COMPARATIVE ANALYSIS OF STANDARD PSO AND MODIFIED PSO

A STUDY ON OPTIMIZATION ALGORITHMS

PROFESSOR
TALHA ALI KHAN

TEAM MEMBER:

MARCELINA OWONO
AISWARYA SAJEEV
SURAJ DESHWAL
AJIL ALEYAS
ANIKET DERE

AGENDA

1. INTRODUCTION
2. STANDARD PSO OVERVIEW
3. MODIFIED PSO OVERVIEW
4. T-VALUE OVERVIEW
5. RESULTS AND DISCUSSIONS
6. CONCLUSION

STANDARD PSO OVERVIEW

BASIC PSO ALGORITHM:

- IMPORT NECESSARY LIBRARIES:
- IMPORT NUMPY AS NP: NUMPY IS **USED FOR NUMERICAL OPERATIONS.**
- IMPORT PANDAS AS PD: PANDAS IS **USED FOR DATA MANIPULATION.**
- IMPORT TIME: THE TIME MODULE IS **USED TO MEASURE THE CONVERGENCE TIME**

.

OBJECTIVE FUNCTIONS:

- DEFINE SEVERAL BENCHMARK FUNCTIONS (E.G., SPHERE, ROSENBROCK, QUARTIC) FOLLOWING A CONSISTENT PATTERN.
- OBJECTIVE FUNCTIONS REPRESENT THE PROBLEMS THE PARTICLE SWARM OPTIMIZATION (PSO) ALGORITHM WILL SOLVE.

```python
import numpy as np
import pandas as pd
import time

def sphere(x):
    return np.sum(x**2)

def rosenbrock(x):
    return sum(100 * (x[i+1] - x[i]**2)**2 + (1 - x[i])**2 for i in range(len(x)-1))

def step_2(x):
    return np.sum(np.floor(x + 0.5)**2)

# Add more functions following the pattern above...
def quartic(x):
    return np.sum(7 * x**4 - 10 * x**3 + 5 * x**2)

def schwefel_2_21(x):
    return np.sum(abs(x) + np.prod(abs(np.sin(x))))

def schwefel_2_22(x):
    return -np.sum(np.sin(np.sqrt(abs(x))))


def foxholes(x):
    return np.sum(
      np.exp(-0.2 * np.sqrt(np.mean(x ** 2, axis=0))) + np.exp(np.sin(np.sqrt(np.mean(x **
    x.shape[0])

def kowalik(x):
    return np.sum([i * (x[j] - 1)**2 + (1 - x[j])**2 for j in range(len(x)) for i in range
```

