X, Y, Z, c=Z, cmap='viridis', alpha=0.1)
4 x,y = x_history[:,0], x_history[:,1]
5 surf = ax.plot3D(x, y, f(x_history.T), color='orange', alpha=1)
```



```

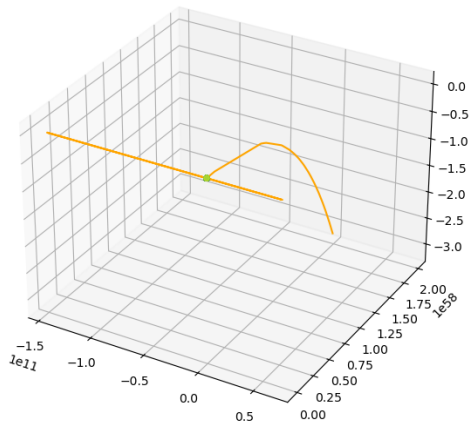
1  cb, history = callback_generator()
2  print(optimize.minimize(f, np.array([-8, -6]), method='trust-constr', tol=10**-10, callback=cb))
3
4  x_history = read_history(history)
5
6  fig = plt.figure(figsize=(10, 7))
7  ax = fig.add_subplot(111, projection='3d')
8  surf = ax.scatter3D(X, Y, Z, c=Z, cmap='viridis', alpha=0.1)
9  x,y = x_history[:,0], x_history[:,1]
10 surf = ax.plot3D(x, y, f(x_history.T), color='orange', alpha=1)

```

```

1  message: The maximum number of function evaluations is exceeded.
2      success: False
3      status: 0
4      fun: -3.112910272680709e+175
5      x: [-5.375e+06  1.982e+58]
6      nit: 1000
7      nfev: 2910
8      njev: 970
9      nhev: 0
10     cg_niter: 1866
11     cg_stop_cond: 2
12     grad: [-0.000e+00 -4.713e+117]
13     lagrangian_grad: [-0.000e+00 -4.713e+117]
14     constr: []
15     jac: []
16     constr_nfev: []
17     constr_njev: []
18     constr_nhev: []
19     v: []
20     method: equality_constrained_sq

```

```

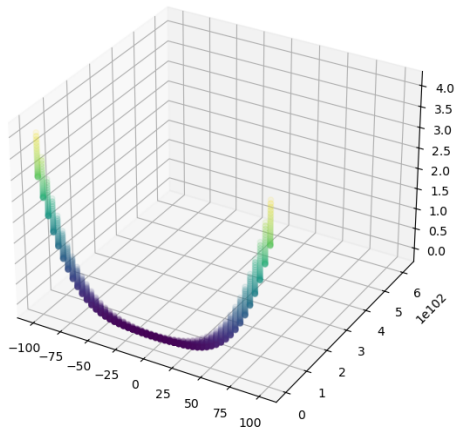
1  cb, history = callback_generator()
2  print(optimize.minimize(f, np.array([-8, -6]), method='powell', tol=10**-10, callback=cb))
3
4  x_history = read_history(history)
5
6  fig = plt.figure(figsize=(10, 7))
7  ax = fig.add_subplot(111, projection='3d')
8  surf = ax.scatter3D(X, Y, Z, c=Z, cmap='viridis', alpha=0.1)
9  x,y = x_history[:,0], x_history[:,1]
10 surf = ax.plot3D(x, y, f(x_history.T), color='orange', alpha=1)

```

```

1  /usr/lib/python3/dist-packages/scipy/optimize/_optimize.py:3049: RuntimeWarning: overflow encountered in scalar
↪ multiply
2      w = xb - ((xb - xc) * tmp2 - (xb - xa) * tmp1) / denom
3  /usr/lib/python3/dist-packages/scipy/optimize/_optimize.py:3048: RuntimeWarning: overflow encountered in scalar
↪ multiply
4      denom = 2.0 * val
5  /usr/lib/python3/dist-packages/scipy/optimize/_optimize.py:3049: RuntimeWarning: invalid value encountered in
↪ scalar divide
6      w = xb - ((xb - xc) * tmp2 - (xb - xa) * tmp1) / denom
7  /usr/lib/python3/dist-packages/scipy/optimize/_optimize.py:3042: RuntimeWarning: overflow encountered in scalar
↪ multiply
8      tmp1 = (xb - xa) * (fb - fc)
9  /usr/lib/python3/dist-packages/scipy/optimize/_optimize.py:3043: RuntimeWarning: overflow encountered in scalar
↪ multiply
10     tmp2 = (xb - xc) * (fb - fa)
11 /usr/lib/python3/dist-packages/scipy/optimize/_optimize.py:3044: RuntimeWarning: invalid value encountered in
↪ scalar subtract
12     val = tmp2 - tmp1
13 /tmp/ipykernel_1005153/2187872282.py:2: RuntimeWarning: overflow encountered in scalar multiply
14     return 3*x[0]**4 - 4*x[1]**3 + x[0]**2 * x[1]**2
15 /tmp/ipykernel_1005153/2187872282.py:2: RuntimeWarning: overflow encountered in scalar power

```

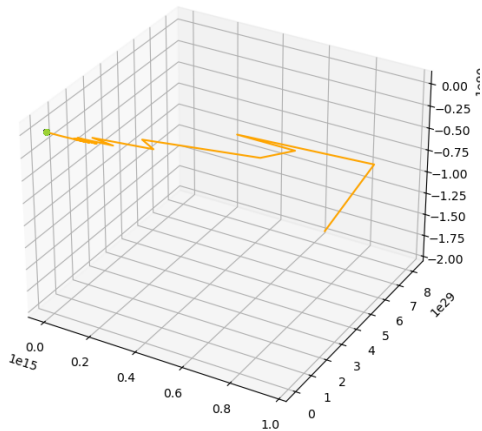


```

1  cb, history = callback_generator()
2  print(optimize.minimize(f, np.array([-8, -6]), method='nelder-mead', tol=10**-10, callback=cb))
3
4  x_history = read_history(history)
5
6  fig = plt.figure(figsize=(10, 7))
7  ax = fig.add_subplot(111, projection='3d')
8  surf = ax.scatter3D(X, Y, Z, c=Z, cmap='viridis', alpha=0.1)
9  x,y = x_history[:,0], x_history[:,1]
10 surf = ax.plot3D(x, y, f(x_history.T), color='orange', alpha=1)

1  message: Maximum number of function evaluations has been exceeded.
2      success: False
3      status: 1
4      fun: -1.9143830178334263e+90
5      x: [ 6.344e+14  8.172e+29]
6      nit: 209
7      nfev: 400
8  final_simplex: (array([[ 6.344e+14,  8.172e+29],
9                      [ 9.600e+14,  6.085e+29],
10                     [ 5.592e+14,  3.393e+29]]), array([-1.914e+90, -5.600e+89, -1.203e+89]))

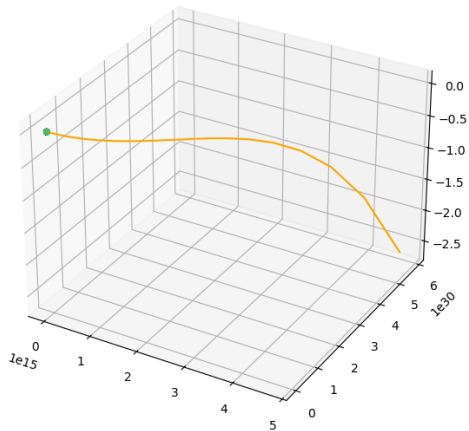
```



```
1 def f_grad(x: np.array):
2     return np.array([ 12*x[0]**3 + 2*x[0]*x[1]**2 , -12*x[1]**2 + 2*x[0]**2*x[1] ]).T
3
4 def f_hessian(x):
5     h = np.array( [[ 36*x[0]**2 + 2*x[1]**2 , 4*x[0]*x[1] ],
6                   [ 4*x[1]*x[0] , -24*x[1] + 2*x[0]**2] ] )
7     return h
```

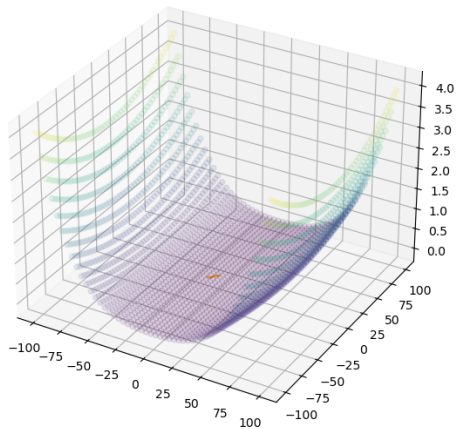
```
1 cb, history = callback_generator()
2 print(optimize.minimize(f, np.array([-8, -6]), method='BFGS', jac=f_grad, tol=10**-10, callback=cb))
3
4 x_history = read_history(history)
5
6 fig = plt.figure(figsize=(10, 7))
7 ax = fig.add_subplot(111, projection='3d')
8 surf = ax.scatter3D(X, Y, Z, c=Z, cmap='viridis', alpha=0.1)
9 x,y = x_history[:,0], x_history[:,1]
10 surf = ax.plot3D(x, y, f(x_history.T), color='orange', alpha=1)
```

```
1 message: Maximum number of iterations has been exceeded.
2 success: False
3 status: 1
4 fun: -2.6246515245352475e+91
5 x: [ 4.738e+15  5.807e+30]
6 nit: 400
7 jac: [ 3.196e+77 -1.439e+62]
8 hess_inv: [[ 4.890e-62  1.208e-46]
9            [ 1.208e-46  2.997e-31]]
10 nfev: 430
11 njev: 430
```

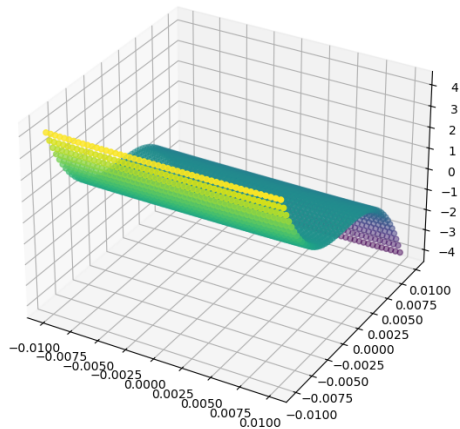



```
1  cb, history = callback_generator()
2  print(optimize.minimize(f, np.array([-8, -6]), method='trust-exact', jac=f_grad, hess=f_hessian, tol=10**-10,
   ↪  callback=cb))
3
4  x_history = read_history(history)
5
6  fig = plt.figure(figsize=(10, 7))
7  ax = fig.add_subplot(111, projection='3d')
8  surf = ax.scatter3D(X, Y, Z, c=Z, cmap='viridis', alpha=0.1)
9  x,y = x_history[:,0], x_history[:,1]
10 surf = ax.plot3D(x, y, f(np.vstack([x,y])), color='orange', alpha=1)
```

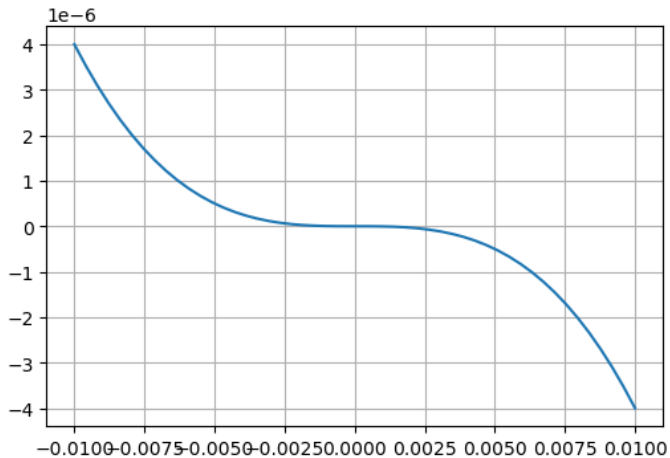
```
1  message: Optimization terminated successfully.
2  success: True
3  status: 0
4      fun: 1.3086566627333613e-15
5      x: [-1.445e-04 -1.771e-07]
6      nit: 28
7      jac: [-3.622e-11 -3.838e-13]
8      nfev: 29
9      njev: 29
10     nhev: 29
11     hess: [[ 7.519e-07  1.024e-10]
12           [ 1.024e-10  4.292e-06]]
```



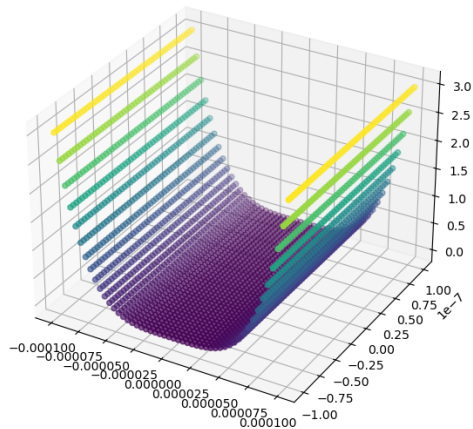
```
1 x = np.linspace(-0.01, 0.01, 50)
2 y = np.linspace(-0.01, 0.01, 50)
3 X, Y = np.meshgrid(x, y)
4 X = X.ravel()
5 Y = Y.ravel()
6
7 Z = [f([X[i],Y[i]]) for i in range(len(X))]
8
9 fig = plt.figure(figsize=(10, 7))
10 ax = fig.add_subplot(111, projection='3d')
11 surf = ax.scatter3D(X, Y, Z, c=Z, cmap='viridis')
```



```
1 plt.plot(y, f([y*0,y]));
```

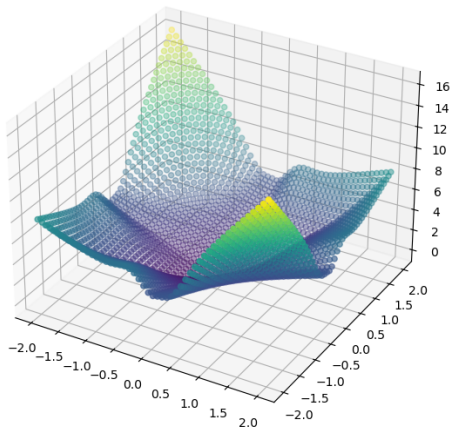


```
1 x = np.linspace(-10**-4, 10**-4, 50)
2 y = np.linspace(-10**-7, 10**-7, 50)
3 X, Y = np.meshgrid(x, y)
4 X = X.ravel()
5 Y = Y.ravel()
6
7 Z = [f([X[i],Y[i]]) for i in range(len(X))]
8
9 fig = plt.figure(figsize=(10, 7))
10 ax = fig.add_subplot(111, projection='3d')
11 surf = ax.scatter3D(X, Y, Z, c=Z, cmap='viridis')
```

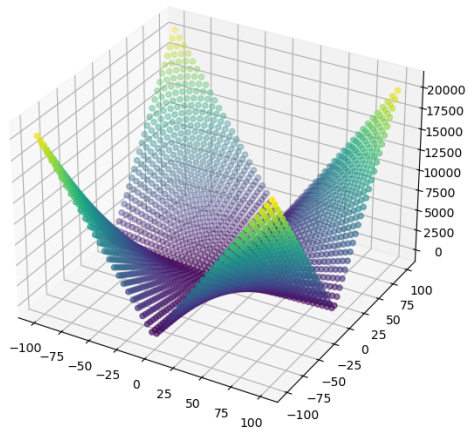



