

The background of the entire page is a light gray overlay of various mathematical sketches, diagrams, and equations. These include geometric shapes like circles and polygons, coordinate planes with plotted curves, and various mathematical symbols and formulas. The sketches are in a hand-drawn style, giving the cover a sense of intellectual exploration and discovery.

MATHEMATICS FOR MACHINE LEARNING

A Comprehensive Guide to Building Mathematical Foundations for AI and Data Science

PART 1 : Beginner level

Mohamed Aazi

MATHEMATICS FOR MACHINE LEARNING

A Comprehensive Guide to Building Mathematical
Foundations for AI and Data Science

Part 1 : Beginner level

Par : Mohamed AAZI

SECTION 1 : LINEAR ALGEBRA

Vector Addition

$$\mathbf{u} + \mathbf{v} = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} u_1 + v_1 \\ u_2 + v_2 \\ \vdots \\ u_n + v_n \end{bmatrix}$$

Explanation: Vector addition combines two vectors component-wise. It is commonly used in machine learning for gradient updates or geometric vector operations.

Example: If $\mathbf{u} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$ and $\mathbf{v} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$, then $\mathbf{u} + \mathbf{v} = \begin{bmatrix} 4 \\ 6 \end{bmatrix}$.

Implementation:

```
import numpy as np
u = np.array([1, 2])
v = np.array([3, 4])
result = u + v
```

Scalar Multiplication of a Vector

$$\alpha \mathbf{v} = \alpha \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} \alpha v_1 \\ \alpha v_2 \\ \vdots \\ \alpha v_n \end{bmatrix}$$

Explanation: Scalar multiplication scales each component of a vector by the same scalar. It is used in scaling gradients or controlling vector magnitudes.

Example: If $\alpha = 3$ and $\mathbf{v} = \begin{bmatrix} 2 \\ -1 \end{bmatrix}$, then $\alpha \mathbf{v} = \begin{bmatrix} 6 \\ -3 \end{bmatrix}$.

Implementation:

```
import numpy as np
alpha = 3
v = np.array([2, -1])
result = alpha * v
```

Dot Product

$$\mathbf{u} \cdot \mathbf{v} = \sum_{i=1}^n u_i v_i = u_1 v_1 + u_2 v_2 + \cdots + u_n v_n$$

Explanation: The dot product calculates a scalar representing the magnitude of projection of one vector onto another. It is widely used in ML for similarity measures or linear operations.

Example: If $\mathbf{u} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$ and $\mathbf{v} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$, then $\mathbf{u} \cdot \mathbf{v} = 1 \cdot 3 + 2 \cdot 4 = 11$.

Implementation:

```
import numpy as np
u = np.array([1, 2])
v = np.array([3, 4])
result = np.dot(u, v)
```

Cross Product (3D)

$$\mathbf{u} \times \mathbf{v} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ u_1 & u_2 & u_3 \\ v_1 & v_2 & v_3 \end{vmatrix}$$

Explanation: The cross product generates a vector perpendicular to two input vectors in 3D space. It is commonly used in physics and computer graphics.

Example: If $\mathbf{u} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$ and $\mathbf{v} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$, then $\mathbf{u} \times \mathbf{v} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$.

Implementation:

```
import numpy as np
u = np.array([1, 0, 0])
v = np.array([0, 1, 0])
result = np.cross(u, v)
```

Norm of a Vector (Euclidean)

$$\|\mathbf{v}\| = \sqrt{\sum_{i=1}^n v_i^2} = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}$$

Explanation: The Euclidean norm measures the magnitude (length) of a vector. It is useful in optimization and distance computations in ML.

Example: If $\mathbf{v} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$, then $\|\mathbf{v}\| = \sqrt{3^2 + 4^2} = 5$.

Implementation:

```
import numpy as np
v = np.array([3, 4])
result = np.linalg.norm(v)
```

Orthogonality Condition

$$\mathbf{u} \cdot \mathbf{v} = 0$$

Explanation: Two vectors are orthogonal if their dot product is zero. This condition is critical in linear algebra and ML for understanding independence and basis construction.

Example: If $\mathbf{u} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$ and $\mathbf{v} = \begin{bmatrix} -2 \\ 1 \end{bmatrix}$, then $\mathbf{u} \cdot \mathbf{v} = 1 \cdot -2 + 2 \cdot 1 = 0$, confirming orthogonality.

Implementation:

```
import numpy as np
u = np.array([1, 2])
v = np.array([-2, 1])
result = np.dot(u, v)
is_orthogonal = result == 0
```


Matrix Addition

$$\mathbf{A} + \mathbf{B} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11} + b_{11} & a_{12} + b_{12} \\ a_{21} + b_{21} & a_{22} + b_{22} \end{bmatrix}$$

Explanation: Matrix addition combines two matrices element-wise. It is used in ML for updating weights and biases or aggregating data.

Example: If $\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and $\mathbf{B} = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$, then $\mathbf{A} + \mathbf{B} = \begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix}$.

Implementation:

```
import numpy as np
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
result = A + B
```

Matrix Scalar Multiplication

$$\alpha \mathbf{A} = \alpha \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} \alpha a_{11} & \alpha a_{12} \\ \alpha a_{21} & \alpha a_{22} \end{bmatrix}$$

Explanation: Scaling a matrix by a scalar is useful in ML for adjusting learning rates or normalization.

Example: If $\alpha = 2$ and $\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, then $\alpha \mathbf{A} = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$.

Implementation:

```
import numpy as np
alpha = 2
A = np.array([[1, 2], [3, 4]])
result = alpha * A
```

Matrix-Vector Multiplication

$$\mathbf{Ax} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 \\ a_{21}x_1 + a_{22}x_2 \end{bmatrix}$$

Explanation: Matrix-vector multiplication transforms a vector using a linear transformation defined by the matrix. It is fundamental in ML for applying weights to inputs.

Example: If $\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and $\mathbf{x} = \begin{bmatrix} 5 \\ 6 \end{bmatrix}$, then $\mathbf{Ax} = \begin{bmatrix} 17 \\ 39 \end{bmatrix}$.

Implementation:

```
import numpy as np
A = np.array([[1, 2], [3, 4]])
x = np.array([5, 6])
result = np.dot(A, x)
```

Matrix Multiplication

$$\mathbf{C} = \mathbf{AB}, \quad c_{ij} = \sum_{k=1}^n a_{ik}b_{kj}$$

Explanation: Matrix multiplication combines two matrices, producing a matrix that represents the composition of linear transformations. It is used in ML for layer operations in neural networks.

Example: If $\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and $\mathbf{B} = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$, then $\mathbf{AB} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$.

Implementation:

```
import numpy as np
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
result = np.dot(A, B)
```

Transpose of a Matrix

$$\mathbf{A}^T = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}^T = \begin{bmatrix} a_{11} & a_{21} \\ a_{12} & a_{22} \end{bmatrix}$$

Explanation: The transpose of a matrix flips it over its diagonal, exchanging rows with columns. It is used in ML for switching between data representations.

Example: If $\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, then $\mathbf{A}^T = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$.

Implementation:

```
import numpy as np
A = np.array([[1, 2], [3, 4]])
result = A.T
```

Determinant of a 2×2 Matrix

$$\det(\mathbf{A}) = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{12}a_{21}$$

Explanation: The determinant measures the scaling factor of the transformation represented by a matrix. It is used to determine matrix invertibility.

Example: If $\mathbf{A} = \begin{bmatrix} 3 & 8 \\ 4 & 6 \end{bmatrix}$, then $\det(\mathbf{A}) = 3 \cdot 6 - 8 \cdot 4 = -14$.

Implementation:

```
import numpy as np
A = np.array([[3, 8], [4, 6]])
result = np.linalg.det(A)
```

Inverse of a 2×2 Matrix

$$\mathbf{A}^{-1} = \frac{1}{\det(\mathbf{A})} \begin{bmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{bmatrix}, \quad \det(\mathbf{A}) \neq 0$$

Explanation: The inverse of a 2×2 matrix reverses the linear transformation it represents. It is used in solving systems of linear equations.

Example: If $\mathbf{A} = \begin{bmatrix} 3 & 8 \\ 4 & 6 \end{bmatrix}$, then $\det(\mathbf{A}) = -14$ and $\mathbf{A}^{-1} = \frac{1}{-14} \begin{bmatrix} 6 & -8 \\ -4 & 3 \end{bmatrix}$.

Implementation:

```
import numpy as np
A = np.array([[3, 8], [4, 6]])
result = np.linalg.inv(A)
```

Cramer's Rule

$$x_i = \frac{\det(\mathbf{A}_i)}{\det(\mathbf{A})}, \quad \det(\mathbf{A}) \neq 0$$

Explanation: Cramer's Rule solves a system of linear equations $\mathbf{Ax} = \mathbf{b}$ by replacing each column of \mathbf{A} with \mathbf{b} and computing determinants. It is a theoretical method often used for small systems.

Example: For $\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix}$ and $\mathbf{b} = \begin{bmatrix} 5 \\ 7 \end{bmatrix}$,

$$\mathbf{A}_1 = \begin{bmatrix} 5 & 1 \\ 7 & 3 \end{bmatrix}, \quad \mathbf{A}_2 = \begin{bmatrix} 2 & 5 \\ 1 & 7 \end{bmatrix}$$

and $\det(\mathbf{A}) = 5$, so $x_1 = \frac{\det(\mathbf{A}_1)}{\det(\mathbf{A})}$, $x_2 = \frac{\det(\mathbf{A}_2)}{\det(\mathbf{A})}$.

Implementation:

```
import numpy as np
A = np.array([[2, 1], [1, 3]])
b = np.array([5, 7])
det_A = np.linalg.det(A)
x = [np.linalg.det(np.column_stack((b if i == j else A[:, j]
                                     for j in range(A.shape[1])))) / det_A
      for i in range(A.shape[1])]
```


Inverse of a Square Matrix

$$\mathbf{A}^{-1} = \frac{1}{\det(\mathbf{A})} \text{adj}(\mathbf{A}), \quad \det(\mathbf{A}) \neq 0$$

Explanation: The inverse of a square matrix generalizes the process for higher dimensions using the adjugate and determinant. It is crucial in linear algebra and ML for solving systems of equations.

Example: If $\mathbf{A} = \begin{bmatrix} 4 & 7 \\ 2 & 6 \end{bmatrix}$, the inverse is computed using cofactor expansion and scaling.

Implementation:

```
import numpy as np
A = np.array([[4, 7], [2, 6]])
result = np.linalg.inv(A)
```

Determinant of a Triangular Matrix

$$\det(\mathbf{A}) = \prod_{i=1}^n a_{ii}$$

Explanation: The determinant of a triangular matrix (upper or lower) is the product of its diagonal elements. This simplifies determinant calculations and is useful in decompositions.

Example: If $\mathbf{A} = \begin{bmatrix} 2 & 1 & 0 \\ 0 & 3 & 4 \\ 0 & 0 & 5 \end{bmatrix}$, then $\det(\mathbf{A}) = 2 \cdot 3 \cdot 5 = 30$.

Implementation:

```
import numpy as np
A = np.array([[2, 1, 0], [0, 3, 4], [0, 0, 5]])
result = np.prod(np.diag(A))
```

Rank-Nullity Theorem

$$\text{rank}(\mathbf{A}) + \text{nullity}(\mathbf{A}) = n$$

Explanation: The Rank-Nullity Theorem states that the sum of the rank (dimension of column space) and nullity (dimension of null space) of a matrix equals the number of columns. It is fundamental in linear algebra for understanding solutions to systems of linear equations.

Example: If \mathbf{A} has 3 columns and its rank is 2, then the nullity is 1 since $2 + 1 = 3$.

Implementation:

```
import numpy as np
from numpy.linalg import matrix_rank
A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
rank = matrix_rank(A)
nullity = A.shape[1] - rank
```

Hadamard (Elementwise) Product

$$\mathbf{C} = \mathbf{A} \circ \mathbf{B} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} \\ a_{21}b_{21} & a_{22}b_{22} \end{bmatrix}$$

Explanation: The Hadamard product performs elementwise multiplication between two matrices. It is used in ML for feature-wise scaling or gating.

Example: If $\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and $\mathbf{B} = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$, then $\mathbf{C} = \begin{bmatrix} 5 & 12 \\ 21 & 32 \end{bmatrix}$.

Implementation:

```
import numpy as np
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
result = np.multiply(A, B)
```

Outer Product

$$\mathbf{C} = \mathbf{u} \otimes \mathbf{v} = \begin{bmatrix} u_1v_1 & u_1v_2 & \cdots & u_1v_n \\ u_2v_1 & u_2v_2 & \cdots & u_2v_n \\ \vdots & \vdots & \ddots & \vdots \\ u_mv_1 & u_mv_2 & \cdots & u_mv_n \end{bmatrix}$$

Explanation: The outer product generates a matrix by multiplying every element of one vector by every element of another. It is used in tensor operations and constructing rank-1 matrices.

Example: If $\mathbf{u} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$ and $\mathbf{v} = \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix}$, then $\mathbf{u} \otimes \mathbf{v} = \begin{bmatrix} 3 & 4 & 5 \\ 6 & 8 & 10 \end{bmatrix}$.

Implementation:

```
import numpy as np
u = np.array([1, 2])
v = np.array([3, 4, 5])
result = np.outer(u, v)
```

Frobenius Norm

$$\|\mathbf{A}\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n |a_{ij}|^2}$$

Explanation: The Frobenius norm measures the magnitude of a matrix by summing the squares of all its elements. It is widely used in optimization and matrix analysis.

Example: If $\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, then $\|\mathbf{A}\|_F = \sqrt{1^2 + 2^2 + 3^2 + 4^2} = \sqrt{30}$.

Implementation:

```
import numpy as np
A = np.array([[1, 2], [3, 4]])
result = np.linalg.norm(A, 'fro')
```

Matrix Norm Inequality

$$\|\mathbf{Ax}\| \leq \|\mathbf{A}\|\|\mathbf{x}\|$$

Explanation: The matrix norm inequality states that the norm of a matrix-vector product is bounded by the product of the matrix norm and the vector norm. It is a key property in numerical linear algebra and ML for error analysis.

Example: For $\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and $\mathbf{x} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$, compute $\|\mathbf{Ax}\| \leq \|\mathbf{A}\|\|\mathbf{x}\|$.

Implementation:

```
import numpy as np
A = np.array([[1, 2], [3, 4]])
x = np.array([1, 1])
left = np.linalg.norm(np.dot(A, x))
right = np.linalg.norm(A) * np.linalg.norm(x)
inequality_holds = left <= right
```

Matrix Trace

$$\text{Tr}(\mathbf{A}) = \sum_{i=1}^n a_{ii}$$

Explanation: The trace of a matrix is the sum of its diagonal elements. It is used in ML for loss functions and characterizing matrix properties.

Example: If $\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, then $\text{Tr}(\mathbf{A}) = 1 + 4 = 5$.

Implementation:

```
import numpy as np
A = np.array([[1, 2], [3, 4]])
result = np.trace(A)
```


Trace of a Product

$$\text{Tr}(\mathbf{AB}) = \text{Tr}(\mathbf{BA})$$

Explanation: The trace of a product of two matrices is invariant under cyclic permutations. This property is useful in ML for simplifying gradients in matrix calculus.

Example: For $\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and $\mathbf{B} = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$, compute $\text{Tr}(\mathbf{AB}) = \text{Tr}(\mathbf{BA})$.

Implementation:

```
import numpy as np
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
trace1 = np.trace(np.dot(A, B))
trace2 = np.trace(np.dot(B, A))
equality_holds = trace1 == trace2
```

Block Matrix Multiplication

$$C = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

Explanation: Block matrix multiplication follows the same rules as scalar matrix multiplication, but each element is a submatrix. It is used in ML for large-scale computations and decompositions.

Example: Compute the block product for two partitioned 4×4 matrices.