STANDARD PSO OVERVIEW

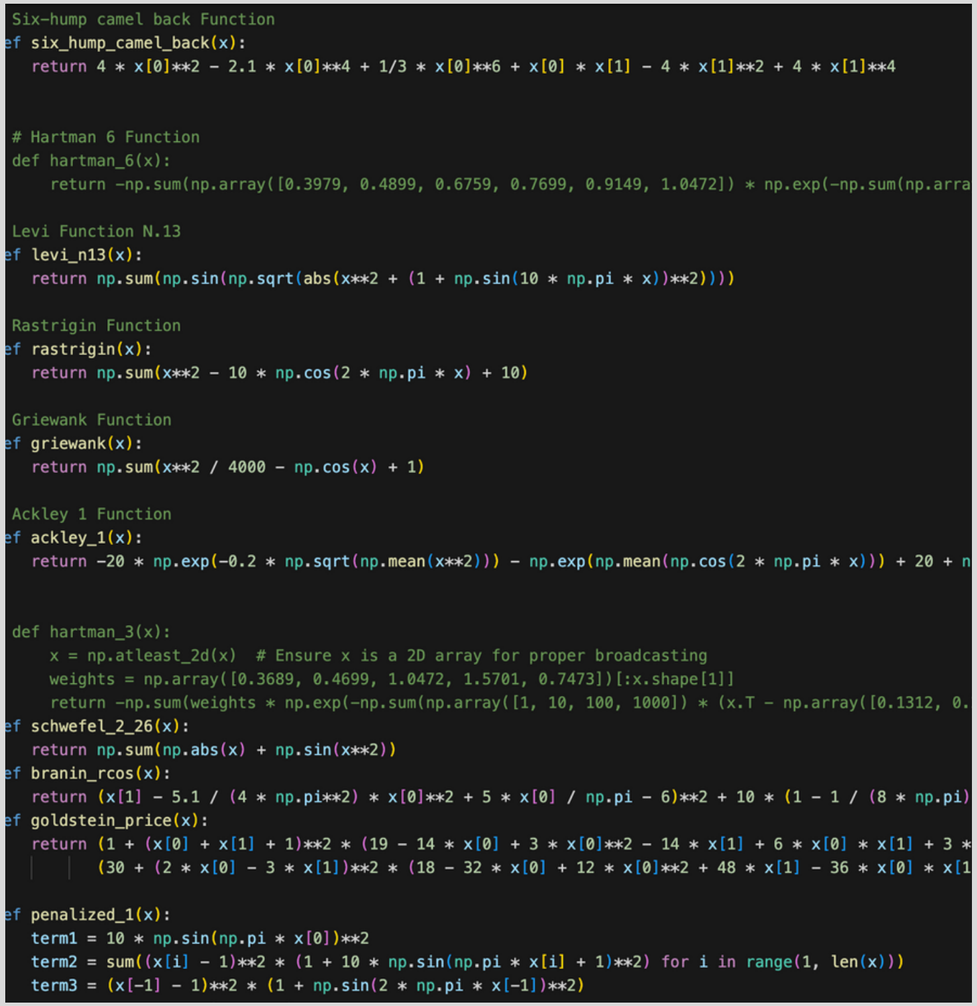
STANDARD PSO PROCESS:

PARTICLE INITIALIZATION:

- GENERATE RANDOM PARTICLES WITHIN SPECIFIED BOUNDS.
- INITIALIZE PARTICLE VELOCITIES.

OBJECTIVE FUNCTION DEFINITIONS:

- IMPLEMENT VARIOUS BENCHMARK FUNCTIONS (E.G., RASTRIGIN, GRIEWANK, ACKLEY) REPRESENTING OPTIMIZATION PROBLEMS.



```
Six-hump camel back Function
ef six_hump_camel_back(x):
   return 4 * x[0]**2 - 2.1 * x[0]**4 + 1/3 * x[0]**6 + x[0] * x[1] - 4 * x[1]**2 + 4 * x[1]**4


 # Hartman 6 Function
 def hartman_6(x):
     return -np.sum(np.array([0.3979, 0.4899, 0.6759, 0.7699, 0.9149, 1.0472]) * np.exp(-np.sum(np.arra

 Levi Function N.13
ef levi_n13(x):
   return np.sum(np.sin(np.sqrt(abs(x**2 + (1 + np.sin(10 * np.pi * x))**2))))

 Rastrigin Function
ef rastrigin(x):
   return np.sum(x**2 - 10 * np.cos(2 * np.pi * x) + 10)

 Griewank Function
ef griewank(x):
   return np.sum(x**2 / 4000 - np.cos(x) + 1)

 Ackley 1 Function
ef ackley_1(x):
   return -20 * np.exp(-0.2 * np.sqrt(np.mean(x**2))) - np.exp(np.mean(np.cos(2 * np.pi * x))) + 20 + n


 def hartman_3(x):
     x = np.atleast_2d(x)  # Ensure x is a 2D array for proper broadcasting
     weights = np.array([0.3689, 0.4699, 1.0472, 1.5701, 0.7473])[:x.shape[1]]
     return -np.sum(weights * np.exp(-np.sum(np.array([1, 10, 100, 1000]) * (x.T - np.array([0.1312, 0.
ef schwefel_2_26(x):
   return np.sum(np.abs(x) + np.sin(x**2))
ef branin_rcos(x):
   return (x[1] - 5.1 / (4 * np.pi**2) * x[0]**2 + 5 * x[0] / np.pi - 6)**2 + 10 * (1 - 1 / (8 * np.pi)
ef goldstein_price(x):
   return (1 + (x[0] + x[1] + 1)**2 * (19 - 14 * x[0] + 3 * x[0]**2 - 14 * x[1] + 6 * x[0] * x[1] + 3 *
           (30 + (2 * x[0] - 3 * x[1])**2 * (18 - 32 * x[0] + 12 * x[0]**2 + 48 * x[1] - 36 * x[0] * x[1

ef penalized_1(x):
   term1 = 10 * np.sin(np.pi * x[0])**2
   term2 = sum((x[i] - 1)**2 * (1 + 10 * np.sin(np.pi * x[i] + 1)**2) for i in range(1, len(x)))
   term3 = (x[-1] - 1)**2 * (1 + np.sin(2 * np.pi * x[-1])**2)
```