```
1 def f(x: np.array):  
2     return (abs(x[0]*x[1]) + abs(x[0]-x[1])) * (2 + np.sin(x[0]*x[1])/(1+(x[0]*x[1])**4))
```

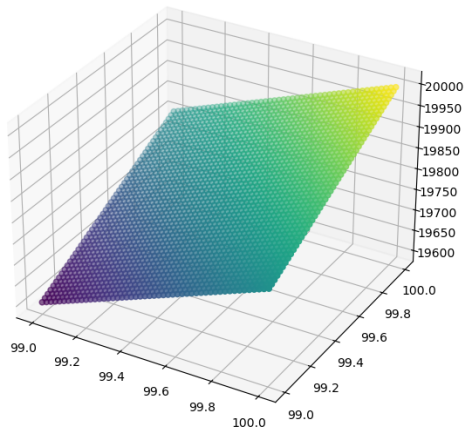
```
1 x = np.linspace(-2, 2, 50)
2 y = np.linspace(-2, 2, 50)
3 X, Y = np.meshgrid(x, y)
4 X = X.ravel()
5 Y = Y.ravel()
6 Z = [f([X[i],Y[i]]) for i in range(len(X))]
7
8 fig = plt.figure(figsize=(10, 7))
9 ax = fig.add_subplot(111, projection='3d')
10 surf = ax.scatter3D(X, Y, Z, c=Z, cmap='viridis')
```



```
1 x = np.linspace(-100, 100, 50)
2 y = np.linspace(-100, 100, 50)
3 X, Y = np.meshgrid(x, y)
4 X = X.ravel()
5 Y = Y.ravel()
6 Z = [f([X[i],Y[i]]) for i in range(len(X))]
7
8 fig = plt.figure(figsize=(10, 7))
9 ax = fig.add_subplot(111, projection='3d')
10 surf = ax.scatter3D(X, Y, Z, c=Z, cmap='viridis')
```



```
1 x = np.linspace(99, 100, 50)
2 y = np.linspace(99, 100, 50)
3 X, Y = np.meshgrid(x, y)
4 X = X.ravel()
5 Y = Y.ravel()
6 Z = [f([X[i],Y[i]]) for i in range(len(X))]
7
8 fig = plt.figure(figsize=(10, 7))
9 ax = fig.add_subplot(111, projection='3d')
10 surf = ax.scatter3D(X, Y, Z, c=Z, cmap='viridis')
```

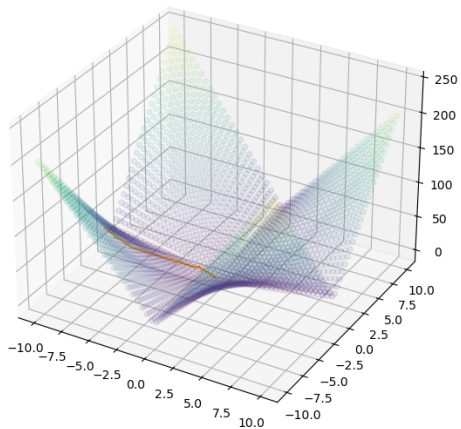


```
1 x = np.linspace(-10, 10, 50)
2 y = np.linspace(-10, 10, 50)
3 X, Y = np.meshgrid(x, y)
4 X = X.ravel()
5 Y = Y.ravel()
6 Z = [f([X[i],Y[i]]) for i in range(len(X))]
```



```
1 cb, history = callback_generator()
2 print(optimize.minimize(f, np.array([-8, -6]), method='BFGS', tol=10**-10, callback=cb))
3
4 x_history = read_history(history)
5
6 fig = plt.figure(figsize=(10, 7))
7 ax = fig.add_subplot(111, projection='3d')
8 surf = ax.scatter3D(X, Y, Z, c=Z, cmap='viridis', alpha=0.1)
9 x,y = x_history[:,0], x_history[:,1]
10 surf = ax.plot3D(x, y, f(x_history.T), color='orange', alpha=1)
```

```
1 message: Desired error not necessarily achieved due to precision loss.
2 success: False
3 status: 2
4 fun: 0.1298089039119022
5 x: [-2.509e-01 -2.509e-01]
6 nit: 6
7 jac: [ 1.259e+00  1.530e+00]
8 hess_inv: [[ 1.547e-01  6.270e-02]
9            [ 6.270e-02  2.541e-02]]
10 nfev: 234
11 njev: 74
```



```

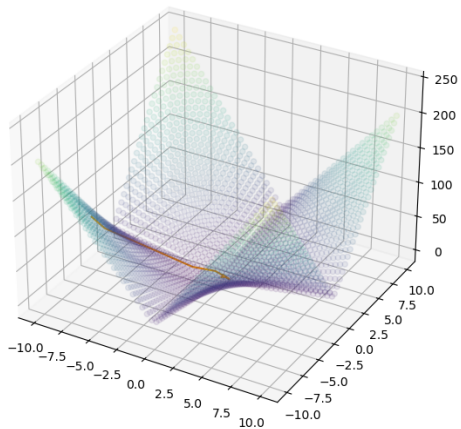
1  cb, history = callback_generator()
2  print(optimize.minimize(f, np.array([-8, -6]), method='trust-constr', tol=10**-10, callback=cb))
3
4  x_history = read_history(history)
5
6  fig = plt.figure(figsize=(10, 7))
7  ax = fig.add_subplot(111, projection='3d')
8  surf = ax.scatter3D(X, Y, Z, c=Z, cmap='viridis', alpha=0.1)
9  x,y = x_history[:,0], x_history[:,1]
10 surf = ax.plot3D(x, y, f(x_history.T), color='orange', alpha=1)

```

```

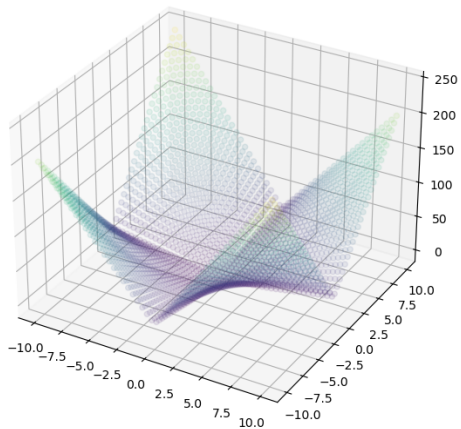
1  message: The maximum number of function evaluations is exceeded.
2      success: False
3      status: 0
4      fun: 0.006067428279899352
5      x: [ 5.504e-02  5.504e-02]
6      nit: 1000
7      nfev: 2991
8      njev: 997
9      nhev: 0
10     cg_niter: 1010
11     cg_stop_cond: 2
12     grad: [ 2.113e+00  2.113e+00]
13     lagrangian_grad: [ 2.113e+00  2.113e+00]
14     constr: []
15     jac: []
16     constr_nfev: []
17     constr_njev: []
18     constr_nhev: []
19     v: []
20     method: equality_constrained_sq

```



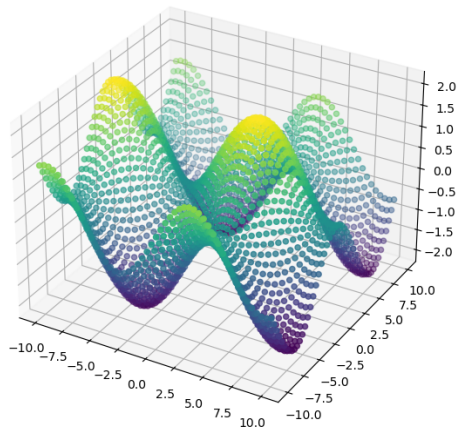
```
1 cb, history = callback_generator()
2 print(optimize.minimize(f, np.array([-8, -6]), method='powell', tol=10**-10, callback=cb))
3
4 x_history = read_history(history)
5
6 fig = plt.figure(figsize=(10, 7))
7 ax = fig.add_subplot(111, projection='3d')
8 surf = ax.scatter3D(X, Y, Z, c=Z, cmap='viridis', alpha=0.1)
9 x,y = x_history[:,0], x_history[:,1]
10 surf = ax.plot3D(x, y, f(x_history.T), color='orange', alpha=1)
```

```
1 message: Optimization terminated successfully.
2   success: True
3     status: 0
4       fun: 0.16697595553808425
5         x: [-2.833e-01 -2.833e-01]
6       nit: 3
7     direc: [[ 0.000e+00  1.000e+00]
8             [ 2.555e-12  0.000e+00]]
9     nfev: 255
```

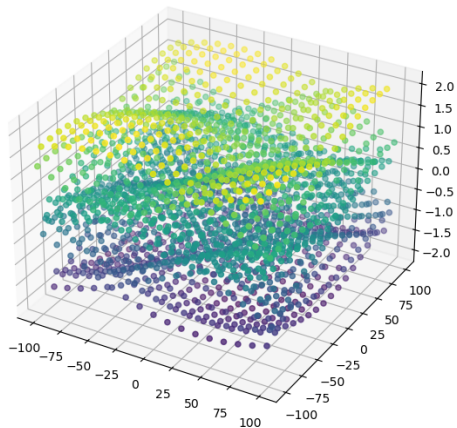


```
1 def f(x: np.array):  
2     return np.sin(x[0]/2)+np.cos(x[1]/2)
```

```
1 x = np.linspace(-10, 10, 50)
2 y = np.linspace(-10, 10, 50)
3 X, Y = np.meshgrid(x, y)
4 X = X.ravel()
5 Y = Y.ravel()
6 Z = [f([X[i],Y[i]]) for i in range(len(X))]
7
8 fig = plt.figure(figsize=(10, 7))
9 ax = fig.add_subplot(111, projection='3d')
10 surf = ax.scatter3D(X, Y, Z, c=Z, cmap='viridis')
```

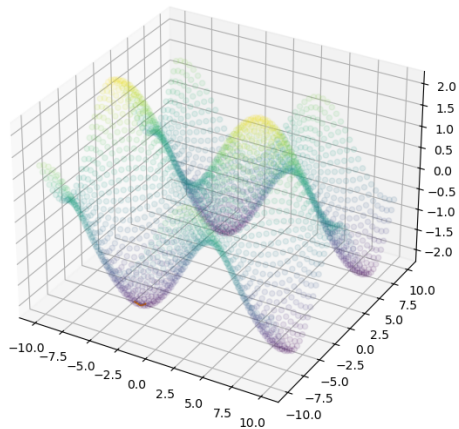
```
1 x = np.linspace(-100, 100, 50)
2 y = np.linspace(-100, 100, 50)
3 X, Y = np.meshgrid(x, y)
4 X = X.ravel()
5 Y = Y.ravel()
6 Z = [f([X[i],Y[i]]) for i in range(len(X))]
7
8 fig = plt.figure(figsize=(10, 7))
9 ax = fig.add_subplot(111, projection='3d')
10 surf = ax.scatter3D(X, Y, Z, c=Z, cmap='viridis')
```



```
1 x = np.linspace(-10, 10, 50)
2 y = np.linspace(-10, 10, 50)
3 X, Y = np.meshgrid(x, y)
4 X = X.ravel()
5 Y = Y.ravel()
6 Z = [f([X[i],Y[i]]) for i in range(len(X))]
```

```
1 cb, history = callback_generator()
2 print(optimize.minimize(f, np.array([-8, -6]), method='BFGS', tol=10**-10, callback=cb))
3
4 x_history = read_history(history)
5
6 fig = plt.figure(figsize=(10, 7))
7 ax = fig.add_subplot(111, projection='3d')
8 surf = ax.scatter3D(X, Y, Z, c=Z, cmap='viridis', alpha=0.1)
9 x,y = x_history[:,0], x_history[:,1]
10 surf = ax.plot3D(x, y, f(x_history.T), color='orange', alpha=1)
```

```
1 message: Optimization terminated successfully.
2   success: True
3   status: 0
4   fun: -2.0
5   x: [-3.142e+00 -6.283e+00]
6   nit: 8
7   jac: [ 0.000e+00  0.000e+00]
8   hess_inv: [[ 4.182e+00  2.863e-01]
9              [ 2.863e-01  3.716e+00]]
10   nfev: 42
11   njev: 14
```



```

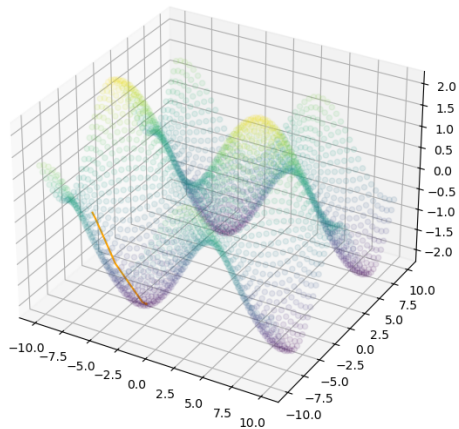
1  cb, history = callback_generator()
2  print(optimize.minimize(f, np.array([-8, -6]), method='trust-constr', tol=10**-10, callback=cb))
3
4  x_history = read_history(history)
5
6  fig = plt.figure(figsize=(10, 7))
7  ax = fig.add_subplot(111, projection='3d')
8  surf = ax.scatter3D(X, Y, Z, c=Z, cmap='viridis', alpha=0.1)
9  x,y = x_history[:,0], x_history[:,1]
10 surf = ax.plot3D(x, y, f(x_history.T), color='orange', alpha=1)

```

```

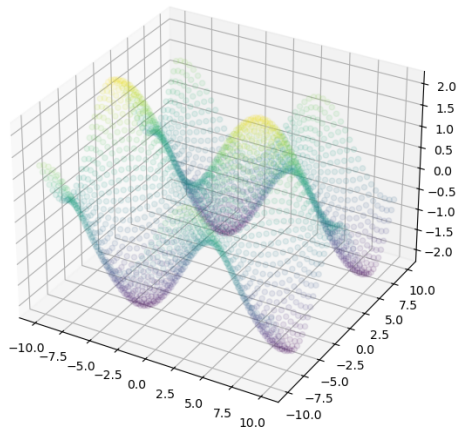
1  message: `gtol` termination condition is satisfied.
2      success: True
3      status: 1
4      fun: -1.9999999999999996
5      x: [-3.142e+00 -6.283e+00]
6      nit: 8
7      nfev: 24
8      njev: 8
9      nhev: 0
10     cg_niter: 7
11     cg_stop_cond: 4
12     grad: [-0.000e+00 -0.000e+00]
13     lagrangian_grad: [-0.000e+00 -0.000e+00]
14     constr: []
15     jac: []
16     constr_nfev: []
17     constr_njev: []
18     constr_nhev: []
19     v: []
20     method: equality_constrained_sq

```




```
1  cb, history = callback_generator()
2  print(optimize.minimize(f, np.array([-8, -6]), method='powell', tol=10**-10, callback=cb))
3
4  x_history = read_history(history)
5
6  fig = plt.figure(figsize=(10, 7))
7  ax = fig.add_subplot(111, projection='3d')
8  surf = ax.scatter3D(X, Y, Z, c=Z, cmap='viridis', alpha=0.1)
9  x,y = x_history[:,0], x_history[:,1]
10 surf = ax.plot3D(x, y, f(x_history.T), color='orange', alpha=1)
```