Implementation:

```
import numpy as np
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
C = np.array([[9, 10], [11, 12]])
D = np.array([[13, 14], [15, 16]])
E = np.array([[17, 18], [19, 20]])
F = np.array([[21, 22], [23, 24]])
G = np.array([[25, 26], [27, 28]])
H = np.array([[29, 30], [31, 32]])
top_left = np.dot(A, E) + np.dot(B, G)
top_right = np.dot(A, F) + np.dot(B, H)
bottom_left = np.dot(C, E) + np.dot(D, G)
bottom_right = np.dot(C, F) + np.dot(D, H)
```

```
result = np.block([[top_left, top_right], [bottom_left, bottom_right]])
```

Kronecker Product

$$\mathbf{C} = \mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & a_{12}\mathbf{B} \\ a_{21}\mathbf{B} & a_{22}\mathbf{B} \end{bmatrix}$$

Explanation: The Kronecker product produces a block matrix by multiplying each element of one matrix by the entirety of another. It is used in ML for tensor operations and signal processing.

Example: If $\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and $\mathbf{B} = \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix}$, compute $\mathbf{A} \otimes \mathbf{B}$.

Implementation:

```
import numpy as np
A = np.array([[1, 2], [3, 4]])
B = np.array([[0, 5], [6, 7]])
result = np.kron(A, B)
```

SECTION 2 : PROBABILITY AND STATISTICS

Conditional Probability

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}, \quad P(B) > 0$$

Explanation: Conditional probability quantifies the likelihood of event A occurring given that event B has occurred. It is fundamental in probabilistic reasoning and Bayesian inference.

Example: If $P(A \cap B) = 0.2$ and $P(B) = 0.5$, then $P(A \mid B) = \frac{0.2}{0.5} = 0.4$.

Implementation:

`P_A_and_B = 0.2`

`P_B = 0.5`

`P_A_given_B = P_A_and_B / P_B`

Law of Total Probability

$$P(A) = \sum_i P(A \mid B_i)P(B_i)$$

Explanation: The law of total probability relates the probability of an event A to the probabilities of A given a partition of events $\{B_i\}$. It is used in scenarios with conditional dependencies.

Example: If $P(A \mid B_1) = 0.3$, $P(A \mid B_2) = 0.7$, $P(B_1) = 0.4$, and $P(B_2) = 0.6$, then $P(A) = 0.3 \cdot 0.4 + 0.7 \cdot 0.6 = 0.54$.

Implementation:

```
P_A_given_B1 = 0.3
```

```
P_A_given_B2 = 0.7
```

```
P_B1 = 0.4
```

```
P_B2 = 0.6
```

```
P_A = P_A_given_B1 * P_B1 + P_A_given_B2 * P_B2
```

Bayes' Theorem

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}, \quad P(B) > 0$$

Explanation: Bayes' Theorem allows the reversal of conditional probabilities, often used in updating beliefs with new evidence in ML and statistics.

Example: If $P(B \mid A) = 0.8$, $P(A) = 0.3$, and $P(B) = 0.5$, then $P(A \mid B) = \frac{0.8 \cdot 0.3}{0.5} = 0.48$.

Implementation:

`P_B_given_A = 0.8`

`P_A = 0.3`

`P_B = 0.5`

`P_A_given_B = (P_B_given_A * P_A) / P_B`

Expectation

$$\mathbb{E}[X] = \sum_i x_i P(X = x_i)$$

Explanation: The expectation (mean) of a random variable is the weighted average of all possible values, weighted by their probabilities. It is central in probability and statistics.

Example: If $X = \{1, 2, 3\}$ with $P(X = 1) = 0.2$, $P(X = 2) = 0.5$, and $P(X = 3) = 0.3$, then $\mathbb{E}[X] = 1 \cdot 0.2 + 2 \cdot 0.5 + 3 \cdot 0.3 = 2.1$.

Implementation:

```
X = [1, 2, 3]
P_X = [0.2, 0.5, 0.3]
expectation = sum(x * p for x, p in zip(X, P_X))
```


Variance

$$\text{Var}(X) = \mathbb{E}[(X - \mathbb{E}[X])^2] = \mathbb{E}[X^2] - (\mathbb{E}[X])^2$$

Explanation: Variance measures the spread of a random variable around its mean. It is widely used in ML for assessing uncertainty and model performance.

Example: For $X = \{1, 2, 3\}$ with $P(X = 1) = 0.2$, $P(X = 2) = 0.5$, and $P(X = 3) = 0.3$, compute $\mathbb{E}[X] = 2.1$ and $\mathbb{E}[X^2] = 4.7$, so $\text{Var}(X) = 4.7 - (2.1)^2 = 0.29$.

Implementation:

```
X = [1, 2, 3]
P_X = [0.2, 0.5, 0.3]
expectation = sum(x * p for x, p in zip(X, P_X))
expectation_X2 = sum(x**2 * p for x, p in zip(X, P_X))
variance = expectation_X2 - expectation**2
```

Standard Deviation

$$\sigma(X) = \sqrt{\text{Var}(X)}$$

Explanation: The standard deviation is the square root of the variance and provides a measure of dispersion in the same units as the random variable. It is widely used in data analysis and ML for variability assessment.

Example: If $\text{Var}(X) = 0.29$, then $\sigma(X) = \sqrt{0.29} \approx 0.54$.

Implementation:

```
variance = 0.29  
std_dev = variance**0.5
```

Covariance

$$\text{Cov}(X, Y) = \mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])]$$

Explanation: Covariance measures the joint variability of two random variables. A positive value indicates that they increase together, while a negative value indicates an inverse relationship.

Example: If $X = \{1, 2\}$, $Y = \{3, 4\}$, $P(X, Y) = \{0.5, 0.5\}$, and $\mathbb{E}[X] = 1.5$, $\mathbb{E}[Y] = 3.5$, compute $\text{Cov}(X, Y) = 0.25$.

Implementation:

```
X = [1, 2]
Y = [3, 4]
P_XY = [0.5, 0.5]
E_X = sum(x * p for x, p in zip(X, P_XY))
E_Y = sum(y * p for y, p in zip(Y, P_XY))
covariance = sum((x - E_X) * (y - E_Y) * p for x, y, p in zip(X, Y, P_XY))
```

Correlation

$$\rho(X, Y) = \frac{\text{Cov}(X, Y)}{\sigma(X)\sigma(Y)}$$

Explanation: Correlation normalizes covariance to a scale of $[-1, 1]$, quantifying the strength and direction of a linear relationship between two variables.

Example: If $\text{Cov}(X, Y) = 0.25$, $\sigma(X) = 0.5$, and $\sigma(Y) = 1.0$, then $\rho(X, Y) = \frac{0.25}{0.5 \cdot 1.0} = 0.5$.

Implementation:

```
covariance = 0.25
std_X = 0.5
std_Y = 1.0
correlation = covariance / (std_X * std_Y)
```

Probability Mass Function (PMF)

$$P(X = x) = \begin{cases} p_i, & \text{if } x = x_i \\ 0, & \text{otherwise} \end{cases}$$

Explanation: The PMF defines the probabilities of discrete outcomes of a random variable. It is a foundational concept in probability theory.

Example: If $X = \{1, 2, 3\}$ with $P(X = 1) = 0.2$, $P(X = 2) = 0.5$, and $P(X = 3) = 0.3$, the PMF is defined for these values.

Implementation:

```
X = [1, 2, 3]
P_X = [0.2, 0.5, 0.3]
def pmf(x):
    return P_X[X.index(x)] if x in X else 0
```

Probability Density Function (PDF)

$$f_X(x) \geq 0, \quad \int_{-\infty}^{\infty} f_X(x) dx = 1$$

Explanation: The PDF defines the relative likelihood of a continuous random variable at a specific value. It is used in probability and statistics for modeling continuous distributions.

Example: For a standard normal distribution, the PDF is $f_X(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$.

Implementation:

```
import numpy as np
from scipy.stats import norm
x = 0 # example point
pdf_value = norm.pdf(x)
```

Joint Probability

$$P(A \cap B) = P(A \mid B)P(B)$$

Explanation: Joint probability quantifies the likelihood of two events occurring together. It is essential in probabilistic modeling and understanding relationships between variables.

Example: If $P(A \mid B) = 0.4$ and $P(B) = 0.5$, then $P(A \cap B) = 0.4 \cdot 0.5 = 0.2$.

Implementation:

```
P_A_given_B = 0.4
```

```
P_B = 0.5
```

```
P_A_and_B = P_A_given_B * P_B
```

CDF (Cumulative Distribution Function)

$$F_X(x) = P(X \leq x)$$

Explanation: The CDF of a random variable gives the probability that the variable takes a value less than or equal to x . It is used to describe the distribution function for both discrete and continuous variables.

Example: For a uniform distribution $X \sim U(0, 1)$, $F_X(0.5) = 0.5$.

Implementation:

```
from scipy.stats import uniform
x = 0.5
cdf_value = uniform.cdf(x, loc=0, scale=1)
```


Entropy (discrete)

$$H(X) = - \sum_i P(X = x_i) \log_2 P(X = x_i)$$

Explanation: Entropy measures the uncertainty of a discrete random variable. It is a fundamental concept in information theory and ML, particularly in decision trees and loss functions.

Example: If $P(X) = \{0.5, 0.5\}$, then $H(X) = -0.5 \log_2(0.5) - 0.5 \log_2(0.5) = 1$.

Implementation:

```
import numpy as np
P_X = [0.5, 0.5]
entropy = -sum(p * np.log2(p) for p in P_X if p > 0)
```

Conditional Expectation

$$\mathbb{E}[X \mid Y] = \sum_x xP(X = x \mid Y)$$

Explanation: Conditional expectation is the expected value of a random variable X given that another variable Y is known. It is critical in Bayesian inference and probabilistic modeling.

Example: If $X = \{1, 2\}$ with $P(X = 1 \mid Y) = 0.7$ and $P(X = 2 \mid Y) = 0.3$, then $\mathbb{E}[X \mid Y] = 1 \cdot 0.7 + 2 \cdot 0.3 = 1.3$.

Implementation:

```
X = [1, 2]
P_X_given_Y = [0.7, 0.3]
conditional_expectation = sum(x * p for x, p in zip(X, P_X_given_Y))
```

Law of Iterated Expectations

$$\mathbb{E}[X] = \mathbb{E}[\mathbb{E}[X \mid Y]]$$

Explanation: The law of iterated expectations states that the expectation of X is the weighted average of its conditional expectations over Y . It is foundational in probability theory and statistics.

Example: Suppose X depends on $Y = \{1, 2\}$, with $\mathbb{E}[X \mid Y = 1] = 3$, $\mathbb{E}[X \mid Y = 2] = 5$, and $P(Y = 1) = 0.6$, $P(Y = 2) = 0.4$. Then $\mathbb{E}[X] = 3 \cdot 0.6 + 5 \cdot 0.4 = 3.8$.

Implementation:

```
E_X_given_Y = [3, 5]
```

```
P_Y = [0.6, 0.4]
```

```
E_X = sum(e * p for e, p in zip(E_X_given_Y, P_Y))
```

Marginal Probability

$$P(A) = \sum_B P(A \cap B)$$

Explanation: Marginal probability calculates the probability of an event A by summing (or integrating, for continuous cases) over all possible outcomes of another variable B . It is used in probabilistic modeling to reduce joint distributions.

Example: If $P(A \cap B_1) = 0.3$ and $P(A \cap B_2) = 0.4$, then $P(A) = 0.3 + 0.4 = 0.7$.

Implementation:

```
P_A_and_B = [0.3, 0.4]  
P_A = sum(P_A_and_B)
```

Skewness

$$\text{Skewness}(X) = \frac{\mathbb{E}[(X - \mu)^3]}{\sigma^3}$$

Explanation: Skewness measures the asymmetry of the probability distribution of a random variable about its mean. Positive skew indicates a longer right tail, and negative skew indicates a longer left tail.

Example: For $X = \{1, 2, 3\}$ with mean $\mu = 2$ and standard deviation $\sigma = 0.816$, compute $\text{Skewness}(X)$ using the third central moment.

Implementation:

```
import numpy as np
X = [1, 2, 3]
mu = np.mean(X)
sigma = np.std(X)
skewness = np.mean(((X - mu) / sigma)**3)
```

Kurtosis

$$\text{Kurtosis}(X) = \frac{\mathbb{E}[(X - \mu)^4]}{\sigma^4}$$

Explanation: Kurtosis measures the "tailedness" of the probability distribution. A high kurtosis indicates heavy tails, while a low kurtosis indicates light tails.

Example: For $X = \{1, 2, 3\}$ with mean $\mu = 2$ and standard deviation $\sigma = 0.816$, compute $\text{Kurtosis}(X)$ using the fourth central moment.

Implementation:

```
import numpy as np
X = [1, 2, 3]
mu = np.mean(X)
sigma = np.std(X)
kurtosis = np.mean(((X - mu) / sigma)**4)
```

Binary Cross-Entropy (special case)

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

Explanation: Binary cross-entropy is a loss function used for binary classification tasks. It measures the dissimilarity between predicted probabilities (\hat{y}) and true labels (y).

Example: For $y = [1, 0]$ and $\hat{y} = [0.8, 0.2]$, compute $\text{BCE} = -\frac{1}{2} (\log(0.8) + \log(0.8))$.

Implementation:

```
import numpy as np
y = np.array([1, 0])
y_hat = np.array([0.8, 0.2])
bce = -np.mean(y * np.log(y_hat) + (1 - y) * np.log(1 - y_hat))
```

Variance (Alternative)

$$\text{Var}(X) = \mathbb{E}[X^2] - (\mathbb{E}[X])^2$$

Explanation: An alternative formula for variance uses the difference between the expected value of the square of X and the square of the expected value of X . This method is computationally efficient.

Example: For $X = \{1, 2, 3\}$, compute $\mathbb{E}[X^2] = \frac{1^2+2^2+3^2}{3} = 4.67$ and $(\mathbb{E}[X])^2 = 2^2 = 4$, so $\text{Var}(X) = 0.67$.

Implementation:

```
import numpy as np
X = np.array([1, 2, 3])
E_X2 = np.mean(X**2)
E_X = np.mean(X)
variance = E_X2 - E_X**2
```


SECTION 3 : CALCULUS

Limit Definition of Derivative

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

Explanation: The derivative of a function is defined as the limit of the difference quotient as h approaches zero. It represents the instantaneous rate of change of the function.

Example: For $f(x) = x^2$, compute $f'(x) = \lim_{h \rightarrow 0} \frac{(x+h)^2 - x^2}{h} = 2x$.

Implementation:

```
def derivative(f, x, h=1e-5):  
    return (f(x + h) - f(x)) / h
```

Power Rule

$$\frac{d}{dx}x^n = nx^{n-1}$$

Explanation: The power rule simplifies differentiation of monomials. It is foundational for calculus and widely used in gradient computations in ML.

Example: For $f(x) = x^3$, $f'(x) = 3x^2$.

Implementation:

```
def power_rule(n, x):  
    return n * x**(n - 1)
```

Product Rule

$$\frac{d}{dx}[u(x)v(x)] = u'(x)v(x) + u(x)v'(x)$$

Explanation: The product rule computes the derivative of the product of two functions. It is crucial for handling multiplicative relationships in ML.

Example: For $f(x) = (x^2)(e^x)$, $f'(x) = 2xe^x + x^2e^x$.

Implementation:

```
def product_rule(u, v, u_prime, v_prime, x):  
    return u_prime(x) * v(x) + u(x) * v_prime(x)
```

Quotient Rule

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

Explanation: The quotient rule computes the derivative of the ratio of two functions. It is essential for operations involving divisions in ML models.

Example: For $f(x) = \frac{x^2}{e^x}$, $f'(x) = \frac{2xe^x - x^2e^x}{e^{2x}}$.

Implementation:

```
def quotient_rule(u, v, u_prime, v_prime, x):  
    return (u_prime(x) * v(x) - u(x) * v_prime(x)) / (v(x)**2)
```

Chain Rule

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

Explanation: The chain rule computes the derivative of a composite function. It is extensively used in backpropagation for training neural networks.

