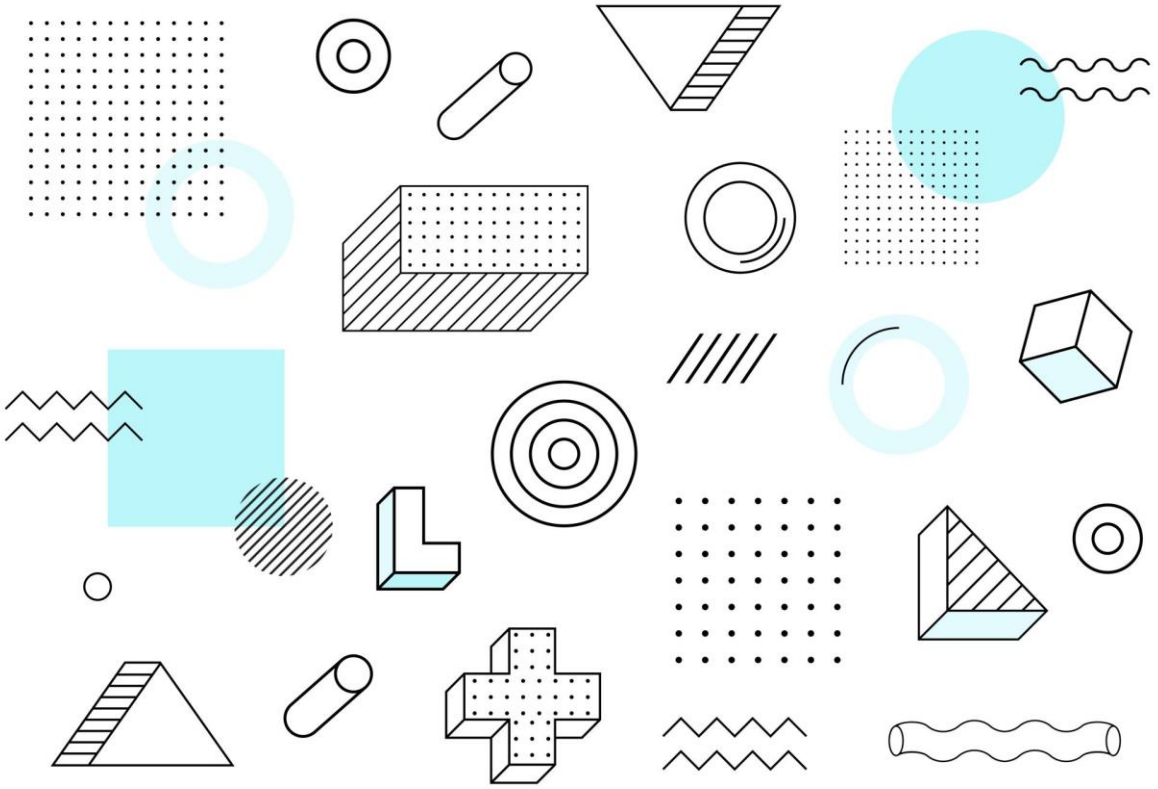
كلية الحاسبات والذكاء الاصطناعي
FACULTY OF COMPUTERS & ARTIFICIAL INTELLIGENCE
BENHA UNIVERSITY



Planning Techniques for Robotics
Lab 09 – Particle Swarm Optimization (Practical)

Eng. Yousef Elbaroudy

2025/2026

GUIDELINES

- You don’t need to memorize what will be mentioned. Instead, try to understand

- Each PowerPoint Presentation will be available as PDF file on Google Drive Folder of the Course

Particle Swarm Optimization (Recap)

- **Particle Swarm Optimization (PSO):** a **nature-inspired optimization algorithm** based on how birds flock or fish school while searching for food. It's commonly used to solve **continuous and numerical optimization problems**, where particles (solutions) adjust their positions by learning from their own experience and the swarm's best-known positions to find the optimal solution among many possibilities.

- In other words, **Particle Swarm Optimization** is inspired from nature social behavior and dynamic movements with communications of insects, birds and fish.

Motivation

The motivation behind **Particle Swarm Optimization (PSO)** was to develop a simple and efficient algorithm that can **balance exploration and exploitation**, avoid getting trapped in local optima, and handle complex non-linear problems. To achieve this, PSO relies on a population of candidate solutions **(particles)** that move through the search space by **updating their velocities and positions based on both their own best experience and the swarm's global best**, enabling the algorithm to efficiently reach near-optimal solutions **without complex control mechanisms**.

Particle Swarm Optimization (Recap)

- **Particle Swarm Optimization (PSO):** a **nature-inspired optimization algorithm** based on how birds flock or fish school while searching for food. It's commonly used to solve **continuous and numerical optimization problems**, where particles (solutions) adjust their positions by learning from their own experience and the swarm's best-known positions to find the optimal solution among many possibilities.

- In other words, **Particle Swarm Optimization** is inspired from nature social behavior and dynamic movements with communications of insects, birds and fish.

- **Particle Swarm Optimization (PSO)** is a **population-based, swarm intelligence and imitation-based algorithm** because it operates using a group of insects (particles) that ***collectively*** explore and exploit the search space, where solutions are improved through social interaction, shared information, and coordinated movement. enabling the system to refine solutions collaboratively **rather than relying on a single solution evolving independently**.

The Three Pillars of Navigation (Recap)

Inertia

Current Direction
The particle's momentum. Its tendency to keep moving in the same direction and at the same velocity

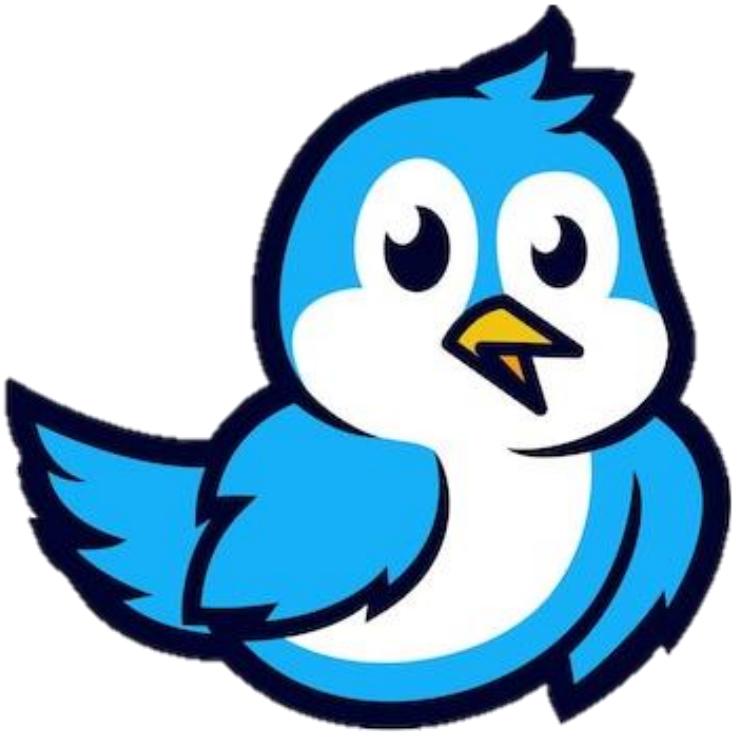
Cognitive

Personal Best Location
Personal Influence. The memory of the absolute best solution this specific particle has discovered so far.

Social

Team Best Location
Social Influence. The knowledge of the best solution discovered by anyone so far in the entire swarm of all time.

