```
# Clone the CEC2017 repo (no-op if already present)
import os
if not os.path.exists('cec2017-py'):
    print('Cloning cec2017-py...')
    # use os.system so this works both in local notebooks and Colab
    rc = os.system('git clone https://github.com/tilleyd/cec2017-py')
    if rc != 0:
        print('Warning: git clone returned', rc)
else:
    print('cec2017-py already present')
```

```
Cloning cec2017-py...
```

```
# Imports: add cloned repo to path, import function factory
import sys, os
sys.path.append(os.path.abspath('cec2017-py'))
import numpy as np
from cec2017.functions import all_functions
import matplotlib.pyplot as plt
import math
import pandas as pd
print('Imports ok')
```

```
Imports ok
```

```
# Small helper to adapt CEC functions to batch input
def evaluate(func, X):
    # CEC2017 functions expect 2D input: (N, D) where N is number of samples
    # Ensure X is always 2D
    if X.ndim == 1:
        X = X.reshape(1, -1)
    # Call function and ensure output is 1D array
    vals = np.asarray(func(X))
    return vals.flatten()
```

```
# Levy flight helper (Mantegna's algorithm)
def levy_flight(D, beta=1.5):
    """Generate a single Levy flight step vector"""
    sigma_u = (math.gamma(1 + beta) * math.sin(math.pi * beta / 2) /
               (math.gamma((1 + beta) / 2) * beta * 2 ** ((beta - 1) / 2))) ** (1 / beta)
    u = np.random.normal(0, sigma_u, size=D)
    v = np.random.normal(0, 1, size=D)
    step = u / (np.abs(v) ** (1.0 / beta))
    return step

def levy_flight_vector(N, D, beta=1.5):
    """Generate Levy flight matrix for N agents with D dimensions"""
    sigma_u = (math.gamma(1 + beta) * math.sin(math.pi * beta / 2) /
               (math.gamma((1 + beta) / 2) * beta * 2 ** ((beta - 1) / 2))) ** (1 / beta)
    u = np.random.normal(0, sigma_u, size=(N, D))
    v = np.random.normal(0, 1, size=(N, D))
    step = u / (np.abs(v) ** (1.0 / beta))
    return step
```

```
# Marine Predators Algorithm (MPA) - Official implementation from paper
# Reference: A. Faramarzi et al., Expert Systems with Applications, 2020
# DOI: doi.org/10.1016/j.eswa.2020.113377
def mpa(func, rng, D=30, N=30, LB=-100, UB=100, Tmax=1000, FADs=0.2, P=0.5):
    """
    Marine Predators Algorithm

    Parameters:
    - func: objective function to minimize
    - rng: numpy random generator
    - D: dimension
    - N: number of search agents (prey)
    - LB, UB: lower and upper bounds
    - Tmax: maximum iterations
    - FADs: Fish Aggregating Devices effect probability (default 0.2)
    - P: constant (default 0.5)
    """
    # Initialize prey population
    Prey = rng.uniform(LB, UB, size=(N, D))

    # Bounds arrays
```

```
Xmin = np.tile(LB * np.ones(D), (N, 1))
Xmax = np.tile(UB * np.ones(D), (N, 1))

# Initialize top predator
Top_predator_pos = np.zeros(D)
Top_predator_fit = np.inf

# Initialize fitness and history
fitness = np.full(N, np.inf)
Convergence_curve = np.zeros(Tmax)
stepsize = np.zeros((N, D))

# Marine Memory
Prey_old = Prey.copy()
fit_old = fitness.copy()

for Iter in range(Tmax):
    # Detecting top predator
    for i in range(N):
        # Boundary checking
        Flag4ub = Prey[i, :] > UB
        Flag4lb = Prey[i, :] < LB
        Prey[i, :] = (Prey[i, :] * (~(Flag4ub | Flag4lb)) +
                      UB * Flag4ub + LB * Flag4lb)

        # Fitness evaluation
        fitness[i] = evaluate(func, Prey[i:i+1, :])[0]