Example: For $f(x) = \sin(x^2)$, $f'(x) = \cos(x^2) \cdot 2x$.

Implementation:

```
def chain_rule(f_prime, g, g_prime, x):  
    return f_prime(g(x)) * g_prime(x)
```

Logarithmic Derivative

$$\frac{d}{dx} \ln(x) = \frac{1}{x}, \quad x > 0$$

Explanation: The derivative of the natural logarithm function is the reciprocal of its argument. It is frequently used in ML for optimization and logarithmic transformations.

Example: For $f(x) = \ln(x)$, $f'(2) = \frac{1}{2}$.

Implementation:

```
import numpy as np
def log_derivative(x):
    return 1 / x
```

Exponential Derivative

$$\frac{d}{dx}e^x = e^x$$

Explanation: The exponential function is unique as its derivative is equal to itself. This property is key in gradient computations and exponential growth models in ML.

Example: For $f(x) = e^x$, $f'(2) = e^2$.

Implementation:

```
import numpy as np
def exp_derivative(x):
    return np.exp(x)
```

Integral of a Power Function

$$\int x^n dx = \frac{x^{n+1}}{n+1} + C, \quad n \neq -1$$

Explanation: The integral of a power function generalizes the antiderivative for monomials. This rule is fundamental in integral calculus and applied in ML for cost function analysis.

Example: For $f(x) = x^2$, $\int x^2 dx = \frac{x^3}{3} + C$.

Implementation:

```
def power_integral(n, x):  
    return x**(n + 1) / (n + 1)
```


Fundamental Theorem of Calculus

$$\int_a^b f(x)dx = F(b) - F(a), \quad \text{where } F'(x) = f(x)$$

Explanation: The Fundamental Theorem of Calculus links differentiation and integration, stating that integration over an interval is the difference of the antiderivative evaluated at the endpoints.

Example: For $f(x) = x^2$ over $[1, 3]$, $\int_1^3 x^2 dx = \left[\frac{x^3}{3} \right]_1^3 = \frac{27}{3} - \frac{1}{3} = \frac{26}{3}$.

Implementation:

```
def definite_integral(f, a, b):  
    from scipy.integrate import quad  
    result, _ = quad(f, a, b)  
    return result
```

Partial Derivatives

$$\frac{\partial f}{\partial x} = \lim_{h \rightarrow 0} \frac{f(x+h, y) - f(x, y)}{h}, \quad \frac{\partial f}{\partial y} = \lim_{h \rightarrow 0} \frac{f(x, y+h) - f(x, y)}{h}$$

Explanation: Partial derivatives measure the rate of change of a multi-variable function with respect to one variable while keeping others constant. They are essential in optimization and gradient-based ML methods.

Example: For $f(x, y) = x^2 + y^2$, $\frac{\partial f}{\partial x} = 2x$, $\frac{\partial f}{\partial y} = 2y$.

Implementation:

```
def partial_derivative(f, var, point, h=1e-5):  
    args = list(point)  
    args[var] += h  
    return (f(*args) - f(*point)) / h
```

Gradient

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

Explanation: The gradient is a vector containing all partial derivatives of a scalar-valued function. It points in the direction of the steepest ascent and is widely used in ML optimization algorithms like gradient descent.

Example: For $f(x, y) = x^2 + y^2$, $\nabla f(x, y) = \begin{bmatrix} 2x \\ 2y \end{bmatrix}$.

Implementation:

```
import numpy as np
def gradient(f, point, h=1e-5):
    grad = np.zeros(len(point))
    for i in range(len(point)):
        args = point.copy()
        args[i] += h
        grad[i] = (f(*args) - f(*point)) / h
    return grad
```

Second Derivative (Hessian)

$$H(f) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}$$

Explanation: The Hessian is a square matrix of second-order partial derivatives. It is used in optimization to assess curvature and convergence properties of a function.

Example: For $f(x, y) = x^2 + y^2$, the Hessian is $H(f) = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$.

Implementation:

```
def hessian(f, point, h=1e-5):
    n = len(point)
    hess = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            args = point.copy()
            args[i] += h
            args[j] += h
            f_ij = f(*args)
            args[j] -= h
            f_i = f(*args)
            args[i] -= h
            args[j] += h
```

```
f_j = f(*args)
f_orig = f(*point)
hess[i, j] = (f_ij - f_i - f_j + f_orig) / (h ** 2)
return hess
```

Directional Derivative

$$D_{\mathbf{v}}f(\mathbf{x}) = \nabla f(\mathbf{x}) \cdot \mathbf{v}$$

Explanation: The directional derivative measures the rate of change of a function in the direction of a given vector. It is critical in optimization and ML for evaluating function behavior in a specific direction.

Example: For $f(x, y) = x^2 + y^2$, $\nabla f(x, y) = \begin{bmatrix} 2x \\ 2y \end{bmatrix}$. In the direction $\mathbf{v} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, $D_{\mathbf{v}}f(x, y) = 2x$.

Implementation:

```
def directional_derivative(f, grad_f, point, direction):  
    grad = grad_f(point)  
    return np.dot(grad, direction)
```

Higher-Order Partial Derivatives

$$\frac{\partial^k f}{\partial x_1^{p_1} \partial x_2^{p_2} \cdots \partial x_n^{p_n}}$$

Explanation: Higher-order partial derivatives extend partial derivatives to greater orders. Mixed derivatives often satisfy equality ($f_{xy} = f_{yx}$) under smoothness conditions.

Example: For $f(x, y) = x^2y$, $\frac{\partial^2 f}{\partial x \partial y} = 2x$.

Implementation:

```
def higher_order_partial(f, point, var_indices, h=1e-5):
    args = list(point)
    for var in var_indices:
        args[var] += h
    f_plus = f(*args)
    for var in var_indices:
        args[var] -= h * len(var_indices)
    f_minus = f(*args)
    return (f_plus - f_minus) / (h ** len(var_indices))
```

Total Derivative

$$\frac{df}{dt} = \sum_{i=1}^n \frac{\partial f}{\partial x_i} \frac{dx_i}{dt}$$

Explanation: The total derivative accounts for changes in all independent variables as functions of an external variable t . It is used in dynamical systems and optimization.

Example: If $f(x, y) = x^2 + y^2$, $x = t$, and $y = t^2$, then $\frac{df}{dt} = 2x \cdot 1 + 2y \cdot 2t = 2t + 4t^3$.

Implementation:

```
def total_derivative(f, partials, dx_dt, point):  
    return sum(partial[i] * dx_dt[i] for i in range(len(point)))
```


Implicit Differentiation

$$\frac{dy}{dx} = -\frac{\frac{\partial F}{\partial x}}{\frac{\partial F}{\partial y}}$$

Explanation: Implicit differentiation computes the derivative of a dependent variable in an equation where the variable cannot be explicitly solved. It is used in ML and calculus for handling complex equations.

Example: For $F(x, y) = x^2 + y^2 - 1 = 0$, $\frac{dy}{dx} = -\frac{x}{y}$.

Implementation:

```
def implicit_differentiation(F, x, y, partial_F_x, partial_F_y):  
    return -partial_F_x(x, y) / partial_F_y(x, y)
```

Taylor Series Expansion

$$f(x) \approx f(a) + f'(a)(x - a) + \frac{f''(a)}{2!}(x - a)^2 + \dots$$

Explanation: The Taylor series approximates a function near a point a using its derivatives. It is used in optimization and numerical analysis.

Example: For $f(x) = e^x$ near $a = 0$, $f(x) \approx 1 + x + \frac{x^2}{2} + \dots$.

Implementation:

```
def taylor_series(f, derivatives, a, x, terms=3):  
    result = 0  
    for n in range(terms):  
        result += derivatives[n](a) * (x - a)**n / np.math.factorial(n)  
    return result
```

Jacobian Matrix

$$J(\mathbf{f}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

Explanation: The Jacobian matrix contains all first-order partial derivatives of a vector-valued function. It is essential in ML for gradient-based optimization in multivariable spaces.

Example: For $\mathbf{f}(x, y) = \begin{bmatrix} x^2 + y \\ y^2 + x \end{bmatrix}$, the Jacobian is $\begin{bmatrix} 2x & 1 \\ 1 & 2y \end{bmatrix}$.

Implementation:

```
def jacobian(f, point, h=1e-5):
    m = len(f)
    n = len(point)
    J = np.zeros((m, n))
    for i in range(m):
        for j in range(n):
            args = point.copy()
            args[j] += h
            J[i, j] = (f[i](args) - f[i](point)) / h
    return J
```

Arc Length of a Curve

$$L = \int_a^b \sqrt{1 + \left(\frac{dy}{dx}\right)^2} dx$$

Explanation: The arc length measures the distance along a curve between two points. It is used in geometry and physics for path analysis.

Example: For $y = x^2$ over $[0, 1]$, $L = \int_0^1 \sqrt{1 + (2x)^2} dx$.

Implementation:

```
from scipy.integrate import quad
def arc_length(f_prime, a, b):
    integrand = lambda x: np.sqrt(1 + f_prime(x)**2)
    return quad(integrand, a, b)[0]
```

Curvature of a Function

$$\kappa(x) = \frac{|y''(x)|}{(1 + [y'(x)]^2)^{3/2}}$$

Explanation: Curvature quantifies how sharply a curve bends at a given point. It is used in geometry and trajectory analysis in robotics and ML.

Example: For $y = x^2$, $y'(x) = 2x$, $y''(x) = 2$, so $\kappa(x) = \frac{2}{(1+4x^2)^{3/2}}$.

Implementation:

```
def curvature(f_prime, f_double_prime, x):  
    numerator = abs(f_double_prime(x))  
    denominator = (1 + f_prime(x)**2)**1.5  
    return numerator / denominator
```

Integral by Parts

$$\int uv' dx = uv - \int u'v dx$$

Explanation: Integration by parts is a technique derived from the product rule of differentiation. It is used to simplify integrals involving products of functions.

Example: For $\int xe^x dx$, let $u = x$ and $v' = e^x$. Then $\int xe^x dx = xe^x - \int e^x dx = xe^x - e^x + C$.

Implementation:

```
from sympy import symbols, integrate, exp
x = symbols('x')
u = x
v_prime = exp(x)
v = integrate(v_prime, x)
integral = u * v - integrate(v * u.diff(x), x)
```

Volume of Revolution (Disk Method)

$$V = \pi \int_a^b [f(x)]^2 dx$$

Explanation: The disk method computes the volume of a solid of revolution by slicing it into disks perpendicular to the axis of rotation. It is common in geometry and physics.

Example: For $y = x^2$ revolved around the x -axis over $[0, 1]$, $V = \pi \int_0^1 (x^2)^2 dx = \pi \int_0^1 x^4 dx = \frac{\pi}{5}$.

Implementation:

```
from scipy.integrate import quad
import numpy as np
def volume_of_revolution(f, a, b):
    integrand = lambda x: np.pi * f(x)**2
    return quad(integrand, a, b)[0]
```

Surface Integral

$$\iint_S f(x, y, z) dS = \iint_R f(x, y, g(x, y)) \sqrt{1 + \left(\frac{\partial g}{\partial x}\right)^2 + \left(\frac{\partial g}{\partial y}\right)^2} dA$$

Explanation: A surface integral extends the idea of a line integral to a surface, summing a scalar field or vector flux over the surface.

Example: Compute the surface integral of $f(x, y, z) = z$ over $z = x^2 + y^2$ for $x^2 + y^2 \leq 1$.

Implementation:

```
from scipy.integrate import dblquad
def surface_integral(f, g, bounds_x, bounds_y):
    def integrand(x, y):
        gx, gy = g(x, y)
        return f(x, y, g(x, y)) * np.sqrt(1 + gx**2 + gy**2)
    return dblquad(integrand, *bounds_x, *bounds_y)
```


Divergence of a Vector Field

$$\operatorname{div} \mathbf{F} = \nabla \cdot \mathbf{F} = \frac{\partial F_1}{\partial x} + \frac{\partial F_2}{\partial y} + \frac{\partial F_3}{\partial z}$$

Explanation: The divergence measures the magnitude of a vector field's source or sink at a given point. It is used in fluid dynamics and electromagnetism.

Example: For $\mathbf{F} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$, $\operatorname{div} \mathbf{F} = 1 + 1 + 1 = 3$.

Implementation:

```
from sympy import symbols, diff
x, y, z = symbols('x y z')
F = [x, y, z]
divergence = sum(diff(F[i], var) for i, var in enumerate([x, y, z]))
```

Curl of a Vector Field

$$\text{curl } \mathbf{F} = \nabla \times \mathbf{F} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ \frac{\partial}{\partial x} & \frac{\partial}{\partial y} & \frac{\partial}{\partial z} \\ F_1 & F_2 & F_3 \end{vmatrix}$$

Explanation: The curl measures the rotation or circulation of a vector field at a point. It is critical in fluid mechanics and electromagnetism.

Example: For $\mathbf{F} = \begin{bmatrix} 0 \\ 0 \\ xy \end{bmatrix}$, $\text{curl } \mathbf{F} = \begin{bmatrix} -y \\ x \\ 0 \end{bmatrix}$.

Implementation:

```
from sympy import symbols, Matrix
x, y, z = symbols('x y z')
F = Matrix([0, 0, x*y])
curl = F.jacobian([x, y, z]).transpose() - F.jacobian([x, y, z])
```

SECTION 4 : OPTIMIZATION

Gradient Descent

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla J(\theta^{(t)})$$

Explanation: Gradient descent is an optimization algorithm that iteratively updates parameters in the direction of the negative gradient to minimize the cost function $J(\theta)$.

Example: For $J(\theta) = \theta^2$ and $\eta = 0.1$, the update is $\theta^{(t+1)} = \theta^{(t)} - 0.2\theta^{(t)}$.

Implementation:

```
def gradient_descent(gradient, theta, eta, steps):  
    for _ in range(steps):  
        theta -= eta * gradient(theta)  
    return theta
```

Stochastic Gradient Descent (SGD)

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla J_i(\theta^{(t)})$$

Explanation: SGD computes gradients on individual data points, updating parameters more frequently. It is widely used in ML due to its efficiency with large datasets.

Example: For $J_i(\theta) = (\theta - y_i)^2$, the update is based on one data point at each iteration.

Implementation:

```
def stochastic_gradient_descent(gradient, theta, eta, data, steps):  
    for _ in range(steps):  
        i = np.random.randint(len(data))  
        theta -= eta * gradient(theta, data[i])  
    return theta
```

Momentum-based Gradient Descent

$$v^{(t+1)} = \beta v^{(t)} - \eta \nabla J(\theta^{(t)}), \quad \theta^{(t+1)} = \theta^{(t)} + v^{(t+1)}$$

Explanation: Momentum adds an exponentially weighted moving average of past gradients to the current update, improving convergence speed and stability.

Example: For $\beta = 0.9$, $\eta = 0.1$, the velocity update smooths oscillations in gradient descent.

Implementation:

```
def momentum_gradient_descent(gradient, theta, eta, beta, steps):  
    v = 0  
    for _ in range(steps):  
        v = beta * v - eta * gradient(theta)  
        theta += v  
    return theta
```

Nesterov Accelerated Gradient (NAG)

$$v^{(t+1)} = \beta v^{(t)} - \eta \nabla J(\theta^{(t)} + \beta v^{(t)}), \quad \theta^{(t+1)} = \theta^{(t)} + v^{(t+1)}$$

Explanation: NAG improves upon momentum by calculating gradients at a lookahead position, resulting in more precise updates.

Example: For $\beta = 0.9$, NAG anticipates the future direction, reducing overshooting in oscillatory scenarios.

Implementation:

```
def nesterov_gradient_descent(gradient, theta, eta, beta, steps):  
    v = 0  
    for _ in range(steps):  
        lookahead = theta + beta * v  
        v = beta * v - eta * gradient(lookahead)  
        theta += v  
    return theta
```

RMSProp

$$s^{(t+1)} = \beta s^{(t)} + (1 - \beta)[\nabla J(\theta^{(t)})]^2, \quad \theta^{(t+1)} = \theta^{(t)} - \frac{\eta}{\sqrt{s^{(t+1)} + \epsilon}} \nabla J(\theta^{(t)})$$

Explanation: RMSProp scales the learning rate by a moving average of squared gradients, improving convergence for non-convex problems.