STANDARD PSO OVERVIEW

PSO IMPLEMENTATION:

IMPLEMENT THE **RUN_STANDARD_PSO FUNCTION**:

- TAKES AN OBJECTIVE FUNCTION, FUNCTION NAME, NUMBER OF PARTICLES, DIMENSIONS, AND OTHER PARAMETERS.
- INITIALIZES PARTICLES, VELOCITIES, AND CALCULATES INITIAL BEST VALUES.
- ITERATES THROUGH A SPECIFIED NUMBER OF ITERATIONS, UPDATING PARTICLE POSITIONS AND VELOCITIES BASED ON PSO EQUATIONS.

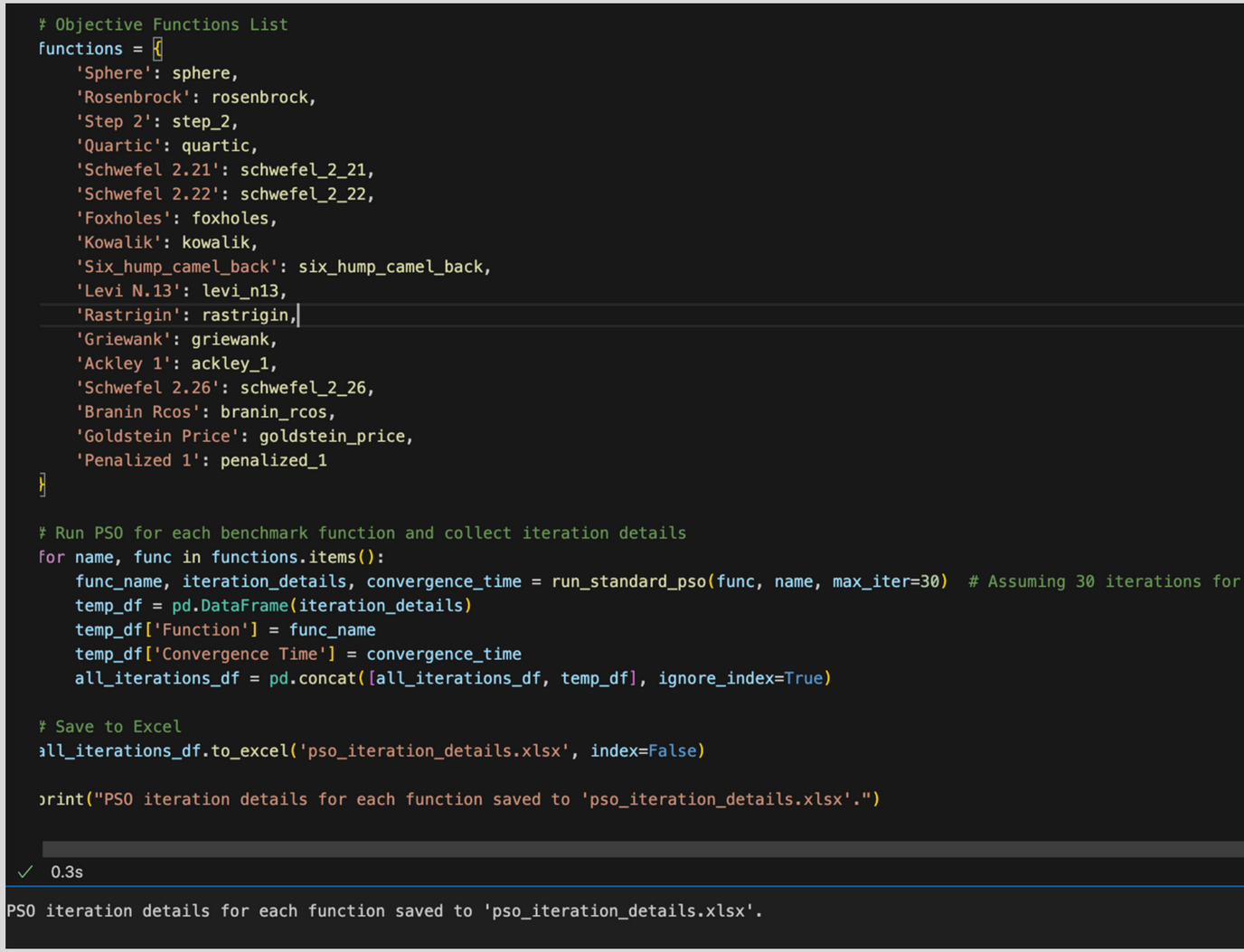
```python
# Modified Enhanced PSO Implementation to capture iteration-wise details
def run_standard_pso(obj_func, func_name, n_particles=30, n_dimensions=2, max_iter=100, bounds=(-10,
    start_time = time.time()  # Start timing
    particles = np.random.uniform(bounds[0], bounds[1], (n_particles, n_dimensions))
    velocity = np.zeros((n_particles, n_dimensions))
    pbest = np.copy(particles)
    pbest_values = np.array([obj_func(x) for x in particles])
    gbest_value = np.min(pbest_values)
    gbest = particles[np.argmin(pbest_values)]
    v_max = (bounds[1] - bounds[0]) * 0.2

    # Initialize a list to store the best value of each iteration
    iteration_details = []

    for t in range(max_iter):
        w = w_max - (w_max - w_min) * t / max_iter
        for i in range(n_particles):
            r1, r2 = np.random.rand(2)
            velocity[i] = w * velocity[i] + 1.49445 * r1 * (pbest[i] - particles[i]) + 1.49445 * r2
            velocity[i] = np.clip(velocity[i], -v_max, v_max)
            particles[i] += velocity[i]
            particles[i] = np.clip(particles[i], bounds[0], bounds[1])
            current_value = obj_func(particles[i])
            if current_value < pbest_values[i]:
                pbest[i] = particles[i]
                pbest_values[i] = current_value
                if current_value < gbest_value:
                    gbest = particles[i]
                    gbest_value = current_value
        # Record the global best value at this iteration
        iteration_details.append({'Iteration': t+1, 'Best Value': gbest_value})

    convergence_time = time.time() - start_time  # Measure convergence time

    return func_name, iteration_details, convergence_time

# Prepare DataFrame for all iteration details
all_iterations_df = pd.DataFrame()

# Objective Functions List
functions = {
    'Sphere': sphere,
```