```
1  message: Optimization terminated successfully.
2  success: True
3  status: 0
4  fun: -2.0
5  x: [-3.142e+00 -6.283e+00]
6  nit: 2
7  direc: [[ 1.000e+00  0.000e+00]
8  [ 0.000e+00  1.000e+00]]
9  nfev: 73
```



Moral of the story...

Checking for convexity is really important...
... but sometimes really difficult.
Make sure the functions you are optimizing are at least limited!



Nelder-Mead

Derivative-free, constrained, trust-region base approach

- ▶ Define a polyhedron (simplex) as domain ($n + 1$ points in \mathbb{R}^n), the volume is the trust region
- ▶ Replace the worst vertex (highest f value) of the polyhedron with a “better” one
- ▶ until the trust region “radius” is within the required $tolx$ tolerance, or $tolf$ condition met

Let the simplex S have vertices $\mathbf{x}_0, \dots, \mathbf{x}_n$. S is nondegenerate if:

- ▶ there are never more than 3 coplanar vertices
- ▶ the matrix $V(S) \in \mathcal{M}^{n \times n} = [\mathbf{x}_n - \mathbf{x}_0, \mathbf{x}_{n-1} - \mathbf{x}_0, \dots, \mathbf{x}_1 - \mathbf{x}_0]$ is nonsingular

Nelder-Mead vertex operations

$S = \mathbf{x}_0, \dots, \mathbf{x}_n$, ordered such that $f(\mathbf{x}_0) \leq f(\mathbf{x}_1) \leq \dots f(\mathbf{x}_n)$.

The centroid is :

$$\bar{\mathbf{x}} = \frac{1}{n} \sum_{i=0}^n \mathbf{x}_i$$

which is used to define the points along the “worst” direction:

$$\bar{\mathbf{x}}(t) = \bar{\mathbf{x}} + t(\mathbf{x}_n - \bar{\mathbf{x}})$$

- ▶ REFLECT: $\mathbf{x}_n = \bar{\mathbf{x}}(-1)$
- ▶ EXPAND: $\mathbf{x}_n = \bar{\mathbf{x}}(-2)$
- ▶ CONTRACT (inside): $\mathbf{x}_n = \bar{\mathbf{x}}(-1/2)$
- ▶ CONTRACT (outside): $\mathbf{x}_n = \bar{\mathbf{x}}(1/2)$
- ▶ SHRINK: $\mathbf{x}_i = (1/2)(\mathbf{x}_0 + \mathbf{x}_i)$ (in direction of \mathbf{x}_0)

Nelder-Mead crawl

1. Given starting point \mathbf{x}_0 , generate $S = \{\mathbf{x}_i\}$, $\mathbf{x}_i = \mathbf{x}_0 + \alpha \mathbf{e}_i$
2. Check: if simplex too small or degenerate, end on best point
3. Compute centroid $\bar{\mathbf{x}} = \frac{1}{n} \sum_{i=0}^n \mathbf{x}_i$
4. Sort vertices $f(\mathbf{x}_0) \leq f(\mathbf{x}_1) \leq \dots f(\mathbf{x}_n)$, select worst \mathbf{x}_n
5. Try REFLECT, EXPAND, CONTRACT in, CONTRACT out. If new \mathbf{x} is better than old \mathbf{x}_n and within constraints, restart from (2) (prefer reflect/expand to contract!)
6. SHRINK and restart from (2)

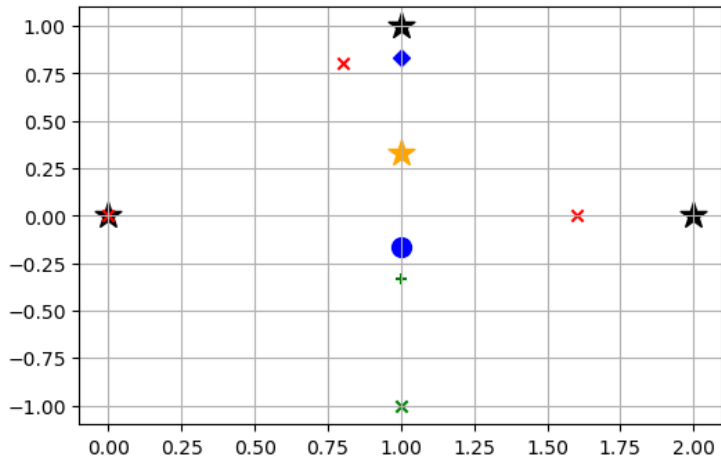
Implementation

```
1 import numpy as np
2 from scipy import optimize
3 from matplotlib import pyplot as plt
4 plt.rcParams['axes.grid'] = True
```



```
1 # m vectors of n dimensions, m x n
2 def centroid(X):
3     return 1/(X.shape[0]) * X.sum(axis=0)
4
5 def modify_x(x, centroid, operation):
6     t = {'REFLECT': -1,
7          'EXPAND': -2,
8          'CONTRACT_IN': -0.75,
9          'CONTRACT_OUT': 0.75}
10    return centroid + t[operation]*(x-centroid)
11
12 def shrink_S(sorted_X, centroid):
13    return (sorted_X + sorted_X[0])/1.25
```

```
1 X = np.array([[0, 0], [2, 0], [1,1]])
2 plt.scatter(X[:,0], X[:,1], marker='*', s=200, color='black')
3
4 C = centroid(X)
5 plt.scatter(C[0], C[1], marker='*', s=200, color='orange')
6
7 RX = modify_x(X[2], C, 'REFLECT')
8 plt.scatter(RX[0], RX[1], marker='+', color='green');
9
10 RX = modify_x(X[2], C, 'EXPAND')
11 plt.scatter(RX[0], RX[1], marker='x', color='green');
12
13 RX = modify_x(X[2], C, 'CONTRACT_IN')
14 plt.scatter(RX[0], RX[1], marker='o', s=100, color='blue');
15
16 RX = modify_x(X[2], C, 'CONTRACT_OUT')
17 plt.scatter(RX[0], RX[1], marker='D', color='blue');
18
19 NX = shrink_S(X, C)
20 plt.scatter(NX[:,0], NX[:,1], marker='x', color='red');
```



```
1 def sort_x(X, f):
2     f_vals = np.apply_along_axis(f, axis=1, arr=X)
3     return X[np.argsort(f_vals)]
4
5 def testf(x):
6     return np.sum(x, axis=0)
7
8 X = np.array([[5,4],[2,2],[3,7],[1,1],[8,4],[0,-1]])
9 sX = sort_x(X, testf)
10
11 print(testf(X.T))
12 print(testf(sX.T))

```

```
1 [ 9  4 10  2 12 -1]
2 [-1  2  4  9 10 12]
```

```
1 def simplex_too_small(X, told):
2     maxdist = 0
3     for x in X:
4         for y in X:
5             d = np.sqrt(np.sum((x-y)**2))
6             maxdist = max(maxdist, d)
7     return maxdist <= told
8
9 X = np.array([[0,0],[2,0],[1,1]])
10 print(simplex_too_small(X, 2))
11 print(simplex_too_small(X, 1.9))
```

```
1 True
2 False
```

```
1 def is_degenerate(X):  
2     return (np.linalg.matrix_rank(X[1:]-X[0]) < (len(X)-1))  
3  
4 X = np.array([[0,0],[2,0],[1,1]])  
5 print(is_degenerate(X))  
6 X = np.array([[0,0],[0.5,0.5],[1,1]])  
7 print(is_degenerate(X))
```

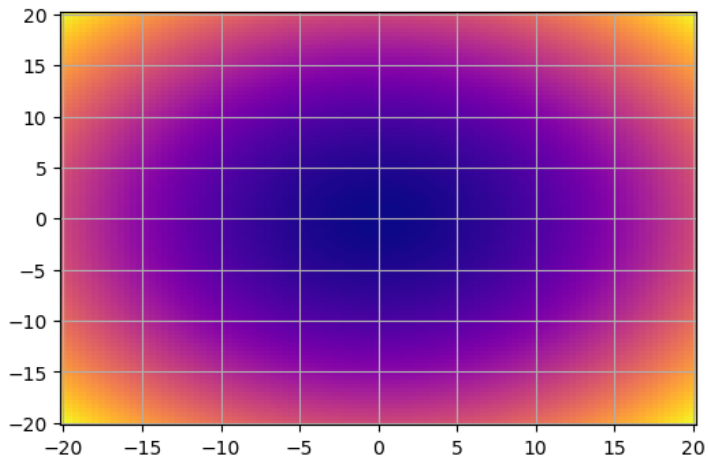
```
1 False  
2 True
```

```
1 def nelder_mead(f, x_0, told=10**-10, maxiters=1000, callback=False):
2     n = x_0.shape[0] # assume x_0 n-dimensional vector
3     X = 5 * np.eye(n) * x_0 + x_0
4     X = np.vstack([X, x_0])
5
6     counter = 0
7     while not simplex_too_small(X,told) and counter < maxiters:
8         counter += 1
9         callback(X)
10        X = sort_x(X, f)
11        C = centroid(X)
12
13        f_worst = f(X[-1])
14
15        x_reflect = modify_x(X[-1], C, 'REFLECT')
16        f_x_reflect = f(x_reflect)
17
18        if f_x_reflect <= f_worst:
19            x_expand = modify_x(X[-1], C, 'EXPAND')
20            f_x_expand = f(x_expand)
21            if f_x_expand <= f_x_reflect:
22                X[-1] = x_expand
23            else:
24                X[-1] = x_reflect
25        else:
26            x_contract_in = modify_x(X[-1], C, 'CONTRACT_IN')
27            f_x_contract_in = f(x_contract_in)
28            x_contract_out = modify_x(X[-1], C, 'CONTRACT_OUT')
29            f_x_contract_out = f(x_contract_out)
30
31            if f_x_contract_in <= f_worst:
```

```
1 def callback_generator():
2     history = list()
3     def cb(*args, **kwargs):
4         history.append([args, kwargs])
5     return cb, history
6
7 def read_history(h):
8     return np.array([h[0][0] for h in history])
9
```



```
1 def f(x):  
2     return np.sum(x**2, axis=0)  
3  
4 x = np.linspace(-20,20,100)  
5 y = np.linspace(-20,20,100)  
6 X,Y = np.meshgrid(x,y)  
7 z = f(np.vstack([X.ravel(),Y.ravel()])).reshape((100,100))  
8  
9 plt.pcolormesh(x, y, z, cmap='plasma', shading='auto');
```

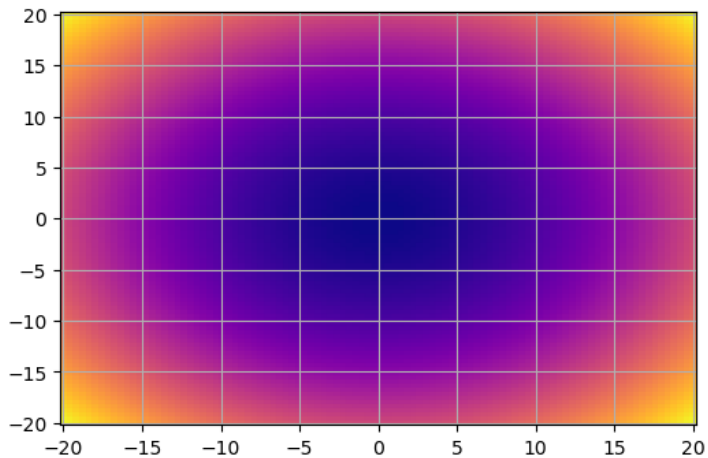


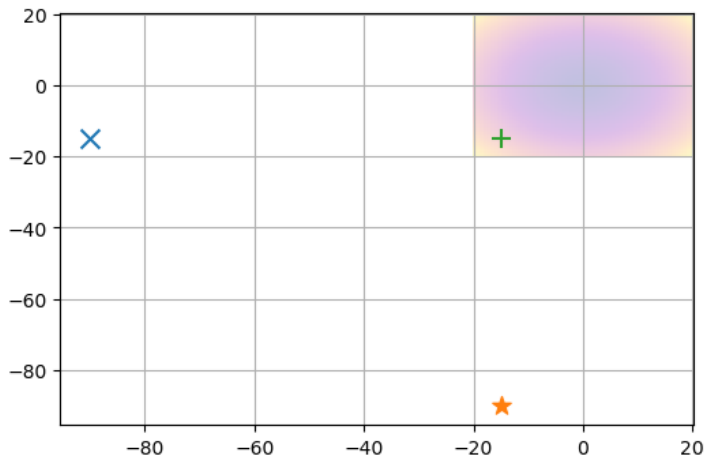
```
1 cb, history = callback_generator()
2 sol = nelder_mead(f, np.array([-15,-15]), callback=cb)
3
4 print(sol, f(sol))
5 print(len(history))
6 print(read_history(history))
7
```

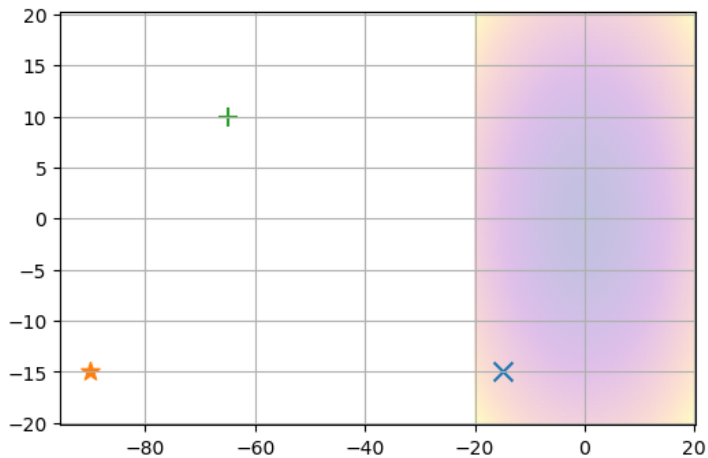
```
1 [ 1.28111797e-09 -6.25620062e-10] 2.032663716633584e-18
2 457
3 [[[-9.00000000e+01 -1.50000000e+01]
4   [-1.50000000e+01 -9.00000000e+01]
5   [-1.50000000e+01 -1.50000000e+01]]
6
7   [[-1.50000000e+01 -1.50000000e+01]
8   [-9.00000000e+01 -1.50000000e+01]
9   [-6.50000000e+01  1.00000000e+01]]
10
11  [[-1.50000000e+01 -1.50000000e+01]
12   [-6.50000000e+01  1.00000000e+01]
13   [ 1.00000000e+01  1.00000000e+01]]
14
15  ...
16
17  [[ 1.28283365e-09 -6.75181284e-10]
18   [ 1.33215250e-09 -5.77558722e-10]
19   [ 1.30663524e-09 -6.01589392e-10]]
20
21  [[ 1.30663524e-09 -6.01589392e-10]
22   [ 1.28283365e-09 -6.75181284e-10]
23   [ 1.25731639e-09 -6.99211954e-10]]
```

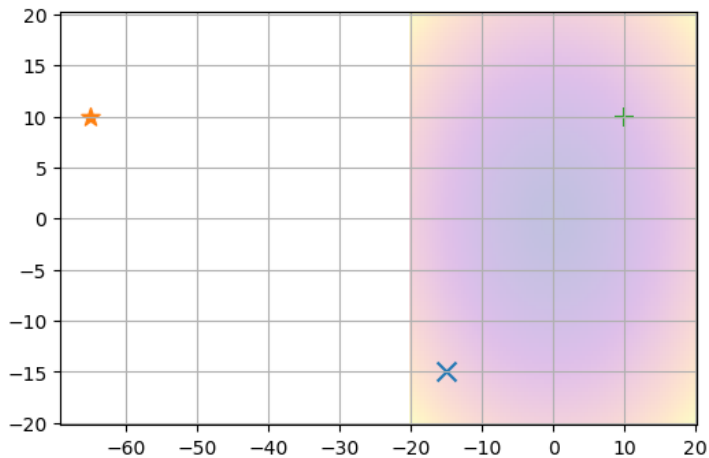
```
1  cb, history = callback_generator()
2  sol = nelder_mead(f, np.array([-15,-15]), told=10**-1, callback=cb)
3
4  print(sol, f(sol))
5  print(len(history))
6  plt.pcolormesh(x, y, z, cmap='plasma', shading='auto');
7  s = lambda n: 100#/(n+1)
8  for n,points in enumerate(read_history(history)[:10]):
9      plt.figure()
10     plt.pcolormesh(x, y, z, cmap='plasma', shading='auto', alpha=0.25);
11     a,b,c, = points
12     plt.scatter(*a, s=s(n), marker='x');
13     plt.scatter(*b, s=s(n), marker='*');
14     plt.scatter(*c, s=s(n), marker='+');