Example: For $\beta = 0.9$, RMSProp adapts the step size for each parameter, stabilizing updates.

Implementation:

```
def rmsprop(gradient, theta, eta, beta, epsilon, steps):  
    s = 0  
    for _ in range(steps):  
        grad = gradient(theta)  
        s = beta * s + (1 - beta) * grad**2  
        theta -= eta / (np.sqrt(s) + epsilon) * grad  
    return theta
```

Adam Optimization

$$m^{(t+1)} = \beta_1 m^{(t)} + (1 - \beta_1) \nabla J(\theta^{(t)}), \quad s^{(t+1)} = \beta_2 s^{(t)} + (1 - \beta_2) [\nabla J(\theta^{(t)})]^2$$
$$\hat{m} = \frac{m^{(t+1)}}{1 - \beta_1^t}, \quad \hat{s} = \frac{s^{(t+1)}}{1 - \beta_2^t}, \quad \theta^{(t+1)} = \theta^{(t)} - \frac{\eta}{\sqrt{\hat{s}} + \epsilon} \hat{m}$$

Explanation: Adam combines momentum and RMSProp, adapting step sizes and smoothing updates. It is one of the most popular optimization algorithms in ML.

Example: For $\beta_1 = 0.9$, $\beta_2 = 0.999$, Adam balances momentum and per-parameter scaling.

Implementation:

```
def adam(gradient, theta, eta, beta1, beta2, epsilon, steps):
    m, s = 0, 0
    for t in range(1, steps + 1):
        grad = gradient(theta)
        m = beta1 * m + (1 - beta1) * grad
        s = beta2 * s + (1 - beta2) * grad**2
        m_hat = m / (1 - beta1**t)
        s_hat = s / (1 - beta2**t)
        theta -= eta / (np.sqrt(s_hat) + epsilon) * m_hat
    return theta
```


Regularized Optimization Objective

$$J_{\text{reg}}(\theta) = J(\theta) + \lambda R(\theta)$$

Explanation: Regularization penalizes model complexity to prevent overfitting. Common regularizers include L1 (lasso) and L2 (ridge) norms.

Example: For $R(\theta) = \|\theta\|_2^2$, $J_{\text{reg}}(\theta) = J(\theta) + \lambda \|\theta\|_2^2$.

Implementation:

```
def regularized_objective(loss, theta, reg, lam):  
    return loss(theta) + lam * reg(theta)
```

Learning Rate Decay

$$\eta_t = \frac{\eta_0}{1 + \gamma t}$$

Explanation: Learning rate decay gradually reduces the learning rate to improve convergence stability as training progresses.

Example: For $\eta_0 = 0.1$, $\gamma = 0.01$, at step $t = 10$, $\eta_t = 0.1 / (1 + 0.01 \cdot 10) = 0.0909$.

Implementation:

```
def learning_rate_decay(eta0, gamma, t):  
    return eta0 / (1 + gamma * t)
```

Gradient Clipping

$$\mathbf{g} = \text{clip}(\mathbf{g}, -\tau, \tau)$$

Explanation: Gradient clipping limits the gradient magnitude to prevent exploding gradients in deep neural networks.

Example: For $\tau = 1.0$, clip gradients to the range $[-1, 1]$.

Implementation:

```
def gradient_clipping(grad, tau):  
    return np.clip(grad, -tau, tau)
```

Minibatch Gradient Descent

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla J_{B_t}(\theta^{(t)})$$

Explanation: Minibatch gradient descent computes updates using small random subsets of data, balancing SGD's noise and batch gradient descent's stability.

Example: Use minibatch size $B = 32$ to compute updates on smaller subsets of data.

Implementation:

```
def minibatch_gradient_descent(gradient, theta, eta, data, batch_size, steps):
    for _ in range(steps):
        batch = np.random.choice(data, batch_size, replace=False)
        theta -= eta * gradient(theta, batch)
    return theta
```

Coordinate Descent

$$\theta_j^{(t+1)} = \theta_j^{(t)} - \eta \frac{\partial J(\theta)}{\partial \theta_j}$$

Explanation: Coordinate descent optimizes a single parameter at a time, cycling through all parameters until convergence. It is effective for high-dimensional problems.

Example: Minimize $J(\theta_1, \theta_2) = (\theta_1 - 1)^2 + (\theta_2 - 2)^2$ by alternately updating θ_1 and θ_2 .

Implementation:

```
def coordinate_descent.gradient, theta, eta, steps):
    for _ in range(steps):
        for j in range(len(theta)):
            theta[j] -= eta * gradient(theta, j)
    return theta
```

Elastic Net Regularization

$$J_{\text{reg}}(\theta) = J(\theta) + \lambda_1 \|\theta\|_1 + \lambda_2 \|\theta\|_2^2$$

Explanation: Elastic Net combines L1 and L2 regularization to handle sparsity and multicollinearity. It is commonly used in regression tasks.

Example: For $\lambda_1 = 0.1$, $\lambda_2 = 0.2$, and $J(\theta) = \|\theta - \mathbf{y}\|_2^2$, compute the regularized objective.

Implementation:

```
def elastic_net_objective(loss, theta, lam1, lam2):  
    return loss(theta) + lam1 * np.sum(np.abs(theta)) + lam2 * np.sum(theta**2)
```

Adagrad Optimization

$$\theta^{(t+1)} = \theta^{(t)} - \frac{\eta}{\sqrt{G^{(t)} + \epsilon}} \nabla J(\theta^{(t)})$$

$$G^{(t)} = \sum_{i=1}^t [\nabla J(\theta^{(i)})]^2$$

Explanation: Adagrad adapts the learning rate for each parameter based on the history of gradients, improving performance on sparse data.

Example: For $\eta = 0.1$, adaptively scale updates for different features.

Implementation:

```
def adagrad(gradient, theta, eta, epsilon, steps):  
    G = 0  
    for _ in range(steps):  
        grad = gradient(theta)  
        G += grad**2  
        theta -= eta / (np.sqrt(G) + epsilon) * grad  
    return theta
```

AdamW Optimization

$$\theta^{(t+1)} = \theta^{(t)} - \frac{\eta}{\sqrt{\hat{s}} + \epsilon} \hat{m} - \lambda \theta^{(t)}$$

Explanation: AdamW modifies Adam by decoupling weight decay from the gradient updates, improving regularization and generalization in ML models.

Example: For $\lambda = 0.01$, regularize weights alongside adaptive learning rates.

Implementation:

```
def adamw(gradient, theta, eta, beta1, beta2, lam, epsilon, steps):
    m, s = 0, 0
    for t in range(1, steps + 1):
        grad = gradient(theta)
        m = beta1 * m + (1 - beta1) * grad
        s = beta2 * s + (1 - beta2) * grad**2
        m_hat = m / (1 - beta1**t)
        s_hat = s / (1 - beta2**t)
        theta -= eta / (np.sqrt(s_hat) + epsilon) * m_hat + lam * theta
    return theta
```


Momentum “Heavy Ball” Method

$$\theta^{(t+1)} = \theta^{(t)} + \beta(\theta^{(t)} - \theta^{(t-1)}) - \eta \nabla J(\theta^{(t)})$$

Explanation: This variant of momentum includes an inertial term to improve convergence speed for strongly convex problems.

Example: For $\beta = 0.9$, the “heavy ball” accelerates gradient descent.

Implementation:

```
def heavy_ball(gradient, theta, eta, beta, steps):
    prev_theta = theta.copy()
    v = 0
    for _ in range(steps):
        grad = gradient(theta)
        v = beta * (theta - prev_theta) - eta * grad
        prev_theta = theta.copy()
        theta += v
    return theta
```

Projection / Projected Gradient Descent

$$\theta^{(t+1)} = \text{Proj}_{\mathcal{C}}(\theta^{(t)} - \eta \nabla J(\theta^{(t)}))$$

Explanation: Projected gradient descent ensures that updates remain within a feasible set \mathcal{C} , often used for constrained optimization.

Example: For $\mathcal{C} = \|\theta\|_2 \leq 1$, project θ onto the unit ball after each step.

Implementation:

```
def projected_gradient_descent(gradient, theta, eta, projection, steps):  
    for _ in range(steps):  
        theta -= eta * gradient(theta)  
        theta = projection(theta)  
    return theta
```

Newton's Method

$$\theta^{(t+1)} = \theta^{(t)} - [H(\theta^{(t)})]^{-1} \nabla J(\theta^{(t)})$$

Explanation: Newton's method uses second-order information via the Hessian to improve convergence, especially for quadratic cost functions.

Example: For $J(\theta) = \theta^2$, the update uses $H = 2$.

Implementation:

```
def newtons_method(gradient, hessian, theta, steps):
    for _ in range(steps):
        grad = gradient(theta)
        hess = hessian(theta)
        theta -= np.linalg.inv(hess).dot(grad)
    return theta
```

Proximal Gradient Method

$$\theta^{(t+1)} = \text{prox}_{\lambda R}(\theta^{(t)} - \eta \nabla J(\theta^{(t)}))$$

Explanation: The proximal gradient method generalizes gradient descent to handle nonsmooth regularization terms such as L1 norm.

Example: For $R(\theta) = \|\theta\|_1$, compute soft thresholding for each parameter.

Implementation:

```
def proximal_gradient(gradient, theta, eta, prox, steps):  
    for _ in range(steps):  
        theta -= eta * gradient(theta)  
        theta = prox(theta)  
    return theta
```

Proximal Gradient with L1 (ISTA)

$$\theta^{(t+1)} = \text{soft}(\theta^{(t)} - \eta \nabla J(\theta^{(t)}), \lambda \eta)$$

Explanation: Iterative Shrinkage-Thresholding Algorithm (ISTA) applies soft thresholding to update parameters for sparse optimization.

Example: For $J(\theta) = \|\theta - \mathbf{y}\|_2^2 + \lambda \|\theta\|_1$, apply shrinkage to each θ_i .

Implementation:

```
def ista(gradient, theta, eta, lam, steps):
    def soft_threshold(x, lam):
        return np.sign(x) * max(0, abs(x) - lam)
    for _ in range(steps):
        theta -= eta * gradient(theta)
        theta = np.vectorize(soft_threshold)(theta, lam * eta)
    return theta
```

Penalty Method

$$J_{\text{penalty}}(\theta) = J(\theta) + \frac{1}{\mu} h(\theta)^2$$

Explanation: The penalty method solves constrained optimization problems by penalizing constraint violations in the objective function.

Example: For $h(\theta) = \|\theta\|_2^2 - 1$, penalize deviations from the unit ball constraint.

Implementation:

```
def penalty_method(loss, theta, penalty, mu):  
    return loss(theta) + penalty(theta)**2 / mu
```

Augmented Lagrangian Method

$$\mathcal{L}(\theta, \lambda, \mu) = J(\theta) + \lambda h(\theta) + \frac{\mu}{2} h(\theta)^2$$

Explanation: The augmented Lagrangian method combines Lagrangian and penalty approaches to solve constrained optimization problems. It alternates between updating parameters and Lagrange multipliers.

Example: For $J(\theta) = \|\theta\|_2^2$ and $h(\theta) = \|\theta\|_1 - 1$, compute updates for θ , λ , and μ .

Implementation:

```
def augmented_lagrangian(loss, h, theta, lam, mu, steps):
    for _ in range(steps):
        lagrangian = loss(theta) + lam * h(theta) + (mu / 2) * h(theta)**2
        theta -= np.gradient(lagrangian)
        lam += mu * h(theta)
    return theta
```

Dual Ascent Method

$$\lambda^{(t+1)} = \lambda^{(t)} + \eta h(\theta^{(t)})$$

Explanation: The dual ascent method optimizes the dual problem of constrained optimization by updating the Lagrange multipliers iteratively.

Example: For $h(\theta) = \|\theta\|_1 - 1$, update λ based on the constraint violation.

Implementation:

```
def dual_ascent(loss, h, theta, lam, eta, steps):  
    for _ in range(steps):  
        theta -= eta * np.gradient(loss(theta) + lam * h(theta))  
        lam += eta * h(theta)  
    return theta, lam
```


Trust Region Method

$$\theta^{(t+1)} = \arg \min_{\Delta} \{J(\theta) + \nabla J(\theta)^T \Delta + \frac{1}{2} \Delta^T H \Delta \mid \|\Delta\| \leq \Delta_{\max}\}$$

Explanation: The trust region method restricts the step size to a region where the quadratic approximation of the cost function is valid, ensuring stability.

Example: For $J(\theta) = \|\theta - \mathbf{y}\|_2^2$, compute steps Δ constrained by $\|\Delta\| \leq \Delta_{\max}$.

Implementation:

```
def trust_region(loss, gradient, hessian, theta, delta_max, steps):
    for _ in range(steps):
        grad = gradient(theta)
        hess = hessian(theta)
        delta = np.linalg.solve(hess, -grad)
        if np.linalg.norm(delta) > delta_max:
            delta *= delta_max / np.linalg.norm(delta)
        theta += delta
    return theta
```

Barrier Method

$$J_{\text{barrier}}(\theta) = J(\theta) - \frac{1}{\mu} \sum_{i=1}^m \ln(-h_i(\theta))$$

Explanation: The barrier method solves constrained optimization by penalizing constraint violations with a logarithmic barrier, keeping updates within the feasible region.

Example: For $h(\theta) = \|\theta\|_1 - 1$, use $-\ln(1 - \|\theta\|_1)$ as the barrier term.

Implementation:

```
def barrier_method(loss, h, theta, mu, steps):
    for _ in range(steps):
        barrier = -np.sum(np.log(-h(theta)))
        theta -= np.gradient(loss(theta) + (1 / mu) * barrier)
        mu *= 0.9
    return theta
```

Simulated Annealing

$$P(\Delta E) = \exp\left(-\frac{\Delta E}{T}\right)$$

Explanation: Simulated annealing is a probabilistic optimization algorithm inspired by annealing in metallurgy. It explores the solution space by accepting worse solutions probabilistically to escape local minima.

Example: Minimize $J(\theta) = \theta^2$ with an initial temperature $T = 1$, gradually cooling down.

Implementation:

```
import numpy as np

def simulated_annealing(loss, theta, T, cooling_rate, steps):
    for _ in range(steps):
        new_theta = theta + np.random.uniform(-1, 1, size=theta.shape)
        delta_E = loss(new_theta) - loss(theta)
        if delta_E < 0 or np.exp(-delta_E / T) > np.random.rand():
            theta = new_theta
        T *= cooling_rate
    return theta
```

SECTION 5 : REGRESSION

Linear Regression Hypothesis

$$\hat{y} = \mathbf{X}\boldsymbol{\beta} + \epsilon$$

Explanation: The hypothesis for linear regression assumes that the target variable y is a linear combination of features \mathbf{X} , coefficients $\boldsymbol{\beta}$, and an error term ϵ .

Example: For $y = 2x_1 + 3x_2 + \epsilon$, predict y as a linear function of x_1 and x_2 .

Implementation:

```
import numpy as np
X = np.array([[1, 2], [3, 4]])
beta = np.array([2, 3])
y_pred = X @ beta
```

Ordinary Least Squares (OLS)

$$\boldsymbol{\beta} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$$

Explanation: OLS finds the coefficient vector $\boldsymbol{\beta}$ that minimizes the sum of squared residuals between predicted and actual values.

Example: For $\mathbf{X} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and $\mathbf{y} = \begin{bmatrix} 5 \\ 11 \end{bmatrix}$, compute $\boldsymbol{\beta}$.

Implementation:

```
beta = np.linalg.inv(X.T @ X) @ X.T @ y
```

Mean Squared Error (MSE)

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

Explanation: MSE quantifies the average squared difference between actual and predicted values. It is a standard loss function in regression.