        # Update top predator
        if fitness[i] < Top_predator_fit:
            Top_predator_fit = fitness[i]
            Top_predator_pos = Prey[i, :].copy()

    # Marine Memory saving
    if Iter == 0:
        fit_old = fitness.copy()
        Prey_old = Prey.copy()

    Inx = fit_old < fitness
    Indx = np.tile(Inx[:, np.newaxis], (1, D))
    Prey = Indx * Prey_old + ~Indx * Prey
    fitness = Inx * fit_old + ~Inx * fitness

    fit_old = fitness.copy()
    Prey_old = Prey.copy()

    # Elite matrix (Eq. 10)
    Elite = np.tile(Top_predator_pos, (N, 1))

    # CF calculation
    CF = (1 - Iter / Tmax) ** (2 * Iter / Tmax)

    # Levy and Brownian random vectors
    RL = 0.05 * levy_flight_vector(N, D, 1.5)
    RB = rng.standard_normal((N, D))

    # Update prey positions based on phase
    for i in range(N):
        for j in range(D):
            R = rng.random()

            # Phase 1: High velocity ratio (Eq. 12)
            if Iter < Tmax / 3:
                stepsize[i, j] = RB[i, j] * (Elite[i, j] - RB[i, j] * Prey[i, j])
                Prey[i, j] = Prey[i, j] + P * R * stepsize[i, j]

            # Phase 2: Unit velocity ratio (Eqs. 13 & 14)
            elif Iter < 2 * Tmax / 3:
                if i > N / 2:
                    stepsize[i, j] = RB[i, j] * (RB[i, j] * Elite[i, j] - Prey[i, j])
                    Prey[i, j] = Elite[i, j] + P * CF * stepsize[i, j]
                else:
                    stepsize[i, j] = RL[i, j] * (Elite[i, j] - RL[i, j] * Prey[i, j])
                    Prey[i, j] = Prey[i, j] + P * R * stepsize[i, j]

            # Phase 3: Low velocity ratio (Eq. 15)
            else:
                stepsize[i, j] = RL[i, j] * (RL[i, j] * Elite[i, j] - Prey[i, j])
                Prey[i, j] = Elite[i, j] + P * CF * stepsize[i, j]
```

```
        # Detecting top predator (after movement)
        for i in range(N):
            # Boundary checking
            Flag4ub = Prey[i, :] > UB
            Flag4lb = Prey[i, :] < LB
            Prey[i, :] = (Prey[i, :] * (~(Flag4ub | Flag4lb)) +
                          UB * Flag4ub + LB * Flag4lb)

            # Fitness evaluation
            fitness[i] = evaluate(func, Prey[i:i+1, :])[0]

            # Update top predator
            if fitness[i] < Top_predator_fit:
                Top_predator_fit = fitness[i]
                Top_predator_pos = Prey[i, :].copy()

        # Marine Memory saving (second time)
        if Iter == 0:
            fit_old = fitness.copy()
            Prey_old = Prey.copy()

        Inx = fit_old < fitness
        Indx = np.tile(Inx[:, np.newaxis], (1, D))
        Prey = Indx * Prey_old + ~Indx * Prey
        fitness = Inx * fit_old + ~Inx * fitness

        fit_old = fitness.copy()
        Prey_old = Prey.copy()

        # Eddy formation and FADs effect (Eq. 16)
        if rng.random() < FADs:
            U = rng.random((N, D)) < FADs
            Prey = Prey + CF * ((Xmin + rng.random((N, D)) * (Xmax - Xmin)) * U)
        else:
            r = rng.random()
            Rs = N
            # Random permutations for FADs effect
            idx1 = rng.permutation(Rs)
            idx2 = rng.permutation(Rs)
            stepsize = (FADs * (1 - r) + r) * (Prey[idx1, :] - Prey[idx2, :])
            Prey = Prey + stepsize

        # Save convergence
        Convergence_curve[Iter] = Top_predator_fit

    return Convergence_curve
```