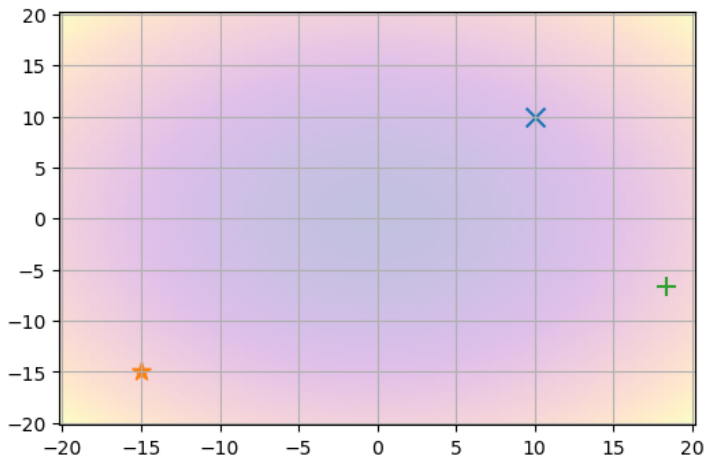
1  [ 0.00812179 -0.02246722] 0.0005707392225003524
2  23
```

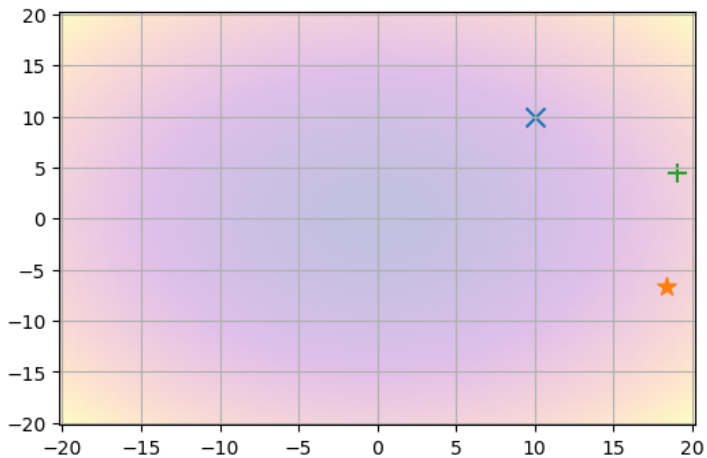


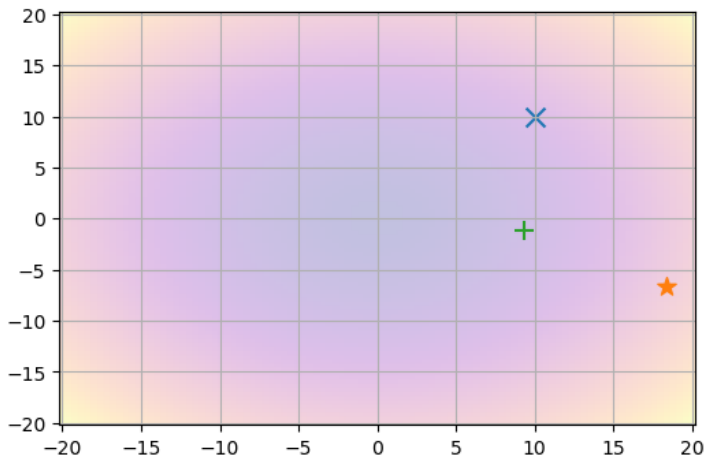


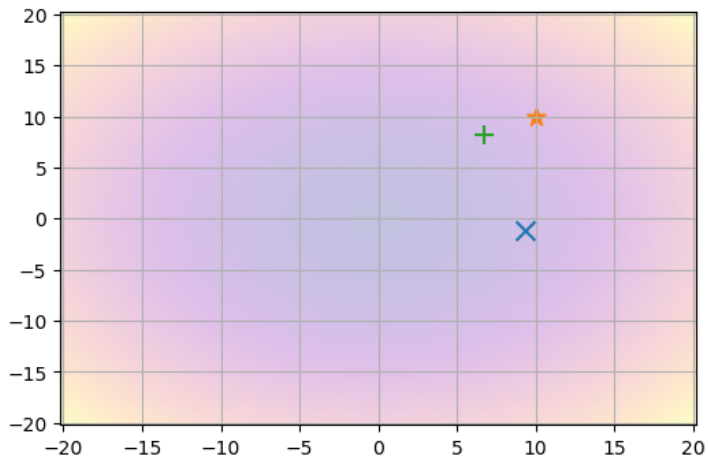


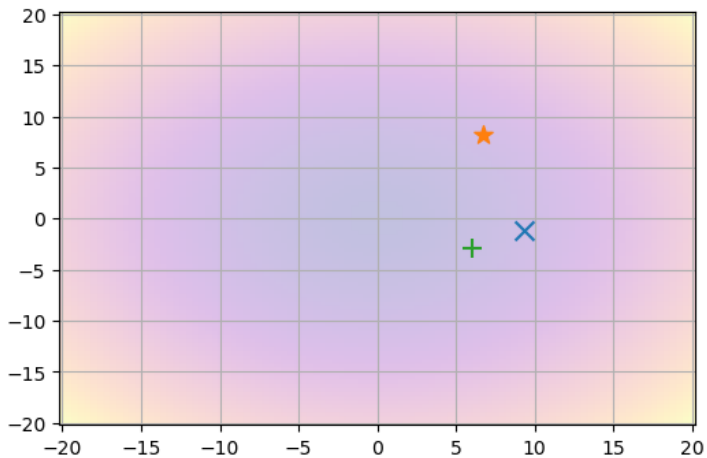


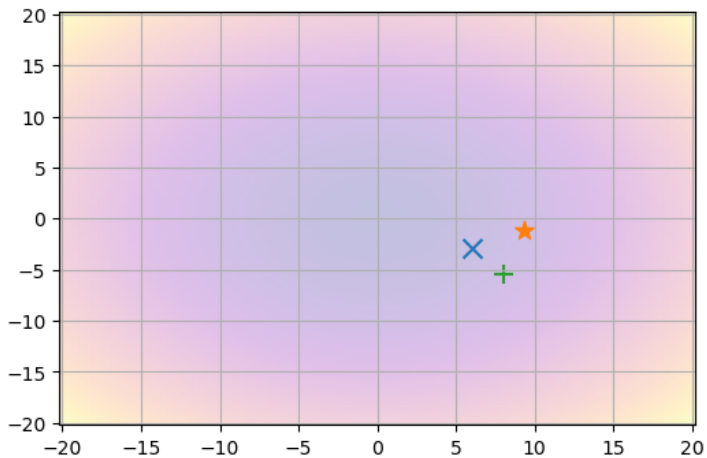


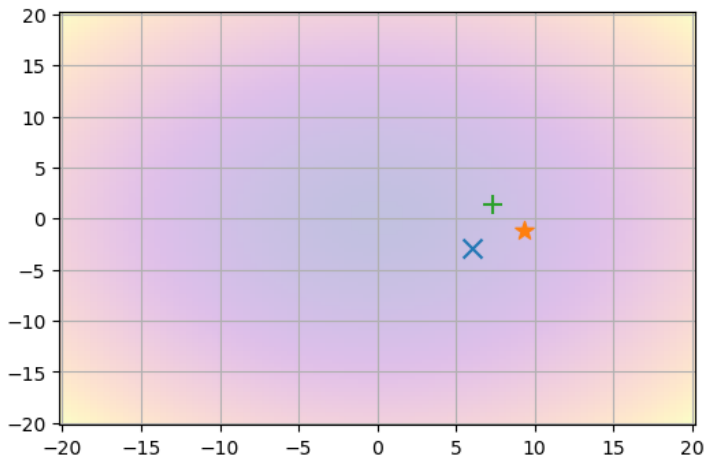












```

1  from scipy.optimize import minimize
2  cb, history = callback_generator()
3  result = minimize(f, np.array([-15,-15]), method='Nelder-Mead', callback=cb)
4
5  print(result)
6  print(history)

```

message: Optimization terminated successfully.

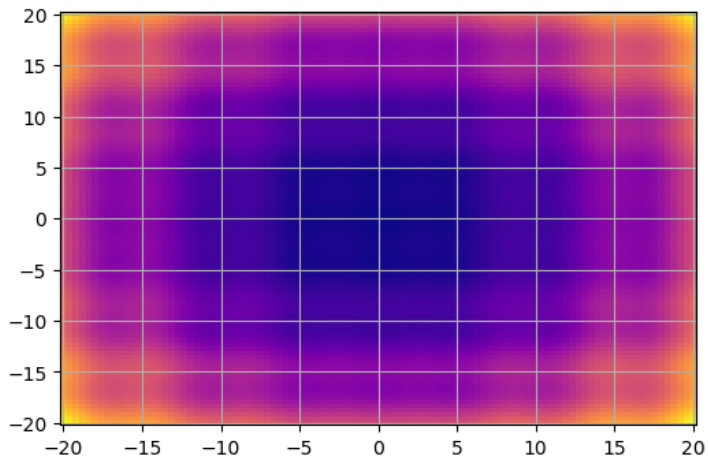
```

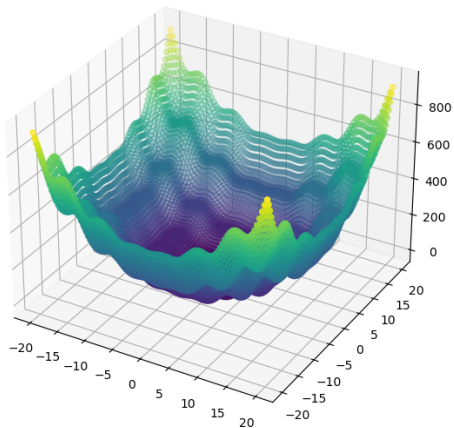
2      success: True
3      status: 0
4      fun: 1.5368435378969786e-09
5      x: [-2.245e-05 -3.214e-05]
6      nit: 48
7      nfev: 87
8      final_simplex: (array([[-2.245e-05, -3.214e-05],
9                          [ 5.734e-05,  9.610e-06],
10                         [ 5.199e-05, -4.571e-05]]), array([ 1.537e-09,  3.381e-09,  4.793e-09]))
11  [[(array([-15., -15.]),), {}], [(array([-14.625, -13.875]),), {}], [(array([-14.625, -13.875]),), {}],
→ [(array([-12.75, -12.75]),), {}], [(array([-12.75, -12.75]),), {}], [(array([-10.125, -9.375]),), {}],
→ [(array([-10.125, -9.375]),), {}], [(array([-3.75, -3.75]),), {}], [(array([-3.75, -3.75]),), {}],
→ [(array([1.875, 2.625]),), {}], [(array([1.875, 2.625]),), {}], [(array([-0.75, -0.75]),), {}],
→ [(array([-0.75, -0.75]),), {}], [(array([-0.75, -0.75]),), {}], [(array([0.46875, 0.28125]),), {}],
→ [(array([0.1171875, 0.4453125]),), {}], [(array([-0.22851562, -0.19335938]),), {}], [(array([0.20654297,
→ 0.20361328]),), {}], [(array([-0.07507324, -0.21496582]),), {}], [(array([-0.08139038, -0.09951782]),), {}],
→ [(array([0.06415558, 0.02318573]),), {}], [(array([0.02461052, 0.05023384]),), {}], [(array([-0.01850367,
→ -0.03140402]),), {}], [(array([-0.02749765, 0.0025295 ]),), {}], [(array([0.00080493, 0.01789829]),), {}],
→ [(array([0.00080493, 0.01789829]),), {}], [(array([0.01237757, 0.00477373]),), {}], [(array([-0.00466688,
→ 0.00037047]),), {}], [(array([-0.00466688, 0.00037047]),), {}], [(array([-0.00085567, -0.00031301]),), {}],
→ [(array([0.00311535, -0.00215119]),), {}], [(array([0.00279572, 0.00015819]),), {}], [(array([-0.00085567,
→ -0.00031301]),), {}], [(array([-0.00085567, -0.00031301]),), {}], [(array([-0.00085567, -0.00031301]),), {}],
→ [(array([-0.00085567, -0.00031301]),), {}], [(array([ 0.00031535, -0.00038227]),), {}],
→ [(array([3.09977062e-04, 7.61451720e-05]),), {}], [(array([3.09977062e-04, 7.61451720e-05]),), {}],
→ [(array([3.09977062e-04, 7.61451720e-05]),), {}], [(array([-1.27477568e-04, -4.11388547e-05]),), {}],
→ [(array([-1.27477568e-04, -4.11388547e-05]),), {}], [(array([9.99153955e-05, 5.81477130e-05]),), {}],
→ [(array([9.99153955e-05, 5.81477130e-05]),), {}], [(array([-2.24467344e-05, -3.21401253e-05]),), {}]]

```



```
1 def f(x):
2     return np.sum(x**2 + 3*x*np.sin(x) , axis=0)
3
4 x = np.linspace(-20,20,100)
5 y = np.linspace(-20,20,100)
6 X,Y = np.meshgrid(x,y)
7 z = f(np.vstack([X.ravel(),Y.ravel()])).reshape((100,100))
8
9 plt.pcolormesh(x, y, z, cmap='plasma', shading='auto');
10 fig = plt.figure(figsize=(10, 7))
11 ax = fig.add_subplot(111, projection='3d')
12 Z = f(np.vstack([X.ravel(),Y.ravel()]))
13 surf = ax.scatter3D(X.ravel(), Y.ravel(), Z, c=Z, cmap='viridis')
```





```
1 cb, history = callback_generator()
2 sol = nelder_mead(f, np.array([-15,-15]), callback=cb)
3
4 print(sol, f(sol))
5 print(len(history))
6 print(read_history(history))
7
```

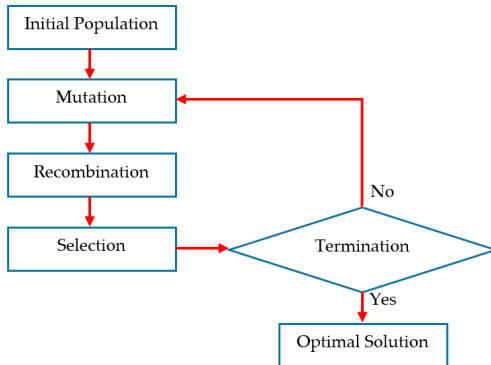
```
1 [6.78086133e+00 2.77454271e-07] 55.691320151649876
2 1000
3 [[-9.00000000e+01 -1.50000000e+01]
4  [-1.50000000e+01 -9.00000000e+01]
5  [-1.50000000e+01 -1.50000000e+01]]
6
7  [[-1.50000000e+01 -1.50000000e+01]
8  [-9.00000000e+01 -1.50000000e+01]
9  [-6.50000000e+01 1.00000000e+01]]
10
11 [[-1.50000000e+01 -1.50000000e+01]
12  [-6.50000000e+01 1.00000000e+01]
13  [1.00000000e+01 1.00000000e+01]]
14
15 ...
16
17 [[ 6.78086134e+00 2.76096811e-07]
18  [ 6.78086134e+00 2.75794364e-07]
19  [ 6.78086134e+00 2.76473094e-07]]
20
21 [[ 6.78086134e+00 2.76473094e-07]
22  [ 6.78086134e+00 2.76096811e-07]
23  [ 6.78086133e+00 2.76775541e-07]]
```

```
1 from scipy.optimize import minimize
2 cb, history = callback_generator()
3 result = minimize(f, np.array([-15,-15]), method='Nelder-Mead', callback=cb)
4
5 print(result)
```