Example: For $y = [1, 2, 3]$ and $\hat{y} = [1.1, 1.9, 3.2]$, compute the MSE.

Implementation:

```
mse = np.mean((y - y_pred)**2)
```

Gradient of the MSE Loss

$$\frac{\partial}{\partial \boldsymbol{\beta}} \text{MSE} = -\frac{2}{n} \mathbf{X}^T (\mathbf{y} - \mathbf{X}\boldsymbol{\beta})$$

Explanation: The gradient of MSE with respect to $\boldsymbol{\beta}$ is used in gradient-based optimization algorithms like gradient descent.

Example: Compute the gradient for $\mathbf{X} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, $\mathbf{y} = [5, 11]$, and $\boldsymbol{\beta} = [1, 1]$.

Implementation:

```
grad = -2 / len(y) * X.T @ (y - X @ beta)
```

Coefficient of Determination (R^2)

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

Explanation: R^2 measures the proportion of variance in the target variable explained by the model. A value close to 1 indicates a good fit.

Example: For $y = [1, 2, 3]$ and $\hat{y} = [1.1, 1.9, 3.2]$, compute R^2 .

Implementation:

```
r2 = 1 - np.sum((y - y_pred)**2) / np.sum((y - np.mean(y))**2)
```


Adjusted R^2

$$\bar{R}^2 = 1 - \frac{(1 - R^2)(n - 1)}{n - p - 1}$$

Explanation: Adjusted R^2 accounts for the number of predictors p in the model, penalizing overfitting.

Example: For $R^2 = 0.9$, $n = 100$, and $p = 5$, compute \bar{R}^2 .

Implementation:

```
adjusted_r2 = 1 - (1 - r2) * (n - 1) / (n - p - 1)
```

Mean Absolute Error (MAE)

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

Explanation: MAE measures the average magnitude of prediction errors. It is less sensitive to outliers compared to MSE.

Example: For $y = [1, 2, 3]$ and $\hat{y} = [1.1, 1.9, 3.2]$, compute the MAE.

Implementation:

```
mae = np.mean(np.abs(y - y_pred))
```

Weighted Least Squares (WLS)

$$\boldsymbol{\beta} = (\mathbf{X}^T \mathbf{W} \mathbf{X})^{-1} \mathbf{X}^T \mathbf{W} \mathbf{y}$$

Explanation: WLS minimizes the sum of weighted residuals, allowing for heteroscedasticity in the data.

Example: For $\mathbf{W} = \text{diag}([1, 2])$, compute $\boldsymbol{\beta}$.

Implementation:

```
W = np.diag([1, 2])
beta = np.linalg.inv(X.T @ W @ X) @ X.T @ W @ y
```

Polynomial Regression Hypothesis

$$\hat{y} = \beta_0 + \beta_1 x + \beta_2 x^2 + \cdots + \beta_n x^n$$

Explanation: Polynomial regression models the relationship between x and y as a polynomial. It generalizes linear regression to non-linear patterns.

Example: Fit $y = 2x + x^2$.

Implementation:

```
from numpy.polynomial.polynomial import Polynomial
poly = Polynomial.fit(X, y, deg=2)
y_pred = poly(X)
```

Non-Linear Regression

$$\hat{y} = f(\mathbf{X}, \boldsymbol{\beta}) + \epsilon$$

Explanation: Non-linear regression models relationships where the target variable is a non-linear function of the parameters.

Example: Fit $y = ae^{bx}$ using optimization.

Implementation:

```
from scipy.optimize import curve_fit
def model(X, a, b):
    return a * np.exp(b * X)
params, _ = curve_fit(model, X, y)
```

Maximum Likelihood Estimation for Regression

$$\hat{\beta} = \arg \max_{\beta} \prod_{i=1}^n p(y_i \mid \mathbf{X}_i, \beta)$$

Explanation: MLE estimates the parameters that maximize the likelihood of observing the data under a probabilistic model.

Example: Estimate β assuming Gaussian noise.

Implementation:

```
from scipy.optimize import minimize
def neg_log_likelihood(beta, X, y):
    residuals = y - X @ beta
    return np.sum(residuals**2)
beta = minimize(neg_log_likelihood, np.zeros(X.shape[1]), args=(X, y)).x
```

Empirical Risk Minimization

$$\hat{\boldsymbol{\theta}} = \arg \min_{\boldsymbol{\theta}} \frac{1}{n} \sum_{i=1}^n \ell(y_i, f(\mathbf{X}_i, \boldsymbol{\theta}))$$

Explanation: ERM minimizes the average loss over the training data to estimate the model parameters.

Example: Minimize MSE loss for linear regression.

Implementation:

```
def empirical_risk(theta, X, y, loss):  
    return np.mean([loss(y[i], np.dot(X[i], theta)) for i in range(len(y))])
```

Logistic Regression Hypothesis

$$\hat{y} = \sigma(\mathbf{X}\boldsymbol{\beta}), \quad \sigma(z) = \frac{1}{1 + e^{-z}}$$

Explanation: Logistic regression predicts probabilities for binary classification using the sigmoid function applied to a linear combination of inputs.

Example: For $\mathbf{X} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and $\boldsymbol{\beta} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$, compute \hat{y} .

Implementation:

```
def sigmoid(z):  
    return 1 / (1 + np.exp(-z))  
y_pred = sigmoid(X @ beta)
```


Binary Cross-Entropy Loss

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

Explanation: Binary cross-entropy measures the dissimilarity between predicted probabilities and true labels in binary classification.

Example: For $y = [1, 0]$ and $\hat{y} = [0.9, 0.1]$, compute the loss.

Implementation:

```
loss = -np.mean(y * np.log(y_pred) + (1 - y) * np.log(1 - y_pred))
```

Cross-Entropy Loss (Multi-Class)

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

Explanation: Cross-entropy loss generalizes to multi-class classification, comparing one-hot-encoded true labels with predicted probabilities.

Example: For $y = [1, 0, 0]$ and $\hat{y} = [0.8, 0.1, 0.1]$, compute the loss.

Implementation:

```
loss = -np.mean(np.sum(y * np.log(y_pred), axis=1))
```

Hinge Loss for SVM

$$\mathcal{L} = \frac{1}{n} \sum_{i=1}^n \max(0, 1 - y_i \hat{y}_i)$$

Explanation: Hinge loss penalizes predictions that are not at least 1 margin away from the correct classification in support vector machines (SVMs).

Example: For $y = [1, -1]$ and $\hat{y} = [0.8, -0.5]$, compute the loss.

Implementation:

```
loss = np.mean(np.maximum(0, 1 - y * y_pred))
```

Lasso Regression Objective

$$\mathcal{L} = \frac{1}{2n} \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|_2^2 + \lambda \|\boldsymbol{\beta}\|_1$$

Explanation: Lasso regression adds an L1 regularization term to the least squares loss, promoting sparsity in the coefficients.

Example: For $\lambda = 0.1$, add $\|\boldsymbol{\beta}\|_1$ as a penalty.

Implementation:

```
loss = 0.5 * np.mean((y - X @ beta)**2) + lam * np.sum(np.abs(beta))
```

Ridge Regression Objective

$$\mathcal{L} = \frac{1}{2n} \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|_2^2 + \lambda \|\boldsymbol{\beta}\|_2^2$$

Explanation: Ridge regression adds an L2 regularization term to reduce overfitting by shrinking coefficients.

Example: For $\lambda = 0.1$, compute the loss with L2 regularization.

Implementation:

```
loss = 0.5 * np.mean((y - X @ beta)**2) + lam * np.sum(beta**2)
```

Negative Binomial Regression

$$\hat{y} = \frac{\Gamma(y + \alpha)}{\Gamma(y + 1)\Gamma(\alpha)} \left(\frac{\alpha}{\alpha + \hat{\mu}} \right)^\alpha \left(\frac{\hat{\mu}}{\alpha + \hat{\mu}} \right)^y$$

Explanation: Negative binomial regression models count data with overdispersion using a generalized linear model.

Example: Fit a model for overdispersed count data.

Implementation:

```
from statsmodels.api import GLM, families
model = GLM(y, X, family=families.NegativeBinomial())
results = model.fit()
```

Poisson Regression Model

$$\hat{\mu} = e^{\mathbf{X}\beta}$$

Explanation: Poisson regression models count data using a log link function, assuming the target variable follows a Poisson distribution.

Example: Predict event counts given feature data.

Implementation:

```
from statsmodels.api import GLM, families
model = GLM(y, X, family=families.Poisson())
results = model.fit()
```

Gamma Regression Objective

$$\mathcal{L} = \frac{1}{\phi} \sum_{i=1}^n \left(-\log(\hat{\mu}_i) + \frac{y_i}{\hat{\mu}_i} \right)$$

Explanation: Gamma regression models positive continuous data with a Gamma distribution, often for skewed datasets.

Example: Predict insurance claims amounts.

Implementation:

```
from statsmodels.api import GLM, families
model = GLM(y, X, family=families.Gamma())
results = model.fit()
```


Probit Regression Model

$$P(y = 1) = \Phi(\mathbf{X}\boldsymbol{\beta})$$

Explanation: Probit regression models binary classification using the cumulative normal distribution function Φ .

Example: Predict binary outcomes using a probit link.

Implementation:

```
from statsmodels.api import GLM, families
model = GLM(y, X, family=families.Binomial(link=families.links.probit()))
results = model.fit()
```

Multinomial Logistic Regression

$$P(y = k) = \frac{e^{\mathbf{X}\beta_k}}{\sum_{j=1}^K e^{\mathbf{X}\beta_j}}$$

Explanation: Multinomial logistic regression generalizes logistic regression for multi-class classification tasks.

Example: Classify samples into one of $K = 3$ classes.

Implementation:

```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression(multi_class='multinomial')
model.fit(X, y)
```

Quantile Regression Loss

$$\mathcal{L} = \sum_{i=1}^n \rho_{\tau}(y_i - \hat{y}_i), \quad \rho_{\tau}(e) = \max(\tau e, (1 - \tau)e)$$

Explanation: Quantile regression minimizes the weighted sum of residuals, modeling conditional quantiles of the target variable.

Example: Estimate the 90th percentile of target values.

Implementation:

```
from statsmodels.api import QuantReg
model = QuantReg(y, X)
results = model.fit(q=0.9)
```

Huber Loss

$$\mathcal{L} = \sum_{i=1}^n \begin{cases} \frac{1}{2}(y_i - \hat{y}_i)^2, & \text{if } |y_i - \hat{y}_i| \leq \delta \\ \delta|y_i - \hat{y}_i| - \frac{1}{2}\delta^2, & \text{otherwise} \end{cases}$$

Explanation: Huber loss combines MSE and MAE, being quadratic for small errors and linear for large errors, robust to outliers.

Example: Fit a regression model robust to outliers with $\delta = 1$.

Implementation:

```
def huber_loss(y, y_pred, delta):  
    diff = np.abs(y - y_pred)  
    return np.where(diff <= delta, 0.5 * diff**2, delta * diff - 0.5 * delta**2)
```

SECTION 6 : NEURAL NETWORKS

Perceptron Update Rule

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \eta(y - \hat{y})\mathbf{x}$$

Explanation: The perceptron update rule adjusts weights based on prediction errors. It is used for binary classification in linearly separable data.

Example: For $\mathbf{x} = [1, 2]$, $y = 1$, $\hat{y} = 0$, and $\eta = 0.1$, update \mathbf{w} .

Implementation:

```
w += eta * (y - y_pred) * x
```

Forward Propagation (Single Layer)

$$\hat{y} = \sigma(\mathbf{X}\mathbf{w} + b)$$

Explanation: Forward propagation computes predictions by applying a weight matrix and activation function to input features.

Example: For $\mathbf{X} = [1, 2]$, $\mathbf{w} = [0.5, 0.5]$, and $b = 0$, compute \hat{y} .

Sigmoid Activation

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Explanation: The sigmoid activation maps inputs to $[0, 1]$, commonly used for binary classification.

Example: For $z = 0.5$, compute $\sigma(0.5)$.

Implementation:

```
def sigmoid(z):  
    return 1 / (1 + np.exp(-z))
```

Tanh Activation

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

Explanation: Tanh activation maps inputs to $[-1, 1]$ and is useful for symmetric data.

Example: For $z = 0.5$, compute $\tanh(0.5)$.

Implementation:

```
def tanh(z):  
    return np.tanh(z)
```


ReLU Activation

$$\text{ReLU}(z) = \max(0, z)$$

Explanation: ReLU introduces non-linearity by zeroing negative values, often used in deep networks.

Example: For $z = -1$, compute $\text{ReLU}(-1)$.

Implementation:

```
def relu(z):  
    return np.maximum(0, z)
```

Heaviside Step Activation

$$H(z) = \begin{cases} 1, & z \geq 0 \\ 0, & z < 0 \end{cases}$$

Explanation: The Heaviside step function outputs binary values for classification tasks.

Example: For $z = -1$, compute $H(-1)$.

Implementation:

```
def heaviside(z):  
    return np.where(z >= 0, 1, 0)
```

Leaky ReLU Activation

$$\text{Leaky ReLU}(z) = \begin{cases} z, & z \geq 0 \\ \alpha z, & z < 0 \end{cases}$$

Explanation: Leaky ReLU allows small gradients for negative inputs, mitigating dead neurons.

Example: For $z = -1$ and $\alpha = 0.01$, compute $\text{Leaky ReLU}(-1)$.

Implementation:

```
def leaky_relu(z, alpha=0.01):  
    return np.where(z >= 0, z, alpha * z)
```

ELU Activation (Exponential Linear Unit)

$$\text{ELU}(z) = \begin{cases} z, & z \geq 0 \\ \alpha(e^z - 1), & z < 0 \end{cases}$$

Explanation: ELU smooths ReLU by providing exponential outputs for negative inputs, improving gradient flow.

Example: For $z = -1$ and $\alpha = 1$, compute $\text{ELU}(-1)$.

Implementation:

```
def elu(z, alpha=1):  
    return np.where(z >= 0, z, alpha * (np.exp(z) - 1))
```

Softmax Function

$$\text{Softmax}(z)_i = \frac{e^{z_i}}{\sum_{j=1}^n e^{z_j}}$$

Explanation: Softmax normalizes a vector into a probability distribution over n classes.

Example: For $\mathbf{z} = [1, 2, 3]$, compute $\text{Softmax}(\mathbf{z})$.

Implementation:

```
def softmax(z):  
    exp_z = np.exp(z - np.max(z)) # Numerical stability  
    return exp_z / exp_z.sum(axis=0)
```

Loss Function for Multi-Class (Cross-Entropy)

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

Explanation: Cross-entropy loss measures the dissimilarity between predicted probabilities and true labels in multi-class classification.

Example: For $y = [1, 0, 0]$ and $\hat{y} = [0.8, 0.1, 0.1]$, compute the loss.

Implementation:

```
loss = -np.mean(np.sum(y * np.log(y_pred), axis=1))
```

Gradient Descent for Neural Networks

$$\theta^{(t+1)} = \theta^{(t)} - \eta \frac{\partial \mathcal{L}}{\partial \theta}$$

Explanation: Gradient descent updates the network's weights by minimizing the loss function using gradients.

Example: Update θ for $\mathcal{L} = (y - \hat{y})^2$.

Backpropagation (Gradient for Weights)

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \delta_j a_i, \quad \delta_j = \frac{\partial \mathcal{L}}{\partial z_j} \sigma'(z_j)$$

Explanation: Backpropagation computes the gradient of the loss function with respect to the weights in a neural network using the chain rule.

Example: Compute gradients for a single-layer neural network.

Implementation:

```
delta = (y_pred - y) * sigmoid_prime(z)
grad_w = np.outer(delta, a)
```


Mean Squared Error Loss

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

Explanation: Mean squared error measures the average squared difference between predictions and actual values, commonly used in regression.

Example: For $y = [1, 2]$ and $\hat{y} = [1.1, 1.8]$, compute the loss.