Marine Predators Algorithm (MPA) - Official Implementation

This is a direct Python translation of the **official MATLAB implementation** from the original paper:

Reference:
A. Faramarzi, M. Heidarinejad, S. Mirjalili, A.H. Gandomi,
Marine Predators Algorithm: A Nature-inspired Metaheuristic
Expert Systems with Applications, 2020
DOI: 10.1016/j.eswa.2020.113377

Key features:

- **3-phase strategy**: High velocity → Unit velocity → Low velocity
- **Marine Memory**: Preserves best solutions found so far
- **Levy and Brownian motion**: For exploration and exploitation
- **FADs effect** (Fish Aggregating Devices): Eddy formation and environmental changes
- **Adaptive CF parameter**: Controls exploration-exploitation balance

```
# PSO Standard implementation
def pso_standard(func, rng, D=30, N=30, LB=-100, UB=100, c1=2.0, c2=2.0, Tmax=1000):
    """Algorithme PSO classique — suit la meilleure valeur globale à chaque itération."""
    x = rng.uniform(LB, UB, size=(N, D))
    v = np.zeros((N, D))
    pbest = x.copy()
    pbest_values = evaluate(func, x)

    g_best = pbest[np.argmin(pbest_values)].copy()
```

```
    history = np.zeros(Tmax)

    for t in range(Tmax):
        w = 0.9 - 0.5 * t / Tmax   # inertie décroissante
        r1, r2 = rng.random((N, D)), rng.random((N, D))

        v = w * v + c1 * r1 * (pbest - x) + c2 * r2 * (g_best - x)
        x = np.clip(x + v, LB, UB)
        fitness = evaluate(func, x)

        improved = fitness < pbest_values
        if np.any(improved):
            pbest[improved] = x[improved]
            pbest_values[improved] = fitness[improved]

        g_best = pbest[np.argmin(pbest_values)]
        history[t] = pbest_values.min()

    return history
```

```
# PSO Hybrid implementation
def pso_hybrid(func, rng, D=30, N=30, LB=-100, UB=100, c1=2.0, c2=2.0, Tmax=1000,
               p_mut=0.05, sigma=25.0):
    """PSO avec mutation gaussienne — améliore l'exploration de l'espace de recherche."""
    x = rng.uniform(LB, UB, size=(N, D))
    v = np.zeros((N, D))
    pbest = x.copy()
    pbest_values = evaluate(func, x)

    g_best = pbest[np.argmin(pbest_values)].copy()
    history = np.zeros(Tmax)

    for t in range(Tmax):
        w = 0.9 - 0.5 * t / Tmax
        r1, r2 = rng.random((N, D)), rng.random((N, D))

        v = w * v + c1 * r1 * (pbest - x) + c2 * r2 * (g_best - x)
        x += v

        # Mutation gaussienne sur certaines coordonnées
        mutation_mask = rng.random((N, D)) < p_mut
        if np.any(mutation_mask):
            x += mutation_mask * rng.normal(0.0, sigma, size=(N, D))

        x = np.clip(x, LB, UB)
        fitness = evaluate(func, x)

        improved = fitness < pbest_values
        if np.any(improved):
            pbest[improved] = x[improved]
            pbest_values[improved] = fitness[improved]

        g_best = pbest[np.argmin(pbest_values)]
        history[t] = pbest_values.min()

    return history
```

```
# Genetic Algorithm implementation
def genetic_algorithm(func, rng, D=30, N=30, LB=-100, UB=100, Tmax=1000, mutation_scale=0.005):
    """
    Algorithme génétique (GA) simple :
    - Croisement à un point (split = D//2)
    - Mutation d'un gène aléatoire par individu
    - Élitisme (on garde toujours les N meilleurs individus)
    """
    split = D // 2
    mutation_amplitude = mutation_scale * (UB - LB)

    parents = rng.uniform(LB, UB, size=(N, D))
    fitness = evaluate(func, parents)
    history = np.zeros(Tmax)

    for t in range(Tmax):
        # Sélection aléatoire de paires de parents