The concept of the Algorithm (Recap)



- **PSO** uses a number of agents (**particles**) that constitute a swarm moving around in the search space looking for the best solution.

- Each particle in search space **adjusts its flying** according to its **(i) own flying experience** as well as **(ii) the flying experience of other particles**.

Movement of a Particle (Recap)

$$\overrightarrow{X_i^{d+1}} = \overrightarrow{X_i^{d}} + \overrightarrow{V_i^{d+1}}$$

Position in day $d+1$ | Position in day d | Velocity in day $d+1$

$$\overrightarrow{V_i^{d+1}} = wV_i^{d} + c_1 r_1 (X_i^{Pbest} - X_i^{d}) + c_2 r_2 (X_i^{Gbest} - X_i^{d})$$

Inertia

Cognitive

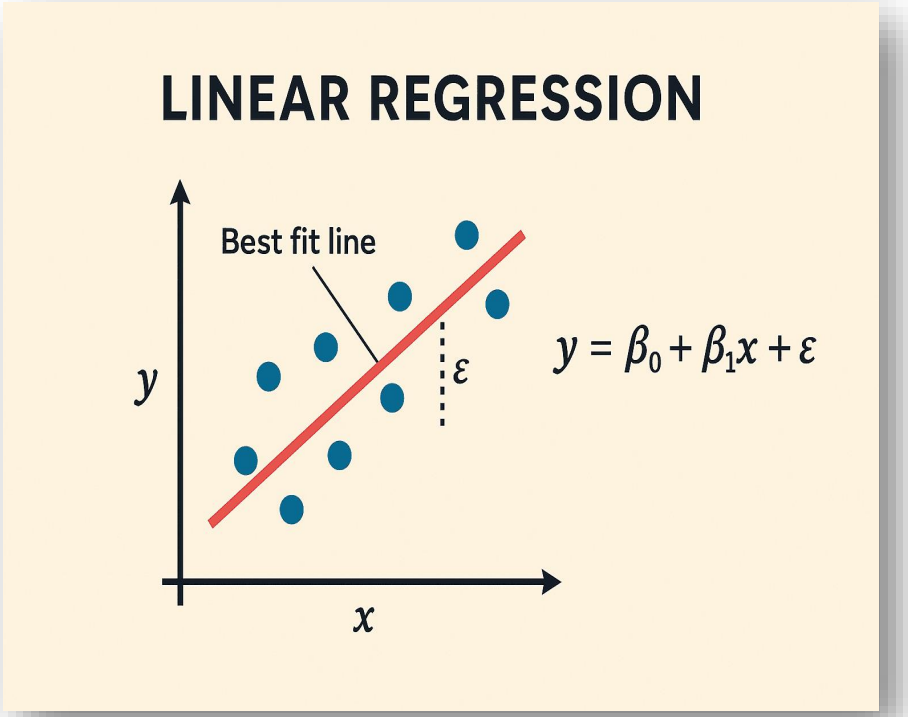
Social

Common Problem to Solve

Linear Regression Problem (LRP)

Linear regression in machine learning is a **supervised learning problem** where the goal is to model the relationship between one or more input features and a continuous output by fitting **a linear equation** to observed data.

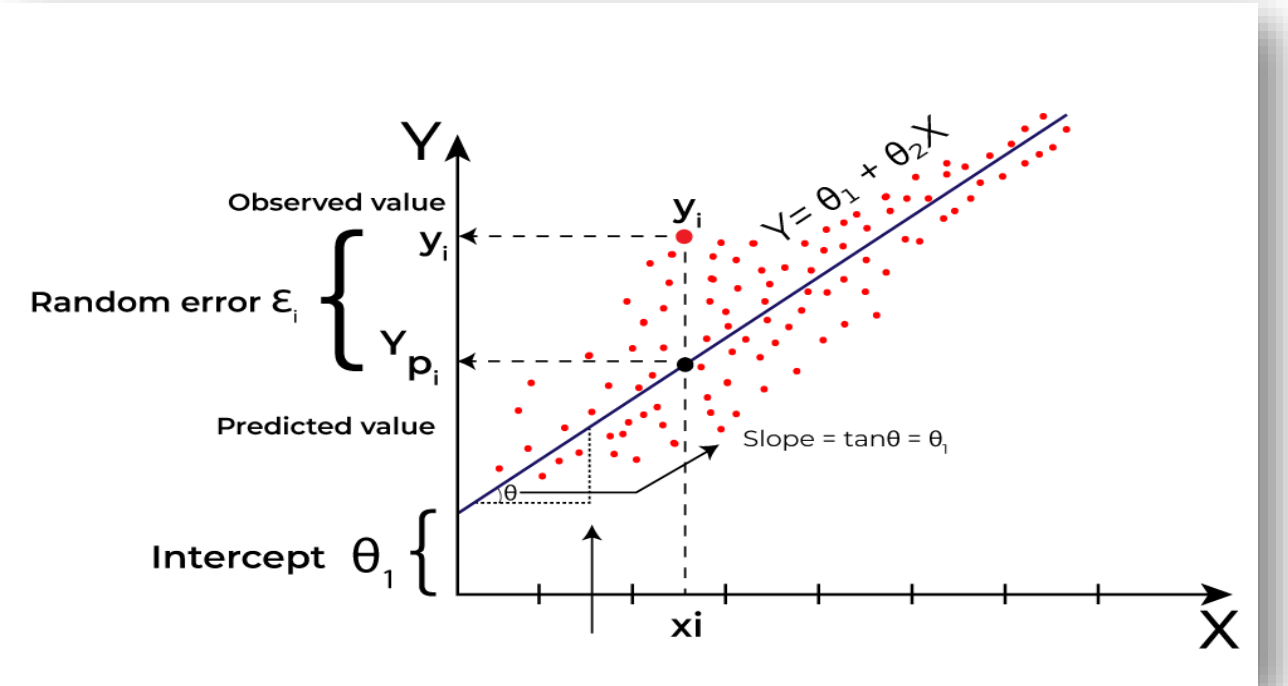
*Given a set of training examples, the model learns coefficients **(weights)** for each feature and a bias term so that the predicted values are as close as possible to the actual targets. This is typically achieved by **MINIMIZING AN ERROR FUNCTION** —most commonly the **Mean Squared Error**— which measures the average squared difference between predicted and true values. The result is a line, plane, or hyperplane (depending on the number of features) **that best represents the underlying pattern in the data**.*