Implementation:

```
loss = np.mean((y - y_pred)**2)
```

Binary Cross-Entropy Loss

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

Explanation: Binary cross-entropy measures the difference between predicted probabilities and true binary labels.

Example: For $y = [1, 0]$ and $\hat{y} = [0.9, 0.1]$, compute the loss.

Implementation:

```
loss = -np.mean(y * np.log(y_pred) + (1 - y) * np.log(1 - y_pred))
```

Batch Normalization

$$\hat{x} = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}, \quad y = \gamma \hat{x} + \beta$$

Explanation: Batch normalization normalizes inputs to a layer, reducing internal covariate shift and accelerating training.

Example: Normalize $x = [1, 2, 3]$ with $\gamma = 1, \beta = 0$.

Implementation:

```
mean = np.mean(x)
var = np.var(x)
x_norm = (x - mean) / np.sqrt(var + epsilon)
y = gamma * x_norm + beta
```

Dropout Regularization

$$\hat{a}_i = \begin{cases} 0, & \text{with probability } p \\ \frac{a_i}{1-p}, & \text{otherwise} \end{cases}$$

Explanation: Dropout randomly sets a fraction p of activations to zero during training to prevent overfitting.

Example: Apply dropout to activations $a = [1, 2, 3]$ with $p = 0.5$.

Implementation:

```
mask = np.random.rand(len(a)) > p
a_dropout = a * mask / (1 - p)
```

Gradient of Sigmoid

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

Explanation: The derivative of the sigmoid function is used in back-propagation to compute gradients efficiently.

Example: For $z = 0.5$, compute $\sigma'(0.5)$.

Implementation:

```
def sigmoid_prime(z):  
    s = sigmoid(z)  
    return s * (1 - s)
```

RMSProp for Weight Updates

$$s^{(t+1)} = \beta s^{(t)} + (1 - \beta)g^2, \quad w^{(t+1)} = w^{(t)} - \frac{\eta}{\sqrt{s^{(t+1)} + \epsilon}}g$$

Explanation: RMSProp adapts the learning rate for each weight based on the moving average of squared gradients.

Implementation:

```
s = beta * s + (1 - beta) * grad**2
w -= eta / (np.sqrt(s) + epsilon) * grad
```

Xavier (Glorot) Initialization

$$w \sim \mathcal{U}\left(-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right)$$

Explanation: Xavier initialization sets weights to maintain variance across layers, improving convergence in deep networks.

Implementation:

```
limit = np.sqrt(6 / (n_in + n_out))  
w = np.random.uniform(-limit, limit, size=(n_in, n_out))
```

L2 Regularization (Weight Decay)

$$\mathcal{L} = \mathcal{L}_0 + \frac{\lambda}{2} \|\mathbf{w}\|_2^2$$

Explanation: L2 regularization adds a penalty proportional to the square of weights to prevent overfitting.

Heaviside vs. Hard Sigmoid

$$\text{Hard Sigmoid}(z) = \max(0, \min(1, 0.2z + 0.5))$$

Explanation: Heaviside is a binary activation function, while Hard Sigmoid approximates sigmoid for efficiency.

Implementation:

```
def hard_sigmoid(z):  
    return np.clip(0.2 * z + 0.5, 0, 1)
```

Swish Activation

$$\text{Swish}(z) = z \cdot \sigma(z)$$

Explanation: Swish is a smooth, non-monotonic activation function that often outperforms ReLU in deep networks.

Implementation:

```
def swish(z):  
    return z * sigmoid(z)
```

Maxout Activation

$$\text{Maxout}(\mathbf{z}) = \max_{i \in [1, k]} z_i$$

Explanation: Maxout selects the maximum value from k linear functions, enabling learnable piecewise linear activations.

Implementation:

```
def maxout(z):  
    return np.max(z, axis=0)
```

Sparse Categorical Cross-Entropy

$$\mathcal{L} = -\frac{1}{n} \sum_{i=1}^n \log(\hat{y}_{i,y_i})$$

Explanation: Sparse categorical cross-entropy simplifies the loss calculation by directly indexing the true class probabilities.

Implementation:

```
loss = -np.mean(np.log(y_pred[range(len(y)), y]))
```

Cosine Similarity / Cosine Loss

$$\text{Cosine Similarity} = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\| \|\mathbf{v}\|}$$

Explanation: Cosine similarity measures the angle between vectors, commonly used in text and embedding similarity.

Implementation:

```
cos_sim = np.dot(u, v) / (np.linalg.norm(u) * np.linalg.norm(v))
```

SECTION 7 : CLUSTERING

Distance Metric (Euclidean)

$$d(\mathbf{u}, \mathbf{v}) = \sqrt{\sum_{i=1}^n (u_i - v_i)^2}$$

Explanation: Euclidean distance measures the straight-line distance between two points in n-dimensional space. It is widely used in clustering and nearest-neighbor methods.

Example: For $\mathbf{u} = [1, 2]$ and $\mathbf{v} = [3, 4]$, $d(\mathbf{u}, \mathbf{v}) = \sqrt{(3-1)^2 + (4-2)^2} = \sqrt{8}$.

Implementation:

```
def euclidean_distance(u, v):  
    return np.sqrt(np.sum((u - v)**2))
```

Manhattan Distance

$$d(\mathbf{u}, \mathbf{v}) = \sum_{i=1}^n |u_i - v_i|$$

Explanation: Manhattan distance measures the sum of absolute differences between corresponding components, resembling city block distances.

Example: For $\mathbf{u} = [1, 2]$ and $\mathbf{v} = [3, 4]$, $d(\mathbf{u}, \mathbf{v}) = |3 - 1| + |4 - 2| = 4$.

Implementation:

```
def manhattan_distance(u, v):  
    return np.sum(np.abs(u - v))
```

Cosine Similarity

$$\text{Cosine Similarity} = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\| \|\mathbf{v}\|}$$

Explanation: Cosine similarity measures the cosine of the angle between two vectors, capturing orientation rather than magnitude.

Example: For $\mathbf{u} = [1, 0]$ and $\mathbf{v} = [0, 1]$, similarity is 0.

Implementation:

```
def cosine_similarity(u, v):  
    return np.dot(u, v) / (np.linalg.norm(u) * np.linalg.norm(v))
```


Jaccard Similarity (Binary Data)

$$\text{Jaccard Similarity} = \frac{|\mathbf{u} \cap \mathbf{v}|}{|\mathbf{u} \cup \mathbf{v}|}$$

Explanation: Jaccard similarity compares the intersection and union of binary data, commonly used in text and set-based similarity.

Example: For $\mathbf{u} = [1, 1, 0]$ and $\mathbf{v} = [1, 0, 1]$, similarity is $\frac{1}{3}$.

Implementation:

```
def jaccard_similarity(u, v):  
    return np.sum(np.logical_and(u, v)) / np.sum(np.logical_or(u, v))
```

k-Means Objective

$$J = \sum_{i=1}^k \sum_{\mathbf{x}_j \in C_i} \|\mathbf{x}_j - \boldsymbol{\mu}_i\|^2$$

Explanation: The k-means objective minimizes the sum of squared distances between data points and their assigned cluster centroids.

Example: For points $[1, 2]$, $[3, 4]$ in cluster C_1 with centroid $[2, 3]$, compute J .

Implementation:

```
def k_means_objective(X, centroids, labels):  
    return np.sum(np.linalg.norm(X - centroids[labels], axis=1)**2)
```

Centroid Update Rule (k-Means)

$$\boldsymbol{\mu}_i = \frac{1}{|C_i|} \sum_{\mathbf{x}_j \in C_i} \mathbf{x}_j$$

Explanation: The centroid of each cluster is updated as the mean of points assigned to it.

Example: For cluster $C_1 = \{[1, 2], [3, 4]\}$, compute $\boldsymbol{\mu}_1 = [2, 3]$.

Implementation:

```
def update_centroids(X, labels, k):  
    return np.array([X[labels == i].mean(axis=0) for i in range(k)])
```

Elbow Method for Optimal k

$$J(k) = \sum_{i=1}^k \sum_{\mathbf{x}_j \in C_i} \|\mathbf{x}_j - \boldsymbol{\mu}_i\|^2$$

Explanation: The elbow method finds the optimal number of clusters k by identifying the "elbow" in the plot of $J(k)$ versus k .

Implementation:

```
def elbow_method(X, max_k):  
    distortions = []  
    for k in range(1, max_k + 1):  
        kmeans = KMeans(n_clusters=k).fit(X)  
        distortions.append(kmeans.inertia_)  
    return distortions
```

k-Medoids Objective

$$J = \sum_{i=1}^k \sum_{\mathbf{x}_j \in C_i} d(\mathbf{x}_j, \mathbf{m}_i)$$

Explanation: k-Medoids minimizes the sum of distances between data points and their cluster medoids, robust to outliers.

Example: Replace centroids with medoids for robust clustering.

Implementation:

```
def k_medoids_objective(X, medoids, labels):  
    return np.sum([np.sum(np.linalg.norm(X[labels == i]  
        - medoids[i], axis=1)) for i in range(len(medoids))])
```

Fuzzy c-Means Objective

$$J = \sum_{i=1}^c \sum_{j=1}^n u_{ij}^m \|\mathbf{x}_j - \mathbf{c}_i\|^2$$

Explanation: Fuzzy c-means assigns membership values u_{ij} to each data point for each cluster, allowing soft clustering.

Implementation:

```
def fuzzy_c_means_objective(X, centroids, memberships, m):  
    return np.sum(memberships**m * np.linalg.norm(X[:, None]  
    - centroids, axis=2)**2)
```

Silhouette Score

$$S = \frac{b - a}{\max(a, b)}, \quad a = \text{intra-cluster distance}, b = \text{nearest-cluster distance}$$

Explanation: Silhouette score evaluates the quality of clustering by comparing intra-cluster and nearest-cluster distances.

Implementation:

```
from sklearn.metrics import silhouette_score  
score = silhouette_score(X, labels)
```

Hierarchical Clustering Dendrogram

$$d(C_1, C_2) = \min_{x \in C_1, y \in C_2} \|x - y\|$$

Explanation: A dendrogram visually represents the hierarchical clustering process, showing cluster merges.

Implementation:

```
from scipy.cluster.hierarchy import dendrogram, linkage
Z = linkage(X, method='ward')
dendrogram(Z)
```


Ward's Linkage

$$d(C_1, C_2) = \frac{|C_1||C_2|}{|C_1| + |C_2|} \|\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2\|^2$$

Explanation: Ward's linkage minimizes the variance increase when merging clusters, resulting in compact clusters.

Implementation:

```
from scipy.cluster.hierarchy import linkage  
Z = linkage(X, method='ward')
```

Single vs. Complete Linkage

$$d_{\text{single}}(C_1, C_2) = \min_{x \in C_1, y \in C_2} \|x - y\|, \quad d_{\text{complete}}(C_1, C_2) = \max_{x \in C_1, y \in C_2} \|x - y\|$$

Explanation: Single linkage merges clusters based on the smallest distance between points, while complete linkage uses the largest distance. They influence the shape of hierarchical clustering.

Implementation:

```
from scipy.cluster.hierarchy import linkage
Z_single = linkage(X, method='single')
Z_complete = linkage(X, method='complete')
```

Average Linkage

$$d_{\text{average}}(C_1, C_2) = \frac{1}{|C_1||C_2|} \sum_{x \in C_1} \sum_{y \in C_2} \|x - y\|$$

Explanation: Average linkage computes the average distance between all pairs of points in two clusters, balancing the extremes of single and complete linkage.

Implementation:

```
Z_average = linkage(X, method='average')
```

Minimum Spanning Tree Criterion

$$\text{MST weight} = \sum_{(u,v) \in E} w(u,v), \quad w(u,v) = \|u - v\|$$

Explanation: The minimum spanning tree (MST) connects all points with the minimum total edge weight, often used in clustering to detect dense regions.

Implementation:

```
from scipy.sparse.csgraph import minimum_spanning_tree
mst = minimum_spanning_tree(distance_matrix(X))
```

DBSCAN Core Point Condition

$$|\text{Neighbors}(\mathbf{x})| \geq \text{MinPts}, \quad \text{where } \text{Neighbors}(\mathbf{x}) = \{\mathbf{y} : \|\mathbf{x} - \mathbf{y}\| \leq \epsilon\}$$

Explanation: A core point in DBSCAN must have at least MinPts neighbors within a distance ϵ .

Implementation:

```
core_condition = len(neighbors) >= MinPts
```

DBSCAN Density Condition

Density-connected: \exists a chain of points $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n$ such that $\|\mathbf{x}_i - \mathbf{x}_{i+1}\| \leq \epsilon$

Explanation: DBSCAN forms clusters by connecting points that are density-reachable through chains of neighbors.

Implementation:

```
from sklearn.cluster import DBSCAN  
dbscan = DBSCAN(eps=epsilon, min_samples=MinPts).fit(X)
```

Cohesion Metric

$$\text{Cohesion} = \sum_{i=1}^k \sum_{\mathbf{x}_j \in C_i} \|\mathbf{x}_j - \boldsymbol{\mu}_i\|^2$$

Explanation: Cohesion measures the compactness of clusters, where smaller values indicate tighter clusters.

Implementation:

```
cohesion = sum(np.linalg.norm(X[labels == i]
- centroids[i], axis=1).sum() for i in range(k))
```

Separation Metric

$$\text{Separation} = \sum_{i=1}^k \sum_{j=i+1}^k \|\boldsymbol{\mu}_i - \boldsymbol{\mu}_j\|^2$$

Explanation: Separation measures the distance between cluster centroids, where larger values indicate well-separated clusters.

Implementation:

```
separation = sum(np.linalg.norm(centroids[i]
- centroids[j])**2 for i in range(k) for j in range(i+1, k))
```


Soft Clustering Membership

$$u_{ij} = \frac{\|\mathbf{x}_j - \mathbf{c}_i\|^{-2/(m-1)}}{\sum_{k=1}^c \|\mathbf{x}_j - \mathbf{c}_k\|^{-2/(m-1)}}$$

Explanation: Soft clustering assigns membership values u_{ij} to each point for each cluster, indicating the degree of belonging.

Implementation:

```
memberships = 1 / (distances**(2/(m-1))) / distances.sum(axis=1, keepdims=True)
```

Entropy for Clustering Evaluation

$$H = - \sum_{i=1}^k \sum_{j=1}^n P_{ij} \log P_{ij}$$

Explanation: Entropy measures the uncertainty in clustering assignments, where lower values indicate clearer clustering.

Implementation:

```
entropy = -np.sum(P * np.log(P))
```

Mutual Information for Clustering

$$I(U, V) = \sum_{i=1}^{|U|} \sum_{j=1}^{|V|} P_{ij} \log \frac{P_{ij}}{P_i P_j}$$

Explanation: Mutual information measures the shared information between true and predicted clusters.

Implementation:

```
from sklearn.metrics import mutual_info_score  
mi = mutual_info_score(true_labels, predicted_labels)
```

F-Measure for Clustering

$$F = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Explanation: The F-measure evaluates clustering performance by balancing precision and recall.

Implementation:

```
from sklearn.metrics import f1_score  
f_measure = f1_score(true_labels, predicted_labels, average='weighted')
```

Adjusted Rand Index (ARI)

$$\text{ARI} = \frac{\text{Index} - \text{Expected Index}}{\text{Max Index} - \text{Expected Index}}$$

Explanation: ARI adjusts the Rand Index for chance, measuring clustering similarity.

Implementation:

```
from sklearn.metrics import adjusted_rand_score  
ari = adjusted_rand_score(true_labels, predicted_labels)
```

Normalized Mutual Information (NMI)

$$\text{NMI} = \frac{2I(U, V)}{H(U) + H(V)}$$

Explanation: NMI normalizes mutual information to compare clustering solutions of different sizes.