        j = rng.integers(0, N, size=N)
        k = rng.integers(0, N, size=N)

        # Croisement à un point
```

```
        enfants = np.hstack((parents[j, :split], parents[k, split:]))

        # Mutation mono-gène
        mutation_idx = rng.integers(0, D, size=N)
        enfants[np.arange(N), mutation_idx] += rng.uniform(
            -mutation_amplitude, mutation_amplitude, size=N
        )
        np.clip(enfants, LB, UB, out=enfants)

        # Évaluation et élitisme
        enfants_fit = evaluate(func, enfants)
        combined = np.vstack((parents, enfants))
        combined_fit = np.concatenate((fitness, enfants_fit))

        best_idx = np.argsort(combined_fit)[:N]
        parents, fitness = combined[best_idx], combined_fit[best_idx]

        history[t] = fitness[0]  # meilleure valeur de la génération

    return history
```

```
# Helper: block averaging (same idea as in your example)
def moyenne_blocs(values, pas=30, evals_par_iter=30):
    n_blocs = len(values) // pas
    moyennes = [np.mean(values[i * pas : (i + 1) * pas]) for i in range(n_blocs)]
    x_axis = np.arange(1, n_blocs + 1) * pas * evals_par_iter
    return np.array(x_axis), np.array(moyennes)
```

```
# Select the target functions f2, f4, f12, f25 from all_functions
funcs = all_functions  # all_functions is already a list, not a callable
# all_functions is a list (0-based index) where f1 is index 0
targets = []
want = [2, 4, 12, 25]
for n in want:
    try:
        # Use 1-based indexing (f2 is funcs[1], etc.)
        targets.append((f'f{n}', funcs[n-1]))
    except (IndexError, TypeError) as e:
        raise RuntimeError(f'Could not find f{n} in all_functions list (error: {e})')

print('Selected target functions:', [t[0] for t in targets])
```

```
Selected target functions: ['f2', 'f4', 'f12', 'f25']
```

```
# Run all 4 algorithms on each target function
DIM = 30
POP = 30
TMAX = 1000
LB, UB = -100, 100
STEP = 30  # block size for averaging

results = {}
for name, f in targets:
    print(f'\n🔹 Running algorithms on {name}...')

    # Use a fresh RNG for each function to ensure reproducibility
    rng = np.random.default_rng(42)

    results[name] = {
        "MPA": mpa(f, rng=np.random.default_rng(42), D=DIM, N=POP, LB=LB, UB=UB, Tmax=TMAX),
        "PSO Standard": pso_standard(f, rng=np.random.default_rng(43), D=DIM, N=POP, LB=LB, UB=UB, Tmax=TMAX
        "PSO Hybrid": pso_hybrid(f, rng=np.random.default_rng(44), D=DIM, N=POP, LB=LB, UB=UB, Tmax=TMAX),
        "GA": genetic_algorithm(f, rng=np.random.default_rng(45), D=DIM, N=POP, LB=LB, UB=UB, Tmax=TMAX)
    }

    # Print best values
    for algo_name, hist in results[name].items():
        print(f'  {algo_name}: best = {hist.min():.6e}')

print('\n✅ All algorithms completed.')
```

```

🔹 Running algorithms on f2...
WARNING: f2 has been deprecated from the CEC 2017 benchmark suite
  MPA: best = 1.162738e+06
  PSO Standard: best = 4.172775e+32
  PSO Hybrid: best = 1.212160e+28
  GA: best = 9.473716e+32
```

```
🔹 Running algorithms on f4...
  MPA: best = 4.673007e+02
  PSO Standard: best = 9.982769e+02
  PSO Hybrid: best = 8.428550e+02
  GA: best = 5.410576e+03

🔹 Running algorithms on f12...
  MPA: best = 4.897408e+04
  PSO Standard: best = 1.050561e+10
  PSO Hybrid: best = 4.526917e+08
  GA: best = 4.780028e+09

🔹 Running algorithms on f25...
  MPA: best = 2.887118e+03
  PSO Standard: best = 4.302897e+03
  PSO Hybrid: best = 3.195238e+03
  GA: best = 4.568456e+03

✅ All algorithms completed.
```