```
1 message: Optimization terminated successfully.
2     success: True
3     status: 0
4     fun: 473.5918048265534
5     x: [-1.661e+01 -1.661e+01]
6     nit: 39
7     nfev: 72
8     final_simplex: (array([[ -1.661e+01,  -1.661e+01],
9                          [ -1.661e+01,  -1.661e+01],
10                         [ -1.661e+01,  -1.661e+01]]), array([ 4.736e+02,  4.736e+02,  4.736e+02]))
```

Genetic Algorithms

- ▶ when function is complex, non-differentiable, multi-modal, vast search-space
- ▶ when dimensionality of search space is really high
- ▶ when there is high sensitivity to starting point
- ▶ when function is noisy or with random components



Differential Evolution

N_p population of vectors

Mutation:

$$\mathbf{u} = \mathbf{x}_{r1} + F(\mathbf{x}_{r2} - \mathbf{x}_{r3})$$

where F is the mutation *scale factor*

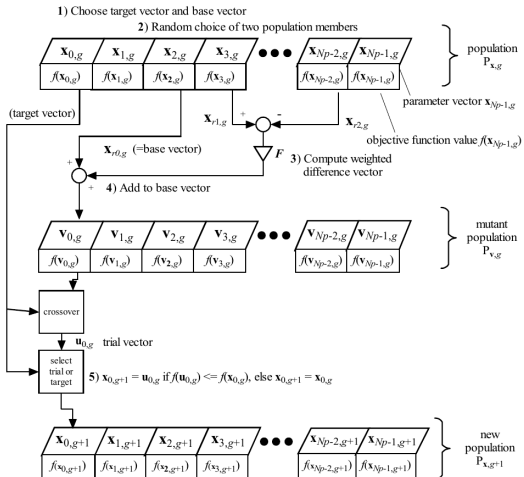
Crossover:

$$(\mathbf{v})_j = \begin{cases} (\mathbf{u}_i)_j, & \text{if } \text{rand}(0, 1) \leq C_r \\ (\mathbf{x}_i)_j, & \text{otherwise} \end{cases}$$

Differential Evolution (2)

```
1  while (convergence criterion not met):
2  for i in range( $N_p$ ):
3      r1 = random_int(0,  $N_p$ )
4      r2 = random_int(0,  $N_p$ )
5      r3 = random_int(0,  $N_p$ )
6
7       $\mathbf{u} = \mathbf{x}_{r1} + F * (\mathbf{x}_{r2} - \mathbf{x}_{r3})$ 
8
9      for j in range(len( $\mathbf{u}_i$ )):
10         if rand(0,1) >  $C_r$ :
11              $\mathbf{u}[j] = \mathbf{x}_{r1}[j]$ 
12
13     if  $f(\mathbf{u}) \leq f(\mathbf{x}_i)$ :
14          $\mathbf{x}_i = \mathbf{u}$ 
```


Differential Evolution (3)



Differential Evolution: Parameters

- ▶ N_p : population size
- ▶ F : combination scaling factor
- ▶ C_r : crossover threshold
- ▶ mutating vector selection: random-to-random, current-to-best, best,
- ▶ number of difference vectors
- ▶ distribution for crossover test (random, exp, binomial...)

DE/<mutating vector selection>/<mutators number>/<crossover distribution>:

DE/rand/1/bin DE/rand-to-best/2/exp

How to initialize the population? (grid...)

```
1 import numpy as np
2 from matplotlib import pyplot as plt
3 from scipy.optimize import differential_evolution
4 import random
5 random.seed(69)
6 plt.rcParams['axes.grid'] = True
```

```
1 def sort_x(X, f):
2     f_vals = np.apply_along_axis(f, axis=1, arr=X)
3     return X[np.argsort(f_vals)]
4
5 def simplex_too_small(X, told):
6     maxdist = 0
7     for x in X:
8         for y in X:
9             d = np.sqrt(np.sum((x-y)**2))
10            maxdist = max(maxdist, d)
11     return maxdist <= told
12
13 def grid(bounds, Np): #Bounds: one row: min/max per dimension
14     d = len(bounds) #dimensions
15     n = int(Np**(1/d)) # grid must have Np^(1/d) points per dimension to have Np ~ n^d
16     G = np.meshgrid(*[np.linspace(m,M,n) for m,M in bounds])
17     return np.vstack([x.ravel() for x in G]).T
```

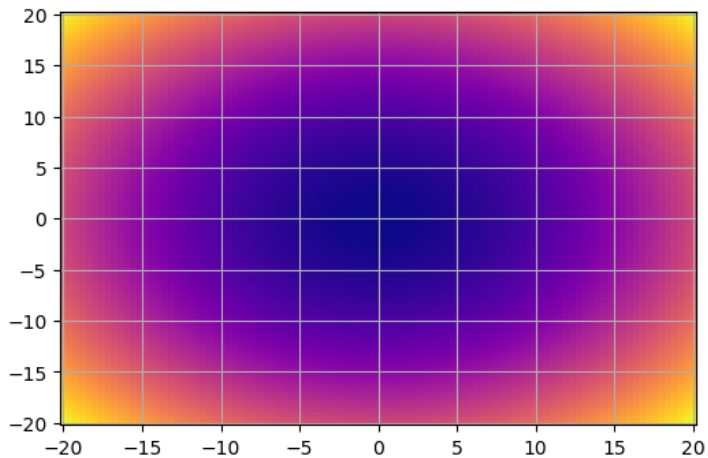
```
1 bounds = np.array([[-20,20],[-20,20]])
2 Np = 10
3 G = grid(bounds, Np)
4 print(G)
5 print(len(G))
```

```
1 [[-20. -20.]
2  [  0. -20.]
3  [ 20. -20.]
4  [-20.  0.]
5  [  0.  0.]
6  [ 20.  0.]
7  [-20. 20.]
8  [  0. 20.]
9  [ 20. 20.]]
10 9
```

```
1 # DE/rand/1/rand
2 def DE_rand_1_rand(f, bounds, Np=10, F=0.9, Cr=0.05, told=1e-6, callback=False):
3     d = len(bounds)
4     P = grid(bounds, Np)
5     Np = len(P)
6     not_assigned_count = 0
7     callback(P.copy())
8
9     while not simplex_too_small(P, told) and not_assigned_count < 10:
10         not_assigned_count += 1
11
12         for i in range(Np):
13             r1, r2, r3 = random.randint(0,Np-1), random.randint(0,Np-1), random.randint(0,Np-1)
14
15             u = P[r1] + F * (P[r2]-P[r3])
16
17             for j in range(d):
18                 if random.random() < Cr:
19                     u[j] = P[r1][j]
20
21             if f(u) < f(P[i]):
22                 P[i] = u
23                 not_assigned_count = 0
24
25         callback(P.copy())
26
27     return sort_x(P, f)[0]
28
```

```
1 def callback_generator():
2     history = list()
3     def cb(*args, **kwargs):
4         history.append([args, kwargs])
5     return cb, history
6
7 def read_history(h):
8     return np.array([h[0][0] for h in history])
```

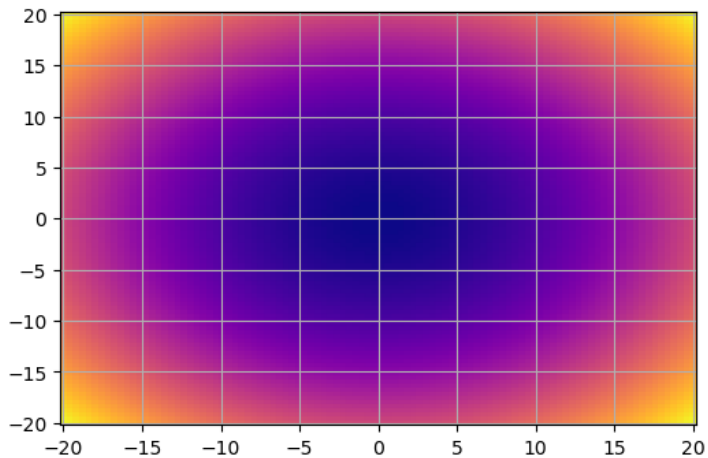
```
1 def f(x):  
2     return np.sum(x**2, axis=0)  
3  
4 x = np.linspace(-20,20,100)  
5 y = np.linspace(-20,20,100)  
6 Xf,Yf = np.meshgrid(x,y)  
7 zf = f(np.vstack([Xf.ravel(),Yf.ravel()])).reshape((100,100))  
8  
9 plt.pcolormesh(x, y, zf, cmap='plasma', shading='auto');
```

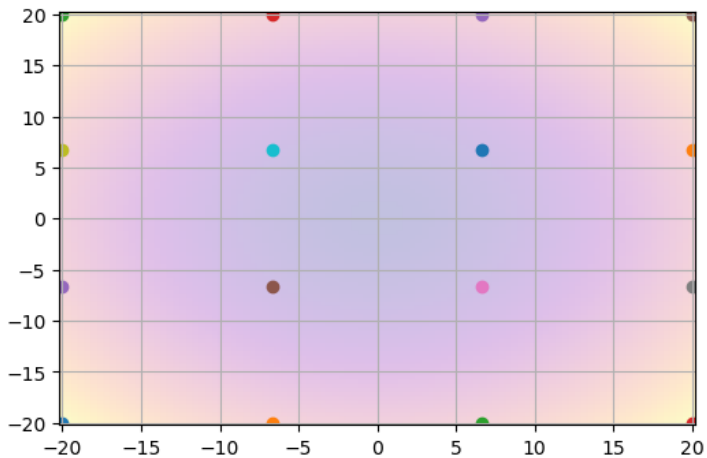



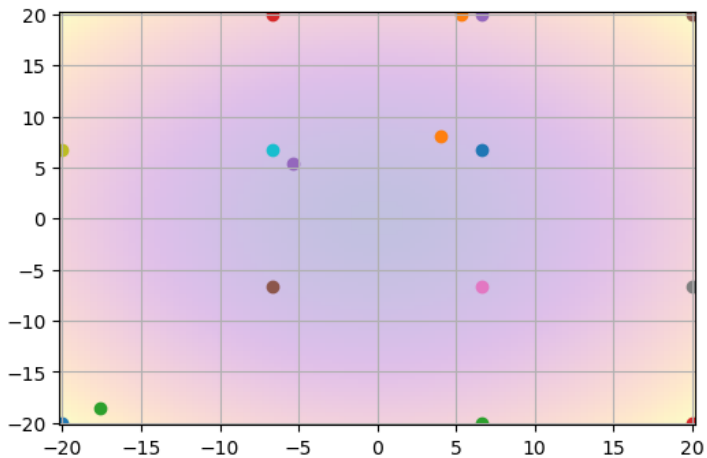
```
1 cb, history = callback_generator()
2 bounds = np.array([[-20,20],[-20,20]])
3 sol = DE_rand_1_rand(f, bounds, Np=20, callback=cb)
4
5 print(sol, f(sol))
6 print(len(history))