Implementation:

```
from sklearn.metrics import normalized_mutual_info_score  
nmi = normalized_mutual_info_score(true_labels, predicted_labels)
```

SECTION 8 : DIMENSIONALITY REDUCTION

Principal Component Analysis (PCA) Objective

$$\text{Maximize: } \text{Var}(\mathbf{z}) = \mathbf{w}^T \mathbf{S} \mathbf{w}, \quad \text{subject to } \|\mathbf{w}\|_2 = 1$$

Explanation: PCA seeks directions (principal components) that maximize the variance of projected data while being orthogonal to each other.

Implementation:

```
from sklearn.decomposition import PCA  
pca = PCA(n_components=k).fit(X)
```

Covariance Matrix for PCA

$$\mathbf{S} = \frac{1}{n-1}(\mathbf{X} - \bar{\mathbf{X}})^T(\mathbf{X} - \bar{\mathbf{X}})$$

Explanation: The covariance matrix captures pairwise feature dependencies and is central to PCA.

Implementation:

```
mean_X = np.mean(X, axis=0)
cov_matrix = np.cov(X - mean_X, rowvar=False)
```


Eigen Decomposition for PCA

$$\mathbf{S}\mathbf{w} = \lambda\mathbf{w}$$

Explanation: PCA uses eigen decomposition of the covariance matrix to find eigenvalues (variances) and eigenvectors (principal components).

Implementation:

```
eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)
```

SVD (Singular Value Decomposition)

$$\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$$

Explanation: SVD factorizes a matrix into orthogonal components, enabling dimensionality reduction by truncating $\mathbf{\Sigma}$.

Implementation:

```
U, S, Vt = np.linalg.svd(X, full_matrices=False)
```

Reconstruction Error for PCA

$$\text{Error} = \|\mathbf{X} - \hat{\mathbf{X}}\|_F^2, \quad \hat{\mathbf{X}} = \mathbf{Z}\mathbf{W}^T + \bar{\mathbf{X}}$$

Explanation: Reconstruction error quantifies the information loss when reducing dimensionality with PCA.

Implementation:

```
X_hat = Z @ W.T + mean_X  
reconstruction_error = np.linalg.norm(X - X_hat, 'fro')**2
```

Explained Variance Ratio

$$\text{Explained Variance Ratio} = \frac{\lambda_i}{\sum_{j=1}^n \lambda_j}$$

Explanation: The explained variance ratio quantifies the proportion of variance captured by each principal component.

Implementation:

```
explained_variance_ratio = eigenvalues / np.sum(eigenvalues)
```

Cumulative Explained Variance

$$\text{Cumulative Explained Variance} = \sum_{i=1}^k \frac{\lambda_i}{\sum_{j=1}^n \lambda_j}$$

Explanation: Cumulative explained variance evaluates the total variance captured by the first k principal components.

Implementation:

```
cumulative_explained_variance = np.cumsum(explained_variance_ratio)
```

Random Projection

$$\mathbf{X}_{\text{proj}} = \mathbf{X}\mathbf{R}, \quad \mathbf{R} \sim \mathcal{N}(0, 1)$$

Explanation: Random projection reduces dimensionality by projecting data onto a lower-dimensional random matrix while approximately preserving distances.

Implementation:

```
from sklearn.random_projection import GaussianRandomProjection
rp = GaussianRandomProjection(n_components=k).fit_transform(X)
```

Isomap Distance Matrix

d_{ij} = Shortest Path Distance on \mathcal{G} , $\mathcal{G} = (\mathbf{X}, \epsilon\text{-Neighborhoods})$

Explanation: Isomap computes geodesic distances in a graph of nearest neighbors to preserve non-linear structures in the data.

Implementation:

```
from sklearn.manifold import Isomap  
isomap = Isomap(n_neighbors=k).fit_transform(X)
```

MDS Stress Function

$$\text{Stress} = \sum_{i < j} \left(d_{ij} - \hat{d}_{ij} \right)^2$$

Explanation: The stress function measures the discrepancy between original and embedded distances in Multidimensional Scaling (MDS).

Implementation:

```
from sklearn.manifold import MDS  
mds = MDS(n_components=2).fit_transform(X)
```


Multidimensional Scaling (MDS)

$$\mathbf{X}_{\text{MDS}} = \arg \min_{\mathbf{Y}} \text{Stress}(\mathbf{Y})$$

Explanation: MDS embeds data into a lower-dimensional space while preserving pairwise distances as much as possible.

Implementation:

```
from sklearn.manifold import MDS  
mds = MDS(n_components=k).fit_transform(X)
```

NMF (Non-Negative Matrix Factorization)

$$\mathbf{X} \approx \mathbf{WH}, \quad \mathbf{W} \geq 0, \mathbf{H} \geq 0$$

Explanation: NMF factorizes a non-negative matrix into two lower-rank non-negative matrices, often used in topic modeling and image processing.

Implementation:

```
from sklearn.decomposition import NMF
nmf = NMF(n_components=k).fit_transform(X)
```

ICA (Independent Component Analysis) Objective

$$\text{Maximize: } \sum_{i=1}^n \log p(s_i), \quad \text{where } \mathbf{s} = \mathbf{W}\mathbf{X}$$

Explanation: ICA separates mixed signals into statistically independent components by maximizing non-Gaussianity.

Implementation:

```
from sklearn.decomposition import FastICA  
ica = FastICA(n_components=k).fit_transform(X)
```

Factor Analysis Model

$$\mathbf{X} = \mathbf{Z}\mathbf{\Lambda} + \boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(0, \Psi)$$

Explanation: Factor analysis models observed variables as linear combinations of latent factors plus noise.

Implementation:

```
from sklearn.decomposition import FactorAnalysis  
fa = FactorAnalysis(n_components=k).fit_transform(X)
```

Kernel PCA Transformation

$$\mathbf{K} = \phi(\mathbf{X})\phi(\mathbf{X})^T, \quad \text{Eigen Decomposition: } \mathbf{K}\alpha = \lambda\alpha$$

Explanation: Kernel PCA applies PCA in a high-dimensional feature space defined by a kernel function.

Implementation:

```
from sklearn.decomposition import KernelPCA  
kpca = KernelPCA(kernel='rbf', n_components=k).fit_transform(X)
```

LDA (Fisher's Criterion)

$$J(\mathbf{w}) = \frac{\mathbf{w}^T \mathbf{S}_B \mathbf{w}}{\mathbf{w}^T \mathbf{S}_W \mathbf{w}}$$

Explanation: LDA finds a projection that maximizes class separation by optimizing the ratio of between-class to within-class variance.

Implementation:

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis  
lda = LinearDiscriminantAnalysis(n_components=k).fit_transform(X, y)
```

Robust PCA (RPCA)

$$\mathbf{X} = \mathbf{L} + \mathbf{S}, \quad \|\mathbf{L}\|_* + \lambda \|\mathbf{S}\|_1$$

Explanation: RPCA decomposes a matrix into a low-rank component (\mathbf{L}) and a sparse component (\mathbf{S}).

Implementation:

```
from r_pca import R_pca
rpca = R_pca(X)
L, S = rpca.fit()
```

Hessian LLE

Minimize: $\|\mathbf{W}\mathbf{X} - \mathbf{X}\|_2^2$, subject to local Hessian alignment

Explanation: Hessian LLE preserves local geometric structures while optimizing a low-dimensional embedding.

Implementation:

```
from sklearn.manifold import LocallyLinearEmbedding
hessian_lle = LocallyLinearEmbedding(n_neighbors=k,
method='hessian').fit_transform(X)
```


Laplacian Eigenmaps Objective

$$\text{Minimize: } \sum_{i,j} w_{ij} \|\mathbf{y}_i - \mathbf{y}_j\|^2, \quad \mathbf{W} = \text{Graph Weights}$$

Explanation: Laplacian Eigenmaps embeds data while preserving local neighborhood information based on a graph structure.

Implementation:

```
from sklearn.manifold import SpectralEmbedding  
laplacian = SpectralEmbedding(n_components=k).fit_transform(X)
```

Autoencoder Reconstruction

$$\hat{\mathbf{X}} = \text{Decoder}(\text{Encoder}(\mathbf{X}))$$

Explanation: Autoencoders minimize reconstruction error by compressing data into a latent representation and reconstructing it.

Implementation:

```
from keras.models import Model
encoded = encoder(X)
decoded = decoder(encoded)
```

Autoencoder Latent Representation

$$\mathbf{Z} = \text{Encoder}(\mathbf{X})$$

Explanation: The latent representation (\mathbf{Z}) compresses input data into a lower-dimensional space for downstream tasks.

Implementation:

```
latent_representation = encoder.predict(X)
```

Sparse PCA Objective

Maximize: $\|\mathbf{XW}\|_2^2$, subject to sparsity constraints on \mathbf{W}

Explanation: Sparse PCA introduces sparsity in the principal components to improve interpretability.

Implementation:

```
from sklearn.decomposition import SparsePCA  
spca = SparsePCA(n_components=k).fit_transform(X)
```

t-SNE Objective

$$\text{Minimize: } KL(P||Q) = \sum_{i \neq j} P_{ij} \log \frac{P_{ij}}{Q_{ij}}$$

Explanation: t-SNE minimizes the Kullback-Leibler divergence between high-dimensional and low-dimensional distributions.

Implementation:

```
from sklearn.manifold import TSNE  
tsne = TSNE(n_components=k).fit_transform(X)
```

Gradient of t-SNE

$$\frac{\partial KL}{\partial y_i} = 4 \sum_j (P_{ij} - Q_{ij})(y_i - y_j) Q_{ij}$$

Explanation: The gradient of the t-SNE objective updates low-dimensional embeddings to align distributions.

UMAP (Uniform Manifold Approximation and Projection)

$$\text{Optimize: } \sum_{i,j} w_{ij} \|y_i - y_j\|^2 - \lambda \sum_{k,l} w_{kl} \log(\|y_k - y_l\|)$$

Explanation: UMAP preserves local and global structures by optimizing a balance between distances and densities.

Implementation:

```
import umap
umap_embedding = umap.UMAP(n_components=k).fit_transform(X)
```

SECTION 9 : PROBABILITY DISTRIBUTIONS

Bernoulli Distribution

$$P(X = x) = p^x(1 - p)^{1-x}, \quad x \in \{0, 1\}, 0 \leq p \leq 1$$

Explanation: The Bernoulli distribution models a single binary event, with success probability p .

Example: For $p = 0.7$, $P(X = 1) = 0.7$, $P(X = 0) = 0.3$.

Implementation:

```
from scipy.stats import bernoulli  
prob = bernoulli.pmf(k=1, p=0.7)
```


Binomial Distribution

$$P(X = k) = \binom{n}{k} p^k (1 - p)^{n-k}, \quad k \in \{0, 1, \dots, n\}$$

Explanation: The Binomial distribution models the number of successes in n independent Bernoulli trials.

Example: For $n = 5$ and $p = 0.5$, $P(X = 3) = \binom{5}{3} (0.5)^3 (0.5)^2 = 0.3125$.

Implementation:

```
from scipy.stats import binom
prob = binom.pmf(k=3, n=5, p=0.5)
```

Poisson Distribution

$$P(X = k) = \frac{\lambda^k e^{-\lambda}}{k!}, \quad k \in \{0, 1, 2, \dots\}$$

Explanation: The Poisson distribution models the number of events in a fixed interval, with a mean rate λ .

Example: For $\lambda = 3$, $P(X = 2) = \frac{3^2 e^{-3}}{2!} = 0.224$.

Implementation:

```
from scipy.stats import poisson
prob = poisson.pmf(k=2, mu=3)
```

Uniform Distribution (Continuous)

$$f(x) = \frac{1}{b-a}, \quad x \in [a, b]$$

Explanation: The continuous uniform distribution assigns equal probability density to all points in $[a, b]$.

Example: For $a = 0$, $b = 2$, $f(1) = \frac{1}{2}$.

Implementation:

```
from scipy.stats import uniform
prob = uniform.pdf(x=1, loc=0, scale=2)
```

Discrete Uniform Distribution

$$P(X = x) = \frac{1}{n}, \quad x \in \{1, 2, \dots, n\}$$

Explanation: The discrete uniform distribution assigns equal probability to n discrete outcomes.

Example: For $n = 6$, $P(X = 3) = \frac{1}{6}$.

Implementation:

```
from scipy.stats import randint  
prob = randint.pmf(k=3, low=1, high=7)
```

Normal (Gaussian) Distribution

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Explanation: The normal distribution models data with a symmetric bell shape, defined by mean μ and standard deviation σ .

Example: For $\mu = 0$, $\sigma = 1$, $f(0) = \frac{1}{\sqrt{2\pi}} \approx 0.398$.

Implementation:

```
from scipy.stats import norm
prob = norm.pdf(x=0, loc=0, scale=1)
```

Exponential Distribution

$$f(x) = \lambda e^{-\lambda x}, \quad x \geq 0$$

Explanation: The exponential distribution models the time between events in a Poisson process.

Example: For $\lambda = 2$, $f(1) = 2e^{-2} \approx 0.271$.

Implementation:

```
from scipy.stats import expon
prob = expon.pdf(x=1, scale=1/2)
```

Geometric Distribution

$$P(X = k) = (1 - p)^{k-1}p, \quad k \in \{1, 2, \dots\}$$

Explanation: The geometric distribution models the number of trials until the first success in repeated Bernoulli trials.

Example: For $p = 0.5$, $P(X = 3) = (0.5)^2(0.5) = 0.125$.

Implementation:

```
from scipy.stats import geom
prob = geom.pmf(k=3, p=0.5)
```

Hypergeometric Distribution

$$P(X = k) = \frac{\binom{K}{k} \binom{N-K}{n-k}}{\binom{N}{n}}$$

Explanation: The hypergeometric distribution models successes in n draws without replacement from a population of N with K successes.

Example: For $N = 20$, $K = 7$, $n = 5$, $P(X = 3)$.

Implementation:

```
from scipy.stats import hypergeom  
prob = hypergeom.pmf(k=3, M=20, n=5, N=7)
```


Beta Distribution

$$f(x) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha, \beta)}, \quad x \in [0, 1]$$

Explanation: The Beta distribution models probabilities as a function of parameters α and β .

Example: For $\alpha = 2$, $\beta = 3$, compute $f(0.5)$.

Implementation:

```
from scipy.stats import beta
prob = beta.pdf(x=0.5, a=2, b=3)
```

Gamma Distribution

$$f(x) = \frac{\beta^\alpha x^{\alpha-1} e^{-\beta x}}{\Gamma(\alpha)}, \quad x > 0$$

Explanation: The Gamma distribution generalizes the exponential distribution, often used for waiting times.

Example: For $\alpha = 2$, $\beta = 1$, compute $f(1)$.

Implementation:

```
from scipy.stats import gamma
prob = gamma.pdf(x=1, a=2, scale=1/1)
```

Multinomial Distribution

$$P(X_1 = k_1, \dots, X_k = k_k) = \frac{n!}{k_1! \dots k_k!} p_1^{k_1} \dots p_k^{k_k}$$

Explanation: The multinomial distribution generalizes the binomial distribution for multiple categories.

Example: For $n = 3$, $\mathbf{p} = [0.2, 0.5, 0.3]$, and $\mathbf{k} = [1, 1, 1]$.

Implementation:

```
from scipy.stats import multinomial
prob = multinomial.pmf(x=[1, 1, 1], n=3, p=[0.2, 0.5, 0.3])
```

Chi-Square Distribution

$$f(x) = \frac{x^{k/2-1}e^{-x/2}}{2^{k/2}\Gamma(k/2)}, \quad x > 0$$

Explanation: The chi-square distribution models the sum of squares of k independent standard normal variables, commonly used in hypothesis testing.

Example: For $k = 3$, compute $f(2)$.

Implementation:

```
from scipy.stats import chi2
prob = chi2.pdf(x=2, df=3)
```

Student's t-Distribution

$$f(x) = \frac{\Gamma((\nu + 1)/2)}{\sqrt{\nu\pi}\Gamma(\nu/2)} \left(1 + \frac{x^2}{\nu}\right)^{-(\nu+1)/2}$$

Explanation: The Student's t-distribution is used for estimating population parameters when the sample size is small.

Example: For $\nu = 5$, compute $f(1)$.

Implementation:

```
from scipy.stats import t
prob = t.pdf(x=1, df=5)
```

F-Distribution

$$f(x) = \frac{\sqrt{\left(\frac{d_1 x}{d_2}\right)^{d_1} \left(1 + \frac{d_1 x}{d_2}\right)^{-(d_1+d_2)/2}}}{x B(d_1/2, d_2/2)}, \quad x > 0$$

Explanation: The F-distribution models the ratio of variances and is commonly used in ANOVA tests.

Implementation:

```
from scipy.stats import f
prob = f.pdf(x=2, dfn=5, dfd=10)
```

Laplace Distribution

$$f(x) = \frac{1}{2b} e^{-\frac{|x-\mu|}{b}}$$

Explanation: The Laplace distribution, also known as the double exponential distribution, is used for modeling differences in data.

Implementation:

```
from scipy.stats import laplace  
prob = laplace.pdf(x=0, loc=0, scale=1)
```

Rayleigh Distribution

$$f(x) = \frac{x}{\sigma^2} e^{-x^2/(2\sigma^2)}, \quad x \geq 0$$

Explanation: The Rayleigh distribution models the magnitude of a two-dimensional vector with independent normal components.

Implementation:

```
from scipy.stats import rayleigh
prob = rayleigh.pdf(x=2, scale=1)
```


Triangular Distribution

$$f(x) = \begin{cases} \frac{2(x-a)}{(b-a)(c-a)}, & a \leq x < c \\ \frac{2(b-x)}{(b-a)(b-c)}, & c \leq x \leq b \end{cases}$$

Explanation: The triangular distribution models data with a known minimum, maximum, and mode.

Implementation:

```
from scipy.stats import triang  
prob = triang.pdf(x=0.5, c=0.5, loc=0, scale=1)
```

Log-Normal Distribution

$$f(x) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-\frac{(\ln x - \mu)^2}{2\sigma^2}}, \quad x > 0$$

Explanation: The log-normal distribution models data whose logarithm follows a normal distribution.

Implementation:

```
from scipy.stats import lognorm
prob = lognorm.pdf(x=2, s=1, scale=np.exp(0))
```

Arcsine Distribution

$$f(x) = \frac{1}{\pi\sqrt{x(1-x)}}, \quad x \in (0, 1)$$

Explanation: The arcsine distribution models probabilities with end-points more likely than the middle.

Implementation:

```
from scipy.stats import arcsine
prob = arcsine.pdf(x=0.5)
```

Beta-Binomial Distribution

$$P(X = k) = \binom{n}{k} \frac{B(k + \alpha, n - k + \beta)}{B(\alpha, \beta)}$$

Explanation: The beta-binomial distribution models overdispersed binomial outcomes using a Beta prior.

Implementation:

```
from scipy.stats import betabinom  
prob = betabinom.pmf(k=2, n=5, a=2, b=3)
```

Cauchy Distribution

$$f(x) = \frac{1}{\pi\gamma \left[1 + \left(\frac{x-x_0}{\gamma} \right)^2 \right]}$$

Explanation: The Cauchy distribution models data with heavy tails, often used in robust statistics.

Implementation:

```
from scipy.stats import cauchy
prob = cauchy.pdf(x=0, loc=0, scale=1)
```

Weibull Distribution

$$f(x) = \frac{k}{\lambda} \left(\frac{x}{\lambda}\right)^{k-1} e^{-(x/\lambda)^k}, \quad x \geq 0$$

Explanation: The Weibull distribution is used for reliability analysis and modeling lifetimes.

Implementation:

```
from scipy.stats import weibull_min  
prob = weibull_min.pdf(x=2, c=1.5, scale=1)
```

Pareto Distribution

$$f(x) = \frac{\alpha x_m^\alpha}{x^{\alpha+1}}, \quad x \geq x_m$$

Explanation: The Pareto distribution models wealth distribution and heavy-tailed phenomena.

Implementation:

```
from scipy.stats import pareto
prob = pareto.pdf(x=2, b=1)
```

Log-Cauchy Distribution

$$f(x) = \frac{1}{x\pi\gamma \left[1 + \left(\frac{\ln x - x_0}{\gamma}\right)^2\right]}, \quad x > 0$$

Explanation: The log-Cauchy distribution is the logarithmic transform of the Cauchy distribution, with heavy tails.

SECTION 10 : REINFORCEMENT LEARNING

Reward Function

$$R(s, a) = \mathbb{E}[\text{Reward} \mid s, a]$$

Explanation: The reward function provides the immediate reward received after taking action a in state s , guiding the agent's behavior.

Implementation:

```
def reward_function(state, action):  
    # Example reward calculation  
    return rewards[state, action]
```

Discounted Return

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \quad 0 \leq \gamma < 1$$

Explanation: The discounted return accumulates rewards over time, weighting future rewards by the discount factor γ .

Implementation:

```
def discounted_return(rewards, gamma):  
    G = 0  
    for t, r in enumerate(rewards):  
        G += (gamma**t) * r  
    return G
```

Bellman Equation (State-Value Function)

$$V(s) = \mathbb{E}_{\pi}[R(s, a) + \gamma V(s')]$$

Explanation: The Bellman equation relates the value of a state to the expected return from it under a policy π .

Implementation:

```
def bellman_state_value(s, rewards, transition_prob, gamma, V):  
    return np.sum(transition_prob[s] * (rewards[s] + gamma * V))
```

Bellman Equation (Action-Value Function)

$$Q(s, a) = \mathbb{E}[R(s, a) + \gamma V(s')]$$

Explanation: The Bellman equation for the action-value function expresses the value of taking action a in state s and following the policy afterward.

Implementation:

```
def bellman_action_value(s, a, rewards, transition_prob, gamma, V):  
    return rewards[s, a] + gamma * np.sum(transition_prob[s, a] * V)
```

Temporal Difference (TD) Update

$$V(s_t) \leftarrow V(s_t) + \alpha [R_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$

Explanation: The TD update improves the value estimate of a state by using the difference between predicted and actual returns.

Implementation:

```
def td_update(V, state, reward, next_state, alpha, gamma):  
    V[state] += alpha * (reward + gamma * V[next_state] - V[state])
```

Monte Carlo Policy Evaluation

$$V(s) \leftarrow \mathbb{E}[G_t \mid s_t = s]$$

Explanation: Monte Carlo evaluation updates the value of a state by averaging returns from multiple episodes starting from that state.

Implementation:

```
def monte_carlo_evaluation(V, state_returns, state_counts):  
    for state, returns in state_returns.items():  
        V[state] = np.mean(returns)
```

Policy Improvement

$$\pi'(s) = \arg \max_a Q(s, a)$$

Explanation: Policy improvement updates the policy by choosing the action that maximizes the action-value function.

Implementation:

```
def policy_improvement(Q):  
    return np.argmax(Q, axis=1)
```

Q-Learning Update

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

Explanation: Q-learning is an off-policy algorithm that updates action-value estimates using the maximum future Q-value.

Implementation:

```
def q_learning_update(Q, state, action, reward, next_state, alpha, gamma):  
    Q[state, action] += alpha * (reward + gamma * np.max(Q[next_state])  
    - Q[state, action])
```


SARSA Update

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [R_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

Explanation: SARSA is an on-policy algorithm that updates Q-values based on the action actually taken under the current policy.

Implementation:

```
def sarsa_update(Q, state, action, reward,
next_state, next_action, alpha, gamma):
    Q[state, action] += alpha * (reward + gamma * Q[next_state, next_action]
    - Q[state, action])
```

Value Iteration Update

$$V(s) \leftarrow \max_a \left[R(s, a) + \gamma \sum_{s'} P(s' | s, a) V(s') \right]$$

Explanation: Value iteration iteratively updates state values by finding the optimal action at each step.

Implementation:

```
def value_iteration(V, rewards, transition_prob, gamma):  
    for s in range(len(V)):  
        V[s] = max(np.sum(transition_prob[s, a] * (rewards[s, a]  
            + gamma * V)) for a in range(num_actions))
```

Actor–Critic Policy Update

$$\theta \leftarrow \theta + \alpha \nabla_{\theta} \log \pi_{\theta}(a_t \mid s_t) \delta_t, \quad \delta_t = R_{t+1} + \gamma V(s_{t+1}) - V(s_t)$$

Explanation: The actor updates the policy using the advantage, while the critic updates the value function to estimate the advantage.

Implementation:

```
def actor_critic_update(actor, critic, state, action, reward, next_state,
alpha, gamma):
    delta = reward + gamma * critic[next_state] - critic[state]
    actor.update(state, action, alpha * delta)
    critic[state] += alpha * delta
```

Deterministic Policy Gradient

$$\nabla J(\theta) = \mathbb{E}_{s \sim \rho^\pi} [\nabla_a Q(s, a) \nabla_\theta \pi_\theta(s)]$$

Explanation: Deterministic policy gradients update the policy directly in a continuous action space using gradients of the Q-function.

Implementation:

```
def deterministic_policy_gradient(policy, q_function, state, alpha):  
    action = policy(state)  
    grad_q = q_function.gradient(state, action)  
    grad_pi = policy.gradient(state)  
    policy.update(state, alpha * np.dot(grad_q, grad_pi))
```

Discount Factor (γ)

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \quad 0 \leq \gamma < 1$$

Explanation: The discount factor determines the weight given to future rewards. A smaller γ prioritizes immediate rewards, while a larger γ considers longer-term rewards.

Implementation:

```
def discounted_return(rewards, gamma):  
    G = 0  
    for t, r in enumerate(rewards):  
        G += (gamma**t) * r  
    return G
```

Expected SARSA

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [R_{t+1} + \gamma \mathbb{E}_{a'}[Q(s_{t+1}, a')] - Q(s_t, a_t)]$$

Explanation: Expected SARSA updates Q-values using the expected value of the next action, improving stability over standard SARSA.

Implementation:

```
def expected_sarsa(Q, state, action, reward, next_state, policy, alpha, gamma):
    expected_value = np.sum(policy[next_state] * Q[next_state])
    Q[state, action] += alpha * (reward + gamma * expected_value
    - Q[state, action])
```

Eligibility Traces Update (TD(λ))

$$\mathbf{e}_t = \gamma\lambda\mathbf{e}_{t-1} + \nabla_{\theta}V(s_t), \quad \theta \leftarrow \theta + \alpha\delta_t\mathbf{e}_t$$

Explanation: TD(λ) combines TD and Monte Carlo methods using eligibility traces, balancing bias and variance in value updates.

Implementation:

```
def td_lambda_update(V, eligibility, state, reward, next_state, alpha,
                    gamma, lambda_):
    delta = reward + gamma * V[next_state] - V[state]
    eligibility[state] += 1
    V += alpha * delta * eligibility
    eligibility *= gamma * lambda_
```

TD Error

$$\delta_t = R_{t+1} + \gamma V(s_{t+1}) - V(s_t)$$

Explanation: The TD error measures the difference between predicted and observed rewards, guiding updates in temporal difference learning.

Implementation:

```
def td_error(V, state, reward, next_state, gamma):  
    return reward + gamma * V[next_state] - V[state]
```


Stochastic Gradient Descent in RL

$$\theta \leftarrow \theta - \alpha \nabla_{\theta} \mathcal{L}(\theta)$$

Explanation: Stochastic gradient descent updates model parameters by minimizing a loss function, often used in function approximation for RL.

Implementation:

```
def sgd_update(theta, grad, alpha):  
    return theta - alpha * grad
```

Double Q-Learning

$$Q_1(s_t, a_t) \leftarrow Q_1(s_t, a_t) + \alpha \left[R_{t+1} + \gamma Q_2(s_{t+1}, \arg \max_a Q_1(s_{t+1}, a)) - Q_1(s_t, a_t) \right]$$

Explanation: Double Q-learning reduces overestimation bias by alternating updates between two Q-functions.

Implementation:

```
def double_q_learning_update(Q1, Q2, state, action, reward, next_state,
alpha, gamma):
    max_action = np.argmax(Q1[next_state])
    target = reward + gamma * Q2[next_state, max_action]
    Q1[state, action] += alpha * (target - Q1[state, action])
```

Advantage Actor–Critic (A2C)

$$\delta_t = R_{t+1} + \gamma V(s_{t+1}) - V(s_t), \quad \theta \leftarrow \theta + \alpha \nabla_{\theta} \log \pi_{\theta}(a_t \mid s_t) \delta_t$$

Explanation: A2C uses the advantage function to reduce variance in policy updates while learning the value function as a baseline.

Implementation:

```
def a2c_update(actor, critic, state, action, reward, next_state, alpha, gamma):
    delta = reward + gamma * critic[next_state] - critic[state]
    actor.update(state, action, alpha * delta)
    critic[state] += alpha * delta
```

Off-Policy Evaluation (Importance Sampling)

$$\mathbb{E}[\hat{G}] = \mathbb{E} \left[\prod_{t=0}^{T-1} \frac{\pi(a_t | s_t)}{\mu(a_t | s_t)} G_t \right]$$

Explanation: Importance sampling corrects for discrepancies between the behavior policy μ and the target policy π when estimating returns.

Implementation:

```
def importance_sampling(weights, returns):  
    return np.sum(weights * returns)
```

Policy Gradient Update Rule

$$\theta \leftarrow \theta + \alpha \nabla_{\theta} \mathbb{E}_{\pi_{\theta}} [G_t \log \pi_{\theta}(a_t \mid s_t)]$$

Explanation: The policy gradient algorithm updates parameters in the direction of performance improvement, directly optimizing the policy.

Implementation:

```
def policy_gradient_update(policy, rewards, states, actions, alpha):  
    for state, action, reward in zip(states, actions, rewards):  
        grad = policy.gradient(state, action)  
        policy.update(state, action, alpha * reward * grad)
```

Soft Q-Learning Objective

$$\mathcal{L} = \mathbb{E}_{s,a} [Q(s, a) - \alpha \log \pi(a \mid s)]$$

Explanation: Soft Q-learning optimizes a policy by balancing reward maximization and entropy regularization.

Implementation:

```
def soft_q_update(Q, policy, state, action, reward, next_state, alpha, gamma):  
    entropy = -policy.log_prob(action, state)  
    target = reward + gamma * (Q[next_state].max() + alpha * entropy)  
    Q[state, action] += alpha * (target - Q[state, action])
```

Entropy-Regularized RL

$$\pi^* = \arg \max_{\pi} \mathbb{E}[G_t] + \alpha H(\pi)$$

Explanation: Entropy regularization encourages exploration by maximizing the entropy of the policy.

Implementation:

```
def entropy_regularized_update(policy, rewards, states,
                               actions, alpha, entropy_coeff):
    for state, action, reward in zip(states, actions, rewards):
        entropy = -policy.log_prob(action, state)
        grad = policy.gradient(state, action)
        policy.update(state, action, alpha *
                      (reward + entropy_coeff * entropy) * grad)
```

Soft Actor–Critic (SAC)

$$\mathcal{L} = \mathbb{E}_{s,a} [Q(s, a) - \alpha \log \pi(a \mid s)], \quad Q(s, a) = R + \gamma V(s')$$

Explanation: SAC combines entropy regularization with actor–critic methods to improve stability and exploration in continuous control.

Implementation:

```
def sac_update(Q, policy, state, action, reward, next_state, alpha, gamma):  
    entropy = -policy.log_prob(action, state)  
    target = reward + gamma * (Q[next_state].max() + alpha * entropy)  
    Q[state, action] += alpha * (target - Q[state, action])
```


Trust Region Policy Optimization (TRPO)

$$\max_{\theta} \mathbb{E}_{\pi_{\theta}} \left[\frac{\pi_{\theta}(a \mid s)}{\pi_{\theta_{\text{old}}}(a \mid s)} A(s, a) \right], \quad \text{subject to } D_{\text{KL}}(\pi_{\theta} \parallel \pi_{\theta_{\text{old}}}) \leq \delta$$