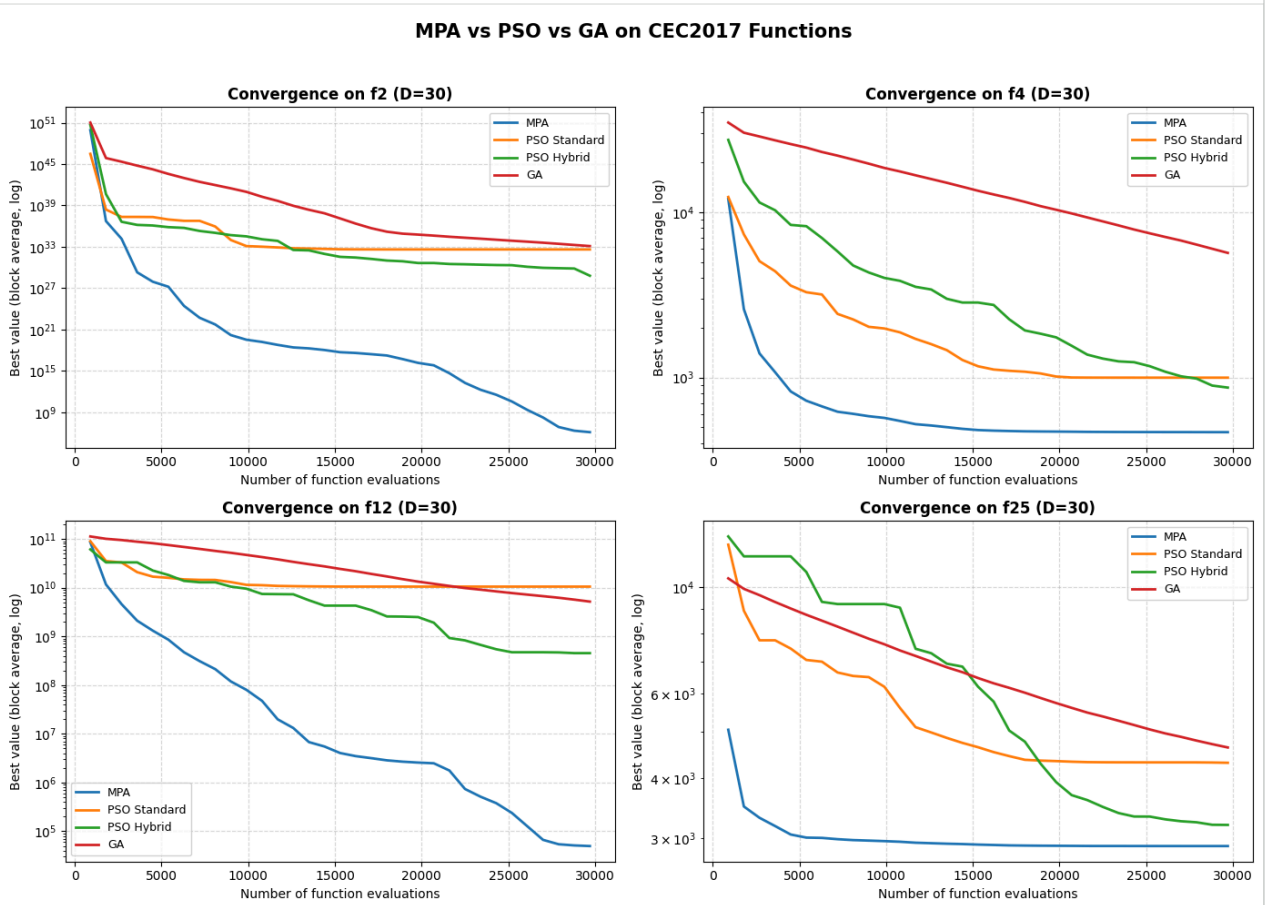
```python
# Create 2x2 subplot comparing all algorithms on each function
plt.figure(figsize=(14, 10))

for i, (func_name, algos) in enumerate(results.items(), start=1):
    plt.subplot(2, 2, i)

    for algo_name, hist in algos.items():
        x, y = moyenne_blocs(hist, pas=STEP, evals_par_iter=POP)
        plt.plot(x, y, label=algo_name, linewidth=2)

    plt.title(f"Convergence on {func_name} (D={DIM})", fontsize=12, fontweight='bold')
    plt.xlabel("Number of function evaluations", fontsize=10)
    plt.ylabel("Best value (block average, log)", fontsize=10)
    plt.grid(True, linestyle="--", alpha=0.5)
    plt.yscale("log")
    plt.legend(fontsize=9, loc='best')

plt.suptitle("MPA vs PSO vs GA on CEC2017 Functions", fontsize=15, fontweight="bold")
plt.tight_layout(rect=[0, 0, 1, 0.96])
plt.show()
```