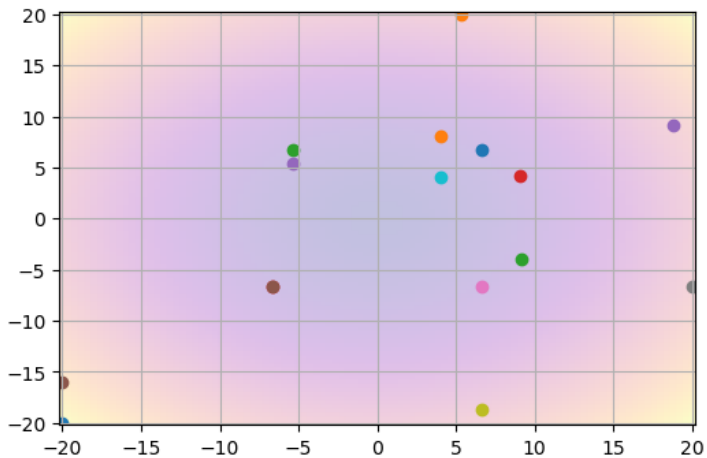
1 [-1.10665989e-07 -4.70027288e-08] 1.4456217611947347e-14
2 81
```

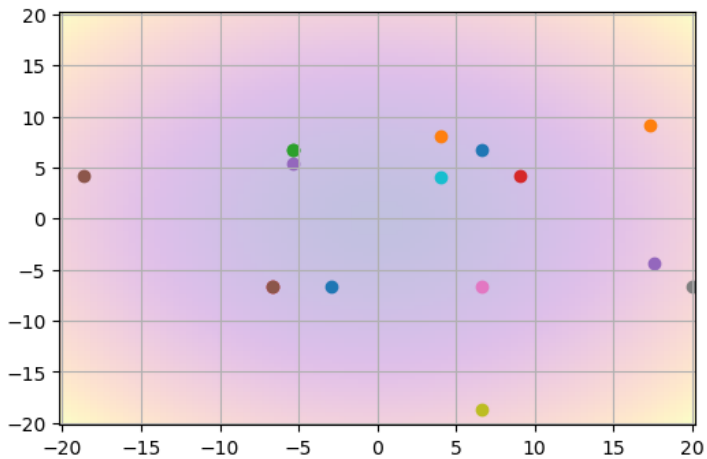
```
1 plt.pcolormesh(x, y, zf, cmap='plasma', shading='auto');  
2 for n,points in enumerate(read_history(history)[:10]):  
3     plt.figure()  
4     plt.pcolormesh(x, y, zf, cmap='plasma', shading='auto', alpha=0.25);  
5     for p in points:  
6         plt.scatter(*p);  
7
```

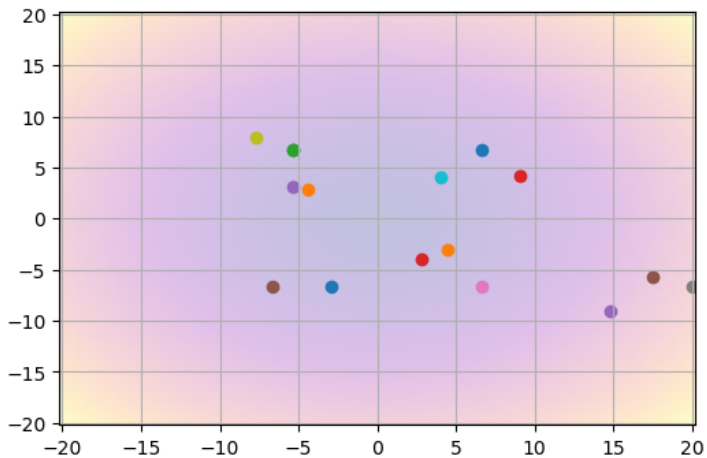


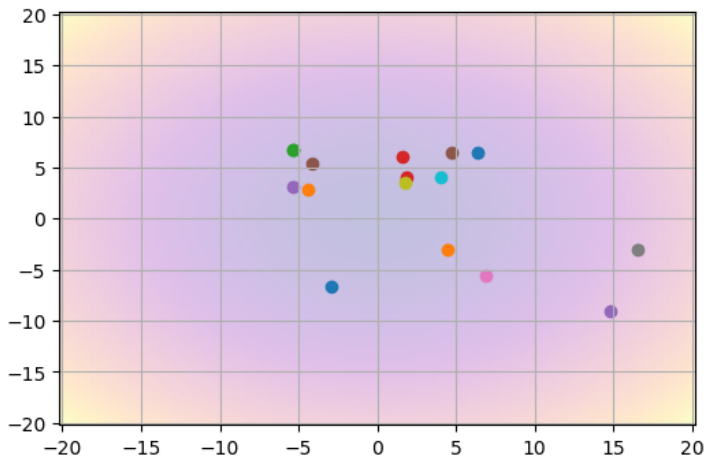


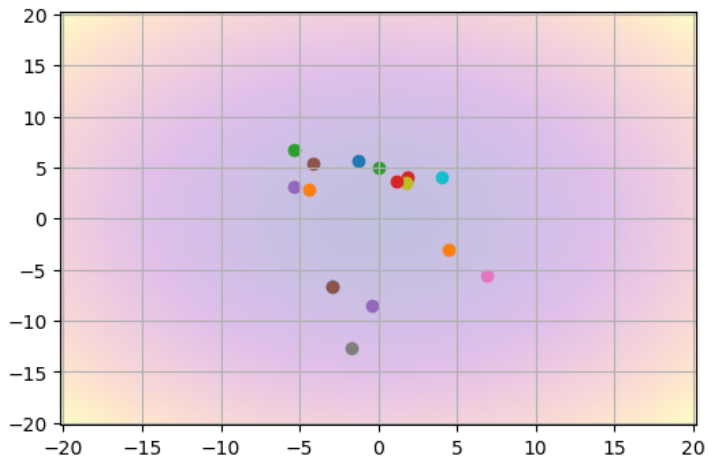


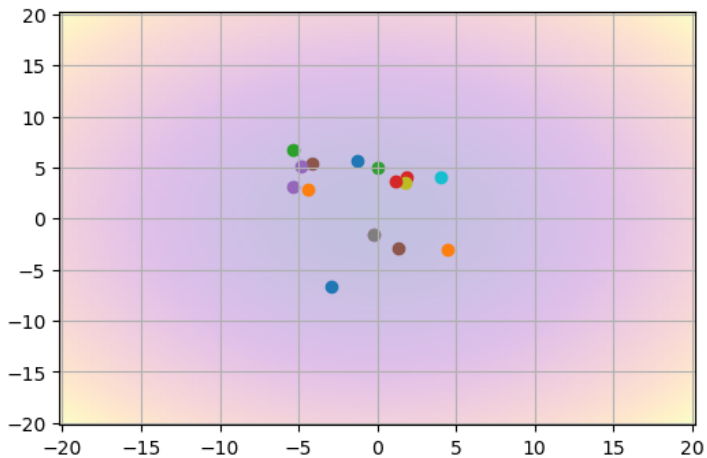


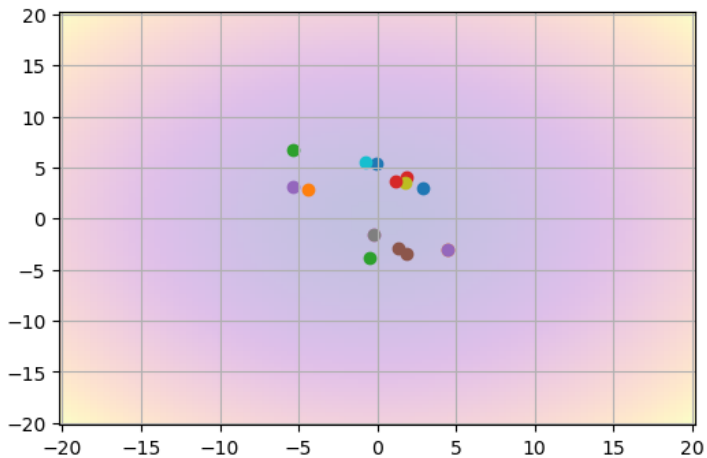


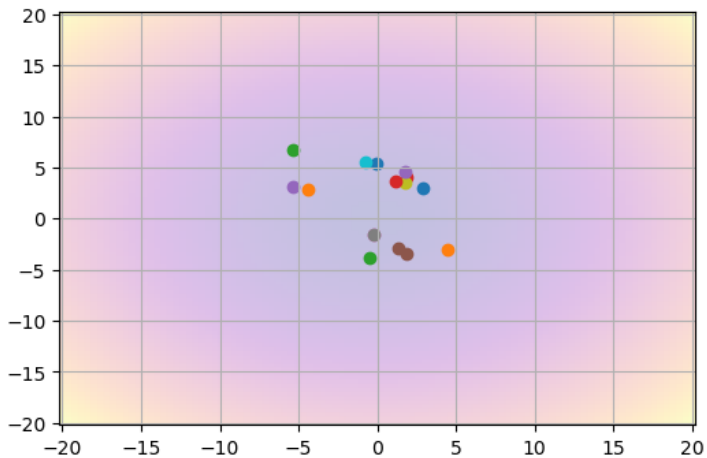




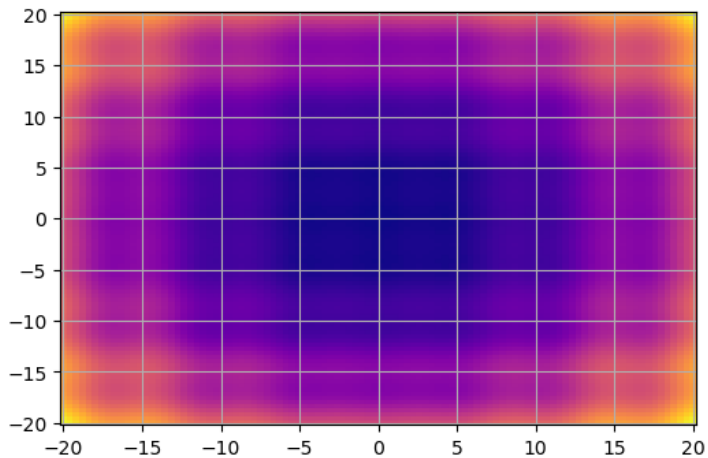








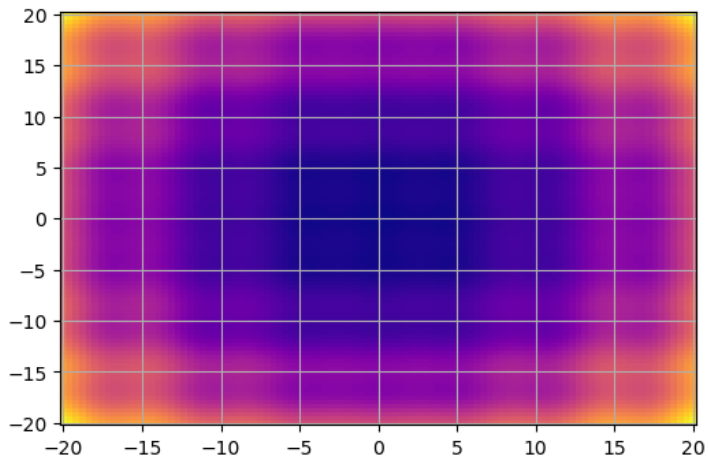
```
1 def g(x):  
2     return np.sum(x**2 + 3*x*np.sin(x) , axis=0)  
3  
4 x = np.linspace(-20,20,100)  
5 y = np.linspace(-20,20,100)  
6 Xg,Yg = np.meshgrid(x,y)  
7 zg = g(np.vstack([Xg.ravel(),Yg.ravel()])).reshape((100,100))  
8  
9 plt.pcolormesh(x, y, zg, cmap='plasma', shading='auto');
```

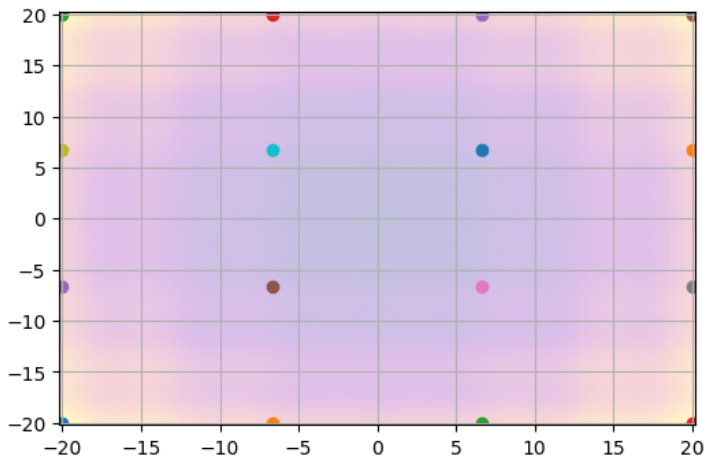


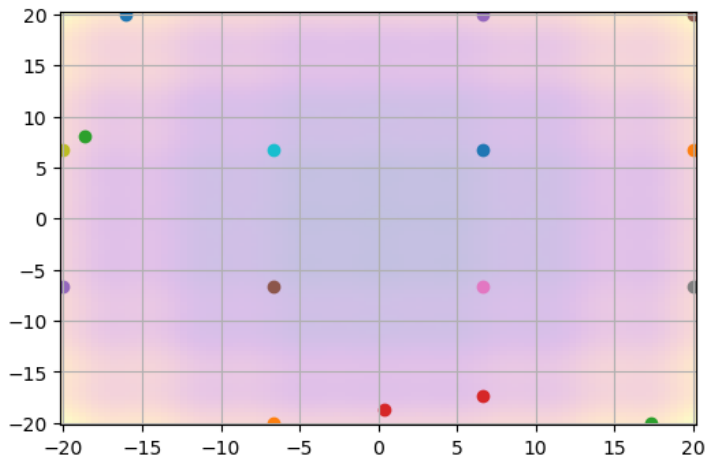

```
1 cb, history = callback_generator()
2 sol = DE_rand_1_rand(g, bounds, Np=20, callback=cb)
3
4 print(sol, f(sol))
5 print(len(history))
```

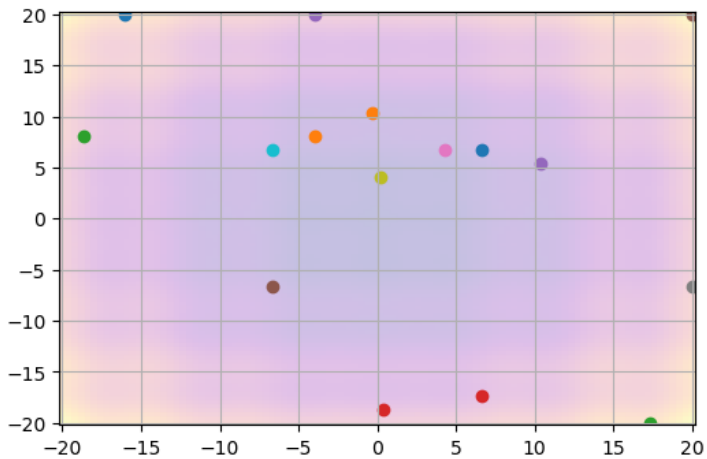
```
1 [4.29994486e-09 1.52807601e-09] 2.0824542143396605e-17
2 91
```

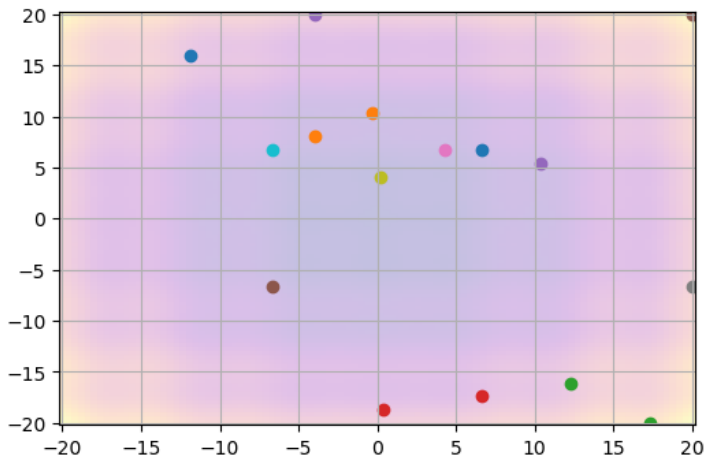
```
1 plt.pcolormesh(x, y, zg, cmap='plasma', shading='auto');
2 for n,points in enumerate(read_history(history)[:10]):
3     plt.figure()
4     plt.pcolormesh(x, y, zg, cmap='plasma', shading='auto', alpha=0.25);
5     for p in points:
6         plt.scatter(*p);
7
```

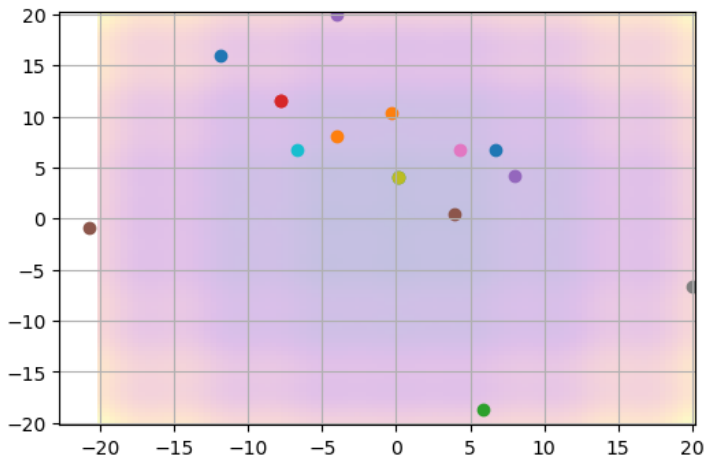


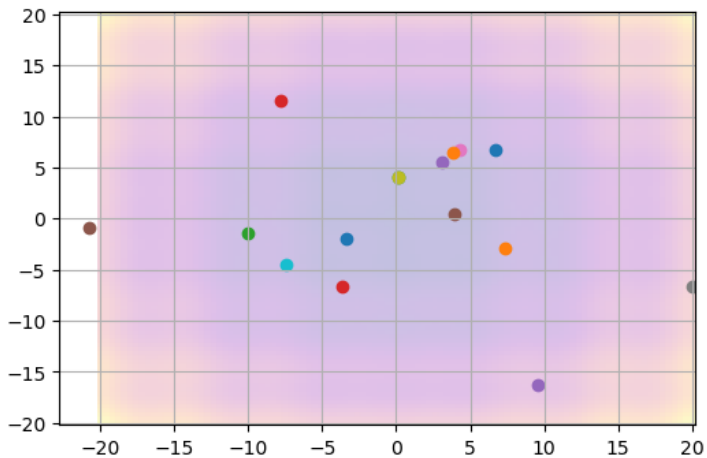


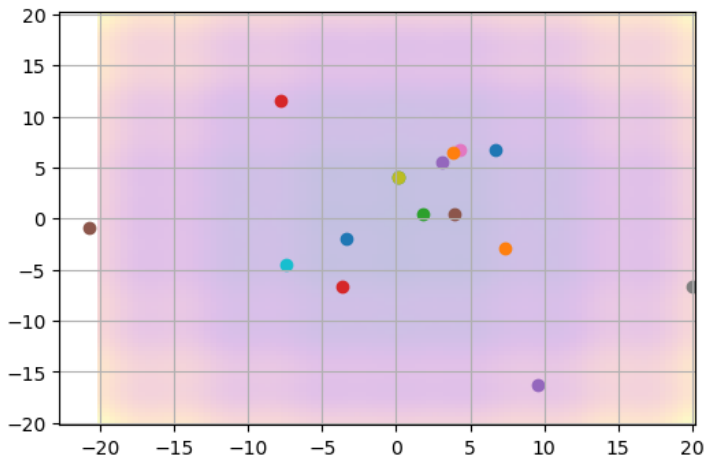


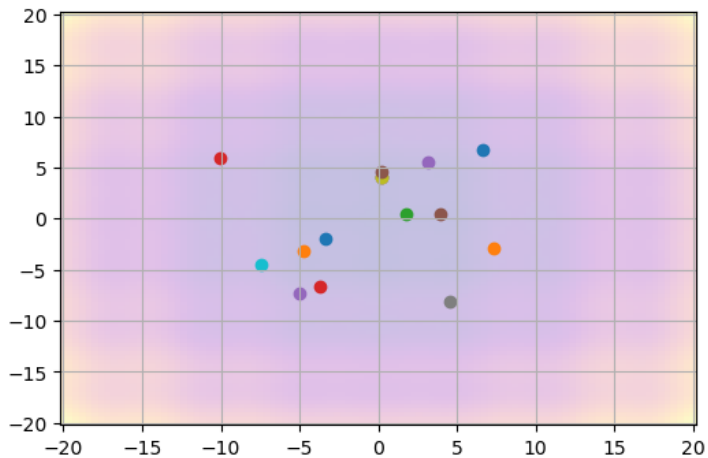


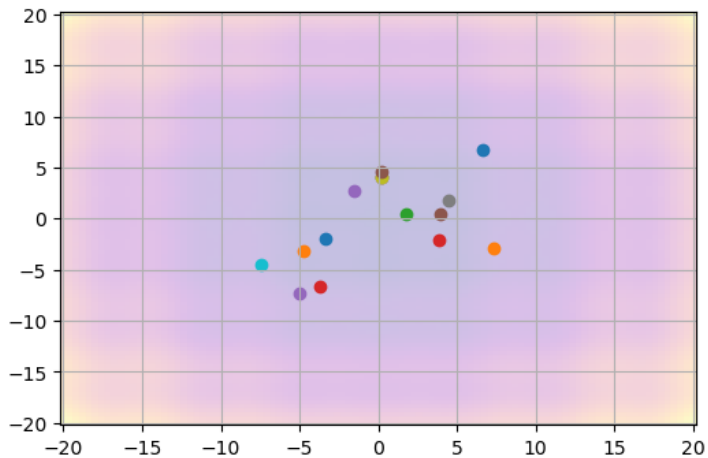


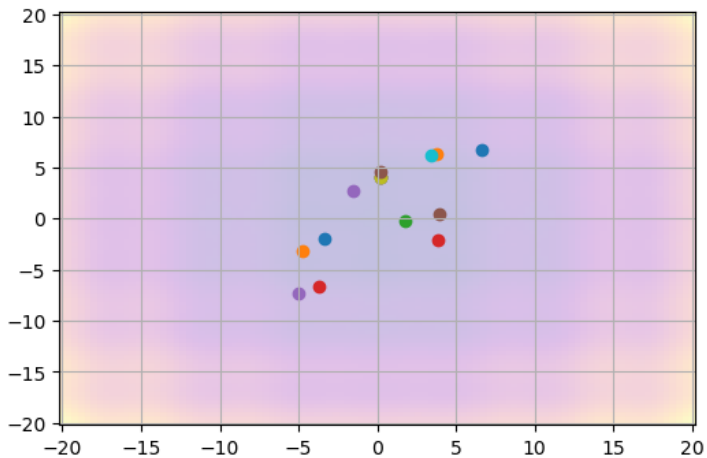










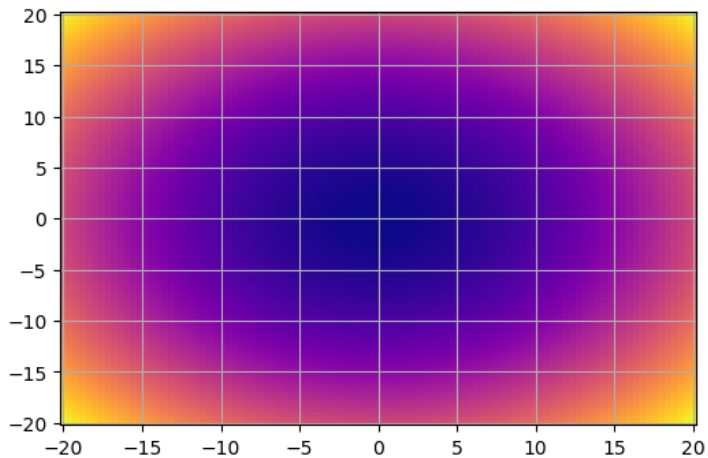


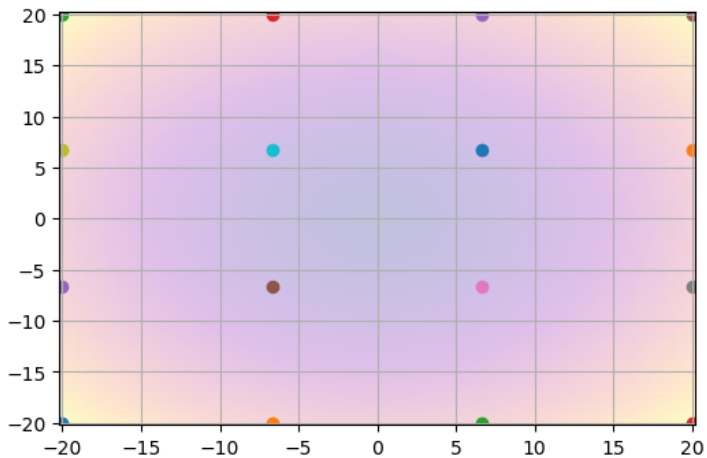
```
1 # DE/best/1/rand
2 def DE_best_1_rand(f, bounds, Np=10, F=0.99, Cr=0.1, told=1e-6, callback=False):
3     d = len(bounds)
4     P = grid(bounds, Np)
5     Np = len(P)
6     not_assigned_count = 0
7     callback(P.copy())
8
9     while not simplex_too_small(P, told) and not_assigned_count < 10:
10         not_assigned_count += 1
11
12         for i in range(Np):
13             P = sort_x(P, f)
14
15             r1, r2, r3 = 0, random.randint(0, Np-1), random.randint(0, Np-1)
16             u = P[r1] + F * (P[r2]-P[r3])
17
18             for j in range(d):
19                 if random.random() < Cr:
20                     u[j] = P[r1][j]
21
22             if f(u) < f(P[i]):
23                 P[i] = u
24                 not_assigned_count = 0
25             callback(P.copy())
26
27     return sort_x(P, f)[0]
28
```

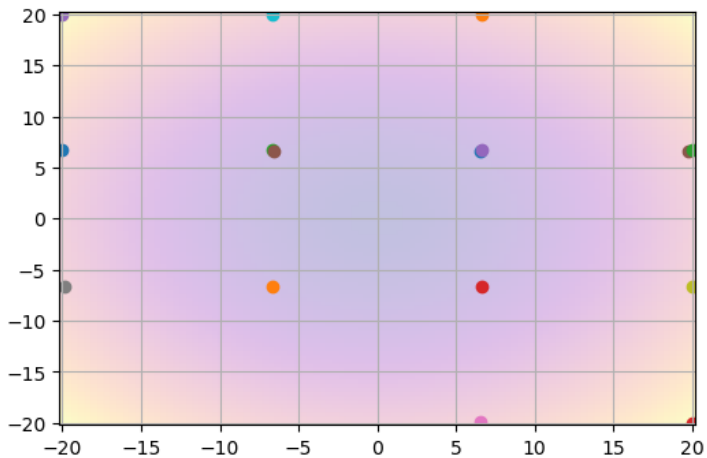
```
1 cb, history = callback_generator()
2 sol = DE_best_1_rand(f, bounds, Np=20, callback=cb)
3
4 print(sol, f(sol))
5 print(len(history))