STANDARD PSO OVERVIEW

PSO IMPLEMENTATION:

IMPLEMENT THE **RUN_STANDARD_PSO FUNCTION**:

- RECORDS THE BEST VALUE FOR EACH ITERATION.
- OUTPUTS THE FUNCTION NAME, ITERATION DETAILS, AND CONVERGENCE TIME.



```python
def adaptive_inertia_weight(iteration, max_iter, w_max=0.9, w_min=0.4):
    return w_max - (iteration / max_iter) * (w_max - w_min)

# Check for convergence (optional, based on problem specifics)
def has_converged(previous_best, current_best, threshold=1e-6):
    return np.abs(previous_best - current_best) < threshold


# Modified PSO Implementation
def run_modified_pso(obj_func, func_name, n_particles=30, n_dimensions=2, max_iter=100, bounds=(-10, 10)):

    start_time = time.time()  # Timing start
    particles = np.random.uniform(bounds[0], bounds[1], (n_particles, n_dimensions))
    velocity = np.zeros((n_particles, n_dimensions))
    pbest = np.copy(particles)
    pbest_values = np.array([obj_func(x) for x in particles])
    gbest_value = np.min(pbest_values)
    gbest = particles[np.argmin(pbest_values)]

    v_max = (bounds[1] - bounds[0]) / 2  # Adjust velocity clamping based on bounds

    previous_best = float('inf')

    iteration_details = []

    for t in range(max_iter):
        w = adaptive_inertia_weight(t, max_iter)  # Example of modifying the inertia weight over time
        for i in range(n_particles):
            r1, r2 = np.random.rand(2)
            velocity[i] = w * velocity[i] + \
                          2.5 * r1 * (pbest[i] - particles[i]) + \
                          2.5 * r2 * (gbest - particles[i])
            velocity[i] = np.clip(velocity[i], -v_max, v_max)
            particles[i] += velocity[i]
            particles[i] = np.clip(particles[i], bounds[0], bounds[1])
            current_value = obj_func(particles[i])
            if current_value < pbest_values[i]:
                pbest[i] = particles[i]
                pbest_values[i] = current_value
                if current_value < gbest_value:
                    gbest = particles[i]
                    gbest_value = current_value
```