Results Summary

The notebook now compares **4 algorithms** on CEC2017 functions f2, f4, f12, and f25:

1. **MPA (Marine Predators Algorithm)** - 3-phase strategy with Levy flights and FADs effect
2. **PSO Standard** - Classic particle swarm with decreasing inertia
3. **PSO Hybrid** - PSO + Gaussian mutation for enhanced exploration
4. **GA (Genetic Algorithm)** - Simple GA with elitism and single-point crossover

Key observations:

- The plot shows convergence curves (log scale) for each algorithm
- X-axis: Total function evaluations (iterations × population size)
- Y-axis: Best fitness value found (block-averaged over 30-iteration windows)

Next steps:

- Run multiple independent trials (e.g., 30 runs) to compute mean ± std for robust comparison
- Tune hyperparameters for each algorithm
- Add statistical tests (Wilcoxon, Friedman) to assess significance
- Test on more CEC2017 functions

Complete CEC2017 Benchmark - MPA Only

Run MPA on all 30 CEC2017 functions with multiple trials to compute statistical metrics.

```
# Run MPA on all 30 CEC2017 functions with 30 independent trials
import time
import warnings
warnings.filterwarnings('ignore')  # Suppress CEC2017 deprecation warnings

# Parameters
DIM = 30
POP = 30
TMAX = 1000
LB, UB = -100, 100
N_RUNS = 30  # Number of independent runs

# Get all 30 CEC2017 functions
all_funcs = all_functions

print(f'Starting MPA benchmark on {len(all_funcs)} CEC2017 functions...')
print(f'Parameters: DIM={DIM}, POP={POP}, TMAX={TMAX}, RUNS={N_RUNS}\n')

# Store results
benchmark_results = []

for func_idx, func in enumerate(all_funcs, start=1):
    func_name = f'f{func_idx}'
    print(f'Running {func_name}... ', end='', flush=True)

    start_time = time.time()
    best_values = []

    # Run MPA N_RUNS times with different seeds
    for run in range(N_RUNS):
        try:
            rng = np.random.default_rng(42 + run)
            convergence = mpa(func, rng=rng, D=DIM, N=POP, LB=LB, UB=UB, Tmax=TMAX)
            best_values.append(convergence[-1])  # Final best value
        except Exception as e:
            print(f'\n  Warning: Run {run+1} failed: {str(e)}')
            continue

    # Calculate statistics
    if len(best_values) == 0:
        print(f'FAILED - All runs failed')
        continue

    best_values = np.array(best_values)
    mean_val = np.mean(best_values)
    std_val = np.std(best_values)
    best_val = np.min(best_values)
```

```
    elapsed = time.time() - start_time
    print(f'Done in {elapsed:.1f}s (Mean: {mean_val:.6e}, Std: {std_val:.6e}, Best: {best_val:.6e})')

    # Store result
    benchmark_results.append({
        'Algorithm': 'MPA',
        'Function': func_name,
        'Mean': mean_val,
        'Std': std_val,
        'Best': best_val
    })

print('\n✅ Benchmark completed!')

# Create DataFrame and save to CSV
df_results = pd.DataFrame(benchmark_results)
csv_filename = 'MPA_CEC2017_Results.csv'
df_results.to_csv(csv_filename, index=False)
print(f'\n📊 Results saved to: {csv_filename}')

# Display results
print('\n' + '='*80)