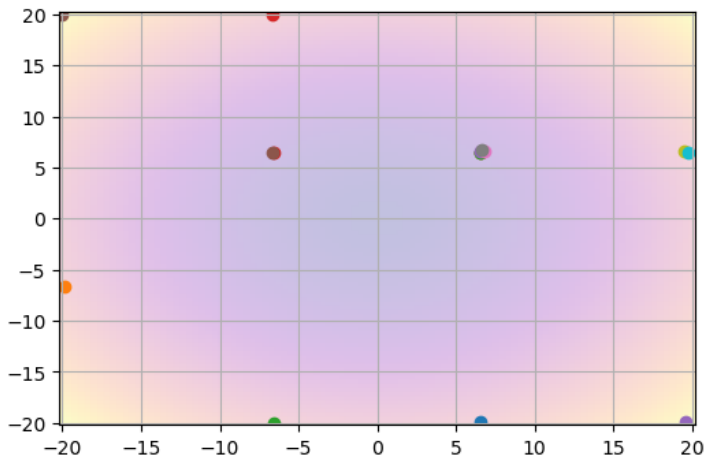
1 [-5.64367123e-02 -3.22203685e-05] 0.003185103531369431
2 31
```

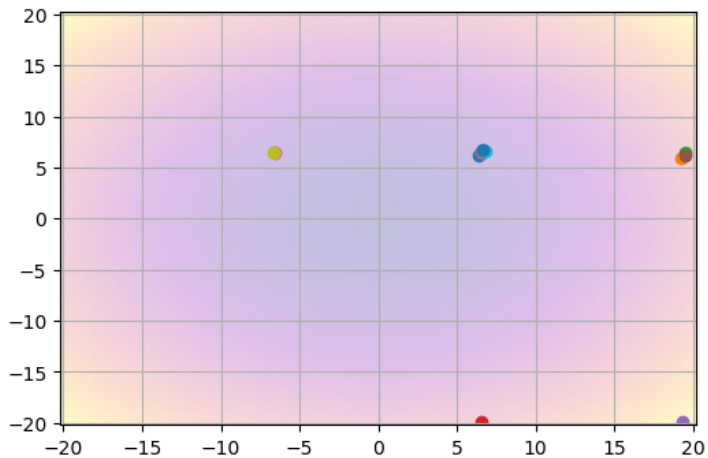
```
1 plt.pcolormesh(x, y, zf, cmap='plasma', shading='auto');
2 for n,points in enumerate(read_history(history)[:10]):
3     plt.figure()
4     plt.pcolormesh(x, y, zf, cmap='plasma', shading='auto', alpha=0.25);
5     for p in points:
6         plt.scatter(*p);
```

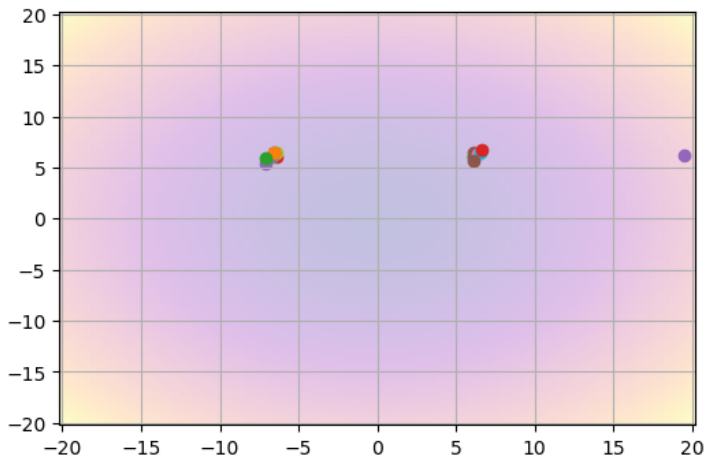



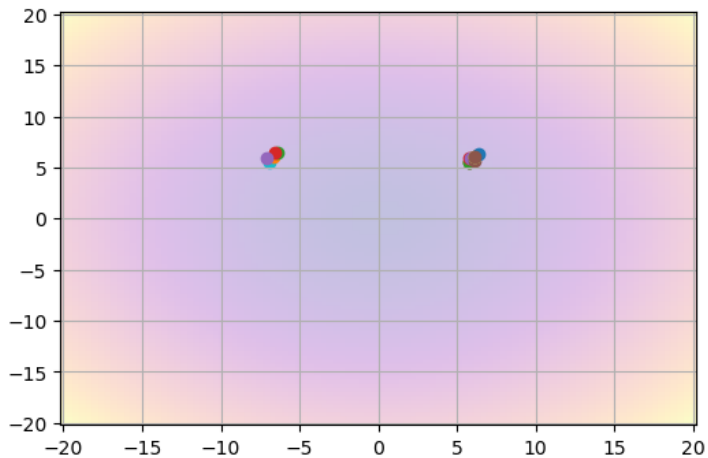


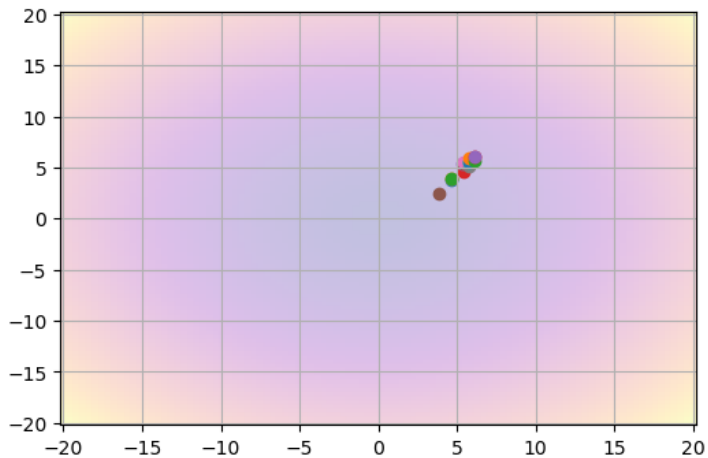


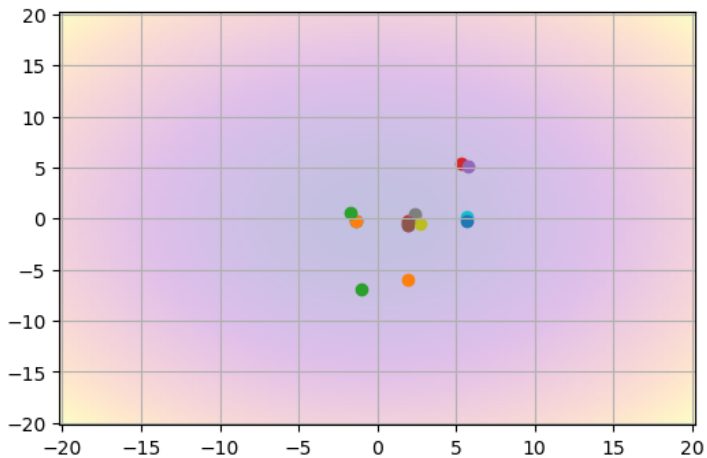


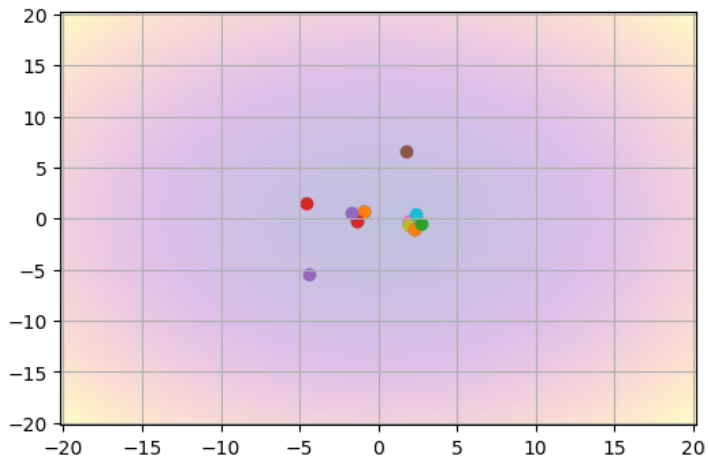


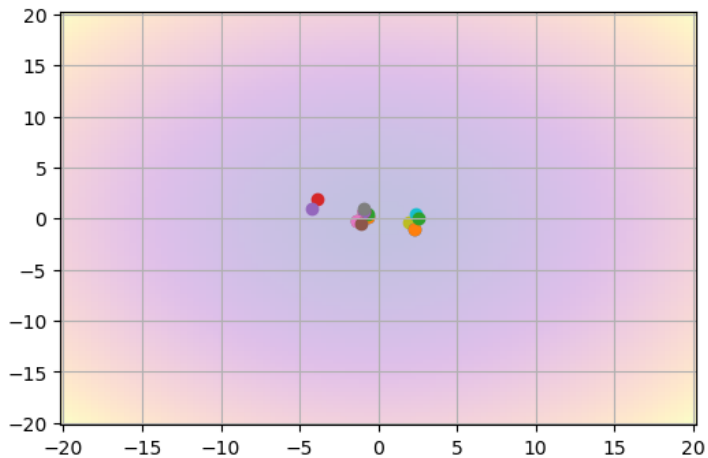








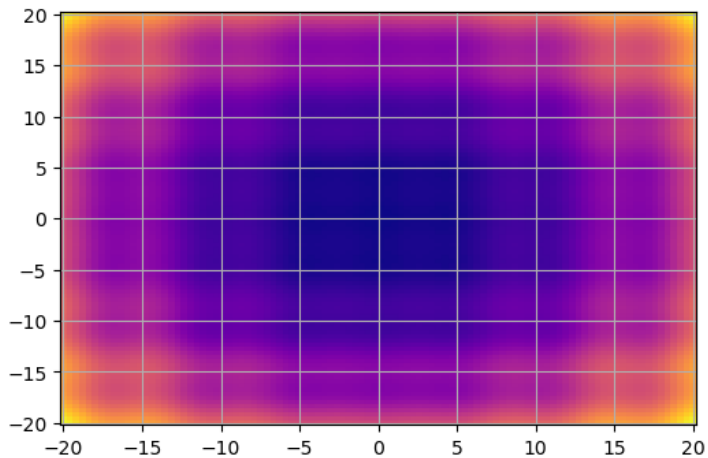


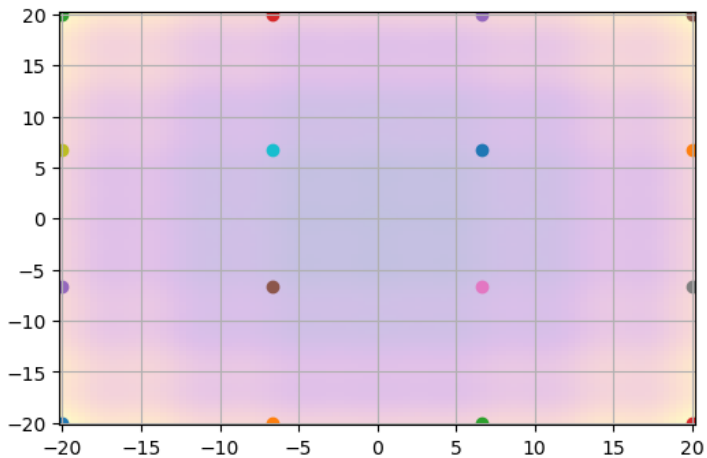


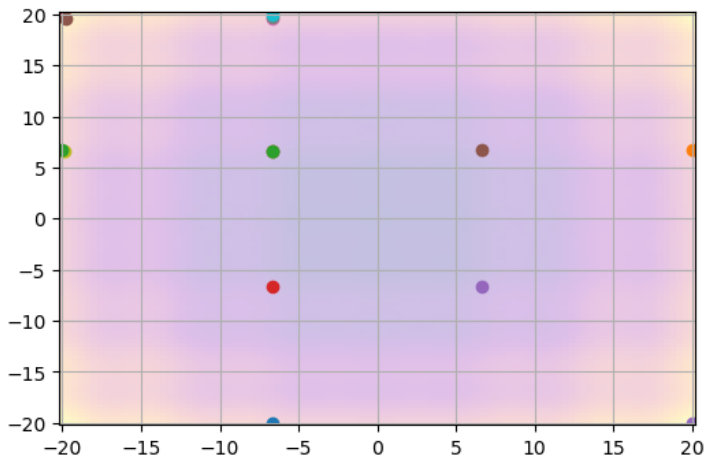
```
1 cb, history = callback_generator()
2 sol = DE_best_1_rand(g, bounds, Np=20, callback=cb)
3
4 print(sol, f(sol))
5 print(len(history))