MODIFIED PSO OVERVIEW

OBJECTIVE:

- THE MODIFIED PSO INTRODUCES THE CONCEPT OF MULTI-SWARM PSO (MSPSO)

.

KEY MODIFICATIONS IN MSPSO:

- ADAPTIVE INERTIA WEIGHT:

INTRODUCES AN ADAPTIVE INERTIA WEIGHT (ADAPTIVE_INERTIA_WEIGHT) THAT CHANGES OVER ITERATIONS. THIS DYNAMIC WEIGHT CAN ENHANCE EXPLORATION AND EXPLOITATION AT DIFFERENT STAGES OF OPTIMIZATION.

- VELOCITY CLAMPING ADJUSTMENT:

ADJUSTS THE VELOCITY CLAMPING RANGE (V_MAX) BASED ON THE PROBLEM BOUNDS. THIS ENSURES THAT PARTICLE VELOCITIES REMAIN WITHIN APPROPRIATE LIMITS.

- CONVERGENCE CHECK:

OPTIONAL HAS_CONVERGED FUNCTION CHECKS FOR CONVERGENCE BASED ON THE DIFFERENCE BETWEEN THE PREVIOUS AND CURRENT BEST VALUES.

```
                    gbest_value = current_value
            iteration_details.append({'Iteration': t+1, 'Best Value': gbest_value})

    convergence_time = time.time() - start_time

    return func_name, iteration_details, convergence_time

# Objective Functions List
# Objective Functions List
functions = {
    'Sphere': sphere,
    'Rosenbrock': rosenbrock,
    'Step 2': step_2,
    'Quartic': quartic,
    'Schwefel 2.21': schwefel_2_21,
    'Schwefel 2.22': schwefel_2_22,
    'Foxholes': foxholes,
    'Kowalik': kowalik,
    'Six_hump_camel_back': six_hump_camel_back,
    'Levi N.13': levi_n13,
    'Rastrigin': rastrigin,
    'Griewank': griewank,
    'Ackley 1': ackley_1,
    'Schwefel 2.26': schwefel_2_26,
    'Branin Rcos': branin_rcos,
    'Goldstein Price': goldstein_price,
    'Penalized 1': penalized_1
}
# Collecting iteration details for each function
all_iterations_df = pd.DataFrame()

for name, func in functions.items():
    func_name, iteration_details, convergence_time = run_modified_pso(func, name, max_iter=30)  # Assuming
    temp_df = pd.DataFrame(iteration_details)
    temp_df['Function'] = func_name
    temp_df['Convergence Time'] = convergence_time
    all_iterations_df = pd.concat([all_iterations_df, temp_df], ignore_index=True)

# Save to Excel
all_iterations_df.to_excel('modified_pso_iteration_details.xlsx', index=False)

print("Modified PSO iteration details for each function saved to 'modified_pso_iteration_details.xlsx'.")
```

MODIFIED PSO OVERVIEW

KEY DIFFERENCES BETWEEN STANDARD PSO AND MODIFIED PSO:

- **ADAPTIVE INERTIA WEIGHT:**
  - STANDARD PSO TYPICALLY USES A FIXED INERTIA WEIGHT. MSPSO ADAPTS THE INERTIA WEIGHT OVER ITERATIONS, PROVIDING MORE FLEXIBILITY.
- VELOCITY CLAMPING ADJUSTMENT:
  - MSPSO ADJUSTS THE VELOCITY CLAMPING RANGE BASED ON THE PROBLEM BOUNDS. THIS HELPS IN HANDLING DIVERSE OPTIMIZATION LANDSCAPES.
- CONVERGENCE CHECK (OPTIONAL):
  - MSPSO INCLUDES AN OPTIONAL CONVERGENCE CHECK BASED ON THE DIFFERENCE BETWEEN THE PREVIOUS AND CURRENT BEST VALUES. THIS CAN BE USEFUL FOR TERMINATING THE OPTIMIZATION PROCESS WHEN CONVERGENCE IS ACHIEVED.