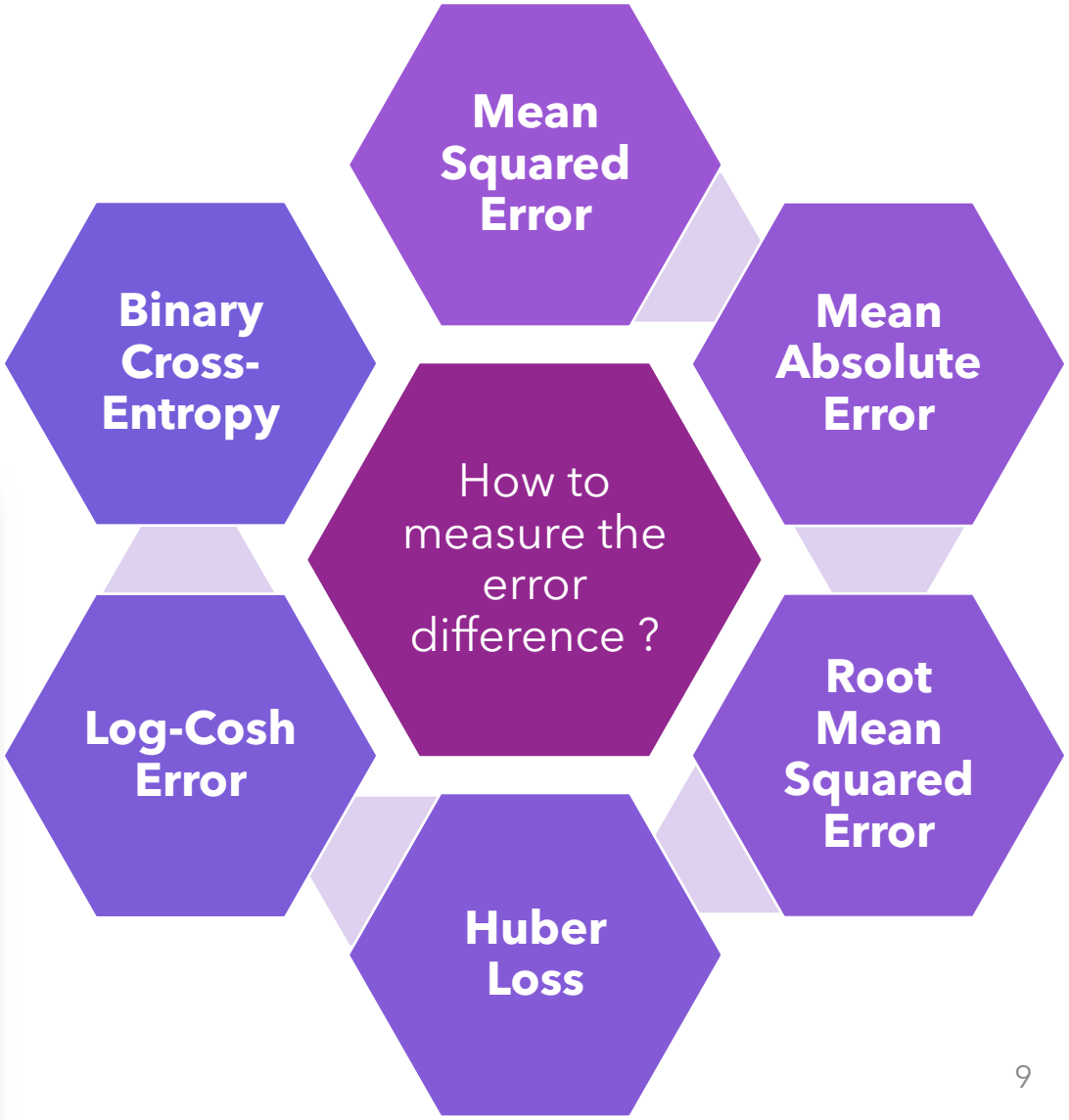


$$y = w_0x_0 + w_1x_1 + w_2x_2 + \ldots + w_nx_n$$

Linear Regression Problem (LRP)

Key Idea
Approximate the relationship between inputs and output with a weighted sum (a straight-line model) that **minimizes prediction error**.





Linear Regression Problem (LRP)

We want to find a set of weights

$W = \{w_0, w_1, w_2, \cdots, w_n\}$

Where to formulate a fitted linear line that achieves MINIMAL error between the actual target and produced output from the line

Implementation

Linear Regression Problem

Example to Implement

x1	x2	x3	y
7	2	6	56.89792
4	1	1	29.273272644758972
8	7	9	84.69510896978399
5	7	6	63.237229404377 21
7	8	3	68.24038476385691
10	5	4	76.52248
3	3	4	35.911036800151784
7	8	3	67.60074445266271
8	6	10	80.92687720164307
5	3	3	42.702644303245215
4	1	3	32.08182
8	3	4	62.111682262517 27
8	5	7	72.315807584896 15
3	3	4	35.178350592851 52
6	1	9	55.706270128407 51
5	5	1	45.640556491123 38
2	10	8	60.334911575338 08

There is a dataset with 100 samples contains 3 features (x_1, x_2, x_3) and a continuous output y required to be used with LR

Notes

We will use Mean Squared Error (MSE) as objective function for LRP

As the dataset contains 3 features (x_1,x_2,x_3) so it means that the linear equation would be $y = w_1x_1 + w_2x_2 + w_3x_3 + b$ (3 weights/dims)

As bias (b) is a constant, it is multiplied by 1 → b x 1

Notes

We will use Mean Squared Error (MSE)
as objective function for LRP

As the dataset contains 3 features (x_1,x_2,x_3) so it means that the linear equation would be
$y = w_1x_1 + w_2x_2 + w_3x_3 + b$ (3 weights/dims)

As bias (b) is a constant, it is multiplied by 1 → b x 1
which means that LR now contains 4 FEATURES

Objective Function

Mean Squared Error (MSE):

$$J(w_i) = \frac{1}{n}\sum_i^n (y_i - \hat{y})^2$$

Importing Libraries

```
import random
import numpy as np
```

LR class (Initialize Parameters)

```python
# Represent Linear Regression Problem (LRP)
class LR:
    def __init__(self, iters=None, lb=None, ub=None, population_size=50, inertia=1, c1=1.5, c2=1.5):
        # Initialize the parameters of PSO
        self.__inertia = inertia
        self.__c1 = c1
        self.__c2 = c2
        self.__pop_size = population_size
        self.__lb = lb
        self.__ub = ub
        self.__epsilon = 1*(10**-100)     # If No. of iterations is not used, an epsilon is taken in consideration
        self.__iters = iters
```

All 5 parameters of PSO is initialized in the constructor

LR class (Initialize method)

```
def __initialize(self, dims):
    # Initialize the solutions (Positions, Velocities ... etc)
    if self.__lb is not None and self.__ub is not None: # If not constrained
        self.__pop = np.random.uniform(self.__lb, self.__ub, size=(self.__pop_size, dims))
    else:
        self.__pop = np.random.rand(self.__pop_size, dims)
    self.__v = np.random.rand(self.__pop_size, dims)
    self.__fx = self.__objective_function()
    self.__pbest = self.__pop.copy()
    self.__pb_fx = self.__fx.copy()
    self.__gbest = self.__pop[np.where(self.__fx==min(self.__fx))][0]
    self.__gb_fx = min(self.__fx)
```

The initialization of solution construction such as positions, velocities, p_{best} and g_{best}

LR class (Objective Function method)

```python
def __objective_function(self):
    # For Effective performance, matrix operations are used instead of loops
    # Mean squared error is used in the Objective function
    # MSE = 1/n * Σ (y - y`)^2
    # y` = w0*x0 + w1*x1 + w2*x2 + ... + wn*xn | Where W = {w0, w1, w2, ... , wn} are weight coefficients (dimensions)
    y_hat = self.__X @ self.__pop.T
    return (1/self.__X.shape[0]) * np.sum((self.__y - y_hat)**2, axis=0)
```

Mean Squared Error (MSE):

$$J(w_i) = \frac{1}{n}\sum_i^n (y_i - \hat{y})^2$$

LR class (Objective Function method)

```python
def __objective_function(self):
    # For Effective performance, matrix operations are used instead of loops
    # Mean squared error is used in the Objective function
    # MSE = 1/n * Σ (y - y`)^2
    # y` = w0*x0 + w1*x1 + w2*x2 + ... + wn*xn | Where W = {w0, w1, w2, ... , wn} are weight coefficients (dimensions)
    y_hat = self.__X @ self.__pop.T
    return (1/self.__X.shape[0]) * np.sum((self.__y - y_hat)**2, axis=0)
```