print('MPA PERFORMANCE ON CEC2017 BENCHMARK')
print('='*80)
print(df_results.to_string(index=False))
print('='*80)
```

```
Running f13... Done in 406.1s (Mean: 1.859695e+03, Std: 2.544882e+02, Best: 1.534686e+03)
Running f14... Done in 395.6s (Mean: 1.474350e+03, Std: 1.426921e+01, Best: 1.449922e+03)
Running f15... Done in 389.5s (Mean: 1.617459e+03, Std: 3.373013e+01, Best: 1.553869e+03)
Running f16... Done in 441.7s (Mean: 2.271378e+03, Std: 1.746138e+02, Best: 1.894554e+03)
Running f17... Done in 646.6s (Mean: 1.854975e+03, Std: 7.585288e+01, Best: 1.752741e+03)
Running f18... Done in 457.5s (Mean: 1.903561e+03, Std: 3.557464e+01, Best: 1.849406e+03)
Running f19... Done in 547.0s (Mean: 1.941943e+03, Std: 8.456194e+00, Best: 1.925631e+03)
Running f20... Done in 626.0s (Mean: 2.234404e+03, Std: 7.589845e+01, Best: 2.054978e+03)
Running f21... Done in 549.0s (Mean: 2.324352e+03, Std: 7.504670e+01, Best: 2.204426e+03)
Running f22... Done in 660.3s (Mean: 2.304819e+03, Std: 3.059921e+00, Best: 2.300399e+03)
Running f23... Done in 747.4s (Mean: 2.704414e+03, Std: 5.268306e+01, Best: 2.445089e+03)
Running f24... Done in 704.1s (Mean: 2.887082e+03, Std: 1.541867e+01, Best: 2.851115e+03)
Running f25... Done in 770.0s (Mean: 2.892334e+03, Std: 1.463563e+01, Best: 2.883677e+03)
Running f26... Done in 893.1s (Mean: 2.911224e+03, Std: 7.356395e+01, Best: 2.826591e+03)
Running f27... Done in 944.8s (Mean: 3.215987e+03, Std: 1.282727e+01, Best: 3.186516e+03)
Running f28... Done in 916.6s (Mean: 3.229899e+03, Std: 2.246851e+01, Best: 3.197526e+03)
Running f29... Done in 1586.0s (Mean: 3.596615e+03, Std: 9.632174e+01, Best: 3.374766e+03)
Running f30... Done in 1423.3s (Mean: 1.015649e+04, Std: 4.093013e+03, Best: 5.775289e+03)

✅ Benchmark completed!

📊 Results saved to: MPA_CEC2017_Results.csv

================================================================================
MPA PERFORMANCE ON CEC2017 BENCHMARK
================================================================================
Algorithm Function         Mean          Std         Best
      MPA       f1 3.347470e+05 3.869072e+05 9.913552e+03
      MPA       f2 3.302031e+13 1.045793e+14 4.576349e+09
      MPA       f3 9.094151e+02 6.560634e+02 3.281961e+02
      MPA       f4 4.924978e+02 2.130697e+01 4.590676e+02
      MPA       f5 5.963076e+02 1.849244e+01 5.527672e+02
      MPA       f6 6.160661e+02 4.999068e+00 6.061396e+02
      MPA       f7 8.518715e+02 2.784211e+01 8.066976e+02
      MPA       f8 8.807702e+02 1.386639e+01 8.518471e+02
      MPA       f9 1.290207e+03 3.680904e+02 9.475264e+02
      MPA      f10 4.085446e+03 4.742278e+02 3.011081e+03
      MPA      f11 1.172833e+03 3.115724e+01 1.125428e+03
      MPA      f12 6.427620e+05 1.140559e+06 3.979146e+04
      MPA      f13 1.859695e+03 2.544882e+02 1.534686e+03
      MPA      f14 1.474350e+03 1.426921e+01 1.449922e+03
      MPA      f30 1.015649e+04 4.093013e+03 5.775289e+03
```

```
MPA      T50 1.015049e+04 4.095015e+03 5.775289e+03
_____________________________________________________________________________
```