BENCHMARK FUNCTIONS:

- **OBJECTIVE FUNCTIONS:**
  - BENCHMARK FUNCTIONS REMAIN CONSISTENT WITH THE STANDARD PSO, REPRESENTING VARIOUS OPTIMIZATION PROBLEMS (E.G., SPHERE, ROSENBROCK, RASTRIGIN).

T-VALUE OVERVIEW

LOAD DATA: THE ITERATION DETAILS FROM EXCEL FILES INTO PANDAS DATAFRAMES (DF_STANDARD AND DF_MODIFIED).

DATA FILTERING (OPTIONAL):
- CALCULATE THE MEAN AND STANDARD DEVIATION OF THE BEST VALUES FOR EACH ALGORITHM.
- DETERMINE THE SAMPLE SIZES

STANDARD ERROR CALCULATION:
CALCULATE THE STANDARD ERROR FOR EACH ALGORITHM.

T-STATISTIC CALCULATION:
COMPUTE THE T-STATISTIC USING THE MEANS AND STANDARD ERRORS.

DEGREES OF FREEDOM CALCULATION:
DETERMINE THE DEGREES OF FREEDOM FOR THE T-TEST.

P-VALUE CALCULATION:
CALCULATE THE TWO-TAILED P-VALUE USING THE SURVIVAL FUNCTION (SF) OF THE T-DISTRIBUTION.

PRINT RESULTS:
OUTPUT THE T-STATISTIC AND P-VALUE FOR INTERPRETATION.

```python
import numpy as np
import pandas as pd
from scipy import stats

# Load your data files. Replace 'standard_pso.xlsx' and 'modified_pso.xlsx' with your actual file paths.
df_standard = pd.read_excel('pso_iteration_details.xlsx')
df_modified = pd.read_excel('modified_pso_iteration_details.xlsx')

# Filter the data if needed, for example by function name
# df_standard = df_standard[df_standard['Function'] == 'some_function']
# df_modified = df_modified[df_modified['Function'] == 'some_function']

# Calculate the means and standard deviations of the best values for each algorithm
mean_standard = df_standard['Best Value'].mean()
mean_modified = df_modified['Best Value'].mean()

std_standard = df_standard['Best Value'].std(ddof=1)  # ddof=1 provides the sample standard deviation
std_modified = df_modified['Best Value'].std(ddof=1)

# Calculate the sample sizes
n_standard = df_standard['Best Value'].count()
n_modified = df_modified['Best Value'].count()

# Calculate the standard error
se_standard = std_standard / np.sqrt(n_standard)
se_modified = std_modified / np.sqrt(n_modified)

# Calculate the t-statistic
t_stat = (mean_standard - mean_modified) / np.sqrt(se_standard**2 + se_modified**2)

# Calculate the degrees of freedom for the t-test
df = ((se_standard**2 + se_modified**2)**2) / ((se_standard**4 / (n_standard - 1)) + (se_modified**4 / (n_modified

# Get the p-value
p_value = stats.t.sf(np.abs(t_stat), df) * 2  # Multiply by 2 for a two-tailed test

print(f'The t-statistic is: {t_stat}')
print(f'The p-value is: {p_value}')
```

✓ 0.0s

```
The t-statistic is: -2.4571849826863588
The p-value is: 0.01430990597682046
```

RESULTS

1. **PERFORMANCE DIFFERENCES:** THE STUDY COMPARED STANDARD AND MODIFIED PSO ALGORITHMS IN FIFTEEN RUNS, REVEALING PERFORMANCE VARIATIONS.