To compute $\hat{y}$ we must use the LR equation $y = w_0x_0 + w_1x_1 \ldots.$

LR class (Objective Function method)

```python
def __objective_function(self):
    # For Effective performance, matrix operations are used instead of loops
    # Mean squared error is used in the Objective function
    # MSE = 1/n * Σ (y - y`)^2
    # y` = w0*x0 + w1*x1 + w2*x2 + ... + wn*xn | Where W = {w0, w1, w2, ... , wn} are weight coefficients (dimensions)
    y_hat = self.__X @ self.__pop.T
    return (1/self.__X.shape[0]) * np.sum((self.__y - y_hat)**2, axis=0)
```

So, we will use the weights/positions that each particle has, and multiply the features of the dataset (x_1,x_2,x_3) to the weights (w_1,w_2,w_3)

LR class (Objective Function method)

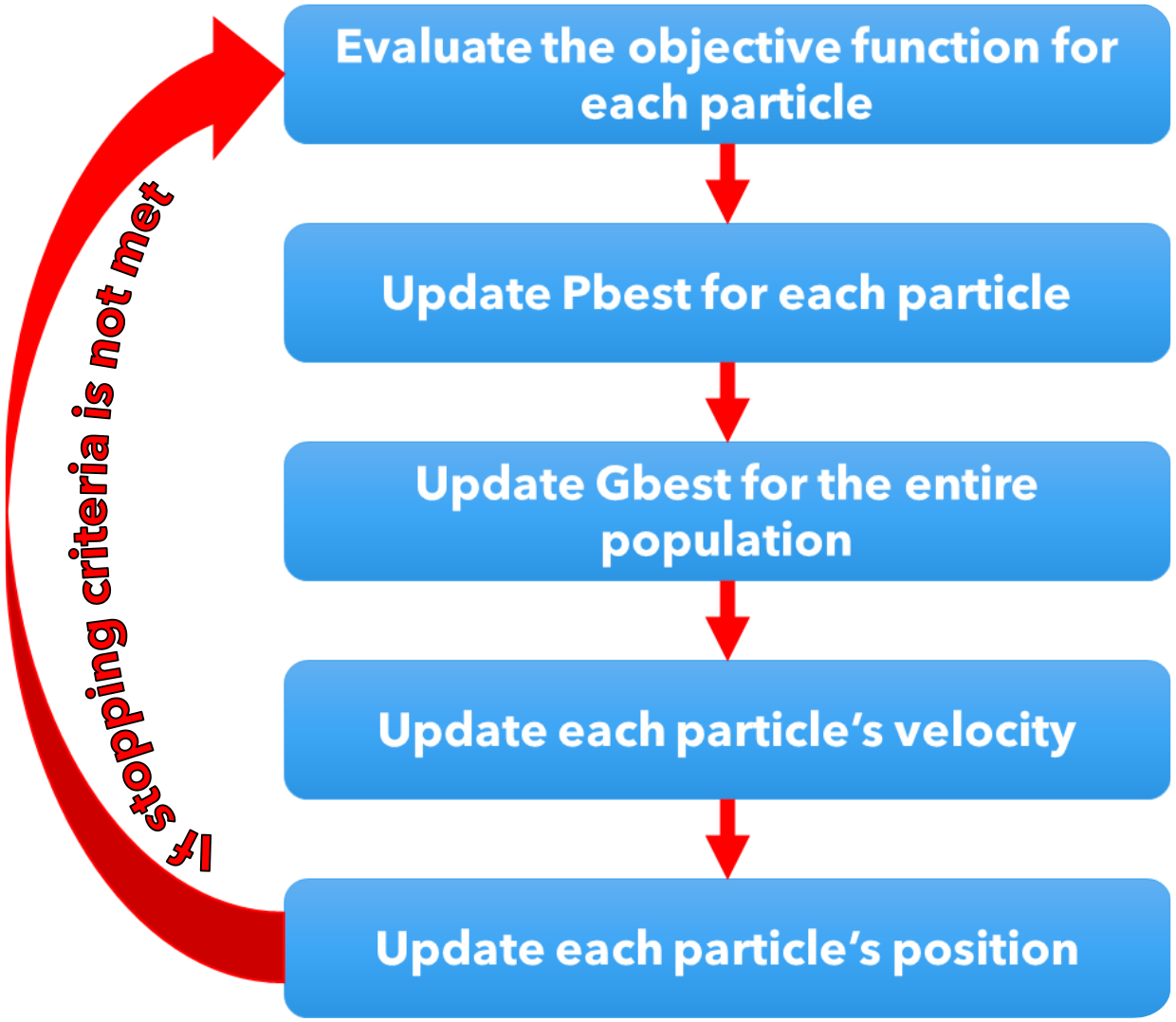
```
def __objective_function(self):
    # For Effective performance, matrix operations are used instead of loops
    # Mean squared error is used in the Objective function
    # MSE = 1/n * Σ (y - y`)^2
    # y` = w0*x0 + w1*x1 + w2*x2 + ... + wn*xn | Where W = {w0, w1, w2, ... , wn} are weight coefficients (dimensions)
    y_hat = self.__X @ self.__pop.T
    return (1/self.__X.shape[0]) * np.sum((self.__y - y_hat)**2, axis=0)
```

For effective performance, we will utilize the matrix operations in NumPy to perform matrix multiplication instead of looping

LR class (Objective Function method)

```python
def __objective_function(self):
    # For Effective performance, matrix operations are used instead of loops
    # Mean squared error is used in the Objective function
    # MSE = 1/n * Σ (y - y`)^2
    # y` = w0*x0 + w1*x1 + w2*x2 + ... + wn*xn | Where W = {w0, w1, w2, ... , wn} are weight coefficients (dimensions)
    y_hat = self.__X @ self.__pop.T
    return (1/self.__X.shape[0]) * np.sum((self.__y - y_hat)**2, axis=0)
```

$$[x_1, x_2, x_3] \times \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = x_1w_1 + x_2w_2 + x_3w_3$$



Particle Swarm Optimization Steps

LR class (fit method)

```
def fit(self, X, y):
    # Same as sklearn, this function to fit the dataset into the model for training
    # As the linear regression equation is defined as:
    # y = w1*x1 + w2*x2 + ... + wn*xn + b
    # You may notice that b is a bias -noise- and considered as WEIGHT
    # b is MULTIPLIED BY 1
    # b*1
    bias = np.ones((X.shape[0],1))
    self.__X = np.hstack((X,bias))  # concat the ones with the dataset to be --> x1, x2, ... , xn , bias (ones)
    self.__y = y.reshape((y.shape[0], 1))
    self.__initialize(self.__X.shape[-1])

    # Save best solution of the previous iteration
    old_gb_fx = self.__gb_fx

    # Check if iterations is defined by user or not
    if self.__iters is not None:
        iterations = self.__iters
    else:
        iterations = 1000000
```

The same method "*fit*" of sklearn is used here to train the LR on the dataset (X features and y targets)

LR class (fit method)

```
def fit(self, X, y):
    # Same as sklearn, this function to fit the dataset into the model for training
    # As the linear regression equation is defined as:
    # y = w1*x1 + w2*x2 + ... + wn*xn + b
    # You may notice that b is a bias -noise- and considered as WEIGHT
    # b is MULTIPLIED BY 1
    # b*1
    bias = np.ones((X.shape[0],1))
    self.__X = np.hstack((X,bias))  # concat the ones with the dataset to be --> x1, x2, ... , xn , bias (ones)
    self.__y = y.reshape((y.shape[0], 1))
    self.__initialize(self.__X.shape[-1])

    # Save best solution of the previous iteration
    old_gb_fx = self.__gb_fx

    # Check if iterations is defined by user or not
    if self.__iters is not None:
        iterations = self.__iters
    else:
        iterations = 1000000
```