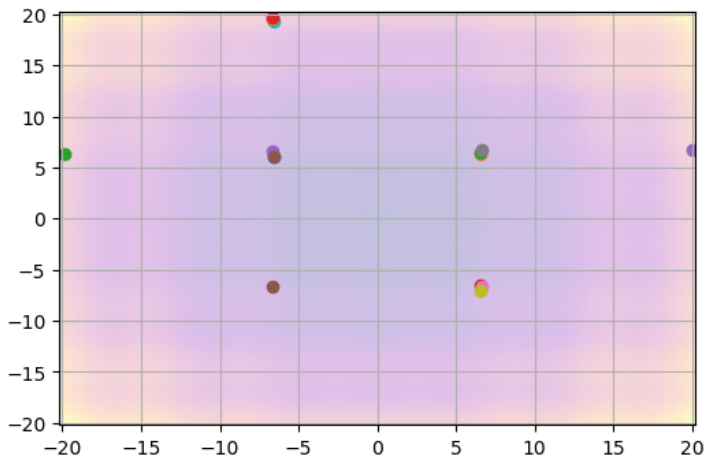
1 [ 2.44785283e-06 -3.02475187e-07] 6.083474707822143e-12
2 61
```

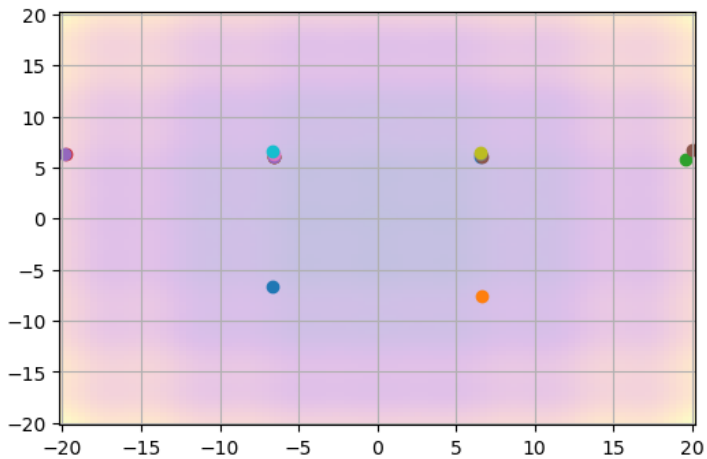
```
1 plt.pcolormesh(x, y, zg, cmap='plasma', shading='auto');
2 for n,points in enumerate(read_history(history)[:10]):
3     plt.figure()
4     plt.pcolormesh(x, y, zg, cmap='plasma', shading='auto', alpha=0.25);
5     for p in points:
6         plt.scatter(*p);
```

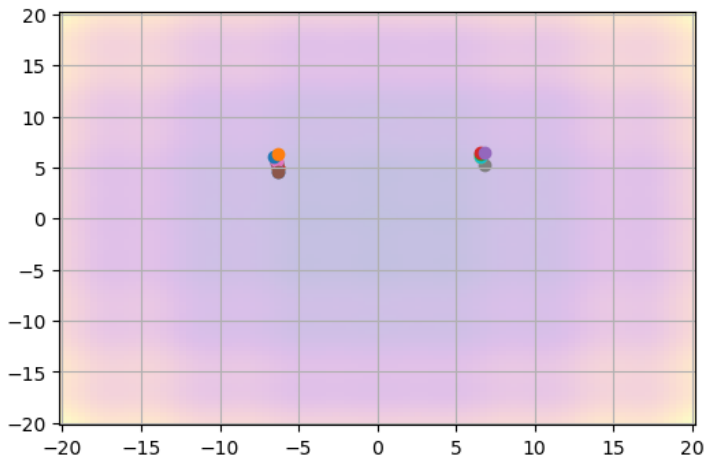


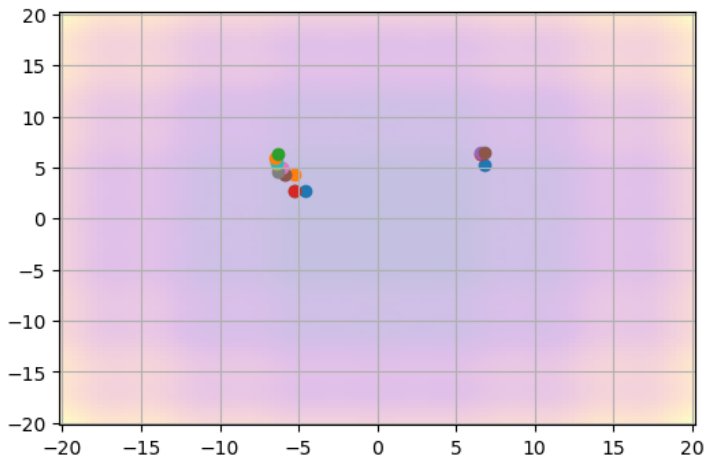


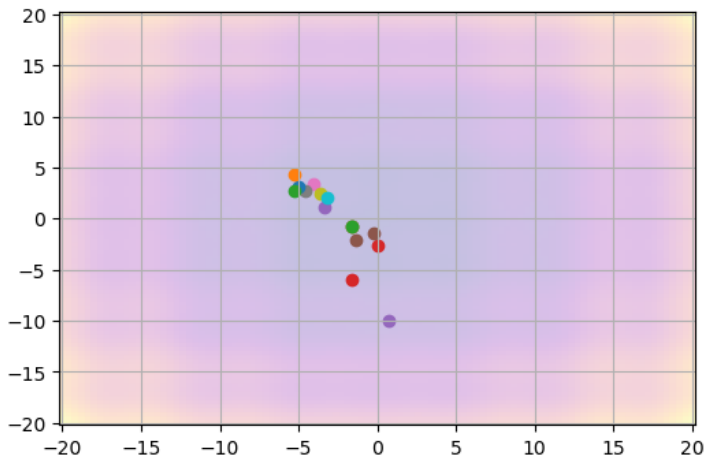


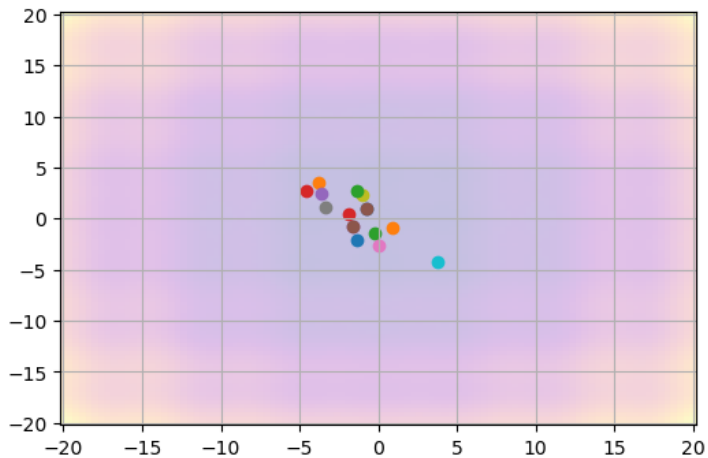


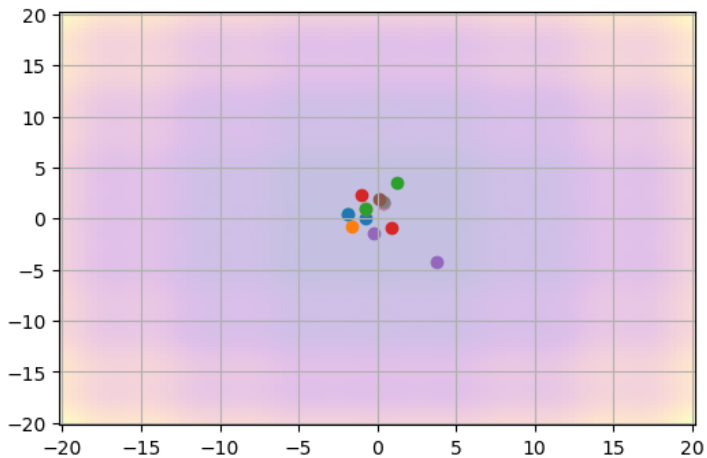


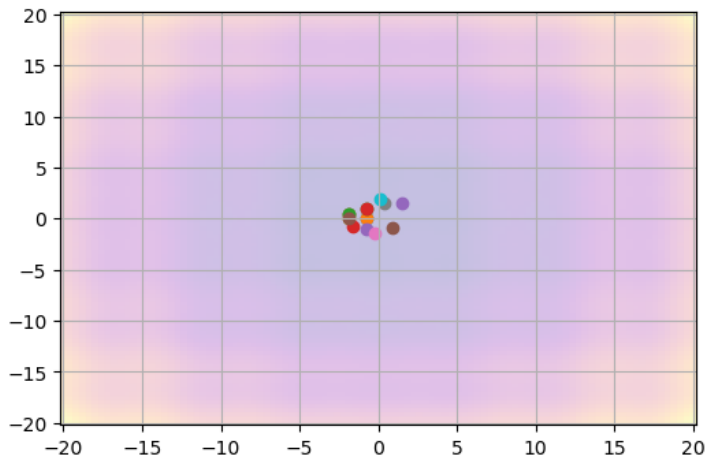










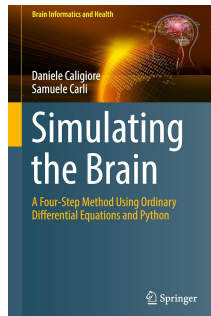



```
1 cb, history = callback_generator()
2 sol = differential_evolution(f, np.array([[-20,20],[-20,20]]), callback=cb)
3
4 print(sol)
5 print(len(history))
```

```
1 message: Optimization terminated successfully.
2     success: True
3     fun: 0.0
4     x: [ 0.000e+00  0.000e+00]
5     nit: 100
6     nfev: 3033
7     population: [[ 0.000e+00  0.000e+00]
8                  [ 0.000e+00  0.000e+00]
9                  ...
10                 [ 0.000e+00  0.000e+00]
11                 [ 0.000e+00  0.000e+00]]
12     population_energies: [ 0.000e+00  0.000e+00 ...  0.000e+00  0.000e+00]
13 100
```

```
1 print(history[:3])  
1 [[(array([-0.89504305, -3.35111496]), np.float64(0.01393568070384353)), {}], [(array([-0.79637558, 1.3880917 ]),  
↪ np.float64(0.008126453853221784)), {}], [(array([-1.06748724, -0.12865847]), np.float64(0.00681722256044361)),  
↪ {}]]
```

Reference literature - I

**Authors:**

Daniele Caligiore &
Samuele Carli

Springer 2025

ISBN: 978-981-96-2717-2

Simulating the Brain

A Four-Step Method Using Ordinary Differential Equations and Python

Core Concept

Comprehensive guide to developing brain models using ODEs and Python

Four-Step Method:

1. Model Design
2. ODE Implementation
3. Python Simulation
4. Analysis & Visualization

Key Features:

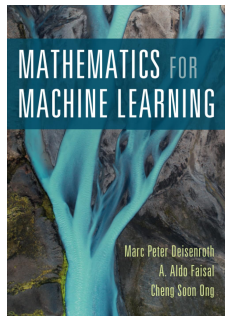
- ▶ Self-consistent textbook
- ▶ Hands-on Python examples
- ▶ Applications to healthy & damaged brains
- ▶ Suitable for beginners

For: Neuroscience students, Computational modelers, AI practitioners

Applications: Healthcare, Research, Education

Available: Amazon, Springer, Barnes & Noble

Reference literature - II

**Authors:**

Marc Peter Deisenroth
A. Aldo Faisal
Cheng Soon Ong

Cambridge University Press 2020

ISBN: 978-1-108-45514-5

Mathematics for Machine Learning

The Fundamental Concepts and Mathematical Foundations of Machine Learning

Core Concept

Bridges the gap between mathematical foundations and machine learning applications

Mathematical Areas:

- ▶ Linear Algebra
- ▶ Analytic Geometry
- ▶ Matrix Decompositions
- ▶ Vector Calculus
- ▶ Probability & Statistics
- ▶ Optimization

Key Features:

- ▶ Clear explanations and examples
- ▶ Visual illustrations throughout
- ▶ Practical ML applications
- ▶ Self-contained textbook
- ▶ Online companion materials

For: ML practitioners, Computer science students, Engineers, Researchers

Available: Amazon, Cambridge University Press, mml-book.com

Thanks!

Samuele Carli

carlisamuele@csspace.net
www.csspace.net www.entersys.it