2. **SIGNIFICANT OUTCOMES:** FIVE RUNS SHOWED SIGNIFICANT DIFFERENCES (P VALUES < 0.05), INDICATING UNIQUE PERFORMANCES.

3. **STANDARD PSO'S WIN:** RUN 1 DISPLAYED STANDARD PSO OUTPERFORMING WITH A T STATISTIC OF -3.98 AND P VALUE 7.56E-05.

4. **MODIFIED PSO'S EDGE:** RUN 2 HAD MODIFIED PSO OUTSHINING WITH A T STATISTIC OF 3.71 AND P VALUE 0.0002.

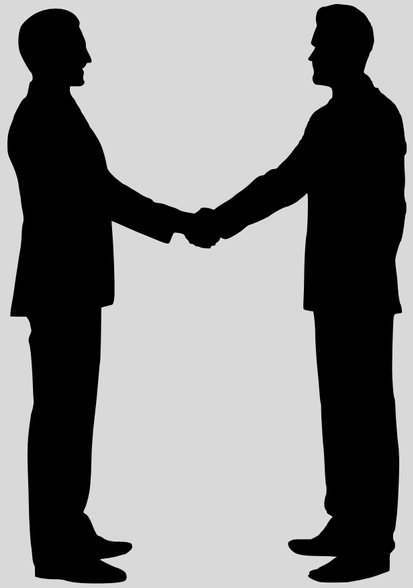
5. **CONTEXT MATTERS:** IN TEN RUNS, BOTH ALGORITHMS PERFORMED SIMILARLY (P VALUES > 0.05), SUGGESTING CONTEXTUAL DEPENDENCE.

6. **QUANTITATIVE INSIGHT:** T STATISTICS AND P VALUES OFFER A QUANTITATIVE BASIS FOR ALGORITHMIC ASSESSMENT, EMPHASIZING THE NEED FOR TAILORED ALGORITHM CHOICES BASED ON SPECIFIC OPTIMIZATION SCENARIOS.

TABLE II
COMPARISON OF STANDARD PSO AND MODIFIED PSO

Run Number	T Statistic	P Value
1	-3.985718982	7.55818E-05
2	3.705283609	0.000233149
3	1.060350368	0.28942394
4	-2.711930369	0.006866426
5	-0.418491619	0.675677445
6	0.333863812	0.738556262
7	-0.838626901	0.401904102
8	1.60782158	0.108218916
9	-0.666309834	0.505491121
10	0.701415135	0.483206898
11	0.952812646	0.341088497
13	-2.712489451	0.006887818
14	-1.596384516	0.110719654
15	-2.250297042	0.024649837

CONCLUSIONS



ALGORITHM CUSTOMIZATION CONSIDERATIONS:

THE STUDY UNDERSCORES THE IMPORTANCE OF TAILORING OPTIMIZATION ALGORITHMS TO THE UNIQUE FEATURES OF THE OPTIMIZATION PROBLEM AT HAND. ALGORITHMIC MODIFICATIONS SHOULD NOT BE UNIVERSALLY APPLIED, AND CAREFUL CONSIDERATION OF THE OPTIMIZATION LANDSCAPE IS CRUCIAL FOR ACHIEVING OPTIMAL RESULTS.

TRADE-OFFS AND COMPLEXITY:

MSPSO DEMONSTRATED PERFORMANCE TRADE-OFFS, EXCELLING IN CERTAIN FUNCTIONS WHILE FACING CHALLENGES IN OTHERS. THIS COMPLEXITY EMPHASIZES THE NEED FOR A NUANCED APPROACH TO ALGORITHM SELECTION AND MODIFICATION, TAKING INTO ACCOUNT BOTH ADVANTAGES AND POTENTIAL DRAWBACKS.

FUTURE DIRECTIONS:

DEEPER EXPLORATION OF MSPSO'S BEHAVIOR, ADAPTIVE STRATEGIES, AND DYNAMIC PARAMETER CHANGES. COMPARATIVE ANALYSES WITH OTHER OPTIMIZATION ALGORITHMS AND REAL-WORLD APPLICATIONS ARE CRUCIAL FOR VALIDATING THE ALGORITHM'S PRACTICAL UTILITY AND IMPROVING ITS OVERALL EFFICIENCY.