As mentioned before, the LR equation has (bias) term which is a weight b that multiplied by one

LR class (fit method)

```python
def fit(self, X, y):
    # Same as sklearn, this function to fit the dataset into the model for training
    # As the linear regression equation is defined as:
    # y = w1*x1 + w2*x2 + ... + wn*xn + b
    # You may notice that b is a bias -noise- and considered as WEIGHT
    # b is MULTIPLIED BY 1
    # b*1
    bias = np.ones((X.shape[0],1))
    self.__X = np.hstack((X,bias))  # concat the ones with the dataset to be --> x1, x2, ... , xn , bias (ones)
    self.__y = y.reshape((y.shape[0], 1))
    self.__initialize(self.__X.shape[-1])

    # Save best solution of the previous iteration
    old_gb_fx = self.__gb_fx

    # Check if iterations is defined by user or not
    if self.__iters is not None:
        iterations = self.__iters
    else:
        iterations = 1000000
```

So, we added a column of ones into the dataset in which we add another dimension/feature that is w_0

LR class (fit method)

```
def fit(self, X, y):
    # Same as sklearn, this function to fit the dataset into the model for training
    # As the linear regression equation is defined as:
    # y = w1*x1 + w2*x2 + ... + wn*xn + b
    # You may notice that b is a bias -noise- and considered as WEIGHT
    # b is MULTIPLIED BY 1
    # b*1
    bias = np.ones((X.shape[0],1))
    self.__X = np.hstack((X,bias))  # concat the ones with the dataset to be --> x1, x2, ... , xn , bias (ones)
    self.__y = y.reshape((y.shape[0], 1))
    self.__initialize(self.__X.shape[-1])

    # Save best solution of the previous iteration
    old_gb_fx = self.__gb_fx

    # Check if iterations is defined by user or not
    if self.__iters is not None:
        iterations = self.__iters
    else:
        iterations = 1000000
```

The rest of the method is preparing PSO to be performed

LR class (fit method) –Cont.

```
# Steps of Particle Swarm Optimization Algorithm
for iters in range(iterations):
    # Step (1): Evaluate objective functions
    self.__fx = self.__objective_function()

    # Step (2) and (3): Update pbest for each particle, and gbest for the entire population
    for particle in range(self.__pop_size):
        # Update pbest for each particle
        if self.__fx[particle] < self.__pb_fx[particle]:
            self.__pbest[particle] = self.__pop[particle].copy()
            self.__pb_fx[particle] = self.__fx[particle]

        # Update gbest for the entire population
        if self.__fx[particle] < self.__gb_fx:
            self.__gbest = self.__pop[particle].copy()
            self.__gb_fx = self.__fx[particle]
```

LR class (fit method) –Cont.

```
# Steps of Particle Swarm Optimization Algorithm
for iters in range(iterations):
    # Step (1): Evaluate objective functions
    self.__fx = self.__objective_function()

    # Step (2) and (3): Update pbest for each particle, and gbest for the entire population
    for particle in range(self.__pop_size):
        # Update pbest for each particle
        if self.__fx[particle] < self.__pb_fx[particle]:
            self.__pbest[particle] = self.__pop[particle].copy()
            self.__pb_fx[particle] = self.__fx[particle]

        # Update gbest for the entire population
        if self.__fx[particle] < self.__gb_fx:
            self.__gbest = self.__pop[particle].copy()
            self.__gb_fx = self.__fx[particle]
```

Step (1): Evaluate f(x) of all particles

LR class (fit method) –Cont.

```
# Steps of Particle Swarm Optimization Algorithm
for iters in range(iterations):
    # Step (1): Evaluate objective functions
    self.__fx = self.__objective_function()

    # Step (2) and (3): Update pbest for each particle, and gbest for the entire population
    for particle in range(self.__pop_size):
        # Update pbest for each particle
        if self.__fx[particle] < self.__pb_fx[particle]:
            self.__pbest[particle] = self.__pop[particle].copy()
            self.__pb_fx[particle] = self.__fx[particle]

        # Update gbest for the entire population
        if self.__fx[particle] < self.__gb_fx:
            self.__gbest = self.__pop[particle].copy()
            self.__gb_fx = self.__fx[particle]
```

Step (2): Update Pbest of each particle

LR class (fit method) –Cont.

```
# Steps of Particle Swarm Optimization Algorithm
for iters in range(iterations):
    # Step (1): Evaluate objective functions
    self.__fx = self.__objective_function()

    # Step (2) and (3): Update pbest for each particle, and gbest for the entire population
    for particle in range(self.__pop_size):
        # Update pbest for each particle
        if self.__fx[particle] < self.__pb_fx[particle]:
            self.__pbest[particle] = self.__pop[particle].copy()
            self.__pb_fx[particle] = self.__fx[particle]

        # Update gbest for the entire population
        if self.__fx[particle] < self.__gb_fx:
            self.__gbest = self.__pop[particle].copy()
            self.__gb_fx = self.__fx[particle]
```

Step (3): Update Gbest of the entire population

LR class (fit method) –Cont.

```
# For effective performance, Matrix operations are used instead of loops
# Step (4): Update velocities
# V = w*V + c1 * r1 * (pbest - X) + c2 * r2 * (gbest - X)
r1 = np.random.rand()
r2 = np.random.rand()
inertia = self.__inertia * (iterations-iters)/iterations # Decay of inertia --> More exploration at beginning and vice versa
self.__v = inertia * self.__v + self.__c1 * r1 * (self.__pbest - self.__pop) + self.__c2 * r2 * (self.__gbest - self.__pop)

# Step (5): Update positions
self.__pop = self.__pop + self.__v

# If boundaries are constrained, any position that exceeds the boundary should be CLIPPED
if self.__lb is not None and self.__ub is not None:
    self.__pop[self.__pop > self.__ub] = self.__ub
    self.__pop[self.__pop < self.__lb] = self.__lb
```

Step (4): Update Velocities

LR class (fit method) –Cont.

```
# For effective performance, Matrix operations are used instead of loops
# Step (4): Update velocities
# V = w*V + c1 * r1 * (pbest - X) + c2 * r2 * (gbest - X)
r1 = np.random.rand()
r2 = np.random.rand()
inertia = self.__inertia * (iterations-iters)/iterations # Decay of inertia --> More exploration at beginning and vice versa
self.__v = inertia * self.__v + self.__c1 * r1 * (self.__pbest - self.__pop) + self.__c2 * r2 * (self.__gbest - self.__pop)

# Step (5): Update positions
self.__pop = self.__pop + self.__v

# If boundaries are constrained, any position that exceeds the boundary should be CLIPPED
if self.__lb is not None and self.__ub is not None:
    self.__pop[self.__pop > self.__ub] = self.__ub
    self.__pop[self.__pop < self.__lb] = self.__lb
```

Step (4): Update Velocities

A tricky move is performed here, since inertia is responsible of Exploration in which it depends on the current direction of the particle only.

LR class (fit method) –Cont.

```
# For effective performance, Matrix operations are used instead of loops
# Step (4): Update velocities
# V = w*V + c1 * r1 * (pbest - X) + c2 * r2 * (gbest - X)
r1 = np.random.rand()
r2 = np.random.rand()
inertia = self.__inertia * (iterations-iters)/iterations # Decay of inertia --> More exploration at beginning and vice versa
self.__v = inertia * self.__v + self.__c1 * r1 * (self.__pbest - self.__pop) + self.__c2 * r2 * (self.__gbest - self.__pop)

# Step (5): Update positions
self.__pop = self.__pop + self.__v

# If boundaries are constrained, any position that exceeds the boundary should be CLIPPED
if self.__lb is not None and self.__ub is not None:
    self.__pop[self.__pop > self.__ub] = self.__ub
    self.__pop[self.__pop < self.__lb] = self.__lb
```

Inertia decay: reducing the value of inertia over iterations to increase exploitation toward best solutions in late iterations.

LR class (fit method) –Cont.

```
# For effective performance, Matrix operations are used instead of loops
# Step (4): Update velocities
# V = w*V + c1 * r1 * (pbest - X) + c2 * r2 * (gbest - X)
r1 = np.random.rand()
r2 = np.random.rand()
inertia = self.__inertia * (iterations-iters)/iterations # Decay of inertia --> More exploration at beginning and vice versa
self.__v = inertia * self.__v + self.__c1 * r1 * (self.__pbest - self.__pop) + self.__c2 * r2 * (self.__gbest - self.__pop)

# Step (5): Update positions
self.__pop = self.__pop + self.__v

# If boundaries are constrained, any position that exceeds the boundary should be CLIPPED
if self.__lb is not None and self.__ub is not None:
    self.__pop[self.__pop > self.__ub] = self.__ub
    self.__pop[self.__pop < self.__lb] = self.__lb
```

Step (5): Update Positions

LR class (fit method) –Cont.

```
# For effective performance, Matrix operations are used instead of loops
# Step (4): Update velocities
# V = w*V + c1 * r1 * (pbest - X) + c2 * r2 * (gbest - X)
r1 = np.random.rand()
r2 = np.random.rand()
inertia = self.__inertia * (iterations-iters)/iterations # Decay of inertia --> More exploration at beginning and vice versa
self.__v = inertia * self.__v + self.__c1 * r1 * (self.__pbest - self.__pop) + self.__c2 * r2 * (self.__gbest - self.__pop)

# Step (5): Update positions
self.__pop = self.__pop + self.__v

# If boundaries are constrained, any position that exceeds the boundary should be CLIPPED
if self.__lb is not None and self.__ub is not None:
    self.__pop[self.__pop > self.__ub] = self.__ub
    self.__pop[self.__pop < self.__lb] = self.__lb
```

If the problem is constrained (other than LRP) so we may use the upper and lower bound and clip the positions that exceeded the limits

LR class (fit method) –Cont.

```
# Print the result
print(f"\rIteration: {iters+1:00000} and gbest = {self.__gb_fx}", end="")

# If iterations is not defined by user, no difference stopping criteria is used
if abs(old_gb_fx - self.__gb_fx) < self.__epsilon and iters > 100 and self.__iters is None:
    break
old_gb_fx = self.__gb_fx
```

For diversity, if maximum iterations stopping criteria is not used. Instead, another one can be applied that is epsilon difference, so if the changes are too small, the algorithm will terminate

LR class (fit method) –Cont.

```
# Print the result
print(f"\rIteration: {iters+1:00000} and gbest = {self.__gb_fx}", end="")

# If iterations is not defined by user, no difference stopping criteria is used
if abs(old_gb_fx - self.__gb_fx) < self.__epsilon and iters > 100 and self.__iters is None:
    break
old_gb_fx = self.__gb_fx
```

For diversity, if maximum iterations stopping criteria is not used. Instead, another one can be applied that is epsilon difference, so if the changes are too small, the algorithm will terminate

LR class (predict method)

```python
def predict(self, X):
    # Same as Sklearn, predict function to obtain the result on samples of data
    bias = np.ones((X.shape[0], 1))
    new_X = np.hstack((X,bias))
    # y = w1*x1 + w2*x2 + ... + wn*xn + b | use the W coefficients of the found positions in PSO
    predicted = new_X @ self.__gbest.T
    return np.reshape(predicted, (predicted.shape[0], 1))
```

The same as Sklearn, the model has a method "predict" to get a test sample and obtain the answer

$$y = w_1x_1 + w_2x_2 + w_3x_3 + b$$

but now we got the optimal Weights (g_{best})

LR class (score method)

```
def score(self, X, y):
    # Use R2 Score as evaluation metric of the model
    # R2 = 1 - (SSresidual / SStotal)
    # SSresidual = Σ(y - y`)^2
    # SStotal    = Σ(y - mean)^2
    y = y.reshape((y.shape[0],1))
    y_hat = self.predict(X)
    avg = sum(y)/len(y)
    SSresidual = sum((y-y_hat)**2)
    SStotal = sum((y-avg)**2)
    r2 = 1 - (SSresidual/SStotal)
    return r2
```

In order to assess the model's performance, we used R^2 score for regression tasks, it gives a measurement similar somehow to Accuracy

```
dataset = np.loadtxt("multivariate_linear_regression_100_samples.csv", delimiter=',', skiprows=1)
dataset
```

```
array([[ 7.        ,  2.        ,  6.        , 56.89791616],
       [ 4.        ,  1.        ,  1.        , 29.27327264],
       [ 8.        ,  7.        ,  9.        , 84.69510897],
       [ 5.        ,  7.        ,  6.        , 63.2372294 ],
       [ 7.        ,  8.        ,  3.        , 68.24038476],
       [10.        ,  5.        ,  4.        , 76.52248213],
       [ 3.        ,  3.        ,  4.        , 35.9110368 ],
       [ 7.        ,  8.        ,  3.        , 67.60074445],
       [ 8.        ,  6.        , 10.        , 80.9268772 ],
       [ 5.        ,  3.        ,  3.        , 42.7026443 ],
       [ 4.        ,  1.        ,  3.        , 32.08182402],
       [ 8.        ,  3.        ,  4.        , 62.11168226],
       [ 8.        ,  5.        ,  7.        , 72.31580758],
       [ 3.        ,  3.        ,  4.        , 35.17835059],
       [ 6.        ,  1.        ,  9.        , 55.70627013],
       [ 5.        ,  5.        ,  1.        , 45.64055649],
       [ 2.        , 10.        ,  8.        , 60.33491158],
```

Reading the dataset

```
X = dataset[:,:-1] # Split the features into variable X
y = dataset[:,-1]  # Split the target into variable y
```

```
model = LR(iters=50000, c1=2, c2=2, population_size=20)
model.fit(X,y)
```

```
Iteration: 50000 and gbest = 0.7084646101621596
```

TESTING

EVALUATING THE MODEL USING R^2

```
print("The model achieved R2 Score= " ,model.score(X,y)[0] * 100, "%")
model.get_best()
```

```
The model achieved R2 Score=  99.70531601432164 %
(array([4.93694715, 2.93813932, 2.00445684, 4.64253663]), 0.7084646101621596)
```

Let's see if it is true answer ?

The original line formed this dataset is:

$$y = 5x_1 + 3x_2 + 2x_3 + 4$$

Our solution is [4.93694715, 2.93813932, 2.00445684, 4.64253663]

Let's see if it is true answer ?

The original line formed this dataset is:

$$y = 5x_1 + 3x_2 + 2x_3 + 4$$

Lets test with unseen sample !

Our solution is [4.93694715, 2.93813932, 2.00445684, 4.64253663]

Let's see if it is true answer ?

The original line formed this dataset is:

$$y = 5x_1 + 3x_2 + 2x_3 + 4$$

[4.5, -2, 3]

Our solution is [4.93694715, 2.93813932, 2.00445684, 4.64253663]

Let's see if it is true answer ?

The original line formed this dataset is:

$$y = 5x_1 + 3x_2 + 2x_3 + 4$$

$$y = 5(4.5) + 3(-2) + 2(3) + 4$$

Our solution is [4.93694715, 2.93813932, 2.00445684, 4.64253663]

Let's see if it is true answer ?

The original line formed this dataset is:

$$y = 5x_1 + 3x_2 + 2x_3 + 4$$

$$y = 26.5$$

Our solution is [4.93694715, 2.93813932, 2.00445684, 4.64253663]

Let's see if it is true answer ?

The original line formed this dataset is:

$$y = 5x_1 + 3x_2 + 2x_3 + 4$$

$$y = 26.5$$

What is our model answer ?

Our solution is [4.93694715, 2.93813932, 2.00445684, 4.64253663]

Let's see if it is true answer ?

The original line formed this dataset is:

$$y = 5x_1 + 3x_2 + 2x_3 + 4$$

$$y = 26.5$$

```
sample = np.array([[4.5, -2, 3]])
print("Model answer's: ", model.predict(sample)[0,0])
print("The original relationship is y = 5x1 + 3x2 + 2x3 + 4 , so the test sample must be (", 5*(4.5) + 3*(-2) + 2*(3) + 4, ")")
```

```
Model answer's:  26.99589067796907
The original relationship is y = 5x1 + 3x2 + 2x3 + 4 , so the test sample must be ( 26.5 )
```

Our solution is [4.93694715, 2.93813932, 2.00445684, 4.64253663]

Let's see if it is true answer ?

The original line formed this dataset is:

$$y = 5x_1 + 3x_2 + 2x_3 + 4$$

$$y = 26.5$$

```
sample = np.array([[4.5, -2, 3]])
print("Model answer's: ", model.predict(sample)[0,0])
print("The original relationship is y = 5x1 + 3x2 + 2x3 + 4 , so the test sample must be (", 5*(4.5) + 3*(-2) + 2*(3) + 4, ")")
```

```
Model answer's:  26.99589067796907
The original relationship is y = 5x1 + 3x2 + 2x3 + 4 , so the test sample must be ( 26.5 )
```

Optimal Solution

Our solution is [4.93694715, 2.93813932, 2.00445 663]

Behavior of Particles

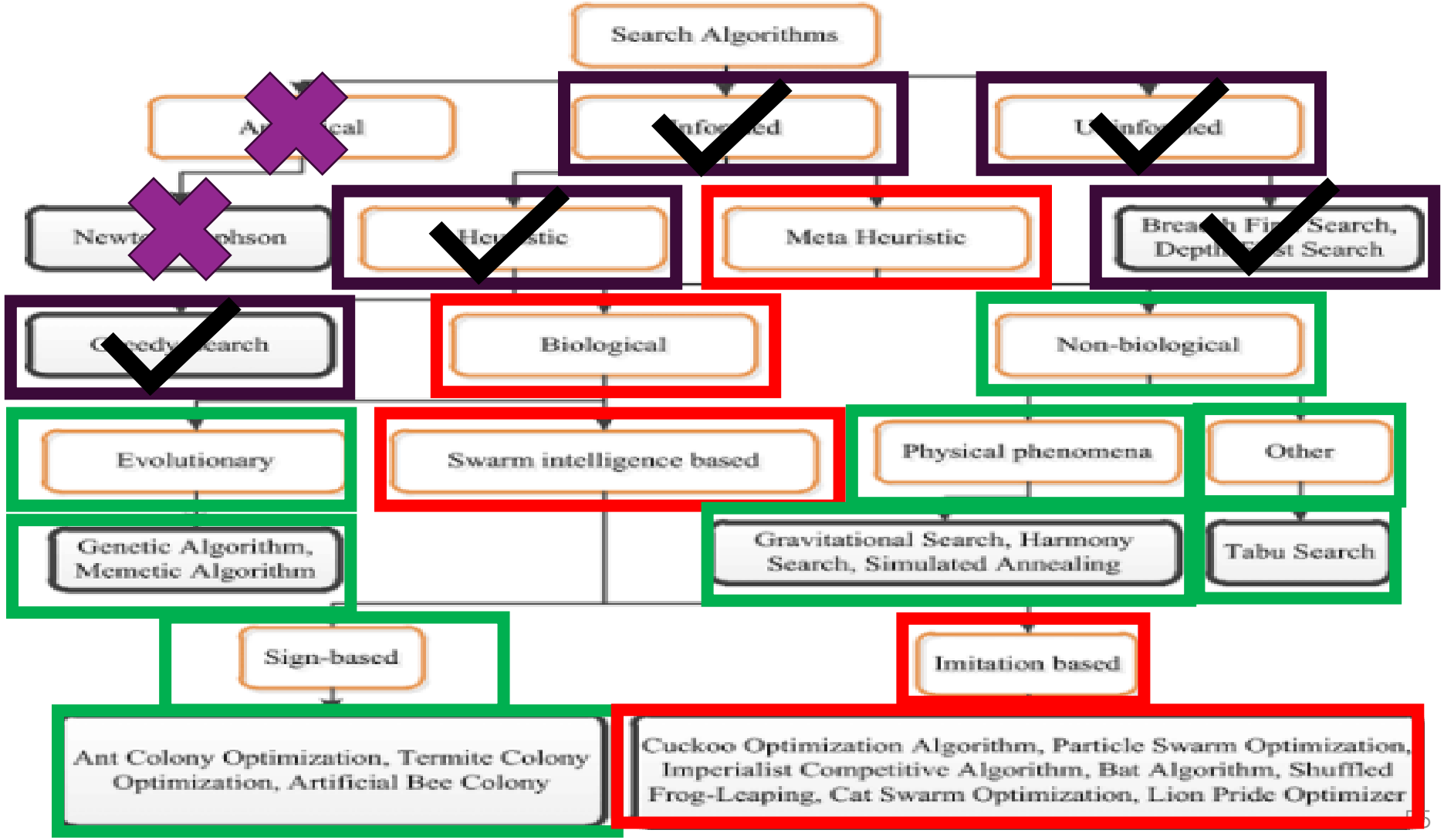
```
array([[4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663]])
```

Behavior of Particles

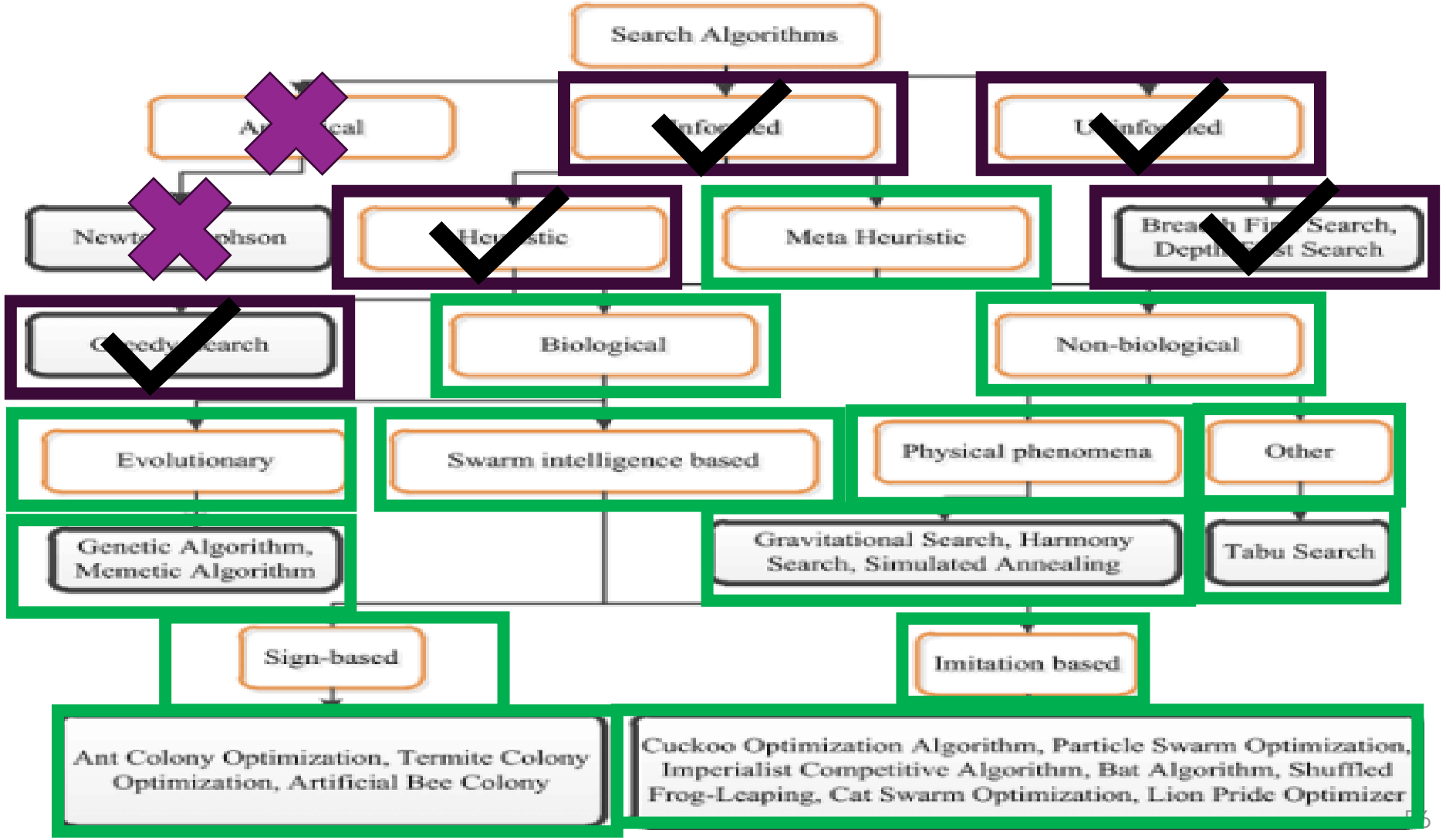
Convergent

```
array([[4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663],
       [4.93694715, 2.93813932, 2.00445684, 4.64253663]])
```

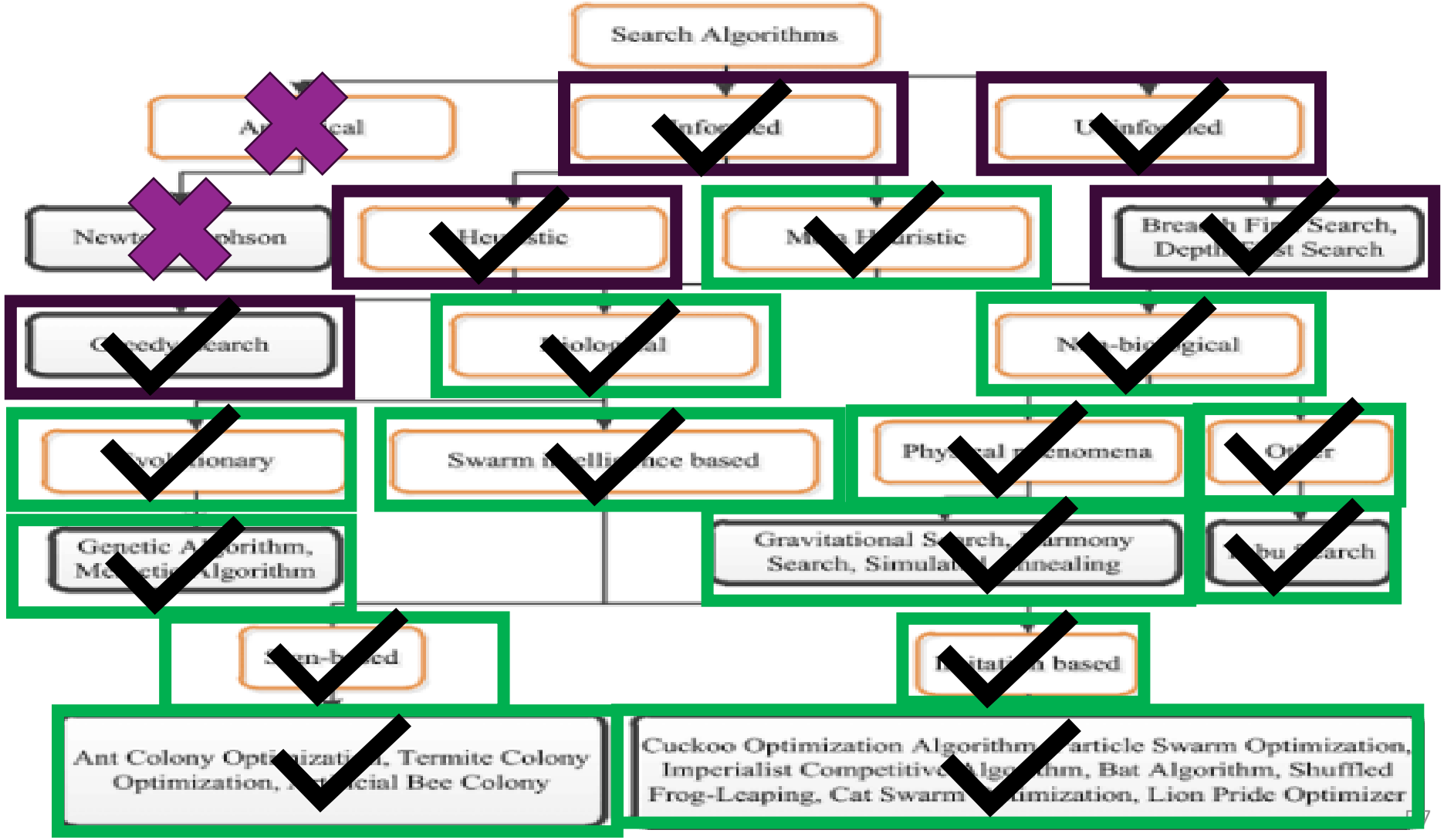
The classification of Searching Algorithms

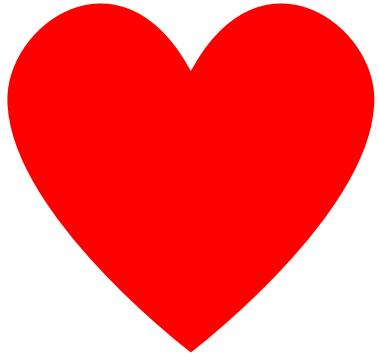


The classification of Searching Algorithms



The classification of Searching Algorithms





End of Course